Virtual Memory 2

Changelog

Changes made in this version not seen in first lecture:

- 23 October: mapped pages (no backing file): fix end of animation to have page on disk
- 23 October: separate out discussion of readahead from other reasons why hit rate not performance

exam notes

exam graded

some things I regret on semaphore, pipe question

- semaphore queue — different shared buffer than text specified
- piping — should have used other than 1 to init, no newline in printf

regrades requests available...

last time (1)

page tables

- mapping from program ('visible') to physical ('real') addresses
- memory divided into fixed-sized chunks called *pages*
- table: for each program page, what's its physical page?

page table entries: physical page and permission bits

- accessible in user-mode or not? writeable or not?

two-level page tables

- too many virtual pages to store entire list
- table of tables
- first bits indicate entry in first table (= loc of second)
- second bits in second table; last bits in actual physical page

last time (2)

xv6 kernel memory layout

- kernel-only space (PTE's marked 'fault if in user mode') at top
- mapping for kernel space: physical page 0, 1, 2, ...,
- same in every process

xv6 page manipulation utility functions

- finding location of last-level PTE — array lookups for each level of page table

allocating new pages

- actually allocate physical page
- if needed, create second-level page table
- set page table entry to point to physical page

exec'ing processes: allocate pages, copy from executable

last time (3)

page table says "not valid"? page fault

- OS get to run
- return from fault handler? reruns instruction

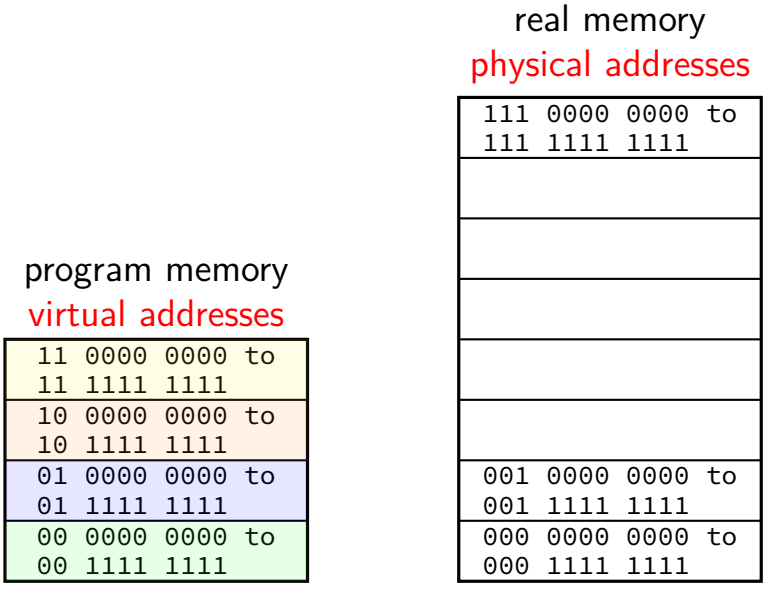
allocate on demand

- on page fault, allocate new page
- set page table entry, return
- program runs as if it was allocated

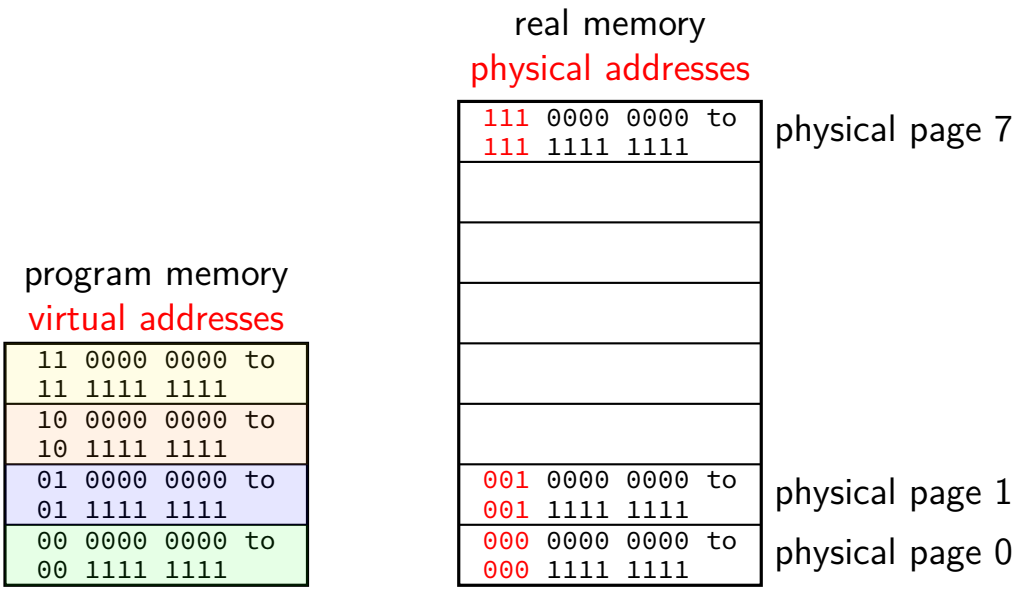
copy-on-write

- on fault for read-only page, make a copy
- set page table entry, return
- program runs as if it had copy all the time

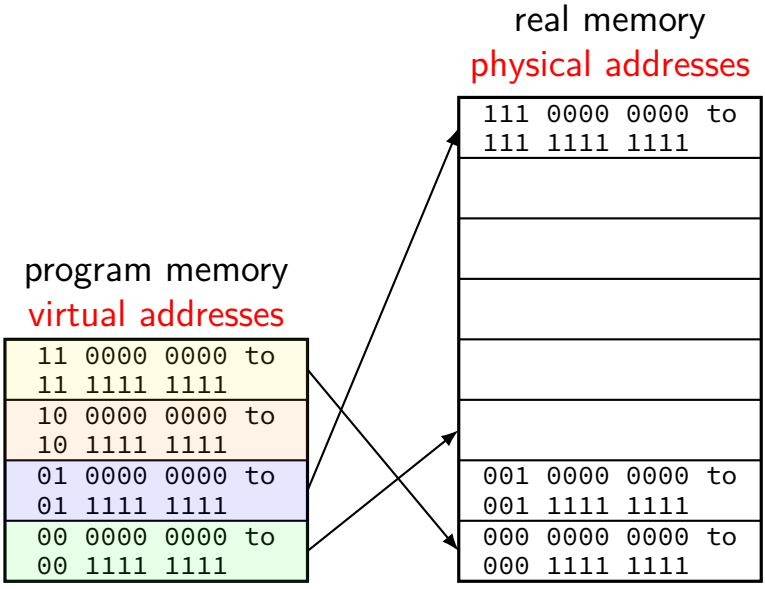
toy virtual and physical memory



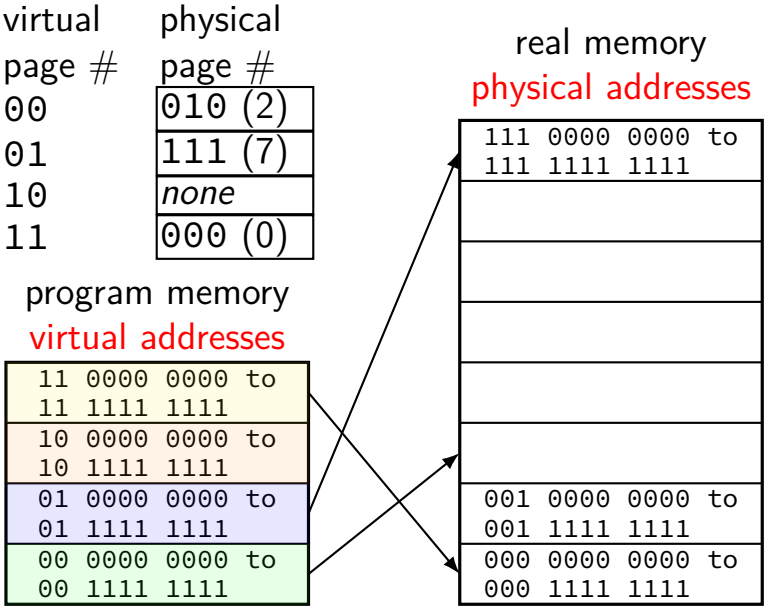
toy virtual and physical memory



toy virtual and physical memory



toy virtual and physical memory



toy virtual and physical memory

page table!

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

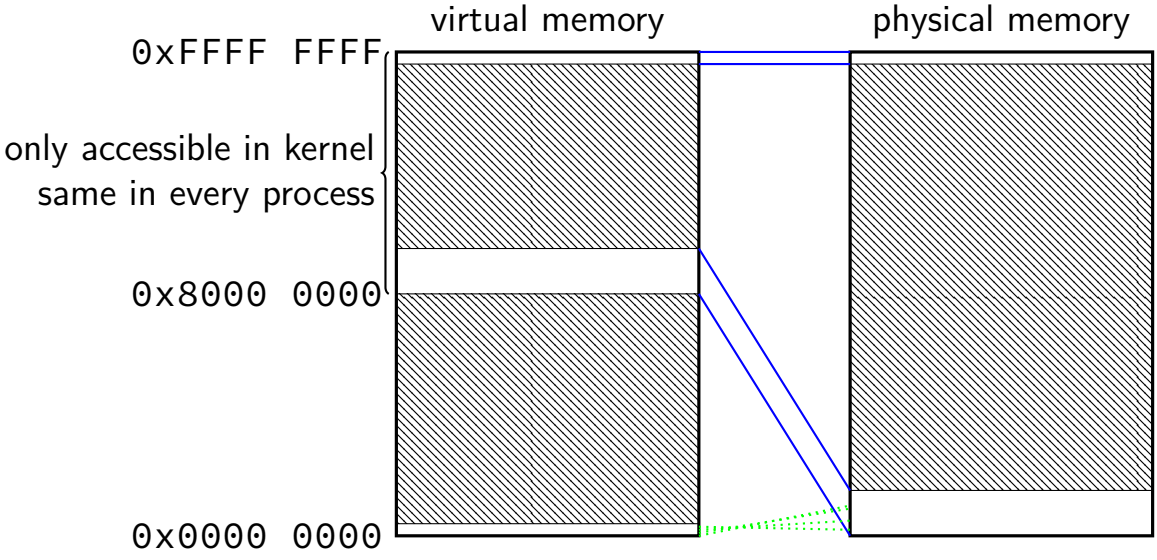
program memory
virtual addresses

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

xv6 memory layout



fast copies

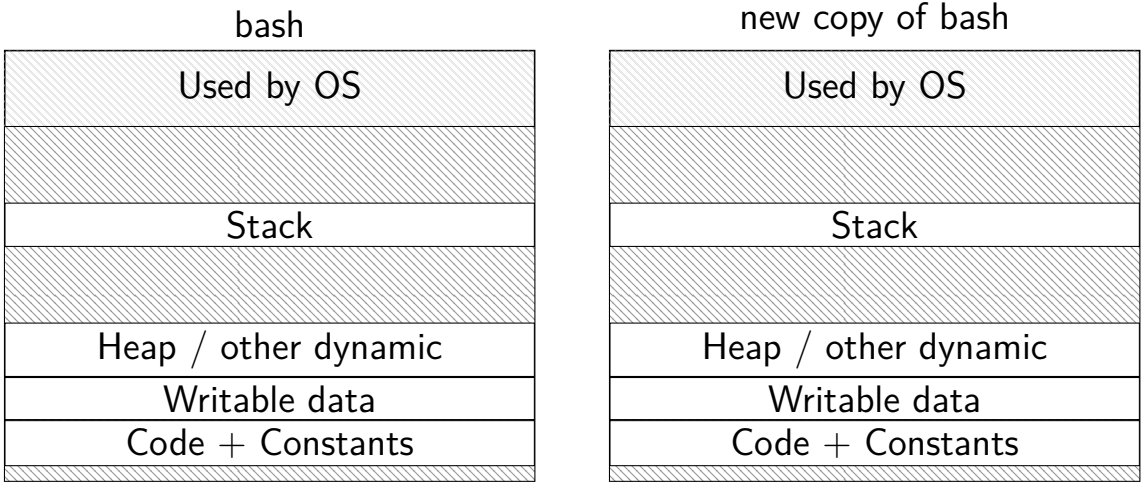
recall : `fork()`

creates a copy of an entire program!

