last time

- dividing memory into pages
  - page numbers (0-based index of page) and page offsets (byte in page)
- address space sizes
  - pointer size v. number of bits actually used
  - more physical addresses than installed memory
- page tables entries
  - lookup using virtual page number
  - valid bit — is something there? (if no, exception)
  - physical page number
  - permission bits — what accesses to allow? (if no, exception)
- allocate-on-demand

quiz Q1D

allegation: child processes waitpid for themselves

some problems:

- execv does not return, so waitpid() never reached by children
- waitpid will fail if current process has no children

quiz Q2

I screwed up and made an off-by-one error here and intended answer

…so correct answer was “none of the above”

- either pointing out that LOCATION 1 code should be changed *or* stating need to wait for things LOCATION 1 did not wait for

intended pattern:

- start 0, start 1, start 2, start 3,
  wait for 0, start 4,
  wait for 1, start 5,
  wait for 2, start 6,
  …wait for final four

quiz Q3

one could take much longer than others

could be waiting for that one for a long time

fix: wait for *any* of the process to finish (once 4 started), then start next

- if no other child processes, can use waitpid(-1, ...) to do this

Q6

13-bit addresses

`0 1010 1010 1010` binary = `1010 1010 1010` binary = `0x0aaa` hexadecimal = `0xaaa` hexadecimal = 2730 decimal

most significant three bits of 13-bit integer are 010 (not 000 or 101)

doesn't matter how I write the value

anonymous feedback (1)

"people in this class whine too much in the anonymous feedback and I think there is no problem with telling them to stop whining... and I'll start with myself... I need to stop whining in the anonymous feedback....."

"I think class time in managed well for content, and honestly others' complaints are better voiced offline, such as on Piazza."

anonymous feedback (2)

"Hi Professor, if it's not too much trouble, would you mind nudging the class about not talking super loudly while you're lecturing? I get a lot out of attending lecture for this class, but sometimes it's even difficult to focus or hear you over the people around me..."

anonymous feedback (2)

“I think that the quiz questions with only one right answer being worth 4 points is very steep, especially since there is only one correct one. If we are between two choices and choose the wrong one, our grade drops by 15 points no questions asked. If possible, I would like to suggest that in the comments we can provide a ”backup answer” for half credit that can help justify and explain our thought process. Personally, I just think that it is a very steep penalty. Thank you for your consideration.”

in some cases it’ll make sense to have partial credit for some wrong answers

in some cases we can give partial credit based on reasoning in comments (when it shows understanding of concept the question is meant to test)

but main mitigation in value of quiz questions is meant to be number of quizzes

anonymous feedback (3)

"I learn better by seeing and understanding examples. I'd like to see more of class time dedicated to livecoding, examples, and interactive questions. This helps me visualize concepts in the way that we'll see them applied. I typically learn the best in CS classes that are structured to have a short conceptual lesson in the beginning of class followed by livecoding and examples to help prepare me for labs and homework. So far, it's taken me a lot longer to complete our HW/labs because it is my first time applying the concepts we use."

- somewhat intentional that HW/labs are meant to be practice applying concepts more than assessment/etc.
  - course design that doesn't do this probably needs students to prepare for lecture more
- agree that I should have more interactive questions

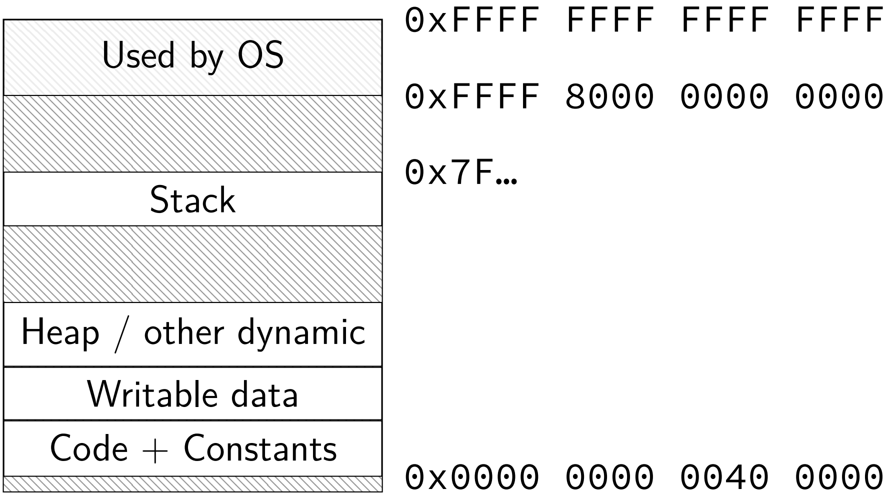
anonymous feedback (4)

“Would you be able to share the median, mean, and standard deviation of the final from this class last semester, as well as the curve that was applied to everyone’s grades at the end?”

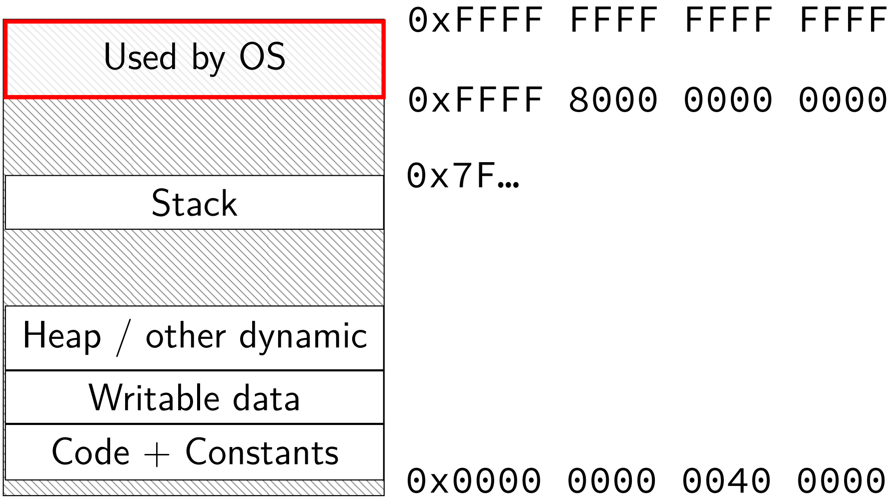
- approx. 25th/median/75th percentile on study materials page
- not a curve (since it’s not based on relative performance of students)

	S 2023	F 2023
A+	96.9	96.5
A	92.7	92.5
A-	89.6	89.7
B+	86.9	86.5
B	83.0	82.5
B-	80.0	79.5
C+	77.0	76.9
C	73.0	73.0
C-	70.0	69.0
D+	67.0	66.0
D	63.0	62.0
D-	60.0	59.0

