

# Exceptions (finish) / Virtual Memory 1

# last time (1)

## SIMD intrinsic examples

process = thread + address space

thread = illusion of own CPU via time multiplexing

address space = illusion of own memory via address translation

exceptions = CPU jumps to operating system

external events (timer, I/O, ...)

internal events (privileged instruction, system call, ...)

## last time (2)

### exception dispatch

*exception table* points to *exception handlers*

exception table/handlers written by operating system

processor uses reorder buffer to save registers as if instructions executed in order

### user versus kernel mode

enter kernel mode via exception

exit kernel mode via return-from-exception

operating system chooses what runs in kernel mode via exception handlers

# types of exceptions

interrupts — externally-triggered

timer — keep program from hogging CPU

I/O devices — key presses, hard drives, networks, ...

aborts — hardware is broken

traps — intentionally triggered exceptions

system calls — ask OS to do something

faults — errors/events in programs

memory not in address space (“Segmentation fault”)

privileged instruction

divide by zero

invalid instruction

**asynchronous**

not triggered by  
running program

**synchronous**

triggered by  
current program

# kernel services

allocating memory? (change address space)

reading/writing to file? (communicate with hard drive)

read input? (communicate with keyboard)

all need privileged instructions!

need to **run code in kernel mode**

# Linux x86-64 system calls

special instruction: `syscall`

triggers `trap` (deliberate exception)

# Linux syscall calling convention

before `syscall`:

`%rax` — system call number

`%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9` — args

after `syscall`:

`%rax` — return value

on error: `%rax` contains -1 times “error number”

**almost** the same as normal function calls

# Linux x86-64 hello world

```
.globl _start
.data
hello_str: .asciz "Hello, World!\n"
.text
_start:
    movq $1, %rax # 1 = "write"
    movq $1, %rdi # file descriptor 1 = stdout
    movq $hello_str, %rsi
    movq $15, %rdx # 15 = strlen("Hello, World!\n")
    syscall

    movq $60, %rax # 60 = exit
    movq $0, %rdi
    syscall
```



# approx. system call handler

```
sys_call_table:  
    .quad handle_read_syscall  
    .quad handle_write_syscall  
    // ...  
  
handle_syscall:  
    ... // save old PC, etc.  
    pushq %rcx // save registers  
    pushq %rdi  
    ...  
    call *sys_call_table(,%rax,8)  
    ...  
    popq %rdi  
    popq %rcx  
    return_from_exception
```

# Linux system call examples

`mmap`, `brk` — allocate memory

`fork` — create new process

`execve` — run a program in the current process

`_exit` — terminate a process

`open`, `read`, `write` — access files  
terminals, etc. count as files, too

# system call wrappers

can't write C code to generate syscall instruction

solution: call “wrapper” function written in assembly

# types of exceptions

interrupts — externally-triggered

timer — keep program from hogging CPU

I/O devices — key presses, hard drives, networks, ...

aborts — hardware is broken

traps — intentionally triggered exceptions

system calls — ask OS to do something

faults — errors/events in programs

memory not in address space (“Segmentation fault”)

privileged instruction

divide by zero

invalid instruction

**asynchronous**

not triggered by  
running program

**synchronous**

triggered by  
current program

# a note on terminology (1)

real world: inconsistent terms for exceptions

we will follow textbook's terms in this course

the real world won't

you might see:

- 'interrupt' meaning what we call 'exception' (x86)

- 'exception' meaning what we call 'fault'

- 'hard fault' meaning what we call 'abort'

- 'trap' meaning what we call 'fault'

- ... and more

## a note on terminology (2)

we use the term “kernel mode”

some additional terms:

- supervisor mode

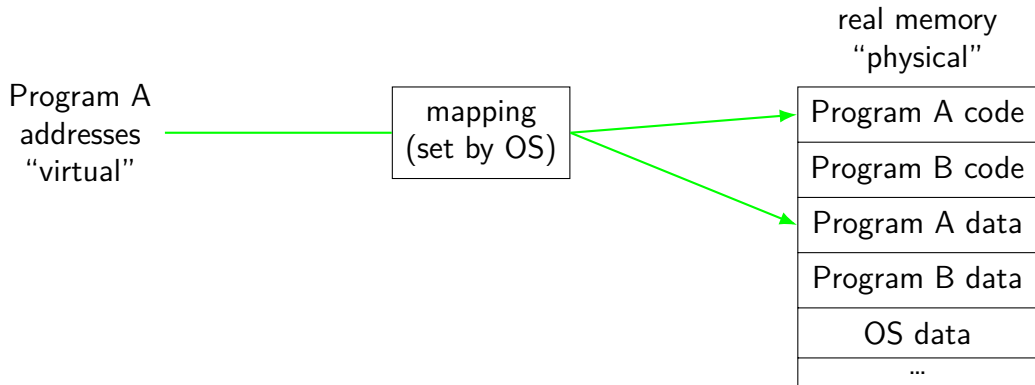
- privileged mode

- ring 0

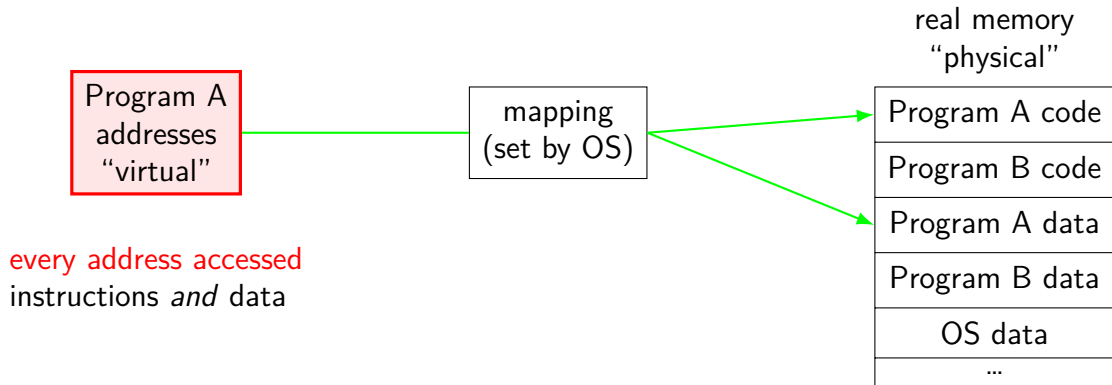
some systems have **multiple levels** of privilege

- different sets of privileged operations work

# address translation

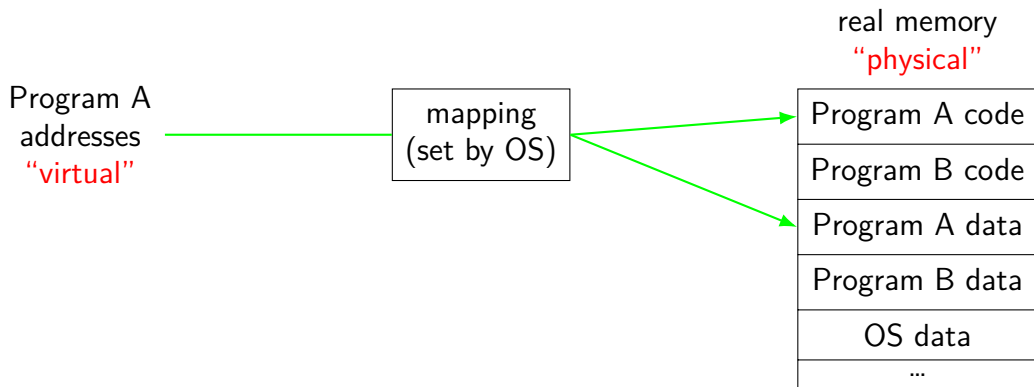


# address translation



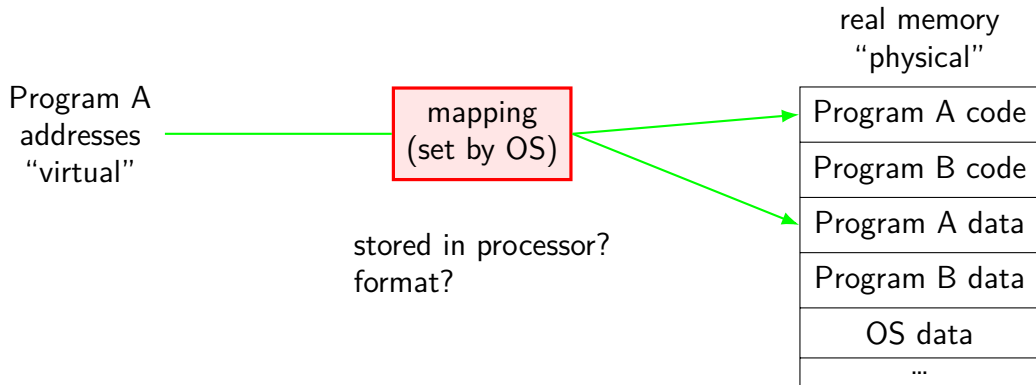


# address translation

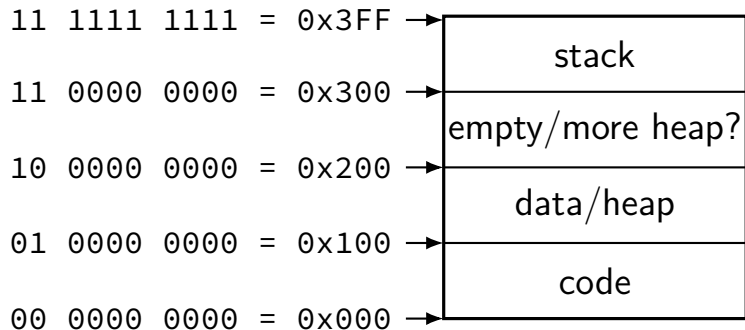


program addresses are 'virtual'  
real addresses are 'physical'  
can be **different sizes!**