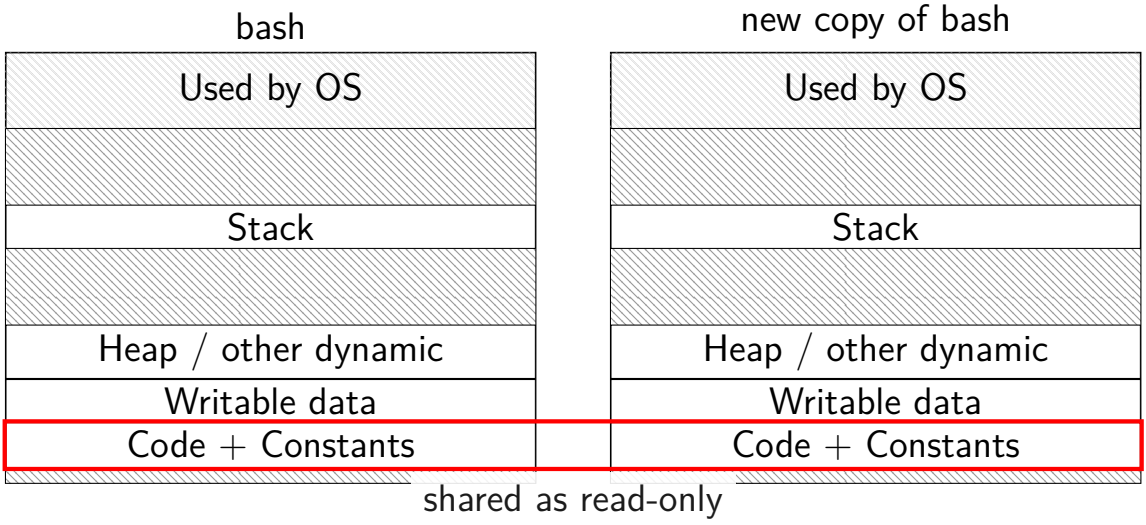
(usually, the copy then calls `execve` — replaces itself with another program)

how isn't this really slow?

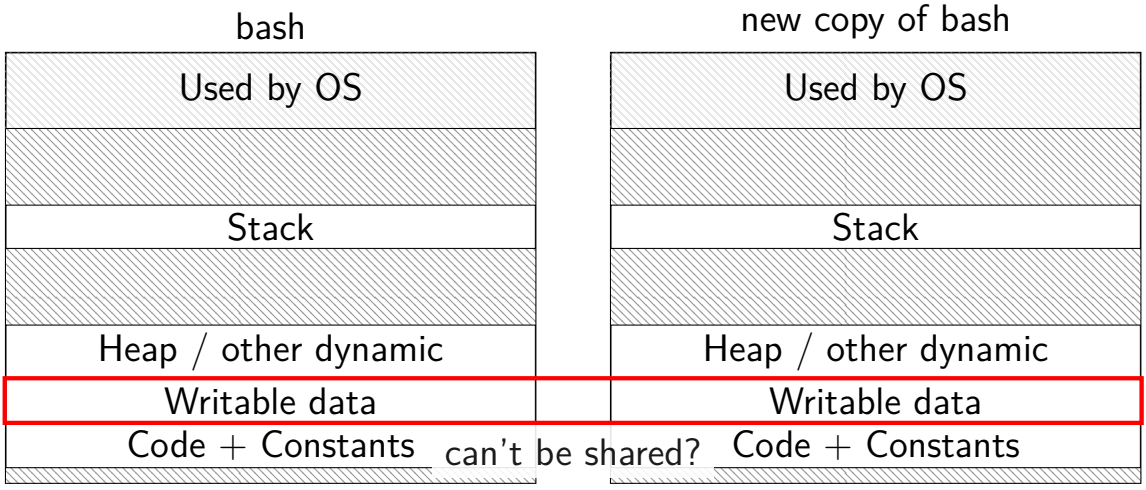
do we really need a complete copy?



do we really need a complete copy?



do we really need a complete copy?



trick for extra sharing

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

can we detect modifications?

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

processor will trigger a fault when it's written

copy-on-write and page tables

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

copy-on-write and page tables

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

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

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

copy-on-write and page tables

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

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

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

copy-on-write and page tables

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

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

after allocating a copy, OS reruns the write instruction

VM assignment

allocate heap on demand

- on page fault — check address
- need to write page fault handler
- if okay, allocate page

copy-on-write

- change fork to keep same page, make read-only
- on page fault for write — copy + allocate new page
- track number of references to each page

VM assignment restrictions

don't handle page faults triggered by kernel

- means system calls will break if passed uninit'd/copy-on-write memory

copy-on-write only reason for read-only pages

- detect copy-on-write via write-only

homework operations

add page fault handler

change growuvm (heap allocation function) to only change sz

edit page fault handler to allocate on demand

add reference counts for each physical copy-on-write page

do marking read-only on fork (in copyuvm)

handle deallocating with reference counts (several places)

xv6: adding space on demand

```
struct proc {
  uint sz;     // Size of process memory (bytes)
  ...
};
```

adding allocate on demand logic:

on page fault: if address $>$ sz

- kill process — out of bounds

on page fault: if address $\leq$ sz

- find virtual page number of address
- allocate page of memory, add to page table
- return from interrupt

versus more complicated OSes

range of valid addresses is not just 0 to maximum

need some more complicated data structure to represent

will get to that later

copy-on write cases

trying to write forbidden page (e.g. kernel memory)

- kill program instead of making it writable

trying to write read-only page and...

only one page table entry refers to it

- make it writeable
- return from fault

multiple process's page table entries refer to it

- copy the page
- replace read-only page table entry to point to copy
- return from fault

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
```

mmap options (1)

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

`length` bytes from open file `fd` starting at byte `offset`

protection flags `prot`, bitwise or together 1 or more of:

- PROT_READ
- PROT_WRITE
- PROT_EXEC
- PROT_NONE (for forcing segfaults)

mmap options (1)

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

`length` bytes from open file `fd` starting at byte `offset`

protection flags `prot`, bitwise or together 1 or more of:

- PROT_READ
- PROT_WRITE
- PROT_EXEC
- PROT_NONE (for forcing segfaults)

mmap options (1)

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

`length` bytes from open file `fd` starting at byte `offset`

protection flags `prot`, bitwise or together 1 or more of:

- PROT_READ
- PROT_WRITE
- PROT_EXEC
- PROT_NONE (for forcing segfaults)

mmap options (2)

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

`flags`, choose at least

- MAP_SHARED — changing memory changes file and vice-versa
- MAP_PRIVATE — make a copy of data in file (using copy-on-write)

…along with additional flags:

- MAP_ANONYMOUS (not POSIX) — ignore `fd`, just allocate space
- … (and more not shown)

`addr`, suggestion about where to put mapping (may be ignored)

- can pass NULL — "choose for me"
- address chosen will be returned

mmap options (2)

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

`flags`, choose at least

- MAP_SHARED — changing memory changes file and vice-versa
- MAP_PRIVATE — make a copy of data in file (using copy-on-write)

…along with additional flags:

- MAP_ANONYMOUS (not POSIX) — ignore `fd`, just allocate space
- … (and more not shown)

`addr`, suggestion about where to put mapping (may be ignored)

- can pass NULL — "choose for me"
- address chosen will be returned

Linux 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-2.19
7f60c764e000-7f60c784e000 ---p 001be000 08:01 96659129                   /lib/x86_64-linux-gnu/libc-2.19
7f60c784e000-7f60c7852000 r--p 001be000 08:01 96659129                   /lib/x86_64-linux-gnu/libc-2.19
7f60c7852000-7f60c7854000 rw-p 001c2000 08:01 96659129                   /lib/x86_64-linux-gnu/libc-2.19
7f60c7854000-7f60c7859000 rw-p 00000000 00:00 0
7f60c7859000-7f60c787c000 r-xp 00000000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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.so
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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

```
$ 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
7f60c764e000-7f60c784e000 -
7f60c784e000-7f60c7852000 r
7f60c7852000-7f60c7854000 rw-p 001c2000 08:01 96659129                   /lib/x86_64-linux-gnu/libc-2.19
7f60c7854000-7f60c7859000 rw-p 00000000 00:00 0
7f60c7859000-7f60c787c000 r-xp 00000000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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.so
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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]
```

at virtual addresses 0x400000–0x40b000

Linux 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 000000[illegible]                                                  -2.19
7f60c764e000-7f60c784e000 ---p 001be0[illegible]                                                  -2.19
7f60c784e000-7f60c7852000 r--p 001be0[illegible]                                                  -2.19
7f60c7852000-7f60c7854000 rw-p 001c20[illegible]                                                  -2.19
7f60c7854000-7f60c7859000 rw-p 000000[illegible]
7f60c7859000-7f60c787c000 r-xp 00000000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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.so
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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]
```

read, not write, execute, private

private = copy-on-write (if writeable)