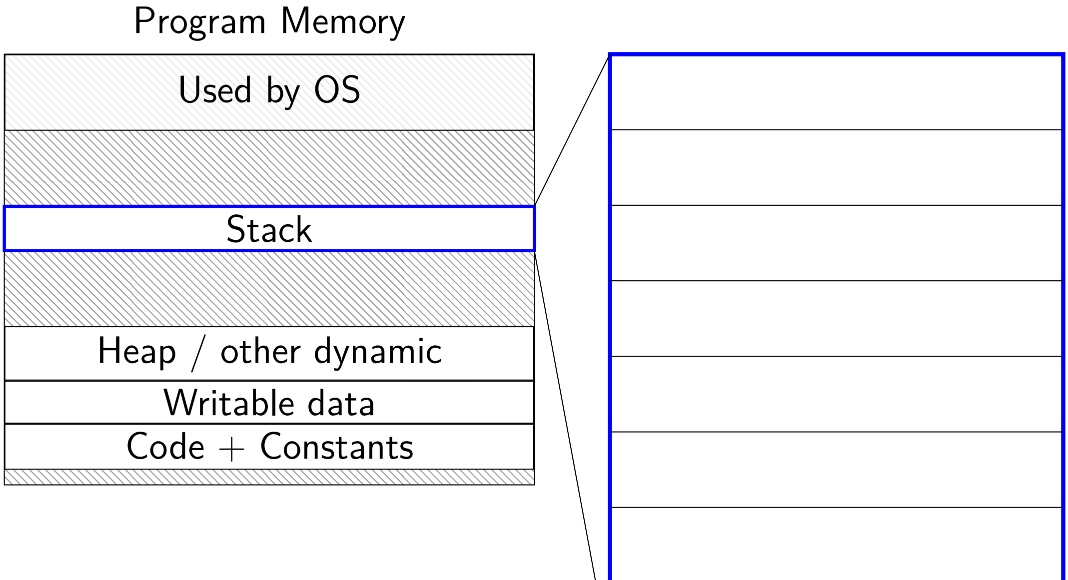
program memory



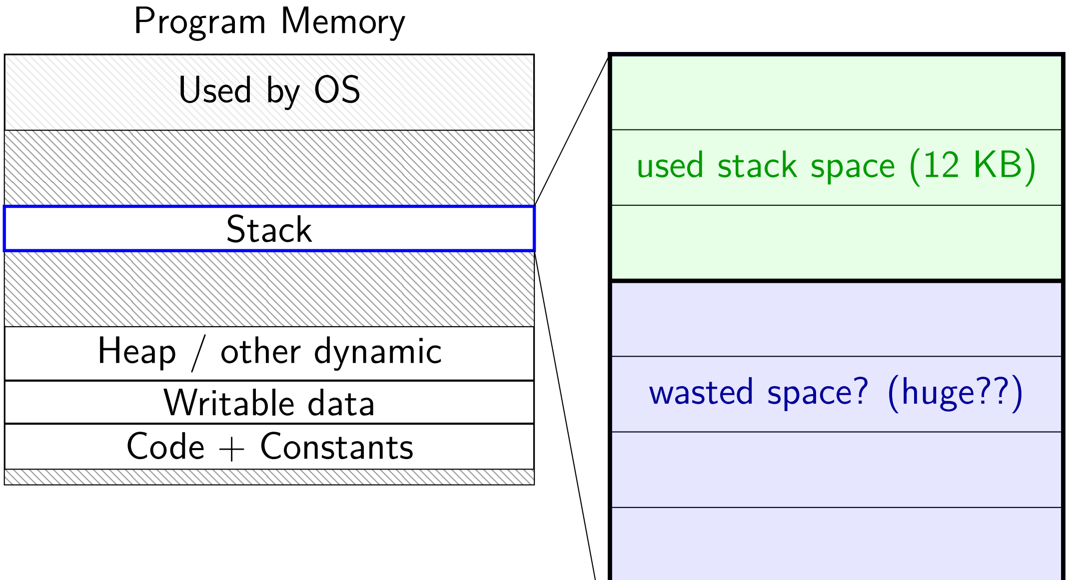
program memory



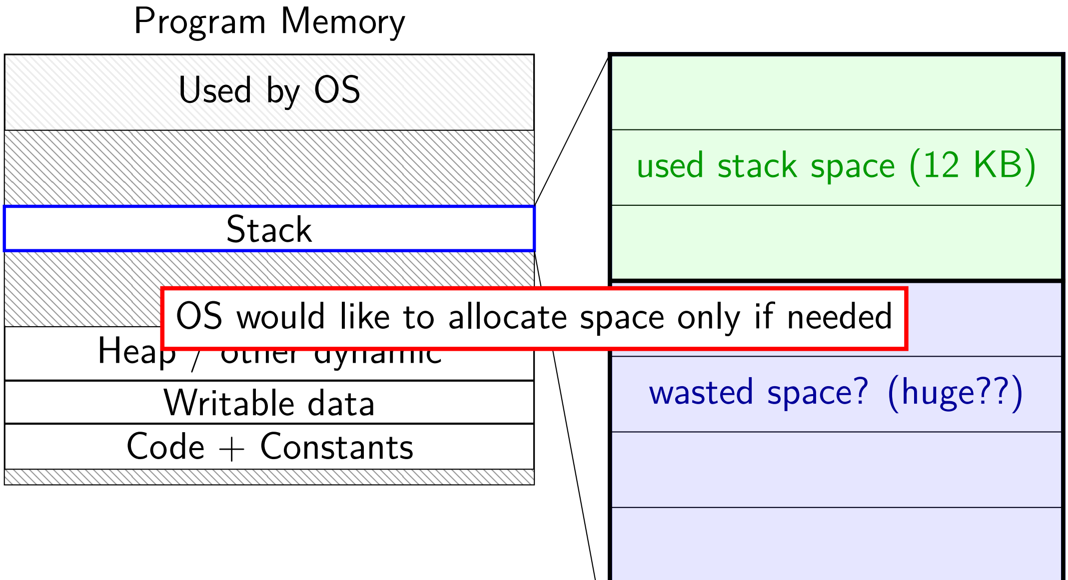
space on demand



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

mmap

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

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

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

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

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

Linux maps: list of maps

```
$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 48328831         /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 48328831         /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 48328831         /bin/cat
01974000-01995000 rw-p 00000000 00:00 0                [heap]
7f60c718b000-7f60c7490000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
7f60c7490000-7f60c764e000 r-xp 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c764e000-7f60c784e000 ---p 001be000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c784e000-7f60c7852000 r--p 001be000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c7852000-7f60c7854000 rw-p 001c2000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c7854000-7f60c7859000 rw-p 00000000 00:00 0
7f60c7859000-7f60c787c000 r-xp 00000000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19.s
7f60c7a39000-7f60c7a3b000 rw-p 00000000 00:00 0
7f60c7a7a000-7f60c7a7b000 rw-p 00000000 00:00 0
7f60c7a7b000-7f60c7a7c000 r--p 00022000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19.s
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19.s
7f60c7a7d000-7f60c7a7e000 rw-p 00000000 00:00 0
7ffc5d2b2000-7ffc5d2d3000 rw-p 00000000 00:00 0        [stack]
7ffc5d3b0000-7ffc5d3b3000 r--p 00000000 00:00 0        [vvar]
7ffc5d3b3000-7ffc5d3b5000 r-xp 00000000 00:00 0        [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Linux maps: list of maps

```
$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 48328831          /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 48328831          /bin/cat
0060b000-(
01974000-(
7f60c718b0                                              cale-archive
7f60c74900                                              gnu/libc-2.1
7f60c764e0                                              gnu/libc-2.1
7f60c784e0                                              gnu/libc-2.1
7f60c78520                                              gnu/libc-2.1
7f60c78540
7f60c78590                                              gnu/ld-2.19.s
7f60c7a390
7f60c7a7a0
7f60c7a7b0                                              gnu/ld-2.19.s
7f60c7a7c0                                              gnu/ld-2.19.s
7f60c7a7d0
7ffc5d2b20
7ffc5d3b00
7ffc5d3b30
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

OS tracks list of `struct vm_area_struct` with:

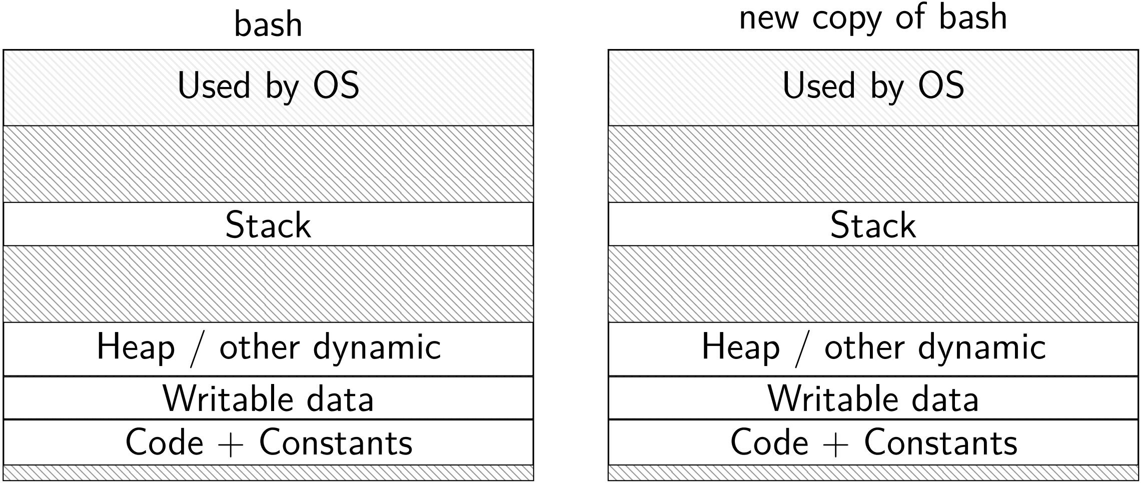
(shown in this output):

- virtual address start, end
- permissions
- offset in backing file (if any)
- pointer to backing file (if any)

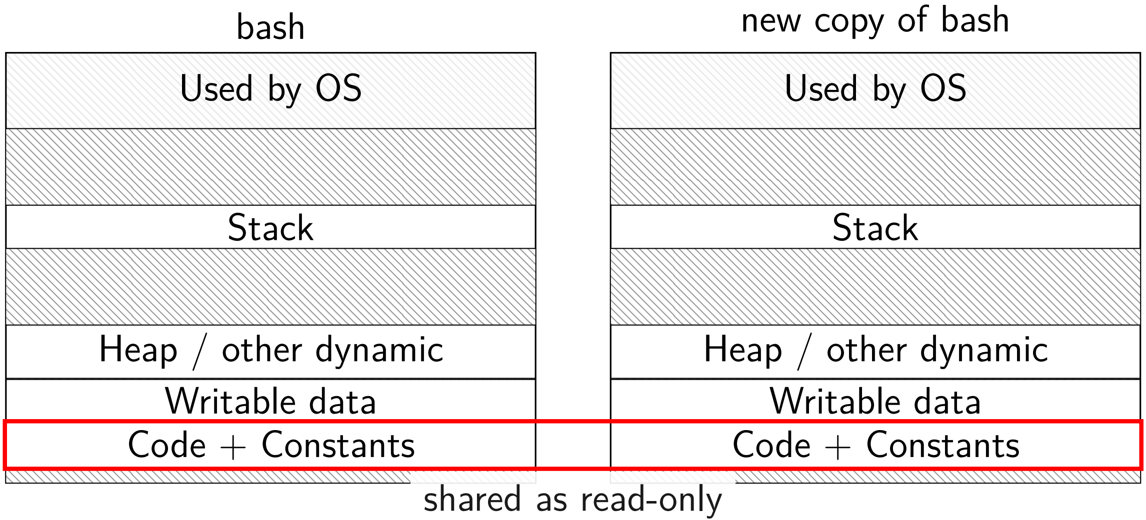
(not shown):

- info about sharing of non-file data ...