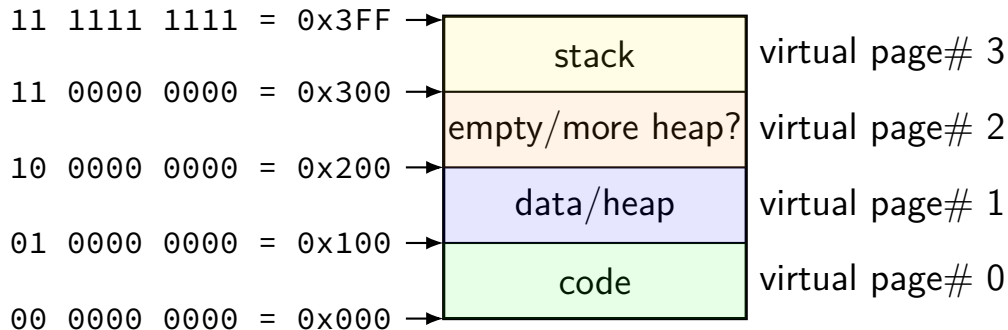
# address translation



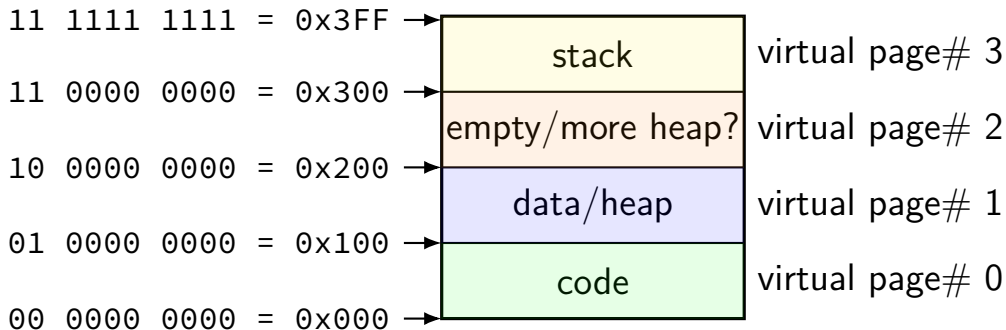
# toy program memory



# toy program memory

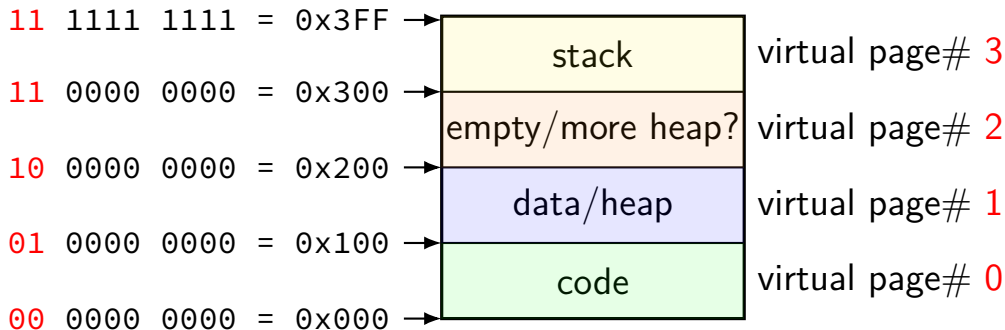


# toy program memory



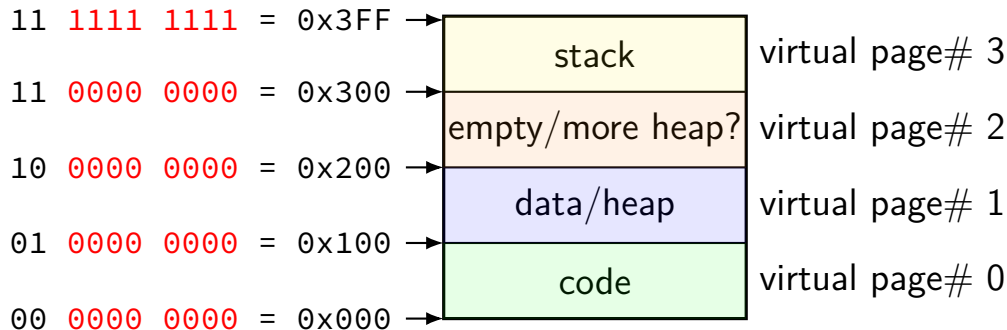
divide memory into **pages** ( $2^8$  bytes in this case)  
“virtual” = addresses the program sees

# 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

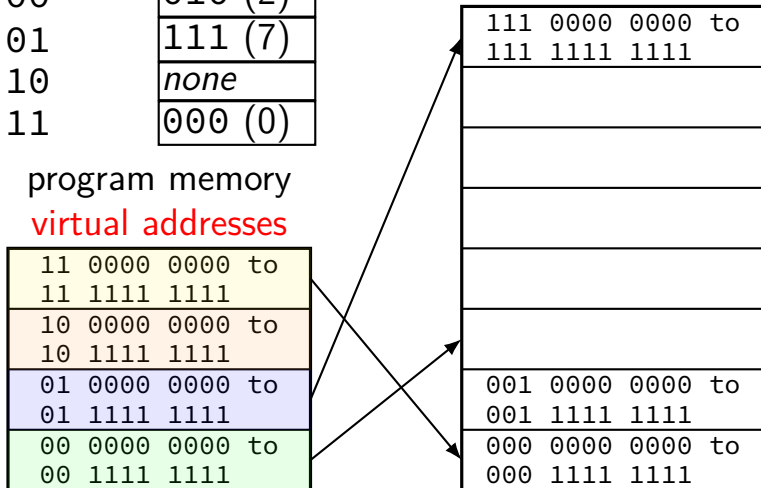
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	<i>none</i>
11	000 (0)

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)

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)

“page table entry”

111 1101 0010

trigger exception if 0?

to cache (data or instruction)

# tov 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?

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

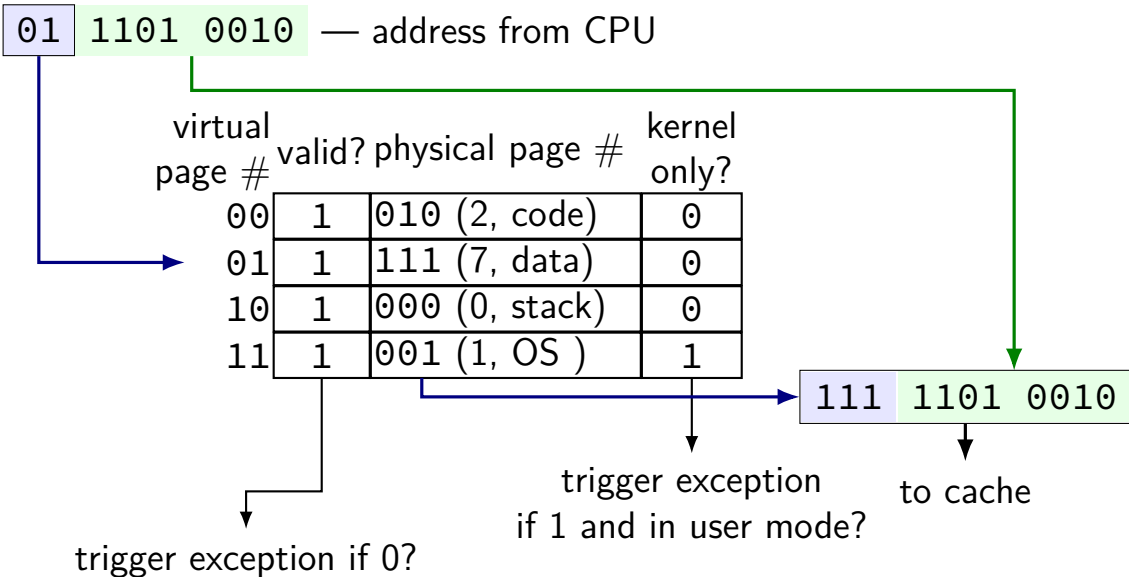
code better not be modified by user program