Linux 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                                                              -2.19
7f60c764e000-7f60c784e000 ---p                                                              -2.19
7f60c784e000-7f60c7852000 r--p                                                              -2.19
7f60c7852000-7f60c7854000 rw-p 001c2000 08:01 96659129           /lib/x86_64-linux-gnu/libc-2.19
7f60c7854000-7f60c7859000 rw-p 00000000 00:00 0
7f60c7859000-7f60c787c000 r-xp 00000000 08:01 96659109           /lib/x86_64-linux-gnu/ld-2.19.so
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.so
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109           /lib/x86_64-linux-gnu/ld-2.19.so
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]
```

starting at offset 0 of the file `/bin/cat`

Linux 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
7f60c7490000-7
7f60c764e000-7
7f60c784e000-7
7f60c7852000-7
7f60c7854000-7
7f60c7859000-7
7f60c7a39000-7
7f60c7a7a000-7
7f60c7a7b000-7f60c7a7c000 r--p 00022000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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]
```

device major number 8

device minor number 1

inode 48328831

more on what this means when we talk about filesystems

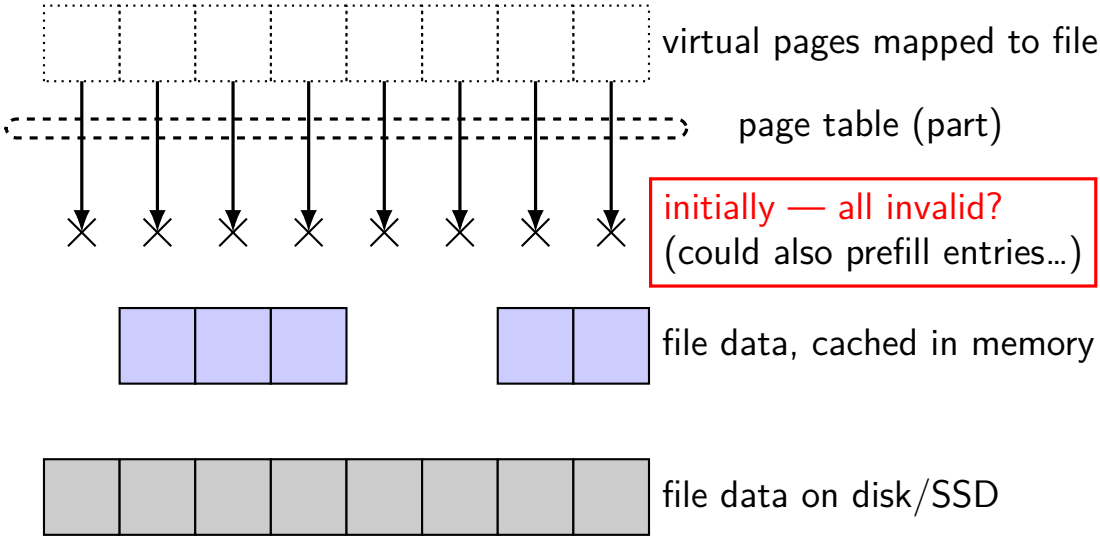
Linux maps

```
$ 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
7f60c7490000-7f6                                                                                    -2.19
7f60c764e000-7f6                                                                                    -2.19
7f60c784e000-7f6                                                                                    -2.19
7f60c7852000-7f6                                                                                    -2.19
7f60c7854000-7f6
7f60c7859000-7f6                                                                                    2.19.so
7f60c7a39000-7f6
7f60c7a7a000-7f60c7a7b000 rw-p 00000000 00:00 0
7f60c7a7b000-7f60c7a7c000 r--p 00022000 08:01 96659109   /lib/x86_64-linux-gnu/ld-2.19.so
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109   /lib/x86_64-linux-gnu/ld-2.19.so
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]
```

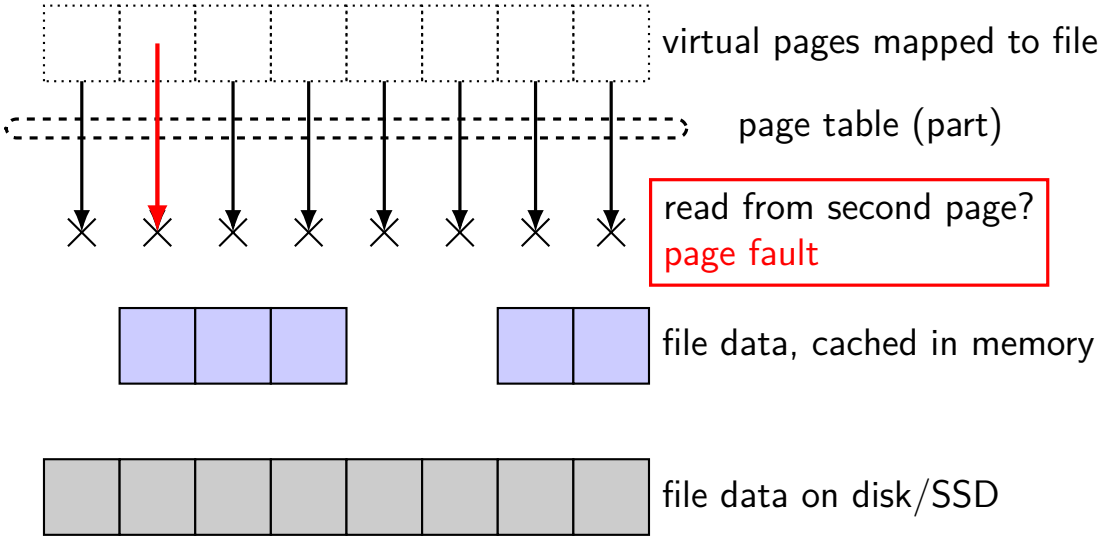
as if:

```
int fd = open("/bin/cat", O_RDONLY);
mmap(0x400000, 0x1000, PROT_READ | PROT_EXEC,
     MAP_PRIVATE, fd, 0xb000);
```

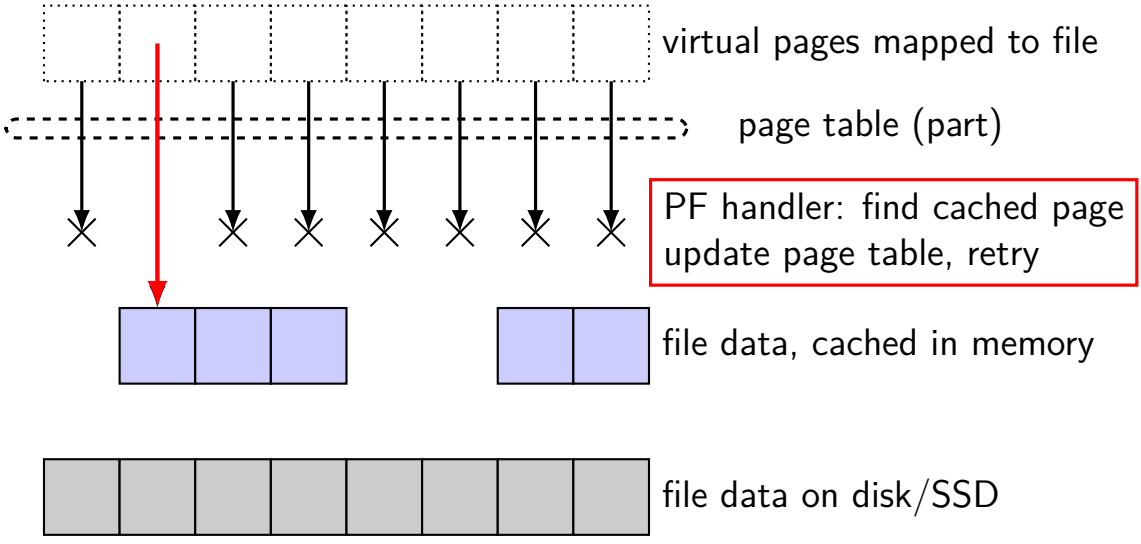
mapped pages (read-only)



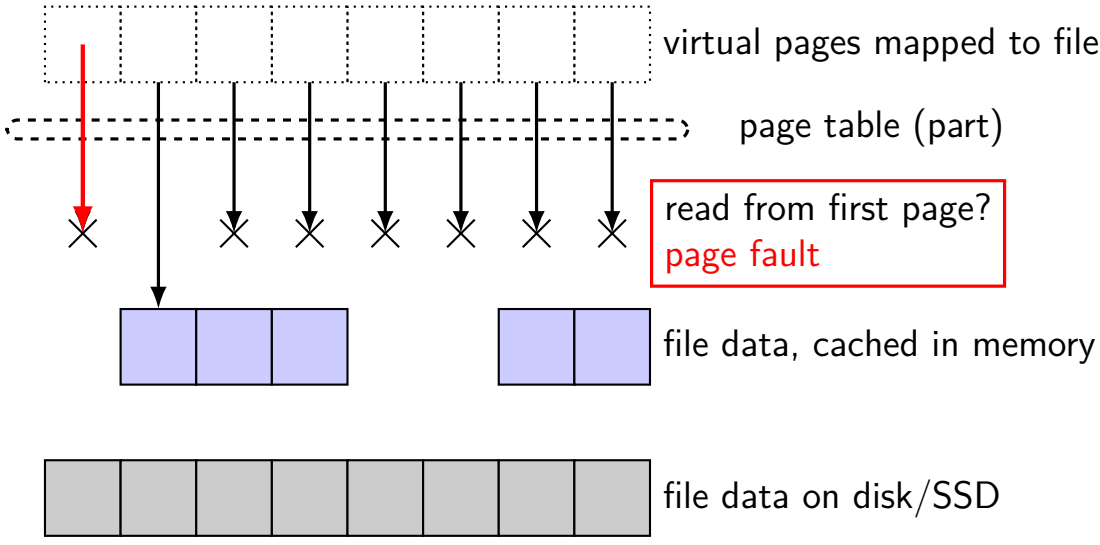
mapped pages (read-only)



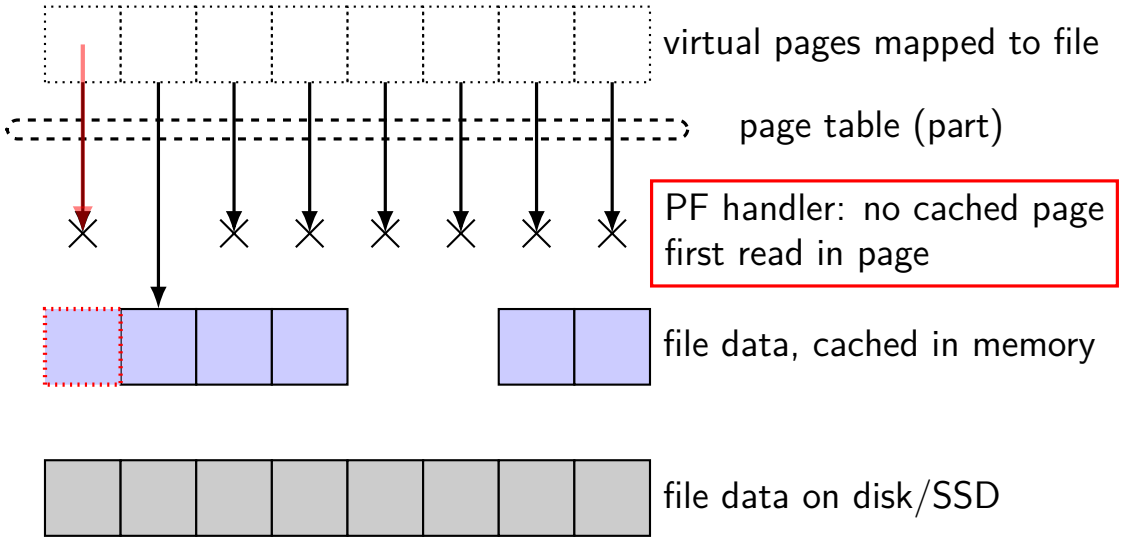
mapped pages (read-only)