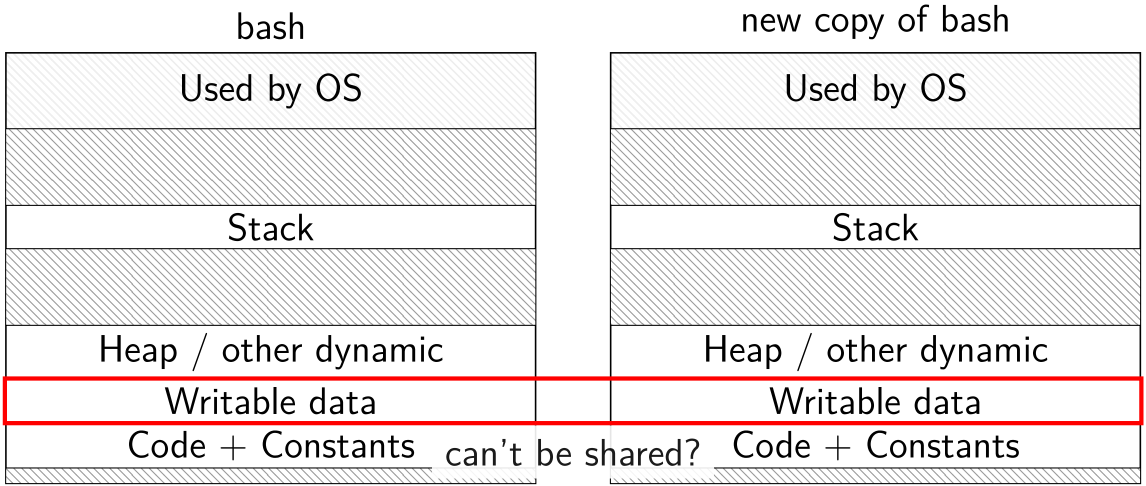
do we really need a complete copy?



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 it

- example: default value of global variables
- might typically not change
- (or OS might have preloaded executable's data anyways)

can we detect modifications?

trick for extra sharing

sharing writeable data is fine — until either process modifies it

- example: default value of global variables
- might typically not change
- (or OS might have preloaded executable's data anyways)

can we detect modifications?

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

processor will trigger a fault when it's written

copy-on-write and page tables

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

copy-on-write and page tables

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

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

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

copy-on-write and page tables

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

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

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

copy-on-write and page tables

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

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

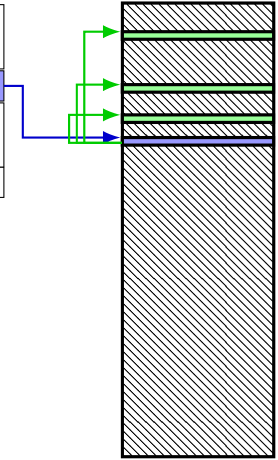
after allocating a copy, OS reruns the write instruction

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

parent process info

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

memory



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

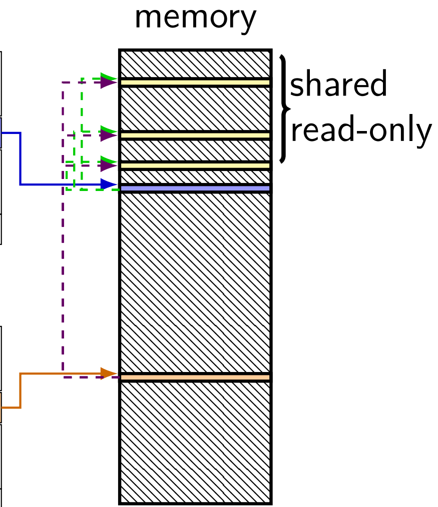
parent process info

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

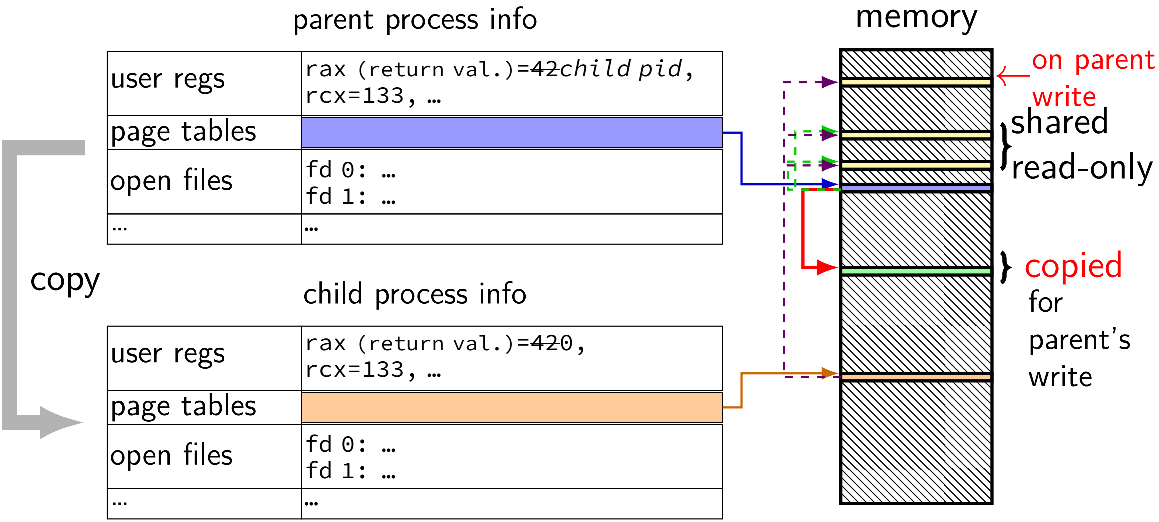
copy

child process info

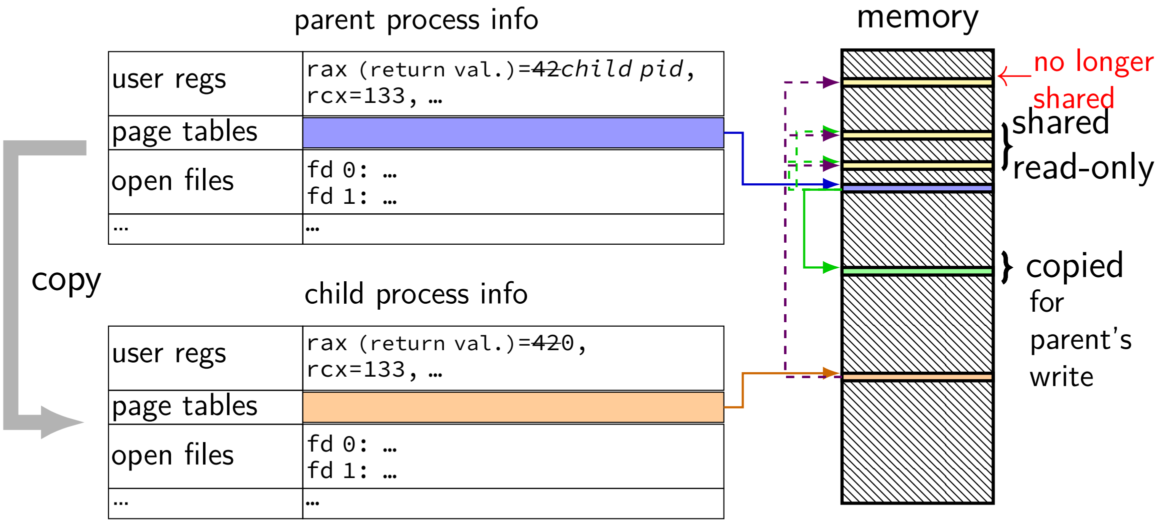
user regs	`rax (return val.)=`~~`42`~~`0,` `rcx=133, …`
page tables	
open files	`fd 0: …` `fd 1: …`
…	…