- otherwise: uncontrolled way to “escape” user mode

# 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





# 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

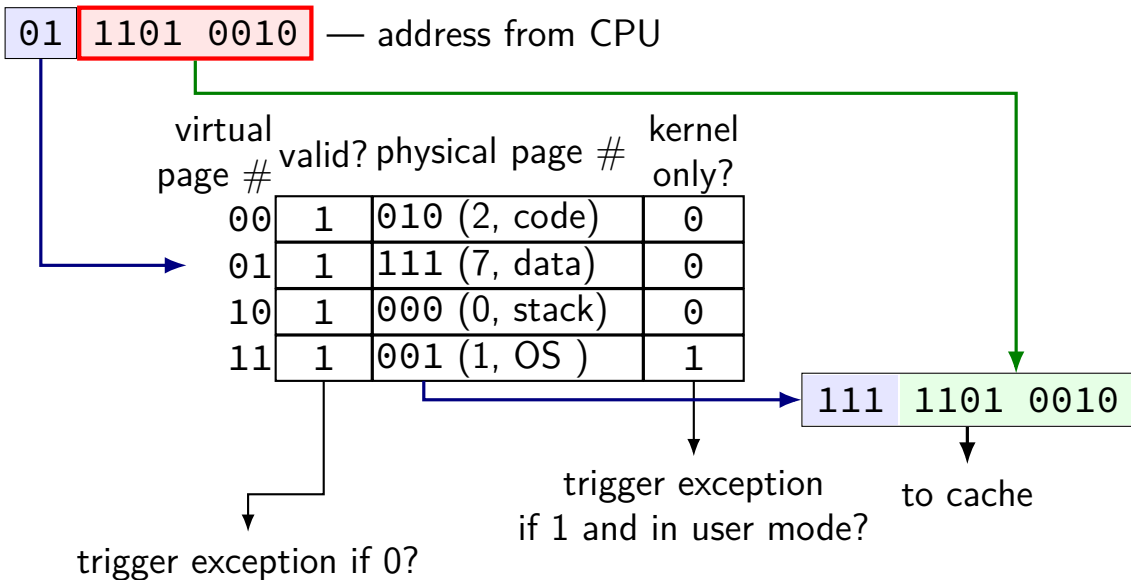
trigger exception if 0?

trigger exception  
if 1 and in user mode?

111 1101 0010

to cache

# kernel-mode only



## 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**

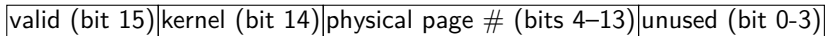
page table entry layout

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

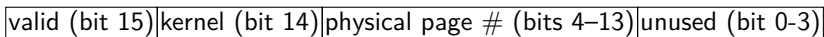
0x00010000



# 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

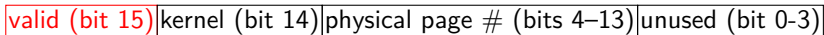
addresses	bytes
0x00000000-1	00000000 00000000
...	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
...	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010



# page tables in memory

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

page table entry layout



page table base register

0x00010000

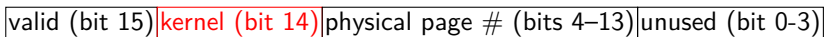
physical memory

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

# page tables in memory

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

page table entry layout



page table base register

0x00010000

physical memory

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

# page tables in memory

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

page table entry layout



page table base register

0x00010000

physical memory

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

# page tables in memory

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

page table entry layout

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

page table base register

0x00010000

page table (logically)

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

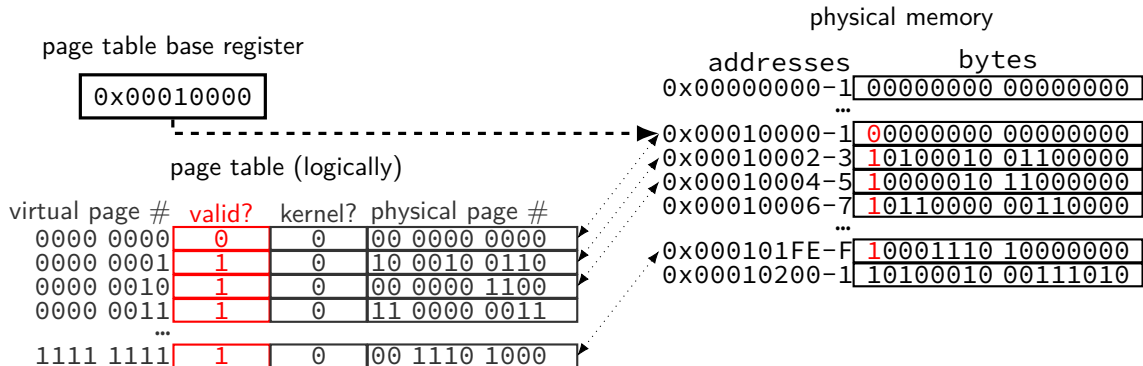
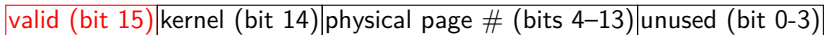
physical memory

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

# page tables in memory

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

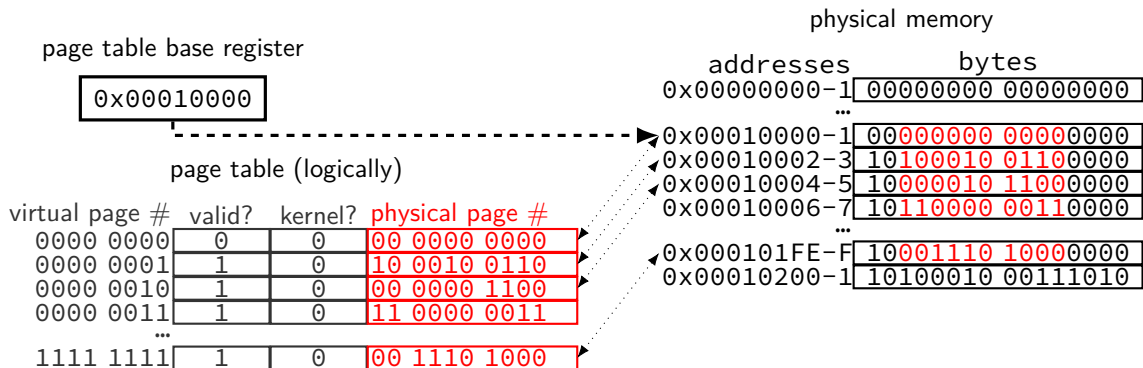
page table entry layout



# page tables in memory

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

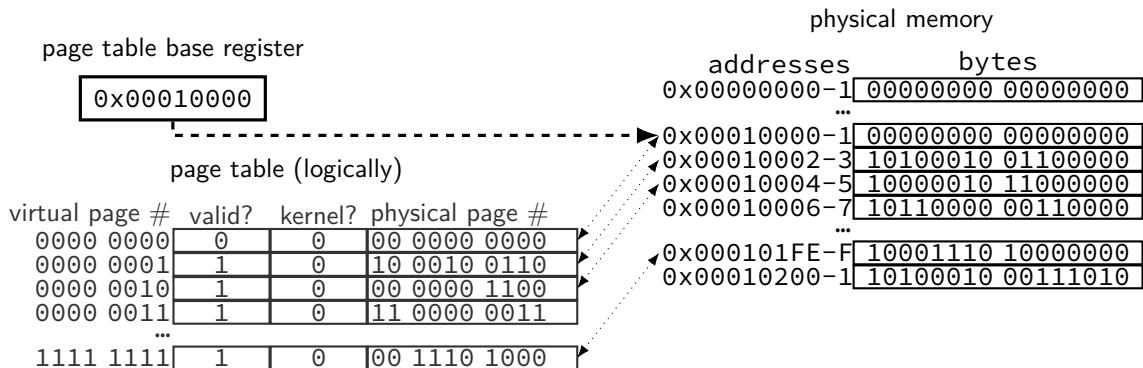
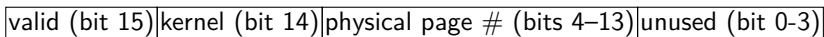
page table entry layout