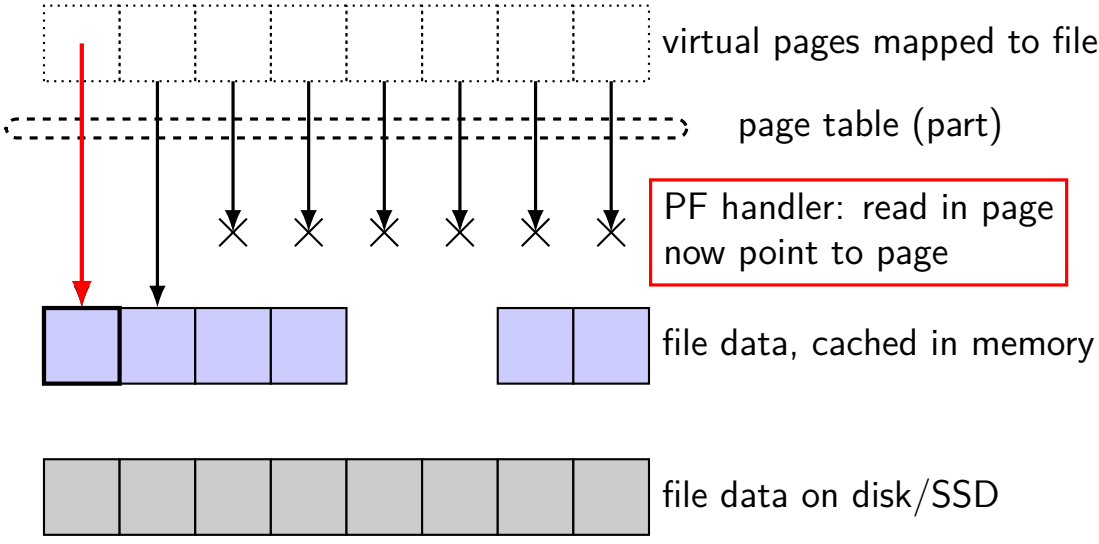
mapped pages (read-only)



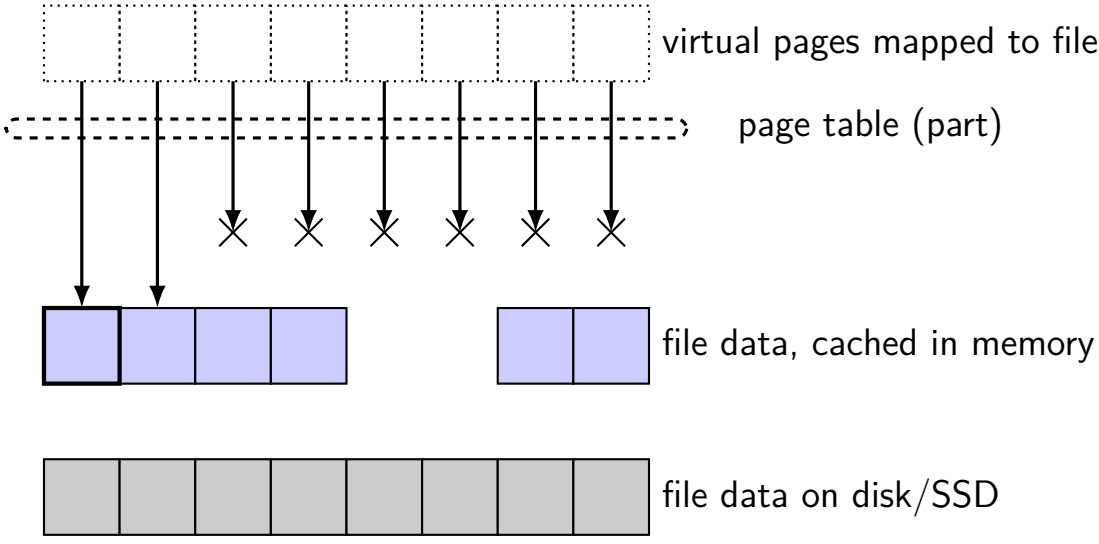
mapped pages (read-only)



mapped pages (read-only)



mapped pages (read-only)



shared mmap

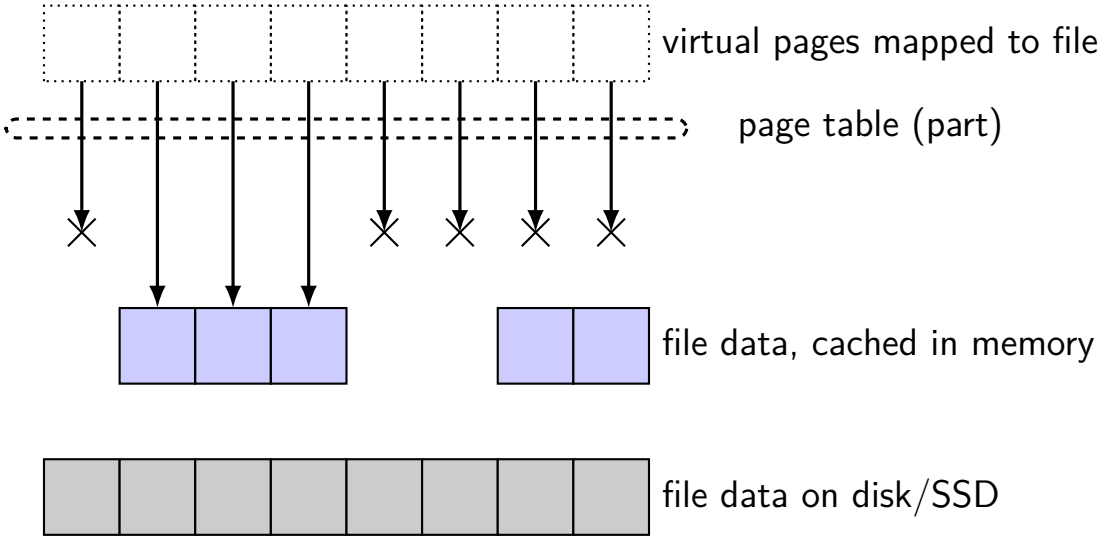
```
int fd  = open("/tmp/somefile.dat", O_RDWR);
mmap(0, 64 * 1024, PROT_READ | PROT_WRITE,
     MAP_SHARED, fd, 0);
```

---

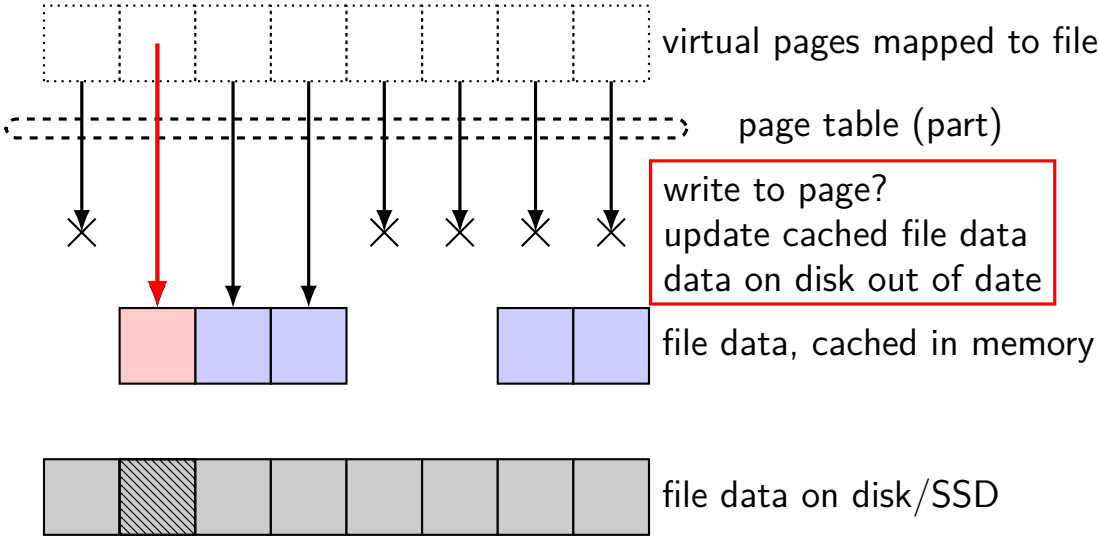
from /proc/PID/maps for this program:

```
7f93ad877000-7f93ad887000 rw-s 00000000 08:01 1839758 /tmp/somefile.dat
```

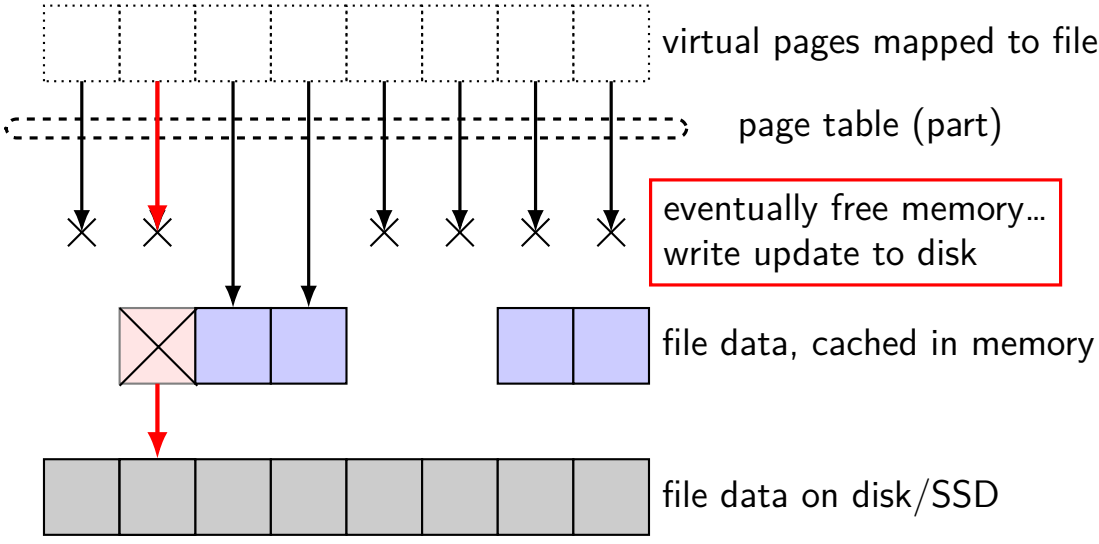
mapped pages (read/write, shared)