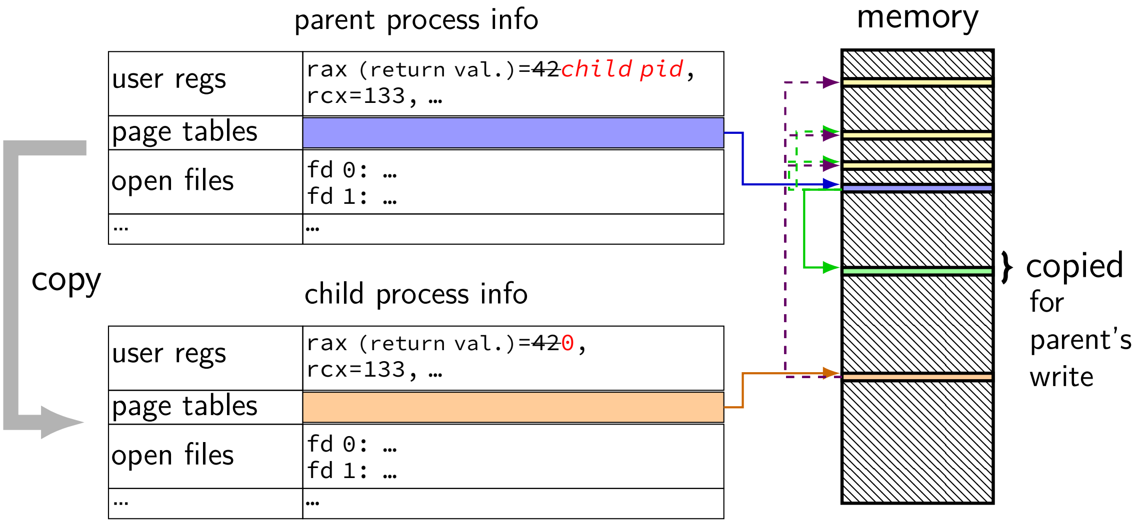
fork (w/ copy-on-write, if parent writes first)



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



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



page tricks generally

deliberately make program trigger page/protection fault

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

have seperate data structures represent logically allocated memory

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

page table is for the hardware and not the OS

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

    "swapping"

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

“swapping”

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

"swapping"

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

"swapping"

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

"swapping"

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

hardware help for page table tricks

information about the address causing the fault

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

(by default) rerun faulting instruction when returning from exception

precise exceptions: no side effects from faulting instruction or after

- e.g. `pushq` that caused did not change `%rsp` before fault
- e.g. can't notice if instructions were executed in parallel

exercise setup

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

page table

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

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

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

exercise setup

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

page table

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

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

phys. page 0

phys. page 1

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

exercise

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

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

page table

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

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

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

exercise

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

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

page table

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

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

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

exercise

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

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

page table

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

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

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

exercise

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

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

page table

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

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

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

exercise

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

(virtual addresses) `0x18` = `00`; `0x03` = 0x4A; `0x0A` = 0xDC; `0x13` = fault

page table

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

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

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

lab tomororw

motivation: page tables are big

real systemns: huge number of virtual pages

not enough space to store page table in the processor core

trick one: store in memory
- processor core just has pointer to place in memory

trick two: avoid storing most invalid entries
- tree-like data structure
- omit nodes of tree that would only have invalid leafs

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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

page table
base register

`0x00010000`

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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

page table base register

0x00010000

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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

page table base register

0x00010000

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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

page table base register

0x00010000

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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

page table
base register

0x00010000

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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

page table base register

`0x00010000`

page table (logically)

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

physical memory

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

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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

page table
base register

0x00010000

page table (logically)

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

physical memory

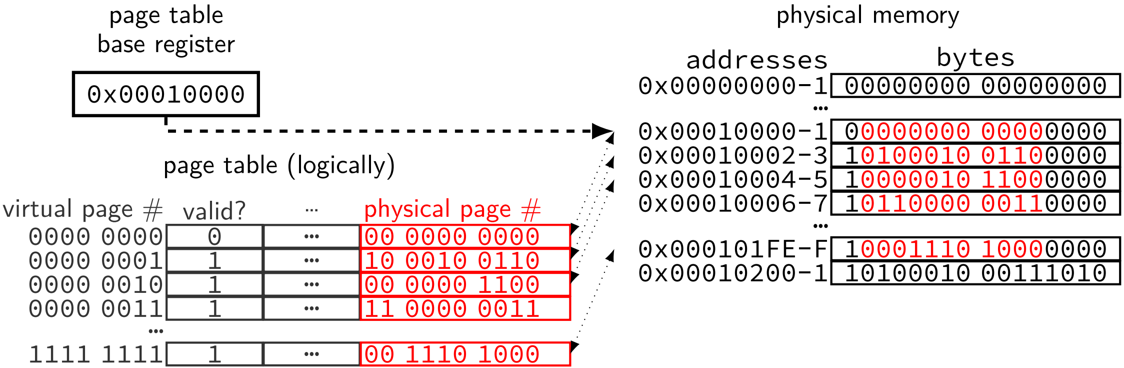
addresses	bytes
0x00000000-1	00000000 00000000
…	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
…	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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



page tables in memory

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

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

page table base register

0x00010000

page table (logically)

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

physical memory

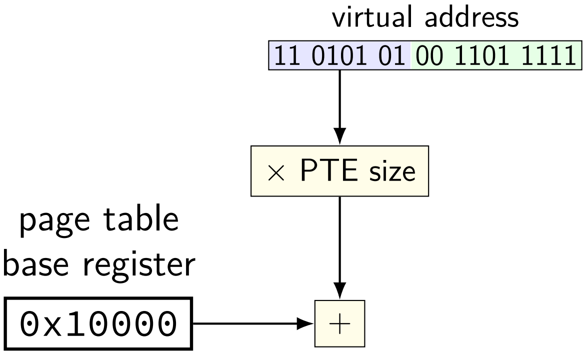
addresses	bytes
0x00000000-1	00000000 00000000
...	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
...	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