# page tables in memory

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

page table entry layout



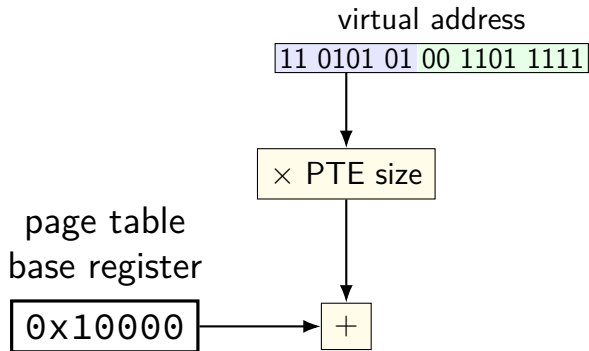
# memory access with page table

virtual address

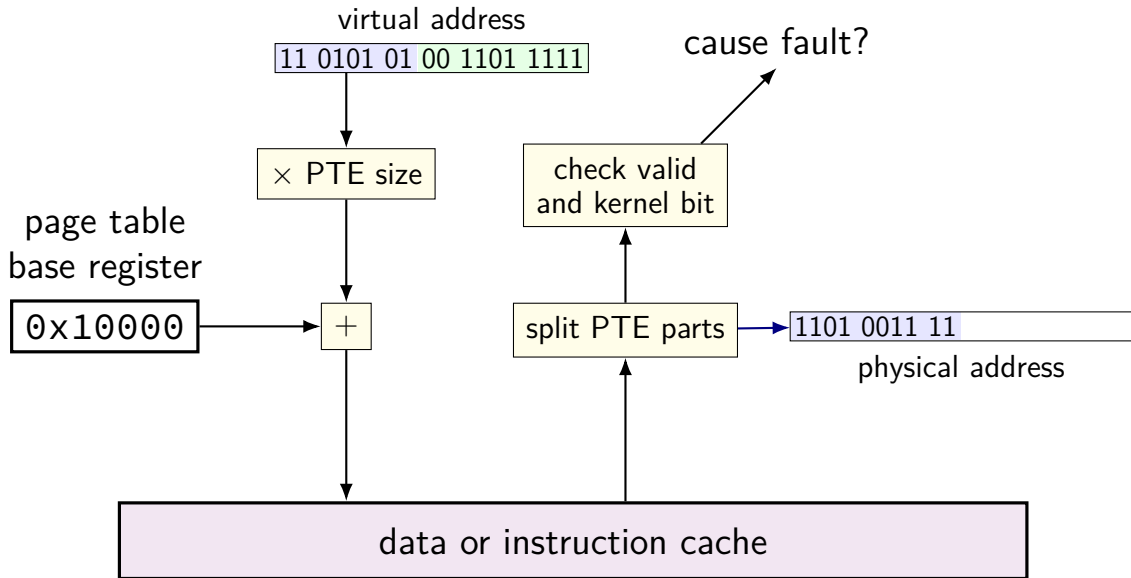
11 0101 01 00 1101 1111



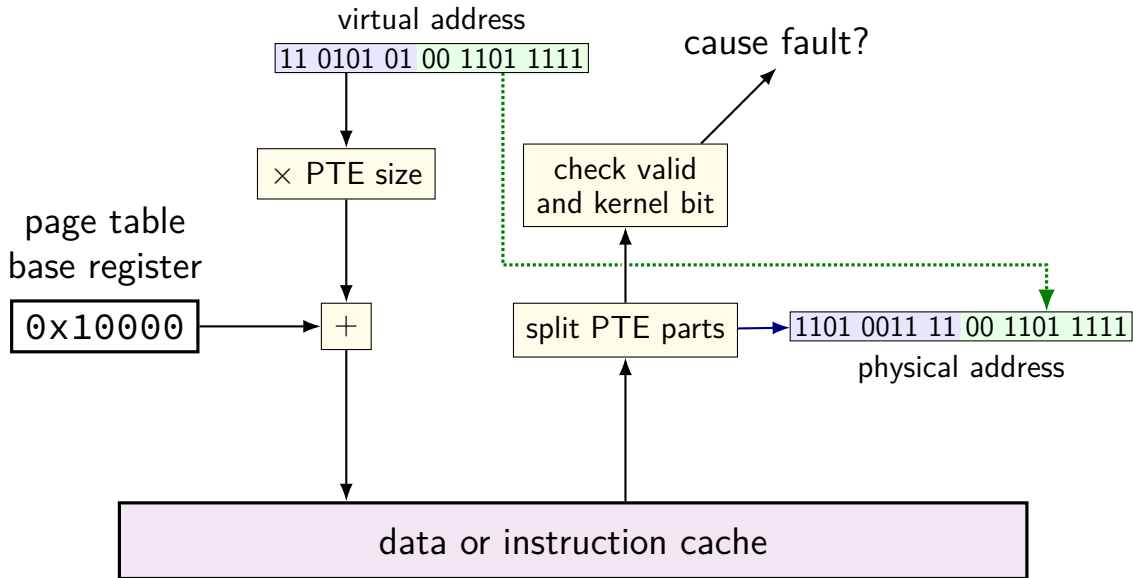
# memory access with page table



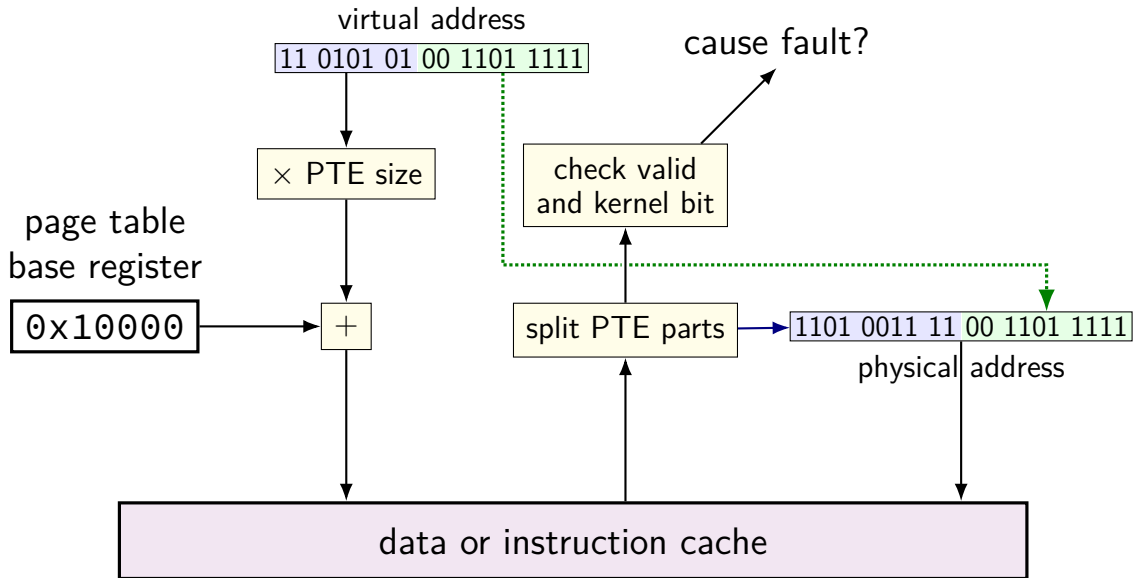
# memory access with page table



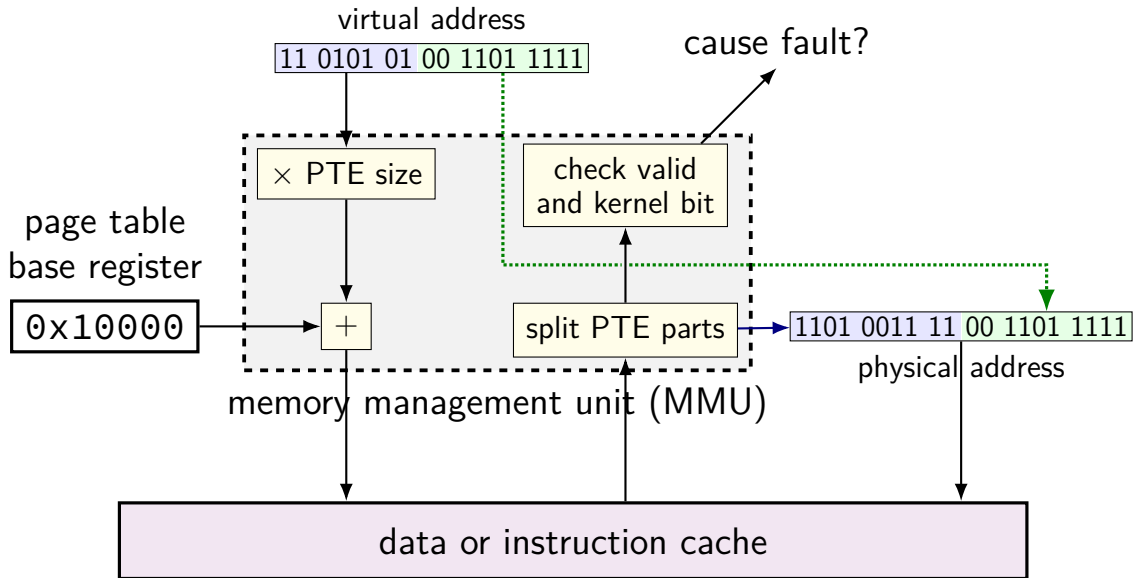
# memory access with page table



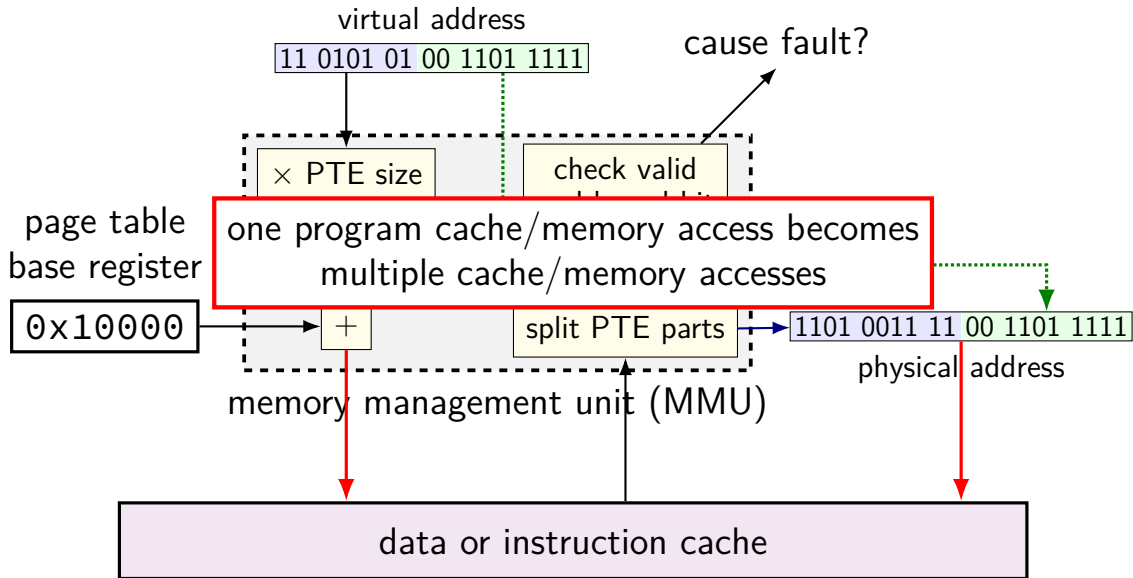
# memory access with page table



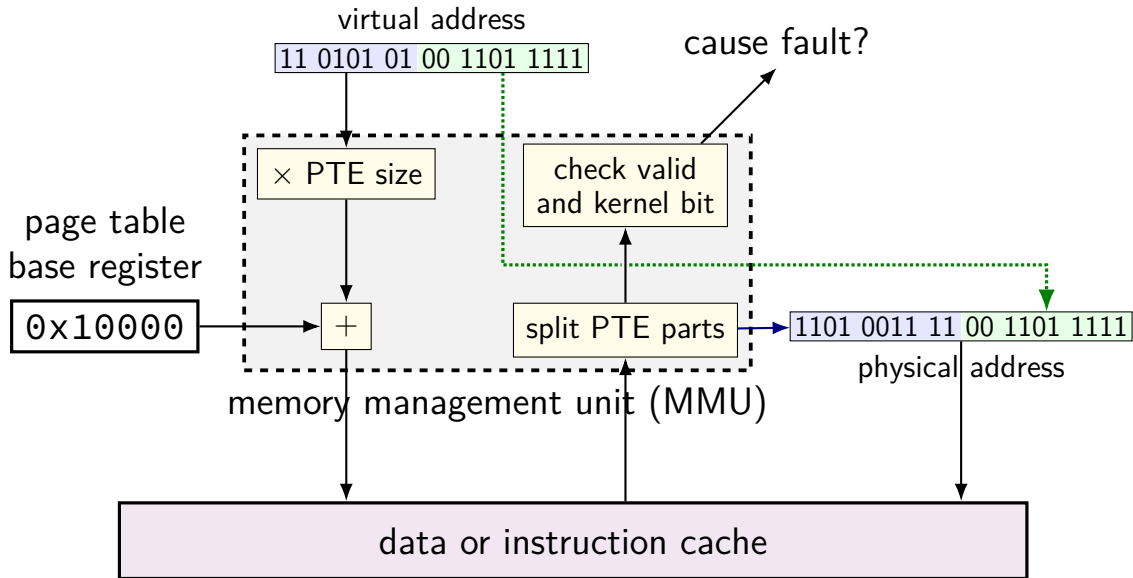
# memory access with page table



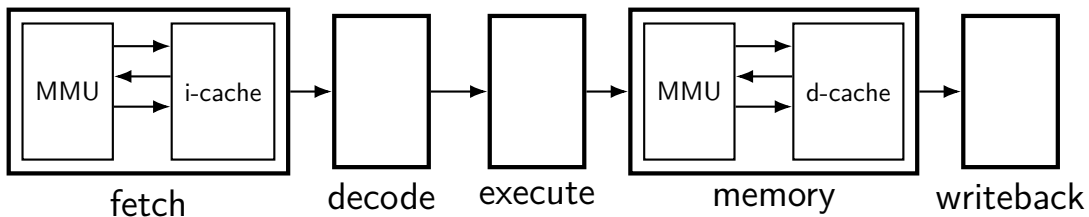
# memory access with page table



# memory access with page table



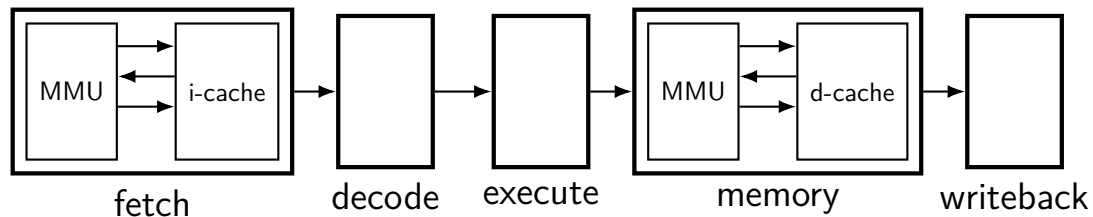
# MMUs in the pipeline



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)

## exercise: 64-bit system

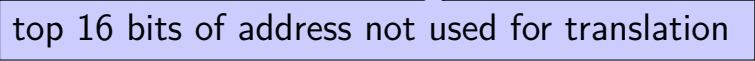
my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

## exercise: 64-bit system

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

4096 byte pages



top 16 bits of address not used for translation

## exercise: 64-bit system

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

4096 byte pages

exercise: how many page table entries?

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?  $2^{48}/2^{12} = 2^{36}$  entries

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

## exercise: 64-bit system

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

4096 byte pages

exercise: how many page table entries?  $2^{48}/2^{12} = 2^{36}$  entries

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

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

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

## two big problems to solve later

two **extra cache accesses** seems really slow

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

page tables seem **really huge**

solution: not just a flat table

# exercise setup

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

page table