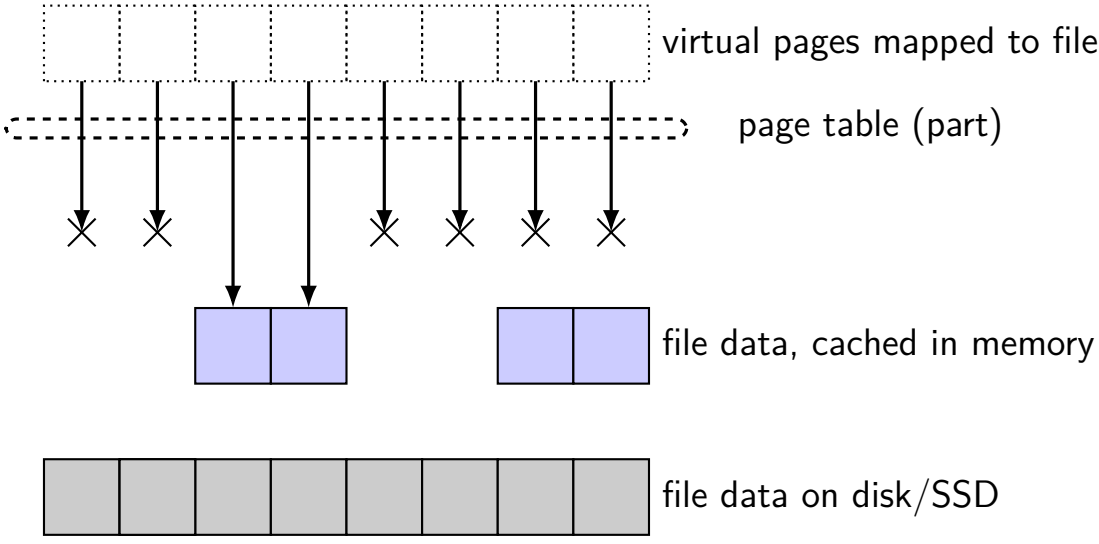
mapped pages (read/write, shared)



mapped pages (read/write, shared)



mapped pages (read/write, shared)



knowing when to write to disk?

need a *dirty bit* per page ("was page modified")

- D bit on PTEs we've seen

x86: 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

multiple dirty bits?

what if a page is in multiple page tables?

each page table has a dirty bit...

check all of them to decide if it was modified

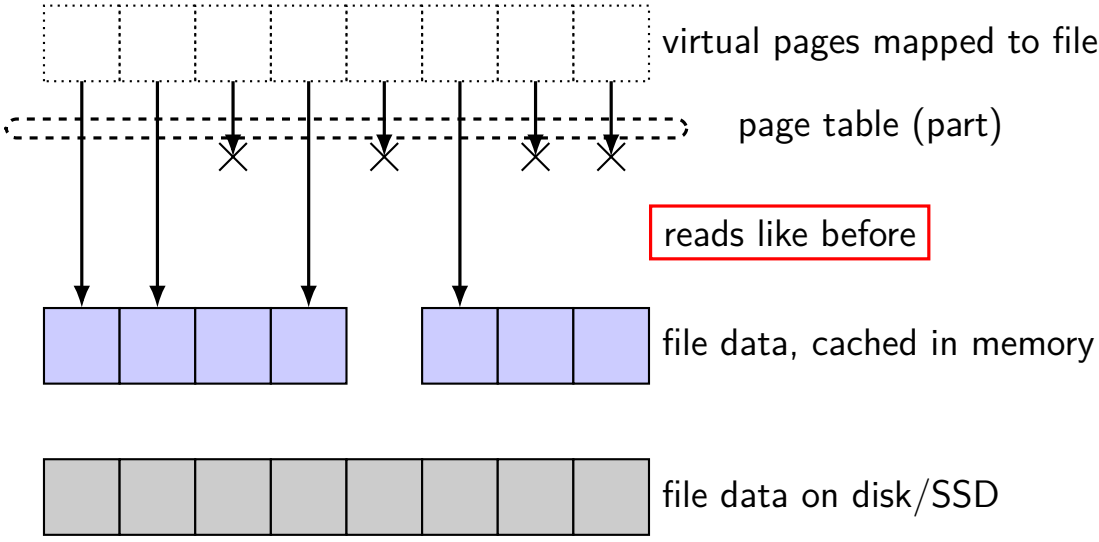
Linux maps

```
$ 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
7f60c7490000-7f
7f60c764e000-7f
7f60c784e000-7f
7f60c7852000-7f
7f60c7854000-7f
7f60c7859000-7f                                                                                    2.19.so
7f60c7a39000-7f
7f60c7a7a000-7f60c7a7b000 rw-p 00000000 00:00 0
7f60c7a7b000-7f60c7a7c000 r--p 00022000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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]
```

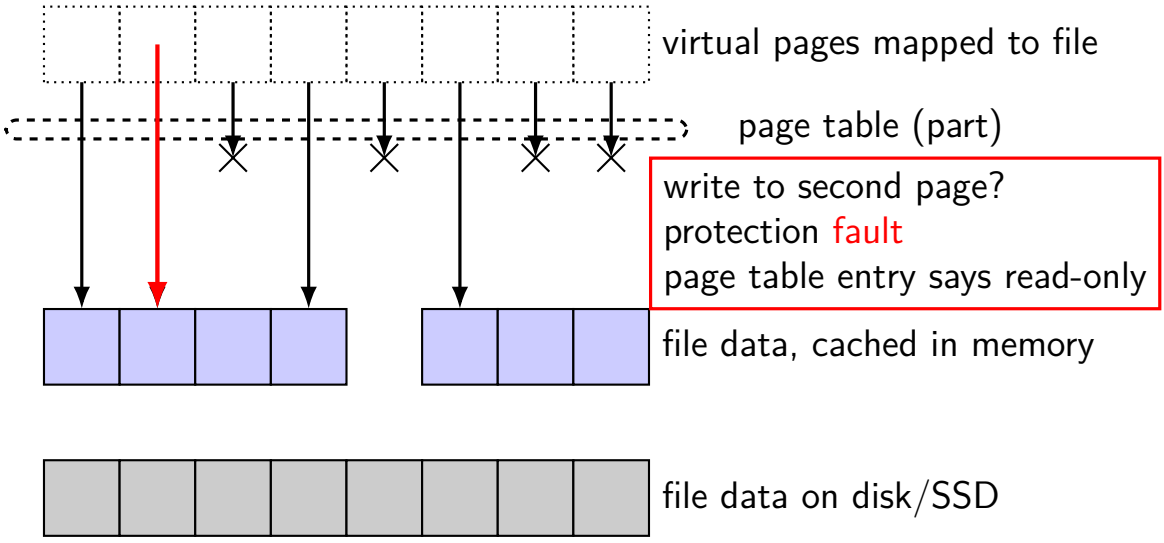
read/write, copy-on-write (private) mapping

```
int fd = open("/bin/cat", O_RDONLY);
mmap(0x60b000, 0x1000, PROT_READ | PROT_WRITE,
     MAP_PRIVATE, fd, 0xb000);
```

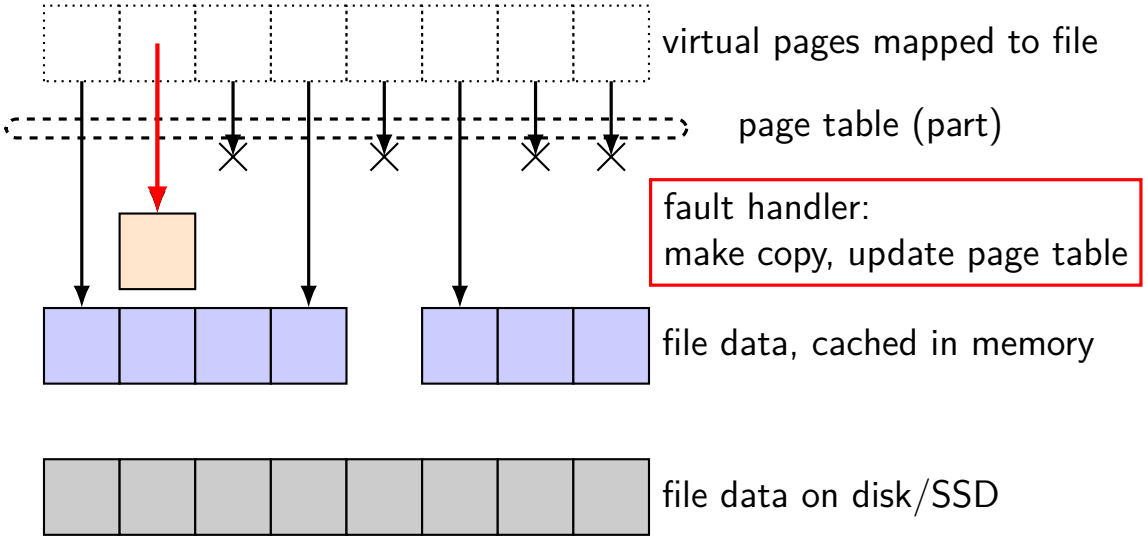
mapped pages (copy-on-write)