memory access with page table

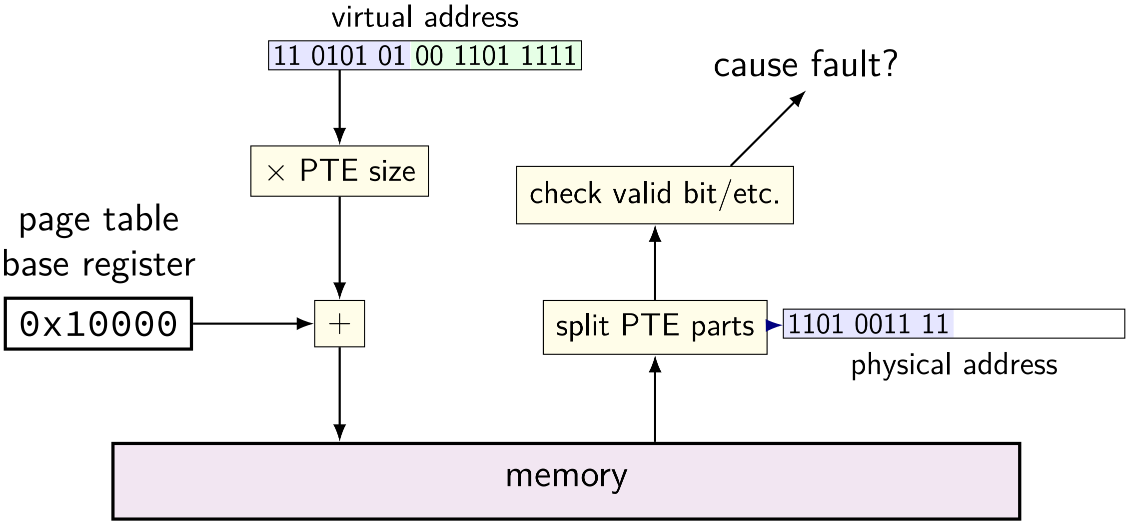
virtual address

11 0101 01	00 1101 1111

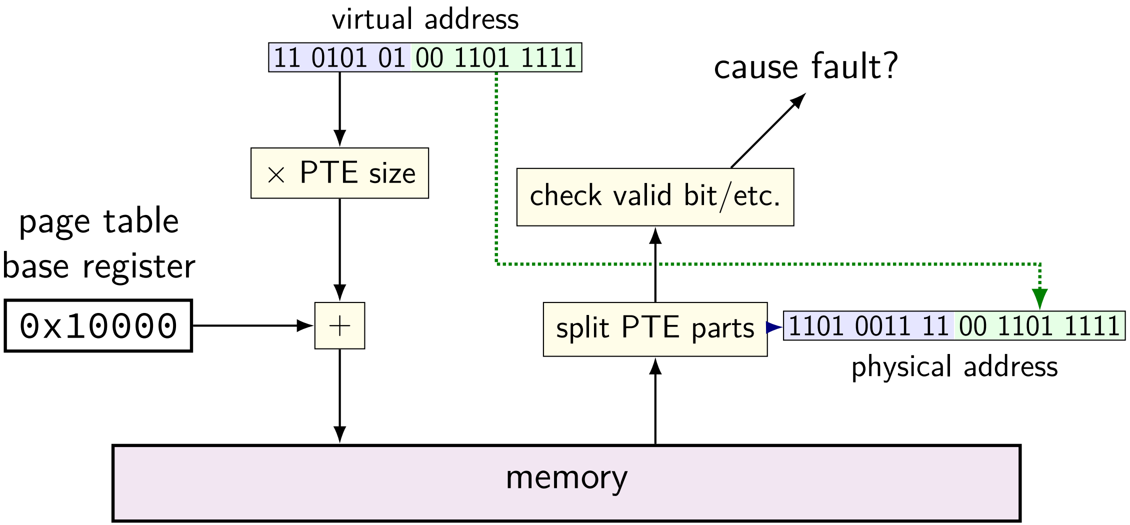
memory access with page table



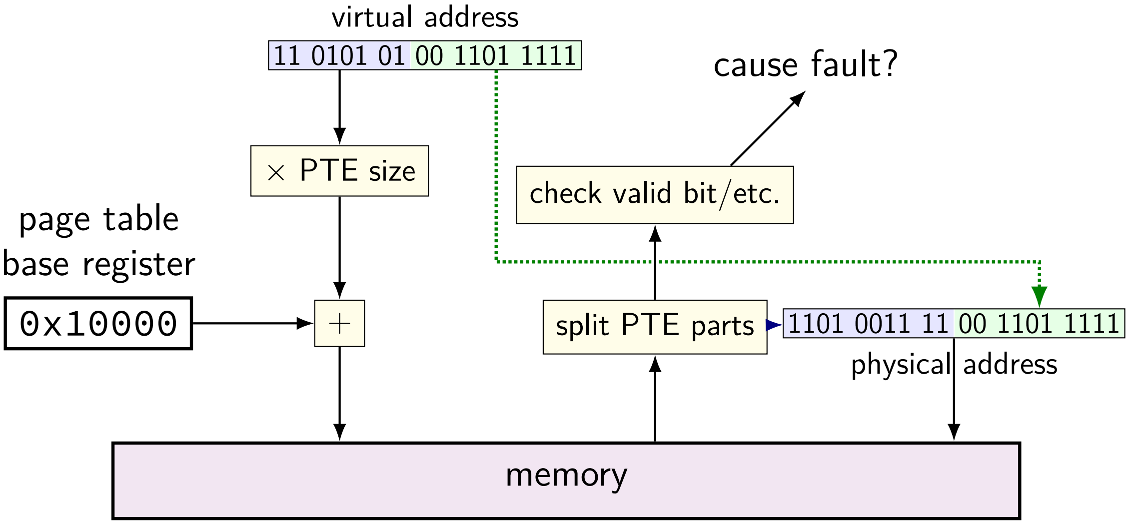
memory access with page table



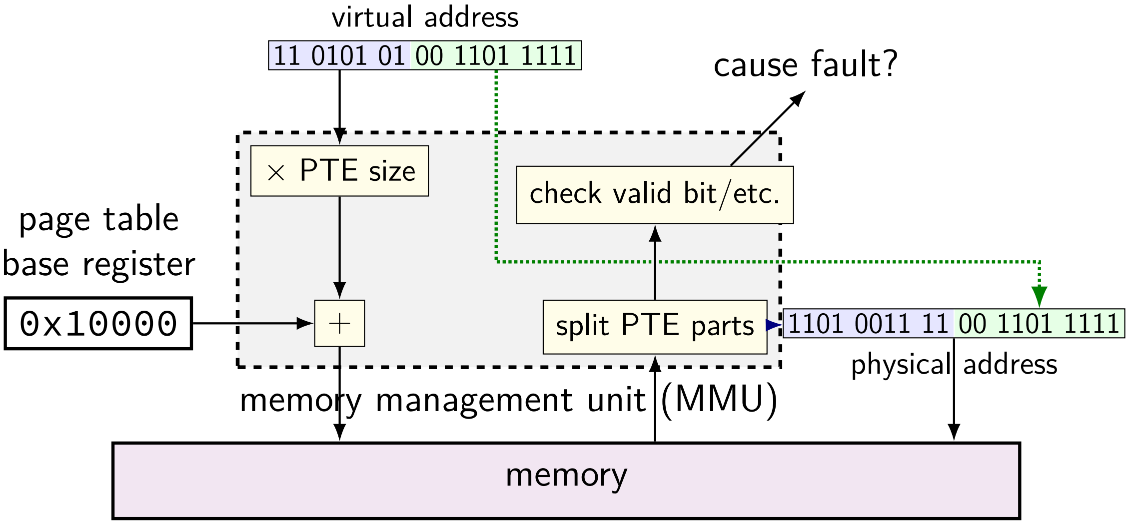
memory access with page table



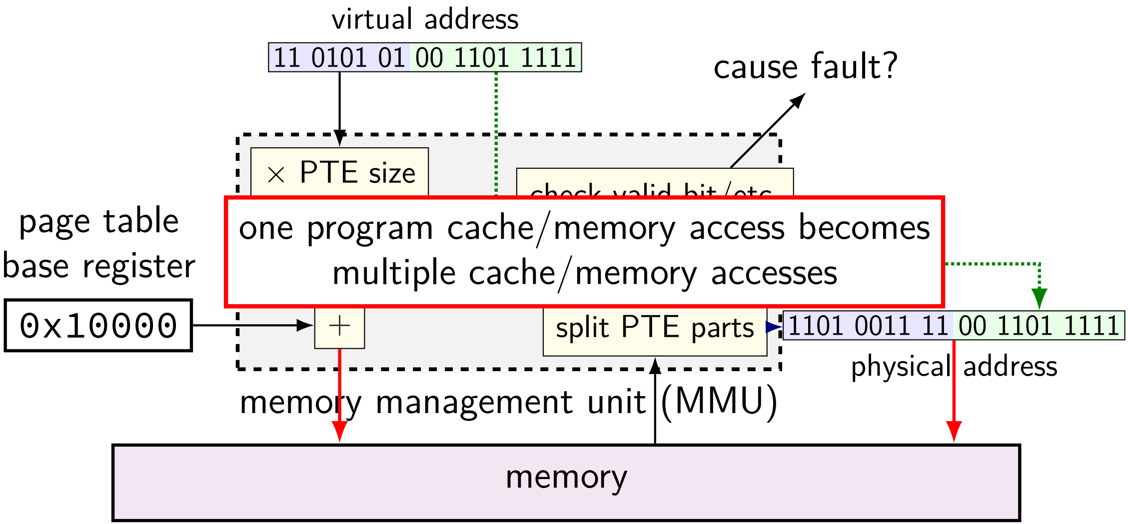
memory access with page table



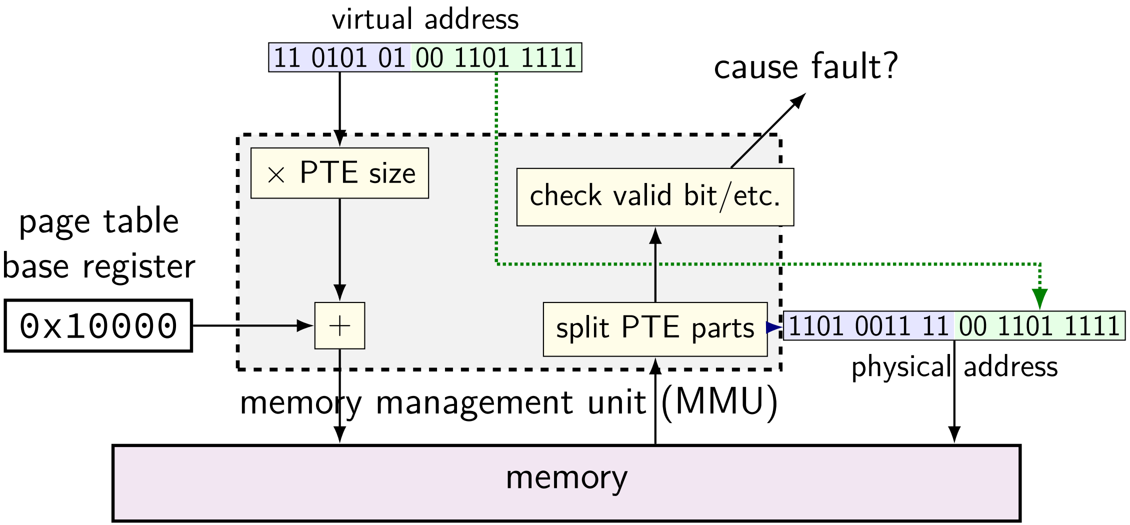
memory access with page table



memory access with page table



memory access with page table



1-level exercise (1)

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

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

page table base register `0x20`; translate virtual address `0x31`

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

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

1-level exercise (1)

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

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

page table base register `0x20`; translate virtual address `0x31`

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

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

`0x31` = `11 0001`

PTE addr:

`0x20` + `110` ×1 = `0x26`

PTE value:

`0xF6` = `1111 0110`

PPN `111`, valid `1`

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

→ `0x0C`

1-level exercise (1)

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

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

page table base register `0x20`; translate virtual address `0x31`

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

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

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

PTE addr:

$\texttt{0x20} + \texttt{110} \times 1 = \texttt{0x26}$

PTE value:

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

PPN `111`, valid 1

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

$\rightarrow$ `0x0C`

1-level exercise (1)

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

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

page table base register `0x20`; translate virtual address `0x31`

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

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

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

PTE addr:

$\texttt{0x20} + \texttt{110} \times 1 = \texttt{0x26}$

PTE value:

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

PPN `111`, valid 1

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

$\rightarrow$ `0x0C`

1-level exercise (1)

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

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

page table base register `0x20`; translate virtual address `0x31`

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

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

`0x31` = `11 0001`

PTE addr:

`0x20` + `110` ×1 = `0x26`

PTE value:

`0xF6` = `1111 0110`

PPN `111`, valid 1

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

→ `0x0C`

1-level exercise (2)

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

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

page table base register `0x20`; translate virtual address `0x12`

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

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

1-level exercise (2)

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

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

page table base register `0x20`; translate virtual address `0x12`

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

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

`0x12` = `01 0010`

PTE addr:

`0x20` + 2 × 1 = `0x22`

PTE value:

`0xD1` = `1101 0001`

PPN `110`, valid `1`

M[`110 001`] = **M[**`0x32`**]**

→ `0xBA`

1-level exercise (2)

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

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

page table base register `0x20`; translate virtual address `0x12`

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

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

`0x12` = `01 0010`

PTE addr:

`0x20` + 2 ×1 = `0x22`

PTE value:

`0xD1` = `1101 0001`

PPN `110`, valid 1

M[`110 001`] = **M[**`0x32`**]**

→ `0xBA`

1-level exercise (2)

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

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

page table base register `0x20`; translate virtual address `0x12`

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

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

`0x12` = `01 0010`

PTE addr:

`0x20` + 2 ×1 = `0x22`

PTE value:

`0xD1` = `1101 0001`

PPN `110`, valid `1`

M[`110 001`] = **M[**`0x32`**]**

→ `0xBA`

1-level exercise (2)

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

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

page table base register `0x20`; translate virtual address `0x12`

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

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

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

PTE addr:

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

PTE value:

$\texttt{0xD1} = \texttt{1101 0001}$

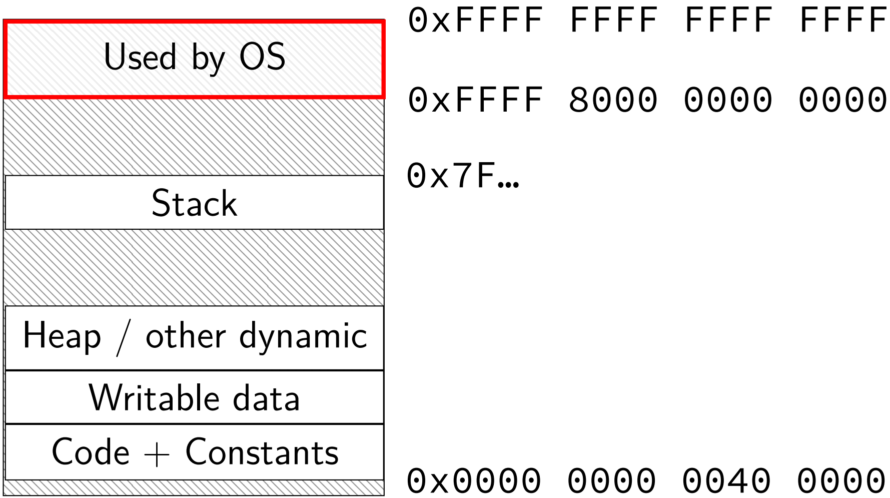
PPN `110`, valid `1`

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

$\rightarrow$ `0xBA`

backup slides

program memory



running OS code

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

running OS code

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

where in memory is this OS code?

running OS code

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

where in memory is this OS code?

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

running OS code

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

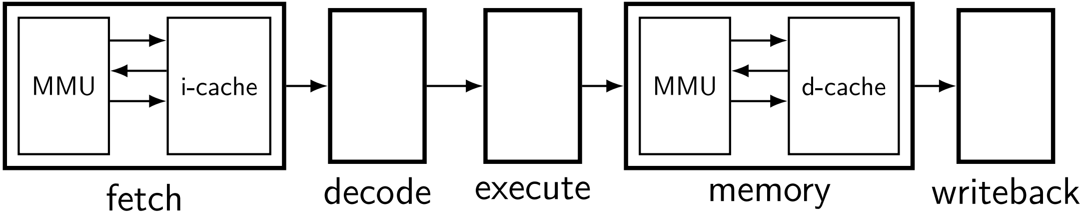
where in memory is this OS code?

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

code better not be modified by user program

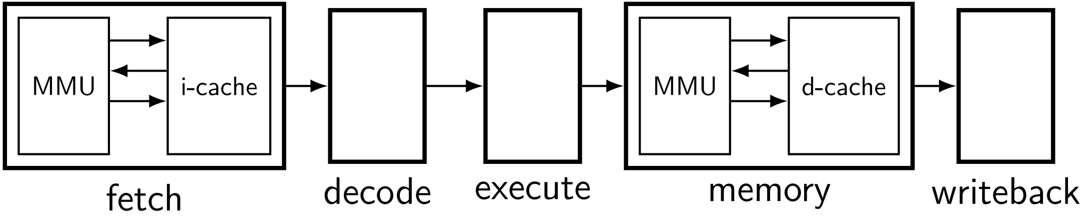
- otherwise: uncontrolled way to "escape" user mode

MMUs in the pipeline



up to four memory accesses per instruction

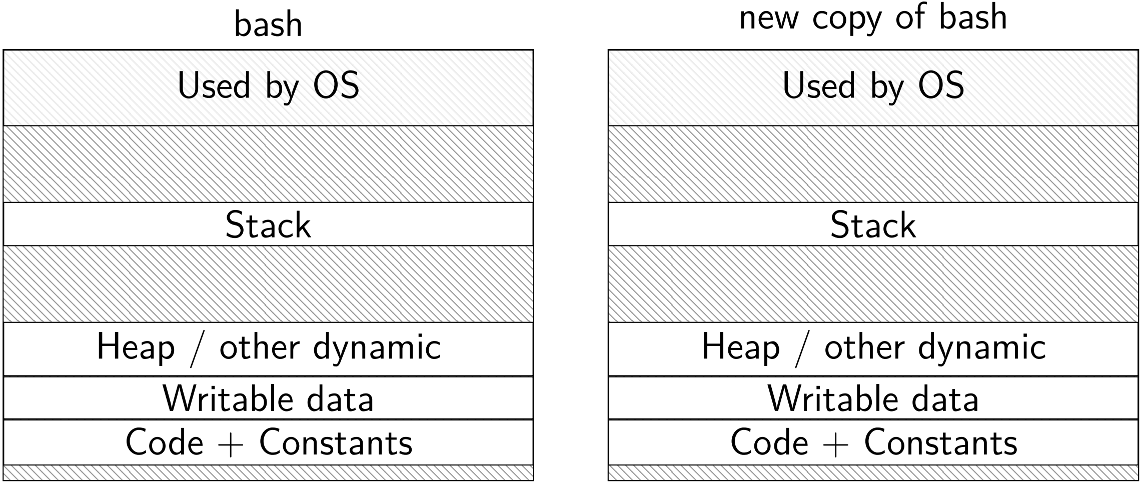
MMUs in the pipeline



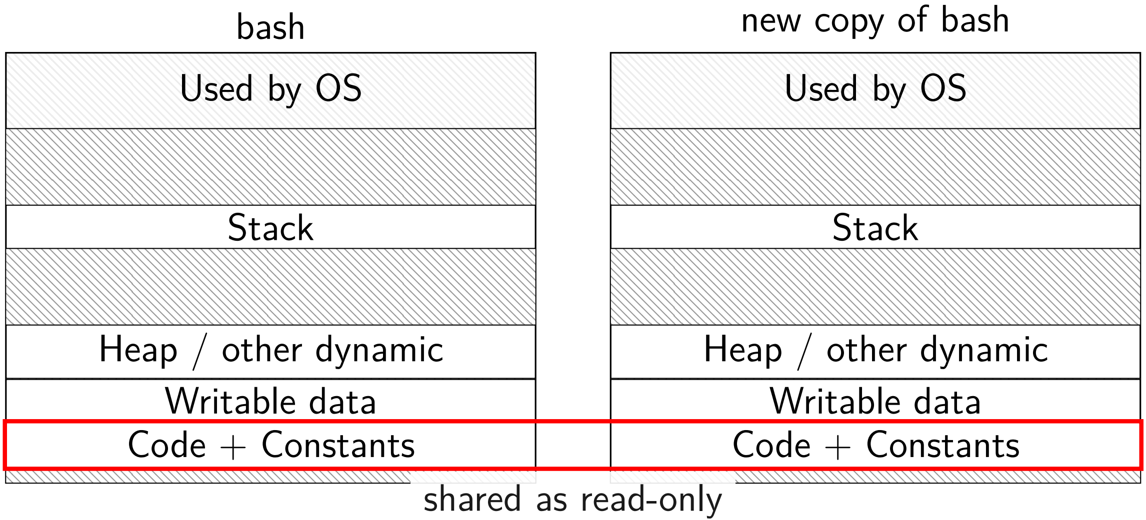
up to four memory accesses per instruction