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

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

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



# exercise setup

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

page table

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

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

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

phys. page 0

phys. page 1

# exercise

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

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

page table

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

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

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

# exercise

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

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

page table

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

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

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

# exercise

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

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

page table

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

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

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

# exercise

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

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

page table

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

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

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

# exercise

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

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

page table

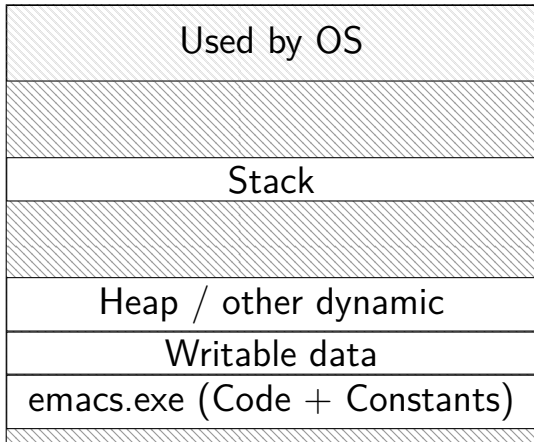
virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

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

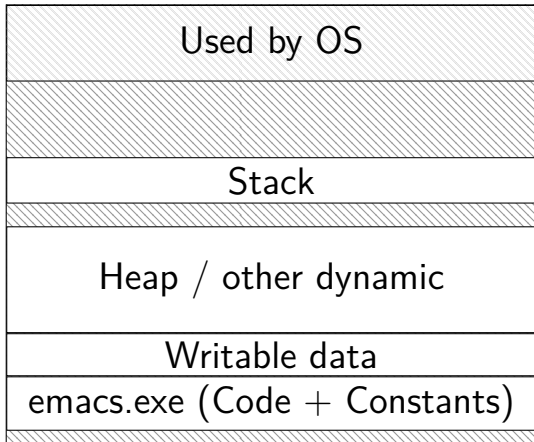
physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	CB 0B CB 0B
$0x38-B$	DC 0C DC 0C
$0x3C-F$	EC 0C EC 0C