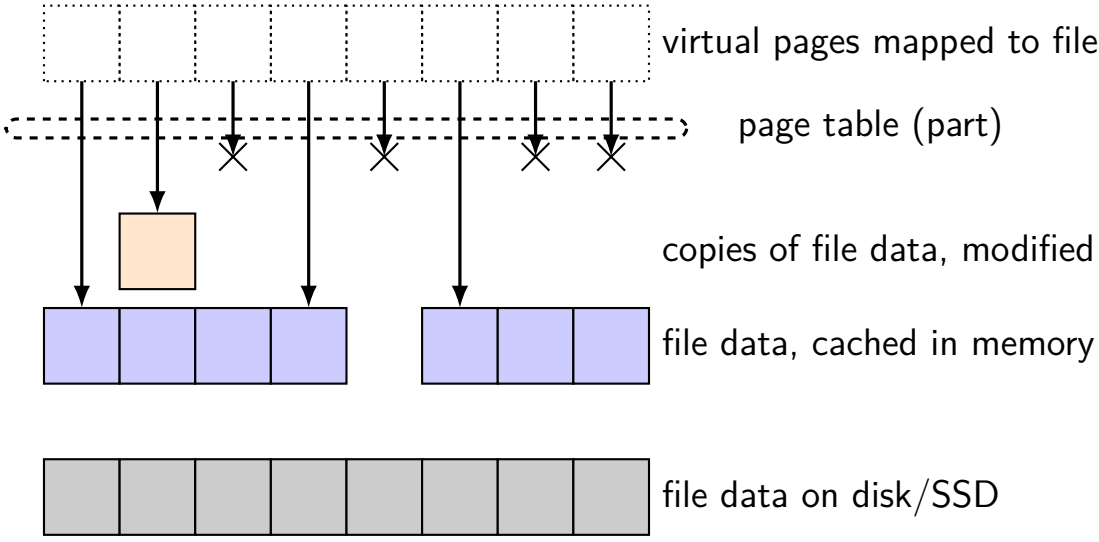
mapped pages (copy-on-write)



mapped pages (copy-on-write)



mapped pages (copy-on-write)

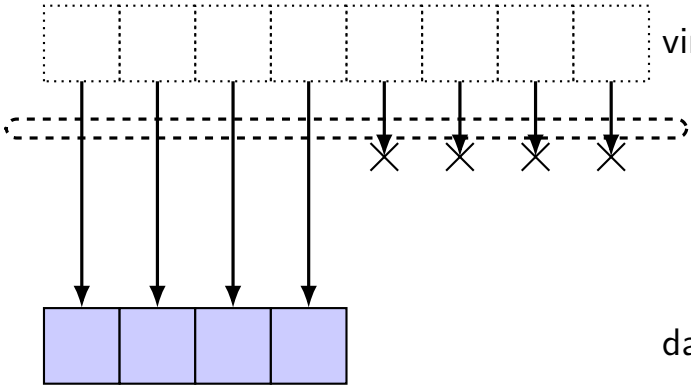


Linux maps

```
$ 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                                                    -2.19
7f60c764e000-7f60c784e000 ---p 001be000 08:01                                                    -2.19
7f60c784e000-7f60c7852000 r--p 001be000 08:01                                                    -2.19
7f60c7852000-7f60c7854000 rw-p 001c2000 08:01                                                    -2.19
7f60c7854000-7f60c7859000 rw-p 00000000 00:00 0
7f60c7859000-7f60c787c000 r-xp 00000000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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.so
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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]
```

heap — no corresponding file
just read/write memory

mapped pages (no backing file)



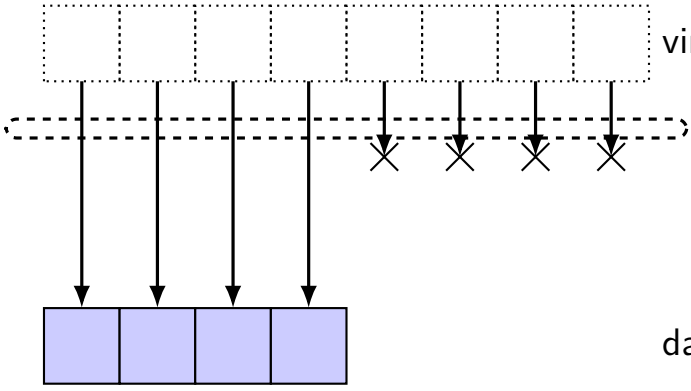
virtual pages w/o backing file

page table (part)

data in memory

swapped out data (if any)

mapped pages (no backing file)



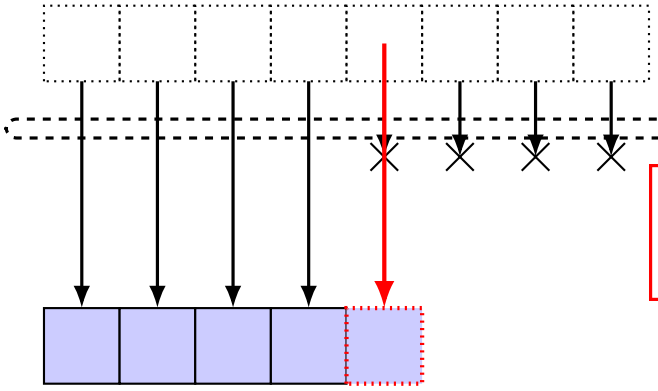
virtual pages w/o backing file

page table (part)

data in memory

swapped out data (if any)

mapped pages (no backing file)



virtual pages w/o backing file

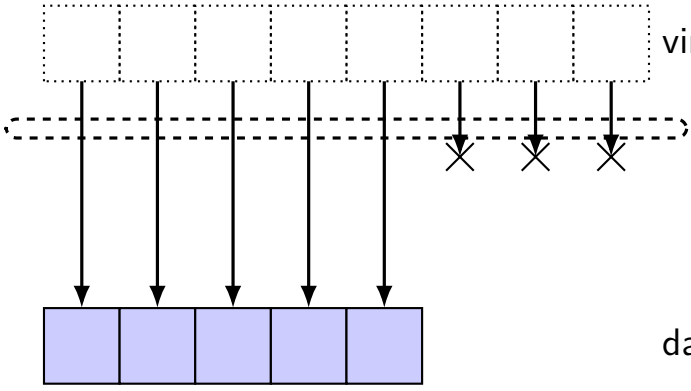
page table (part)

access new page
page fault handler
allocates on demand

data in memory

swapped out data (if any)

mapped pages (no backing file)



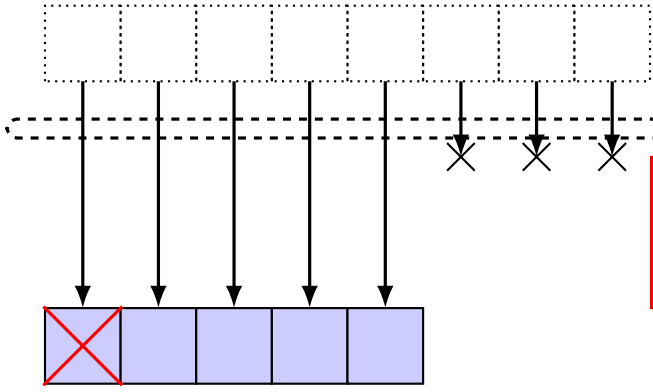
virtual pages w/o backing file

page table (part)

data in memory

swapped out data (if any)

mapped pages (no backing file)



virtual pages w/o backing file

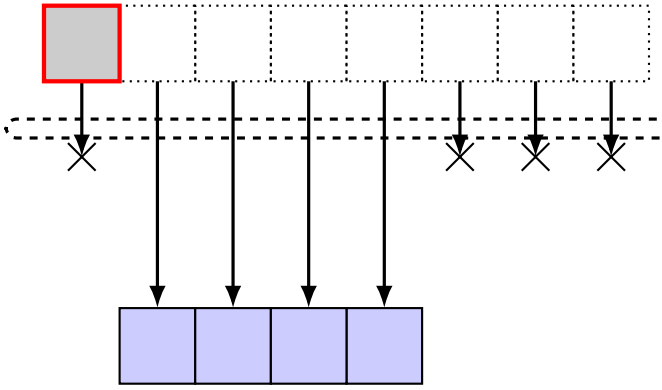
page table (part)

need more memory
save page to disk
temporarily

data in memory

swapped out data (if any)

mapped pages (no backing file)



virtual pages w/o backing file

page table (part)

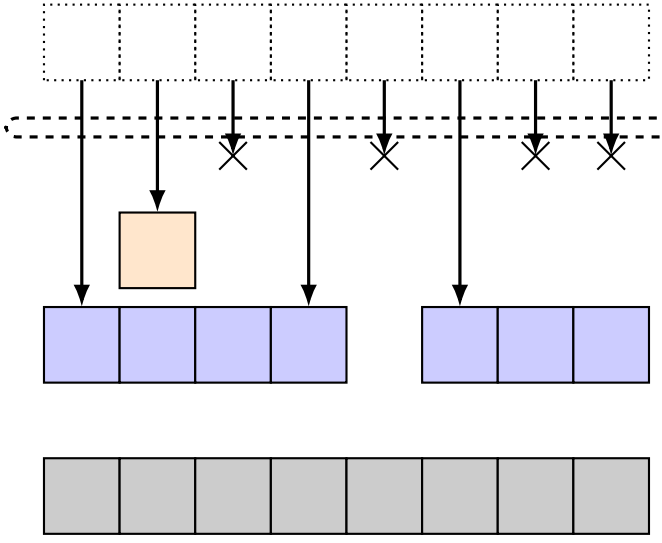
data in memory

swapped out data (if any)

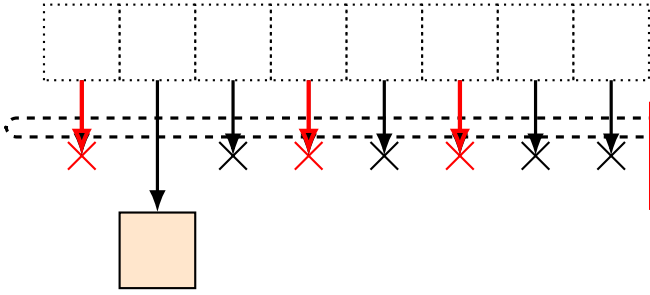
Linux maps

```
$ 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-2.19
7f60c764e000-7f60c784e000 ---p 001be000 08:01 96659129                   /lib/x86_64-linux-gnu/libc-2.19
7f60c784e000-7f60c7852000 r--p 001be000 08:01 96659129                   /lib/x86_64-linux-gnu/libc-2.19
7f60c7852000-7f60c7854000 rw-p 001c2000 08:01 96659129                   /lib/x86_64-linux-gnu/libc-2.19
7f60c7854000-7f60c7859000 rw-p 00000000 00:00 0
7f60c7859000-7f60c787c000 r-xp 00000000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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.so
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109                   /lib/x86_64-linux-gnu/ld-2.19.so
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]
```