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

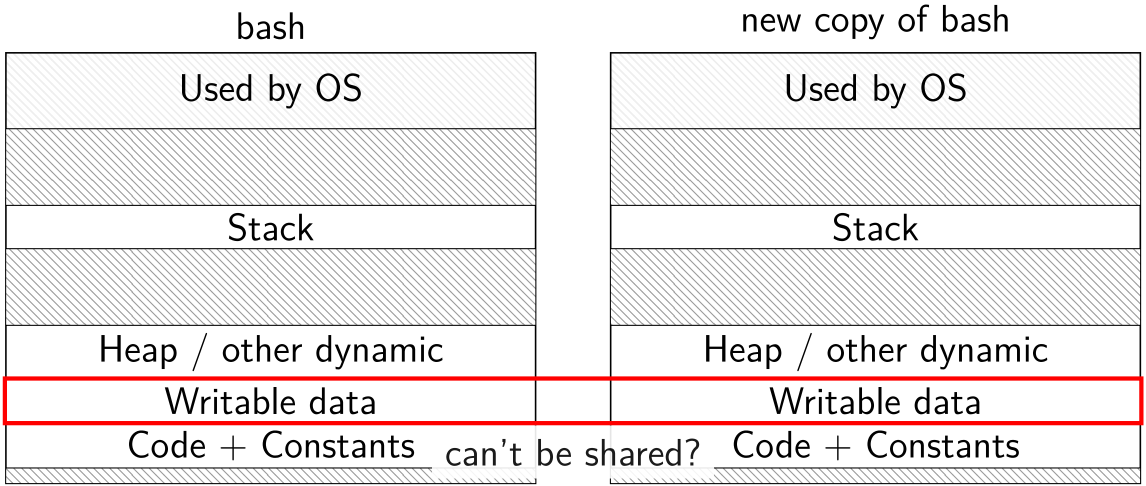
do we really need a complete copy?



do we really need a complete copy?



do we really need a complete copy?



trick for extra sharing

sharing writeable data is fine — until either process modifies it

- example: default value of global variables
- might typically not change
- (or OS might have preloaded executable's data anyways)

can we detect modifications?

trick for extra sharing

sharing writeable data is fine — until either process modifies it

- example: default value of global variables
- might typically not change
- (or OS might have preloaded executable's data anyways)

can we detect modifications?

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

processor will trigger a fault when it's written

copy-on-write and page tables

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

copy-on-write and page tables

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

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

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

copy-on-write and page tables

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

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

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

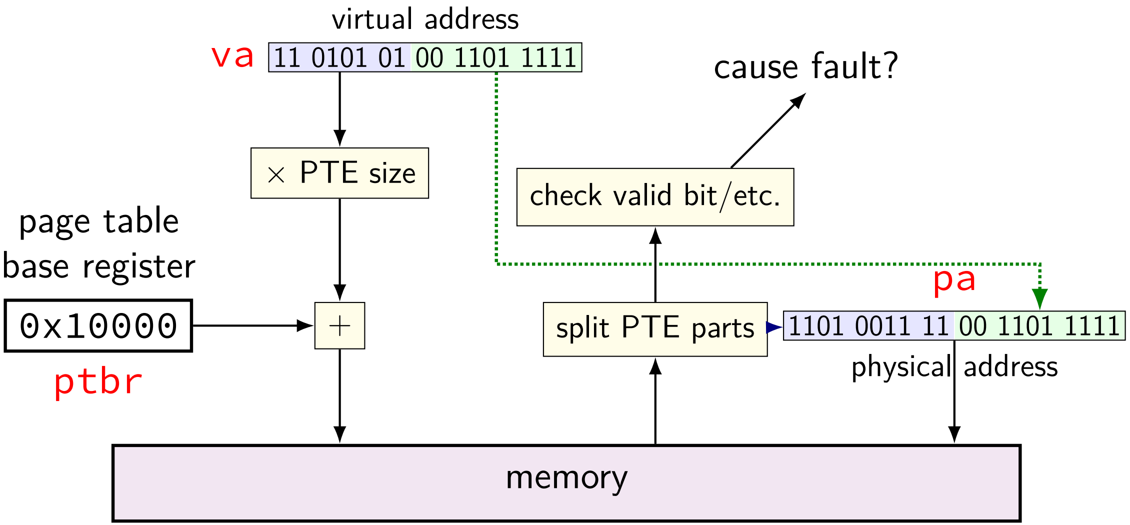
copy-on-write and page tables

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

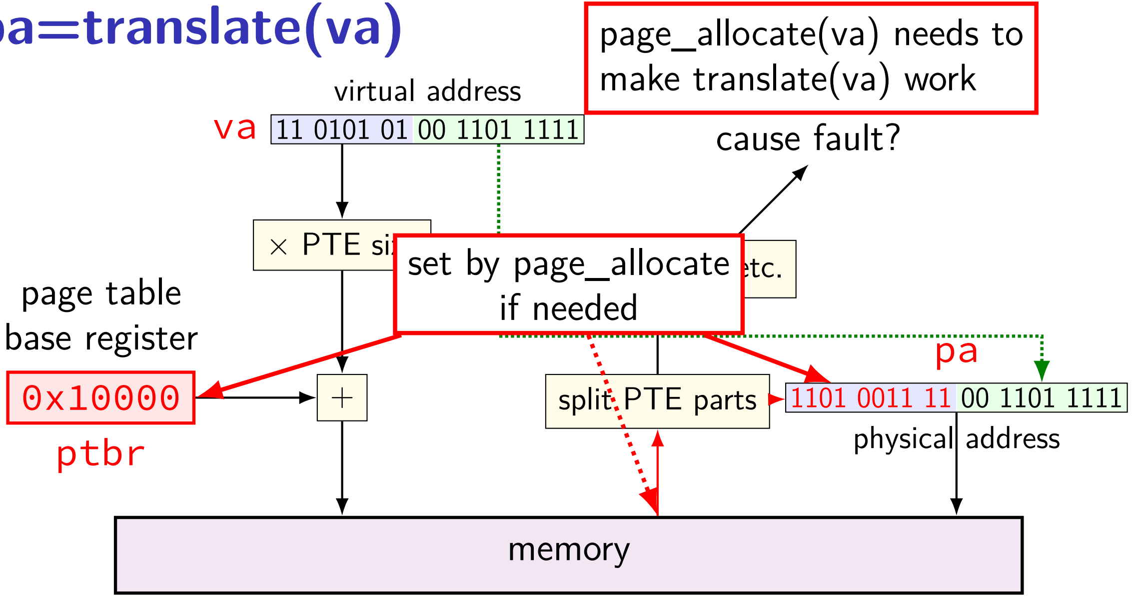
VPN	valid?	write?	physical page
…	…	…	…
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	1	0x300FD
…	…	…	…

after allocating a copy, OS reruns the write instruction

pa=translate(va)



pa=translate(va)



swapping

early motivation for virtual memory: swapping

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

- how our textbook and many other sources presents virtual memory

OS allocates program space on disk

- own mapping of virtual addresses to location on disk

DRAM is a cache for disk

swapping

early motivation for virtual memory: swapping

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

- how our textbook and many other sources presents virtual memory

OS allocates program space on disk

- own mapping of virtual addresses to location on disk

DRAM is a cache for disk

swapping components

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

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

“swap out” a page

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

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

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

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

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

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

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

HDD/SDDs are slow

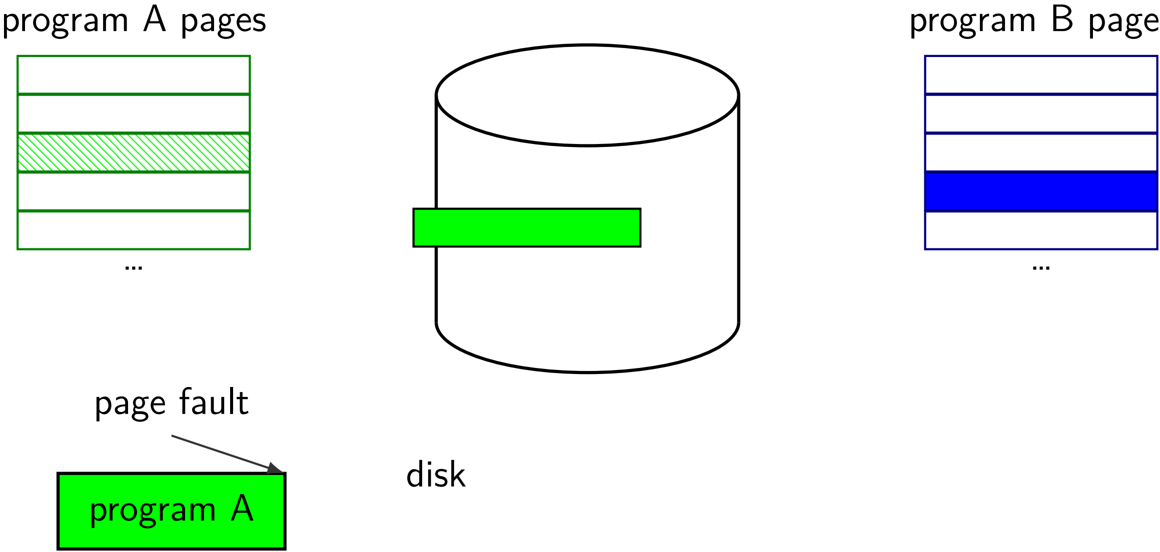
HDD reads and writes: milliseconds to tens of milliseconds