# emacs (two copies)

Emacs (run by user mst3k)

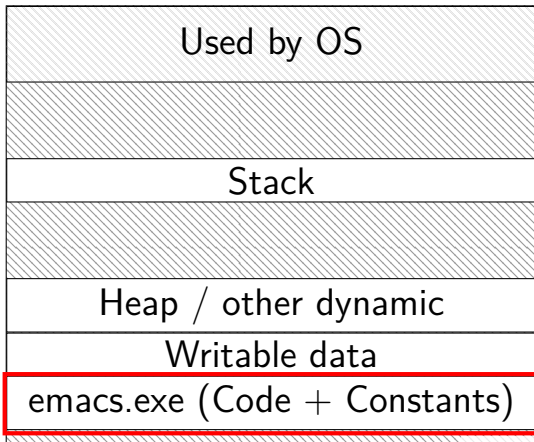


Emacs (run by user xyz4w)

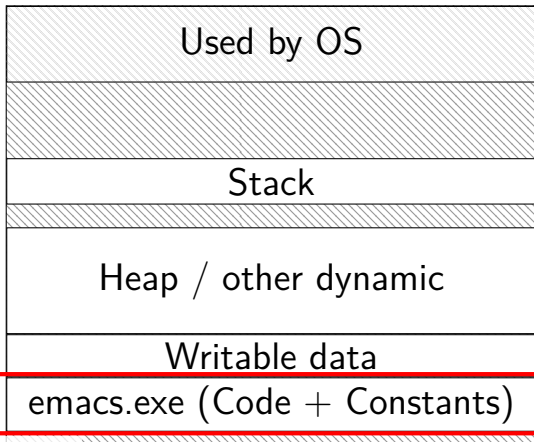


# emacs (two copies)

Emacs (run by user mst3k)



Emacs (run by user xyz4w)



same data?



## two copies of program

would like to only have one copy of program

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

would break process abstraction:

“illusion of own memory”

# typical page table entries

solution: same idea as kernel-only bit

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

can read?

can write?

can execute?

checked by MMU like valid/kernel bit

page table (logically)

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

# shared libraries

C standard library has lots of common code:

```
printf  
fopen  
qsort  
...
```

would like to not have multiple copies

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

# efficient shared libraries

recall: linking

- placeholders to fill in with addresses

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

different code depending on where library ends up?

one solution: **position-independent code**

**backup slides**

# 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**

# page table tricks

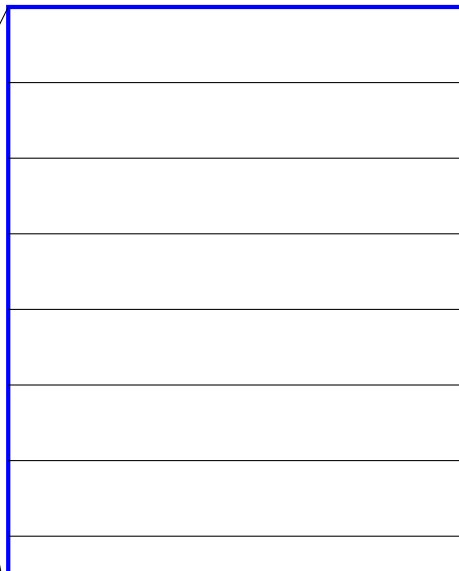
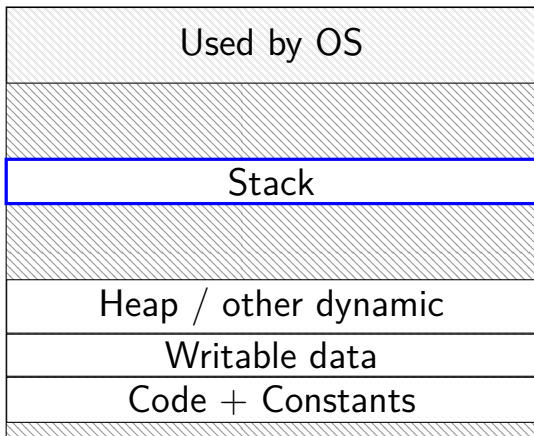
page tables let operating systems do 'magic' with memory

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

instead: change page tables and rerun memory access  
similar idea to trap-and-emulate

# space on demand

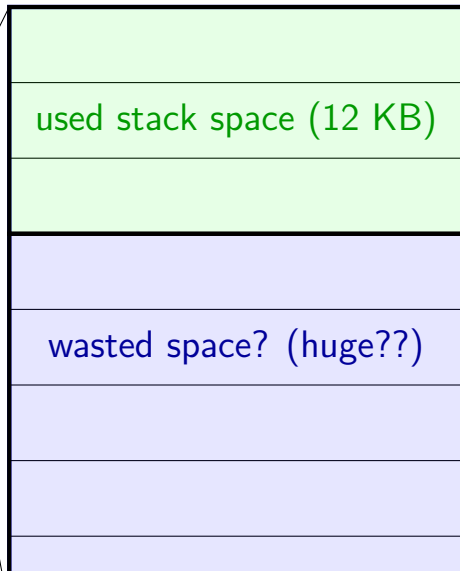
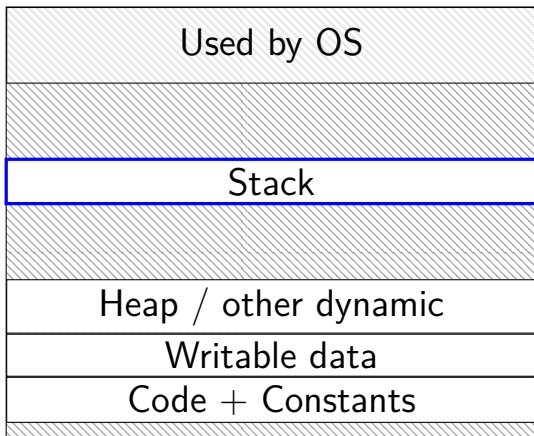
Program Memory





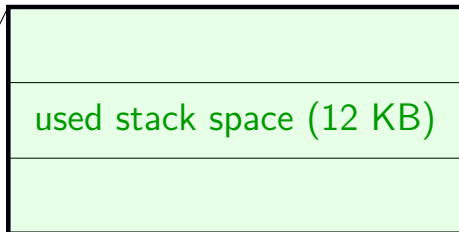
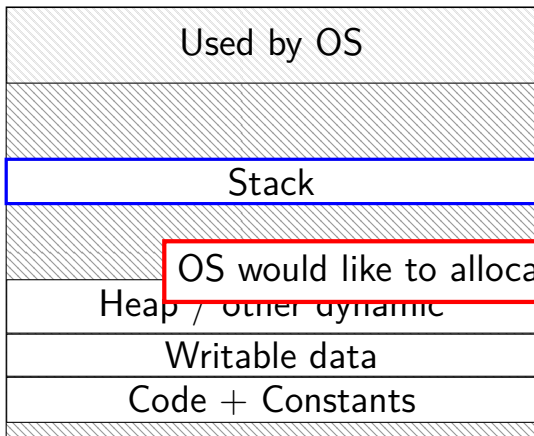
# space on demand

Program Memory

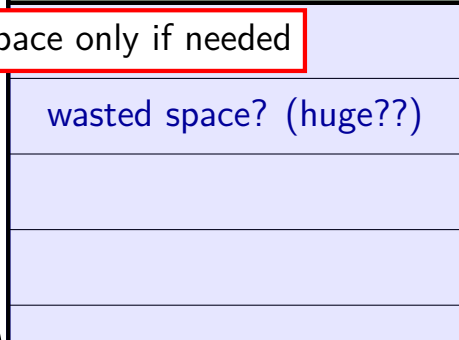


# space on demand

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
...	...	...
<b>0x7FFFB</b>	<b>1</b>	<b>0x200D8</b>
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

# page tricks generally

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

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

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

e.g. “addresses `0x7FFF8000` to `0x7FFFFFFF` are the stack”

might talk about Linux data structures later (book section 9.7)

page table is for the hardware and not the OS

# hardware help for page table tricks

information about the address causing the fault

e.g. special register with memory address accessed

harder alternative: OS disassembles instruction, look at registers

(by default) rerun faulting instruction when returning from exception

precise exceptions: no side effects from faulting instruction or after

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

e.g. instructions reordered after faulting instruction not visible



# fast copies

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

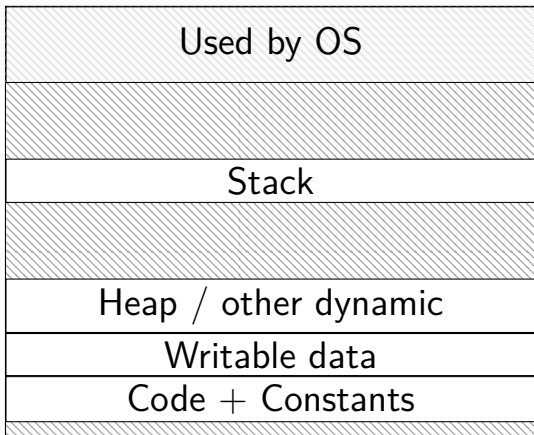
creates a **copy** of an entire program!