swapping with copy-on-write



swapping with copy-on-write



virtual pages mapped to file

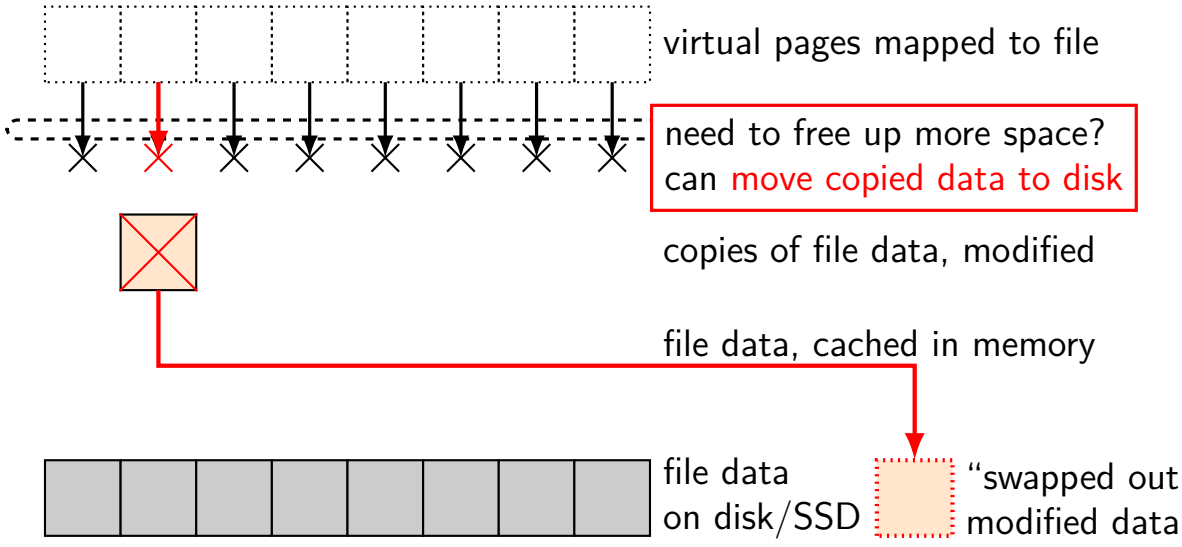
free up space by removing cached copies of file

copies of file data, modified

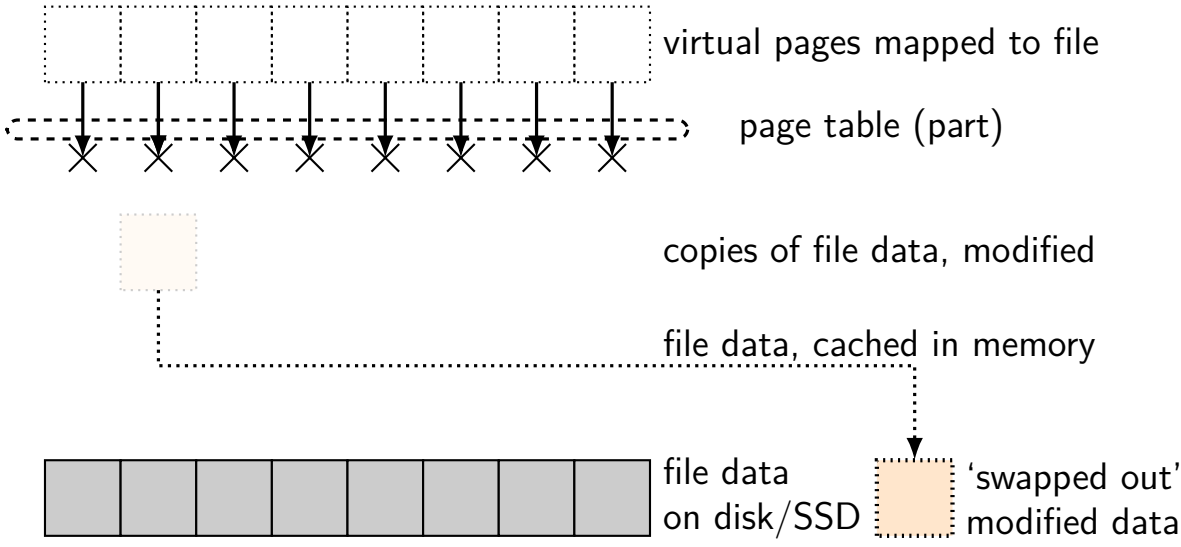
file data, cached in memory

file data on disk/SSD

swapping with copy-on-write



swapping with copy-on-write



swapping

historical major use of virtual memory is supporting "swapping"

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

process is allocated space on disk/SSD

memory is a cache for disk/SSD

- only need keep 'currently active' pages in physical memory

swapping

historical major use of virtual memory is supporting "swapping"

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

process is allocated space on disk/SSD

memory is a cache for disk/SSD

- only need keep 'currently active' pages in physical memory

swapping $\approx$ mmap with "default" files to use

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)

the page cache

memory is a cache for disk

files, program memory has a place on disk

- running low on memory? always have room on disk
- assumption: disk space approximately infinite

physical memory pages: disk data 'temporarily' kept in faster storage

- possibly being used by one or more processes?
- possibly part of a file on disk?
- possibly both

goal: manage this cache intelligently

memory as a cache for disk

"cache block" $\approx$ physical page

fully associative

- any virtual address/file part 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

page cache components

mapping: virtual address or file+offset → physical page

- handle cache hits

find backing location based on virtual address/file+offset

- handle cache misses

track information about each physical page

- handle page allocation
- handle cache eviction

page cache components

mapping: virtual address or file+offset → physical page

- handle cache hits

find backing location based on virtual address/file+offset

- handle cache misses

track information about each physical page

- handle page allocation
- handle cache eviction

virtual address/file offset → physical page

"cache hits"

page table: virtual address → physical page (if any)

- mapping found? cache hit on program memory access
- structure determined by hardware — involved in every memory access

kernel data structures: file offset → physical page (if any)

- mapping found? cache hit on read/write system call
- (or cache hit on page fault for mmap'd memory)
- multiple possible designs (software data structure)
- one idea: balanced tree: offset → physical page

Linux: tracking files in memory

```
struct file {
    ...
    struct inode *f_inode;
    ...
};
...
struct inode {
    ...
    struct address_space i_data;
    ...
};
...
struct address_space {
    ...
    struct radix_tree_root  i_pages;        /* cached pages */
    atomic_t                i_mmap_writable;/* count VM_SHARED mappi
    struct rb_root_cached   i_mmap;         /* tree of private and s
    ...
```

Linux: tracking files in memory

```
struct file {
    ...
    struct inode *f_inode;
    ...
};
...
struct inode {
    ...
    struct address_space i_data;
    ...
};
...
struct address_space {
    ...
    struct radix_tree_root  i_pages;        /* cached pages */
    atomic_t                i_mmap_writable;/* count VM_SHARED mappi
    struct rb_root_cached   i_mmap;         /* tree of private and s
    ...
```

`struct inode` represents file on disk (versus a file that's a pipe, terminal, etc.)

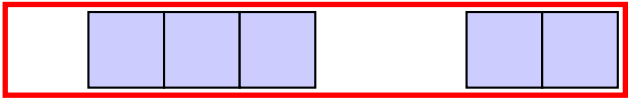
Linux: tracking files in memory

```
struct file {
    ...
    struct inode *f_inode;
    ...
};
...
struct inode {
    ...
    struct address_space i_data;
    ...
};
...
struct address_space {
    ...
    struct radix_tree_root  i_pages;        /* cached pages */
    atomic_t                i_mmap_writable;/* count VM_SHARED mappi
    struct rb_root_cached   i_mmap;         /* tree of private and s
    ...
```

way to find cached copies of parts of file
copies can be shared between processes

Linux's choice: tree of cached pages

mapped pages (read/write, shared)



file data, cached in memory

file data on disk/SSD

page cache components

mapping: virtual address or file+offset → physical page

- handle cache hits

find backing location based on virtual address/file+offset

- handle cache misses

track information about each physical page

- handle page allocation
- handle cache eviction

virtual address/file offset → location on disk

"cache miss"

for memory mapped to files:

- need data structure saying where files are mapped
- (then rely on filesystem)
- seen dump of this data structure in Linux: /proc/PID/maps

for "swapped out" data outside of files:

- (heap memory, modified copy-on-write copies of files, etc.)
- need some way to track swapped out, modified pages
- hopefully not too big...

for data in files: depends on filesystem (topic for later)

virtual address/file offset → location on disk

"cache miss"

for memory mapped to files:

- need data structure saying where files are mapped
- (then rely on filesystem)
- seen dump of this data structure in Linux: /proc/PID/maps

for "swapped out" data outside of files:

- (heap memory, modified copy-on-write copies of files, etc.)
- need some way to track swapped out, modified pages
- hopefully not too big...

for data in files: depends on filesystem (topic for later)

Linux: tracking memory regions

```
struct vm_area_struct { ...
    unsigned long vm_start;          /* Our start address within vm_m
    unsigned long vm_end;            /* The first byte after our end
                                        within vm_mm. */
    ...
    pgprot_t vm_page_prot;           /* Access permissions of this VM
    unsigned long vm_flags;          /* Flags, see mm.h. */
    ...
    struct anon_vma *anon_vma;       /* Serialized by page_table_lock
    ...
    unsigned long vm_pgoff;          /* Offset (within vm_file) in PA
                                        units */
    struct file * vm_file;           /* File we map to (can be NULL).
    ...
} __randomize_layout;
```

Linux: tracking memory regions

```
struct vm_area_struct { ...
    unsigned long vm_start;
    unsigned long vm_end;

    ...
    pgprot_t vm_page_prot;          /* Access permissions of this VM
    unsigned long vm_flags;         /* Flags, see mm.h. */
    ...
    struct anon_vma *anon_vma;      /* Serialized by page_table_lock
    ...
    unsigned long vm_pgoff;         /* Offset (within vm_file) in PA
                                       units */
    struct file * vm_file;          /* File we map to (can be NULL).
    ...
} __randomize_layout;
```

virtual addresses of mapping
mapping are part of sorted list/tree
to allow finding by start/end address

Linux: tracking memory regions

```
struct vm_area_struct { ...
    unsigned long vm_start;          /* Ou                    vm_m
    unsigned long vm_end;            /* Th                    end
                                        within vm_mm. */

    ...
    pgprot_t vm_page_prot;           /* Access permissions of this VM
    unsigned long vm_flags;          /* Flags, see mm.h. */
    ...
    struct anon_vma *anon_vma;       /* Serialized by page_table_lock
    ...
    unsigned long vm_pgoff;          /* Offset (within vm_file) in PA
                                        units */
    struct file * vm_file;           /* File we map to (can be NULL).
    ...
} __randomize_layout;
```

file to get data from
and location within file

Linux: tracking memory regions

```
struct vm_area_struct { ...
    unsigned long vm_start;
    unsigned long vm_end;

    ...
    pgprot_t vm_page_prot;
    unsigned long vm_flags;
    ...
    struct anon_vma *anon_vma;
    ...
    unsigned long vm_pgoff;

    struct file * vm_file;
    ...
} __randomize_layout;
```

permissions (read/write/execute)

```
                                vm_m
                    /* The first byte after our end
                       within vm_mm. */

                    /* Access permissions of this VM
                    /* Flags, see mm.h. */

                    /* Serialized by page_table_lock

                    /* Offset (within vm_file) in PA
                       units */
                    /* File we map to (can be NULL).
```