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

SSD writes and writes: hundreds of microseconds

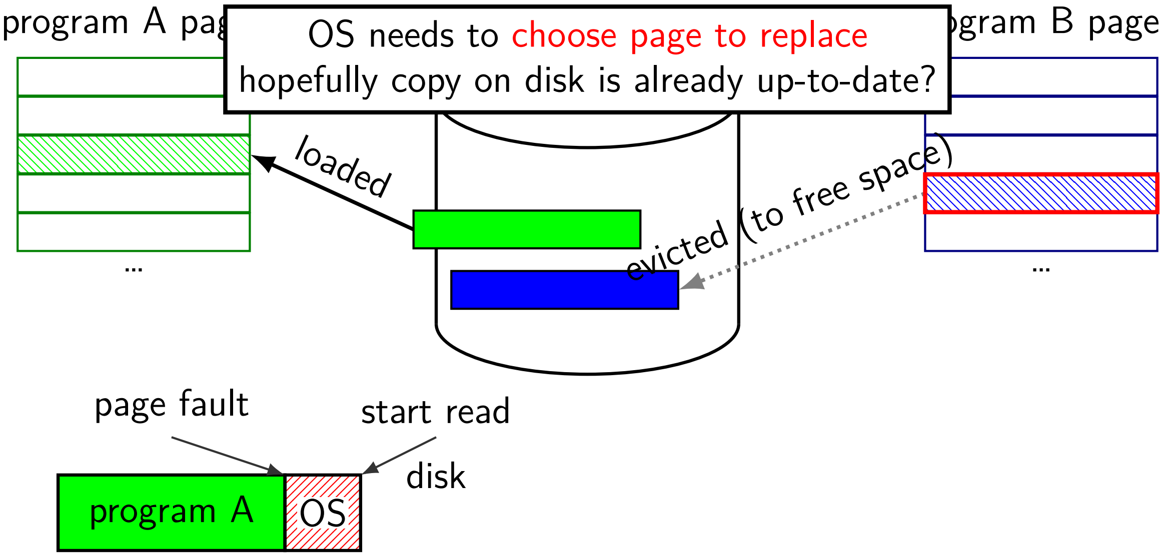
- designed for writes/reads of kilobytes (not much smaller)

swapping timeline

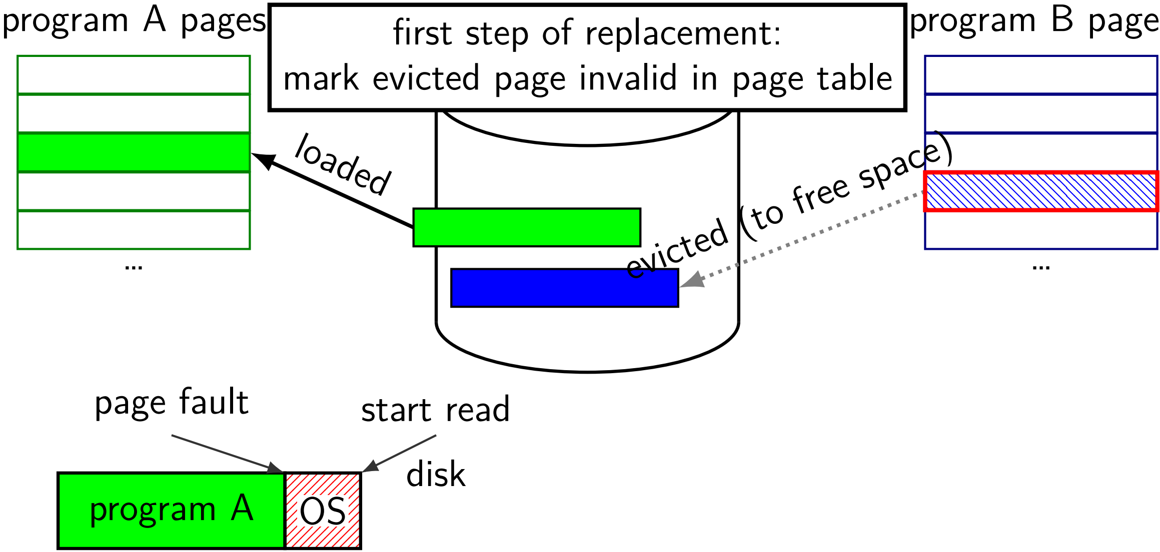


swapping timeline

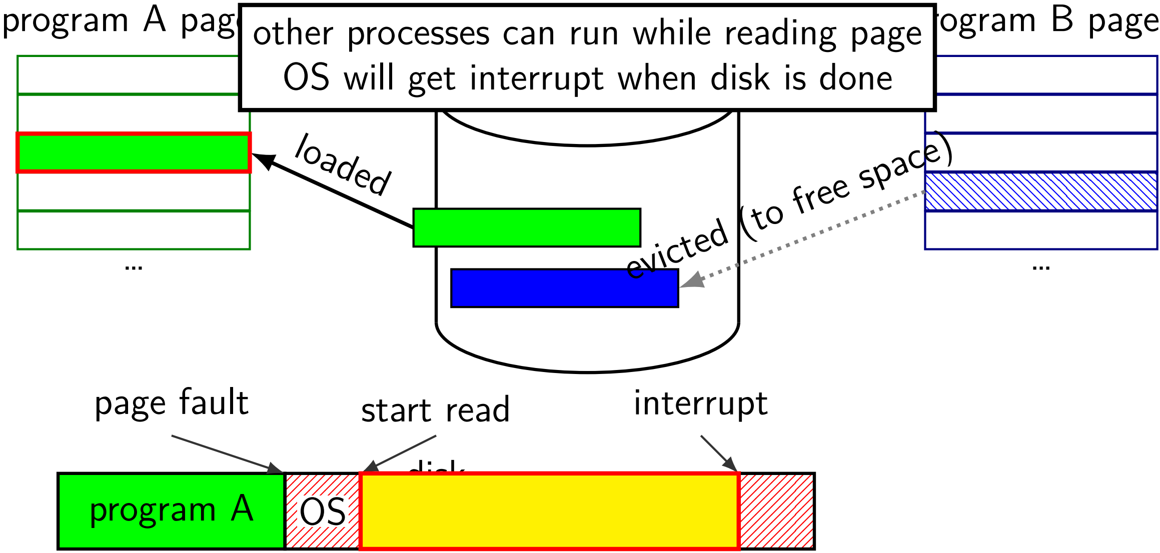
OS needs to choose page to replace
hopefully copy on disk is already up-to-date?



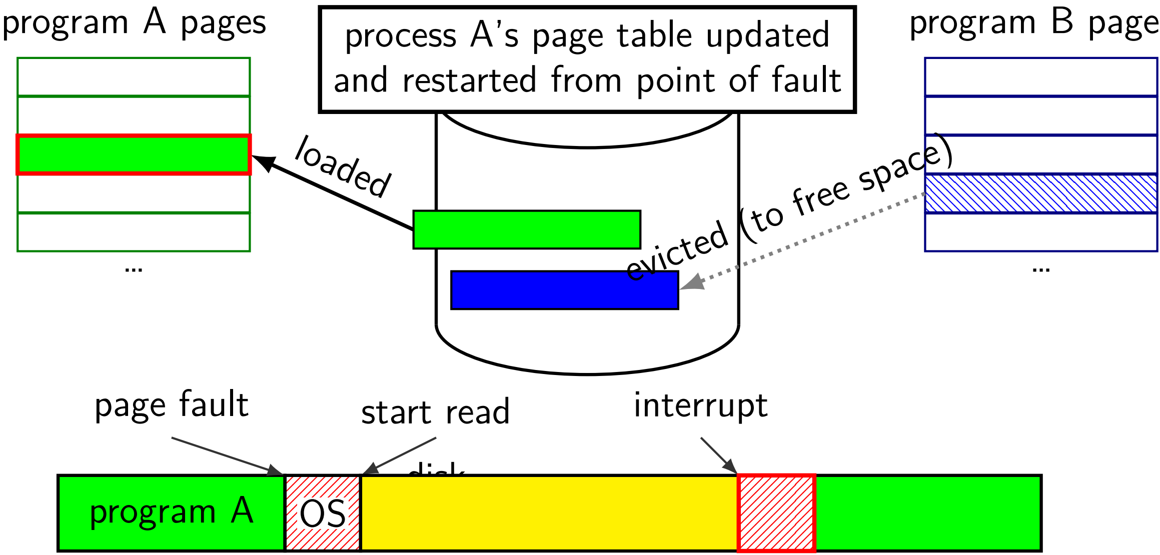
swapping timeline



swapping timeline



swapping timeline



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