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

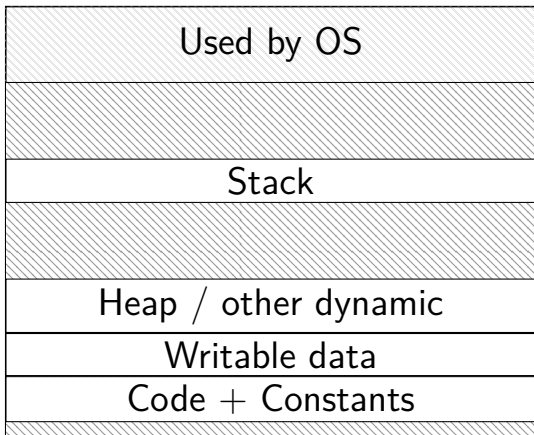
how isn't this really slow?

# do we really need a complete copy?

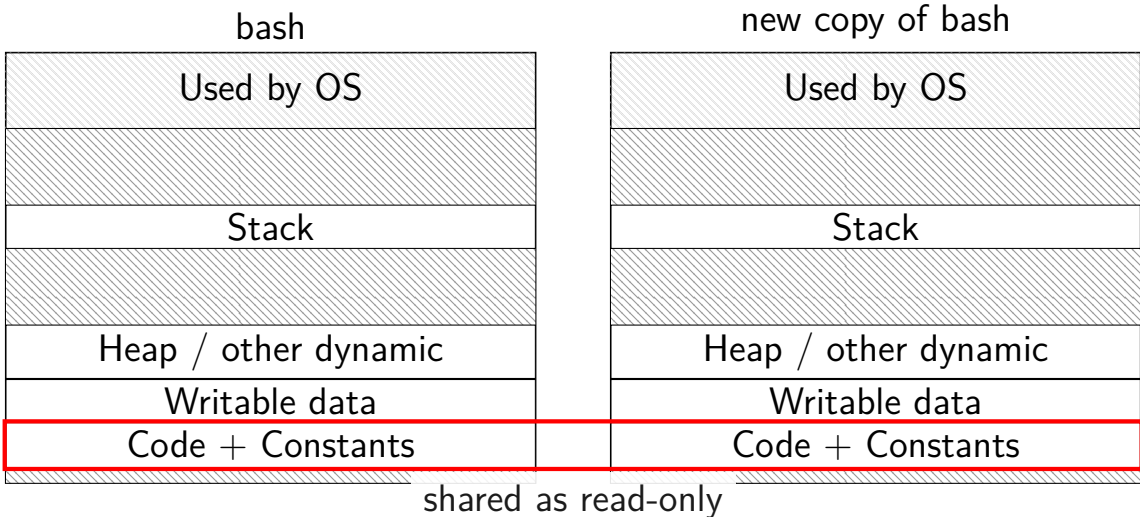
bash



new copy of bash

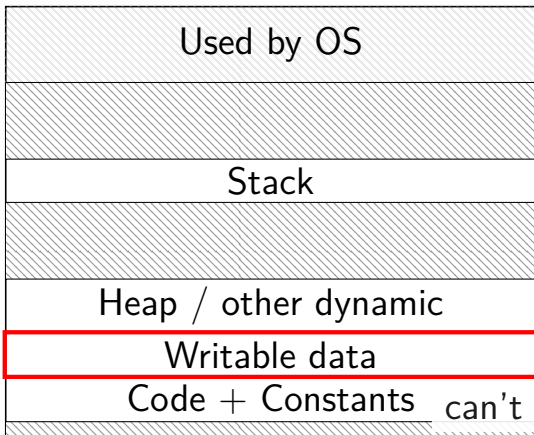


# do we really need a complete copy?

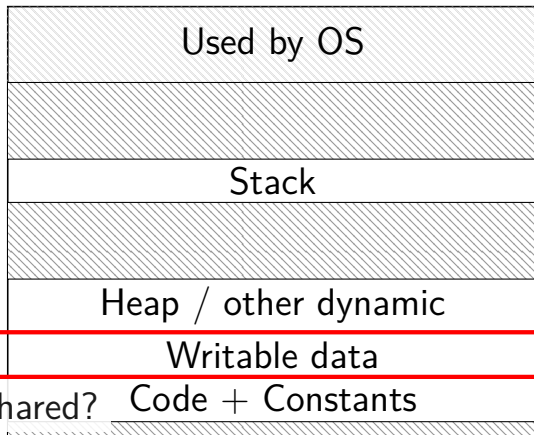


# do we really need a complete copy?

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

# swapping

early motivation for virtual memory: **swapping**

using disk (or SSD, ...) as the next level of the memory hierarchy  
how our textbook and many other sources presents virtual memory

OS allocates **program space on disk**

own mapping of virtual addresses to location on disk

DRAM is a cache for disk

# swapping

early motivation for virtual memory: **swapping**

using disk (or SSD, ...) as the next level of the memory hierarchy  
how our textbook and many other sources presents virtual memory

OS allocates **program space on disk**

own mapping of virtual addresses to location on disk

**DRAM is a cache for disk**

# swapping versus caching

“cache block”  $\approx$  physical page

fully associative

every virtual page can be stored in any physical page

replacement is managed by the OS

normal cache hits happen without OS

common case that needs to be fast

# swapping components

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

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

“swap out” a page

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

# HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

- minimum size: 512 bytes

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

SSD reads and writes: hundreds of microseconds

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

# HDD/SDDs are slow

HDD reads and writes: **milliseconds to tens of milliseconds**

minimum size: 512 bytes

writing tens of kilobytes basically as fast as writing 512 bytes

SSD reads and writes: **hundreds of microseconds**

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

# HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

minimum size: 512 bytes

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

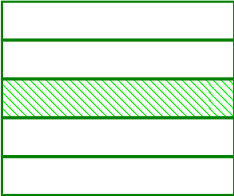
SSD reads and writes: hundreds of microseconds

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

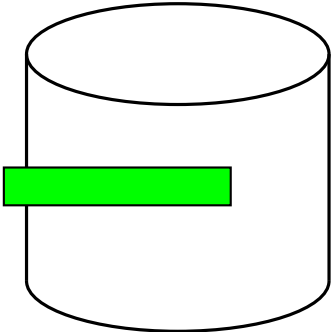


# swapping timeline

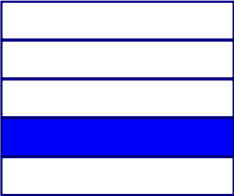
program A pages



...



program B pages

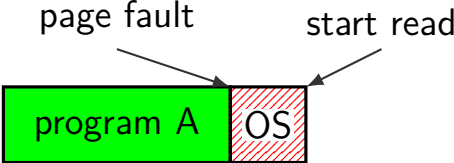
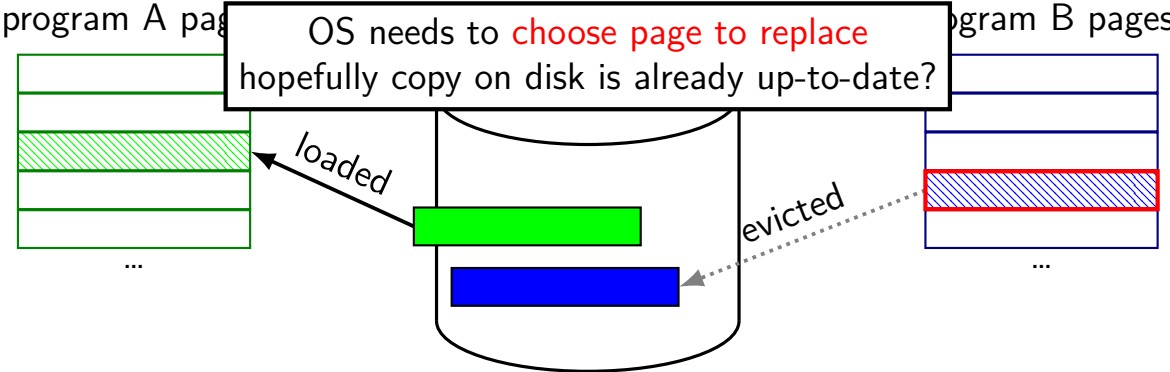


...

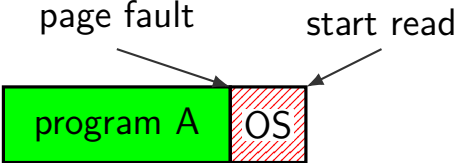
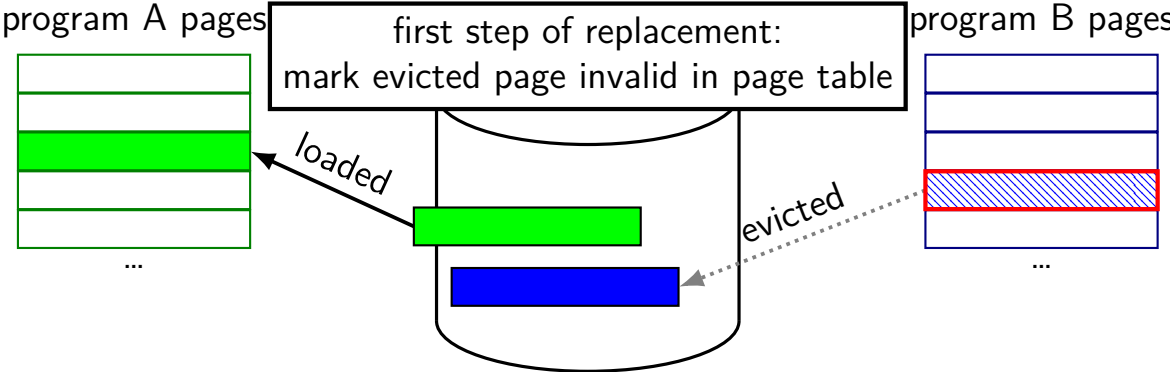
page fault



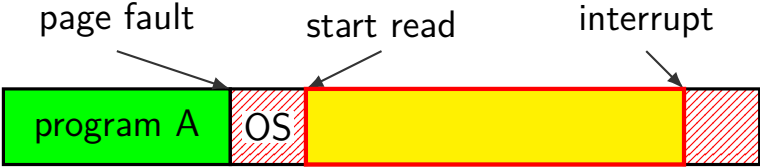
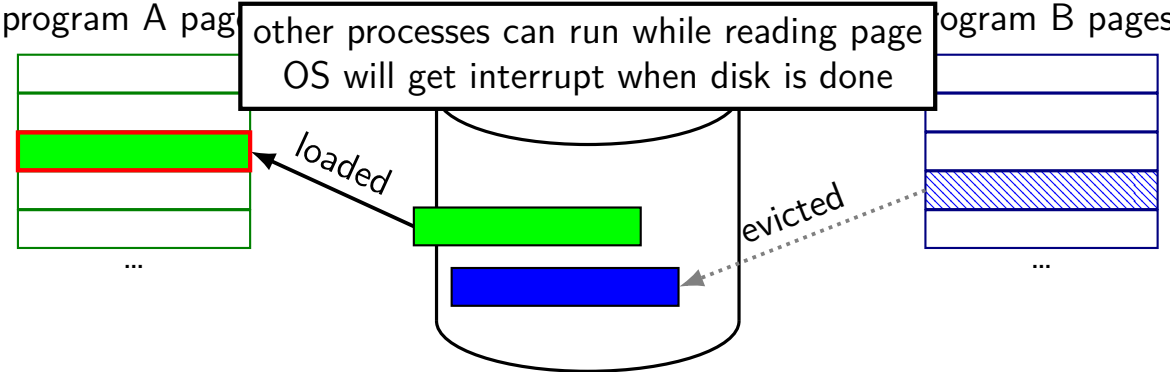
# swapping timeline



# swapping timeline

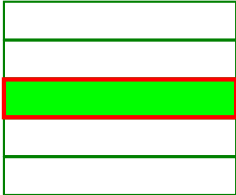


# swapping timeline

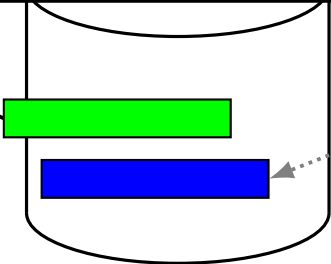


# swapping timeline

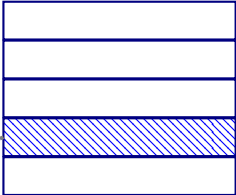
program A pages



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



program B pages



page fault

start read

interrupt



# swapping decisions

write policy

replacement policy

# swapping decisions

write policy

replacement policy

# swapping is writeback

implementing write-through is hard

- when fault happens — physical page not written

- when OS resumes process — no chance to forward write

- HW itself doesn't know how to write to disk

write-through would also be really slow

- HDD/SSD perform best if one writes **at least a whole page** at a time



# implementing writeback

need a *dirty bit* per page (“was page modified”)

often **kept in the page table!**

option 1 (most common): **hardware sets dirty bit** in page table entry (on write)

bit means “physical page was modified using this PTE”

option 2: OS sets page read-only, flips read-only+dirty bit on fault

# swapping decisions

write policy

replacement policy

# replacement policies really matter

huge cost for “miss” on swapping (milliseconds!)

many millions of computations on modern processor  
much much worse than even L3 caches

replacement policy implemented **in software**

a lot more room for fancy policies

# LRU replacement?

problem: need to identify when pages are used

ideally **every single time**

not practical to do this exactly

HW would need to keep a list of when each page was accessed, or

SW would need to force every access to trigger a fault

# approximating LRU

one policy: “not recently used”

OS periodically marks all pages as unreadable/writeable

when page fault happens:

- make page accessible again
- put on list of “used” pages

OS replaces pages not on “used” list

- tiebreaker: when was page last on used list?

# hardware help for not-recently-used

hardware usually implements **accessed** bit in page table entry

whenever page is read/written — hardware sets this bit (if not set)

makes it easier (faster?) for OS to maintain “used ” list

OS can periodically scan/clear “accessed” bit  
instead of marking pages invalid temporarily

construct lists of “used” pages

## accessed/dirty bits

information about how a page table entry was used  
*indirectly* about underlying physical page

kept in **page table entries** themselves

# accessed/dirty bits

information about how a page table entry was used  
*indirectly* about underlying physical page

kept in **page table entries** themselves

multiple page table entries refer to same page  
separate valid/accessed bits for each  
OS needs to consolidate them all



# swapping worst-cases

replacing dirty page?

need to perform write before starting read  
OSes prefer non-dirty pages for this reason

instruction accesses multiple pages?

e.g. machine code and stack!  
better not evict one to load the other

# 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

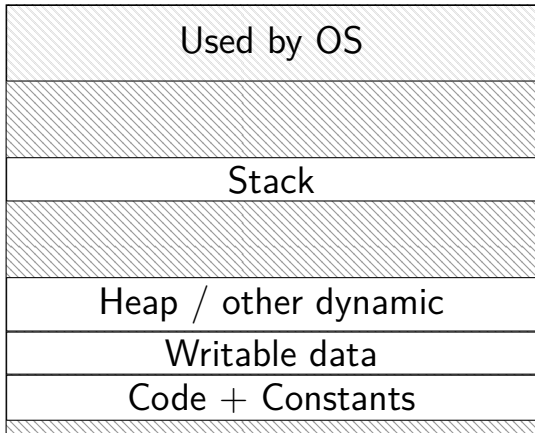
access mapped file for first time, read from disk  
(like swapping when memory was swapped out)

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**

# tracking memory regions

Emacs (run by user mst3k)



OS tracks **regions of memory**

# Linux `vm_area_struct`

start, end address

permissions (read/write/etc.)

underlying file (if any)

checked on every page fault

used to allocate new pages

# position-independent code: one idea

Y86 encoding for jump: include **target address**

alternate encoding: include **target address - PC**  
“relative address”

handles jumps **within standard library**

# real shared libraries

Linux tool `ldd` — what shared libraries does this program use:

```
$ ldd /bin/ls
linux-vdso.so.1 => (0x00007ffebbd49000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so
                (0x00007f4fac17c000)
libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1
             (0x00007f4fabf74000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
           (0x00007f4fabbab000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3
              (0x00007f4fab96d000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2
            (0x00007f4fab769000)
/lib64/ld-linux-x86-64.so.2 (0x00007f4fac39f000)
libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1
              (0x00007f4fab564000)
```

# position-independent code: indirection

what about code between libraries?

what about changing the library code?

use **indirection**:

```
/* instead of: */  
    call printf  
/* use something like: */  
    relative_movq LOOKUP_TABLE + 100, %rax  
    // x86-64 syntax: movq LOOKUP_TABLE+100(%rip), %rax  
    call *%rax
```

populate **lookup table for each function used**

lookup table **not shared between programs**

add version of move instruction that uses relative address