Linux: tracking memory regions

```
struct vm_area_struct { ·
    unsigned long vm_star                                   vm_m
    unsigned long vm_end;                                   end

    ...
    pgprot_t vm_page_prot;          /* Access permissions of this VM
    unsigned long vm_flags;         /* Flags, see mm.h. */
    ...
    struct anon_vma *anon_vma;      /* Serialized by page_table_lock
    ...
    unsigned long vm_pgoff;         /* Offset (within vm_file) in PA
                                       units */
    struct file * vm_file;          /* File we map to (can be NULL).
    ...
} __randomize_layout;
```

flags: private or shared? ...
private = copy-on-write
shared = make changes to underlying file

virtual address/file offset → location on disk

"cache miss"

for memory mapped to files:

- need data structure saying where files are mapped
- (then rely on filesystem)
- seen dump of this data structure in Linux: /proc/PID/maps

for "swapped out" data outside of files:

- (heap memory, modified copy-on-write copies of files, etc.)
- need some way to track swapped out, modified pages
- hopefully not too big...

for data in files: depends on filesystem (topic for later)

Linux: tracking swapped out pages

need to lookup location on disk

potentially one location for every virtual page

trick: store location in page table entry

instead of physical page #, permission bits, etc., store offset on disk

on page fault: examine page table entry to read from disk

page cache components

mapping: virtual address or file+offset → physical page

- handle cache hits

find backing location based on virtual address/file+offset

- handle cache misses

track information about each physical page

- handle page allocation
- handle cache eviction

tracking physical pages: finding free pages

Linux has list of "least recently used" pages:

```
struct page {
    ...
    struct list_head lru;   /* list_head ~ next/prev pointer */
    ...
};
```

how we're going to find a page to allocate
(and evict from something else)

later — what this list actually looks like (how many lists, ...)

page cache components

mapping: virtual address or file+offset → physical page

- handle cache hits

find backing location based on virtual address/file+offset

- handle cache misses

track information about each physical page

- handle page allocation
- handle cache eviction

tracking physical pages: finding mappings

want to evict a page? remove from page tables, etc.

need to track where every page is used!

Linux: physical page → file → PTE

Linux tracking where file pages are in page tables:

```
struct page {
    ...
    struct address_space *mapping;
    pgoff_t index;              /* Our offset within mapping. */
    ...
};
struct address_space {
    ...
    struct rb_root_cached       i_mmap; /* tree of private and share
    ...
};
```

tree of mappings lets us find vm_area_structs and PTEs

rather complicated look up (but writing ot disk is already slow)

Linux: physical page → PTE w/o file

Linux also tracks location of "anonymous" (non-file) pages

mapping from page to list of vm_area_structs that contain page

- recall: vm_area_struct: one memory allocation in one process

exercise: why a list?

- what's one case when non-file memory is shared between processes?

list of allocations per page

naive solution: seperate list for each page?

- a lot of overhead (many tens of bytes per 4K page?)

but, trick: many pages 'copied' at the same time (e.g. fork)

idea: share list between all pages

- Linux represents each list as `struct anon_vma` (mmap region)
- initially: list one of mmap region
- on fork: add to existing list; create a new one

Linux: tracking memory regions

```
struct vm_area_struct { ...
    unsigned long vm_start;
    unsigned long vm_end;

    ...
    pgprot_t vm_page_prot;
    unsigned long vm_flags;
    ...
    struct anon_vma *anon_vma;
    ...
    unsigned long vm_pgoff;

    struct file * vm_file;
    ...
} __randomize_layout;
```

```
/                                   vm_
/                                   end
/                                   is VM
/* Flags, see mm.h. */
/* Serialized by page_table_lock
/* Offset (within vm_file) in PA
   units */
/* File we map to (can be NULL).
```

for tracking pages
that aren't part of file
(from copy-on-write, or
for non-file-backed memory)

page replacement

step 1: evict a page to free a physical page

step 2: load new, more important in its place

evicting a page

find a 'victim' page to evict

remove victim page from page table, etc.

- every page table it is referenced by
- every list of file pages
- ...

if needed, save victim page to disk

page replacement goals

hit rate: minimize number of misses

throughput: minimize overhead/maximize performance

fairness: every process/user gets its 'share' of memory

will start with optimizing hit rate

max hit rate $\approx$ max throughput

optimizing hit rate almost optimizes throughput, but...

max hit rate $\approx$ max throughput

optimizing hit rate almost optimizes throughput, but...

cache miss costs are variable

- creating zero page versus reading data from slow disk?
- write back dirty page before reading a new one or not?
- reading multiple pages at a time from disk (faster per page read)?
- ...

being proactive?

can avoid misses by "reading ahead"

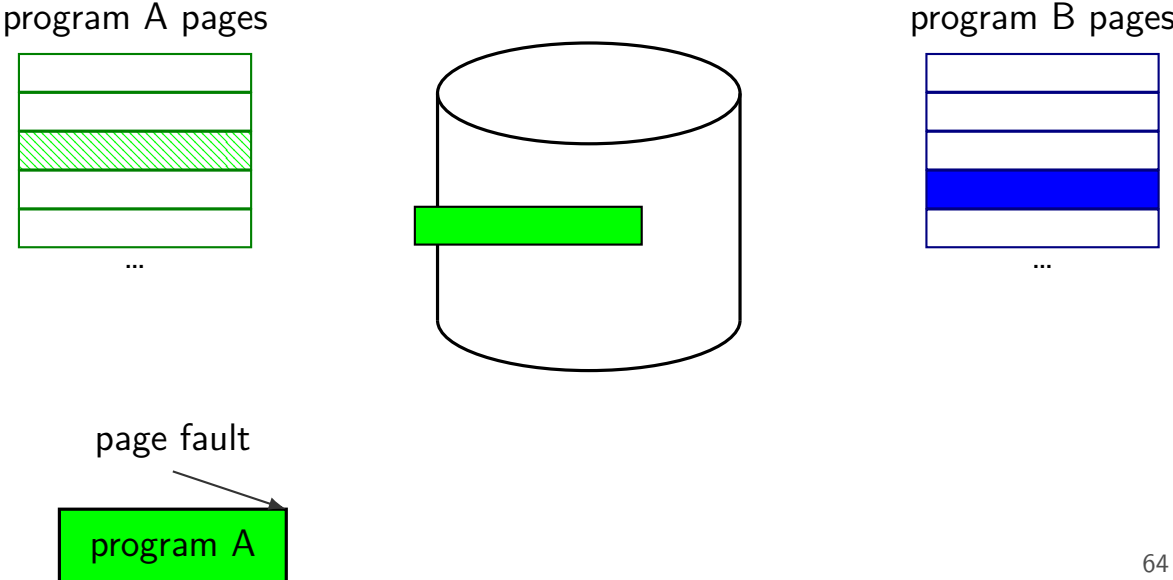
- guess what's needed — read in ahead of time
- wrong guesses can have costs besides more cache misses

we will get back to this later

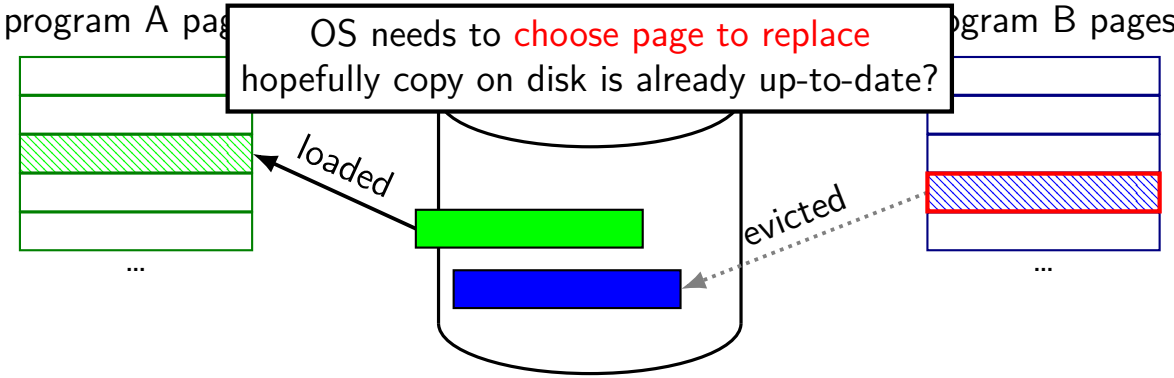
for now — only access/evict on demand

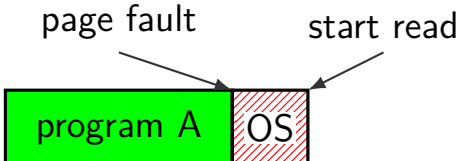
backup slides

swapping timeline

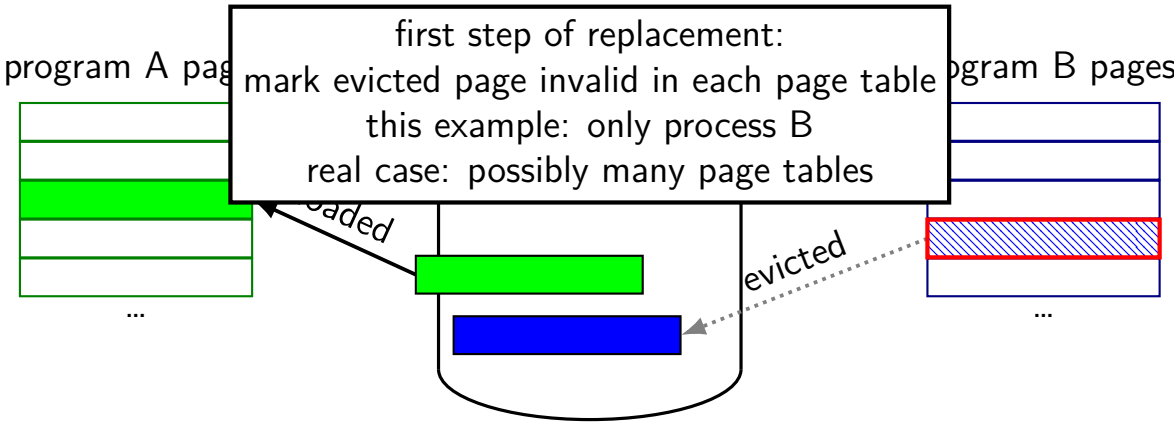


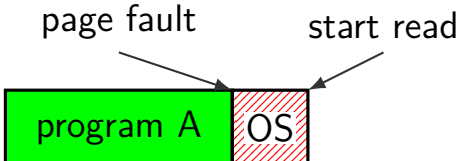
swapping timeline



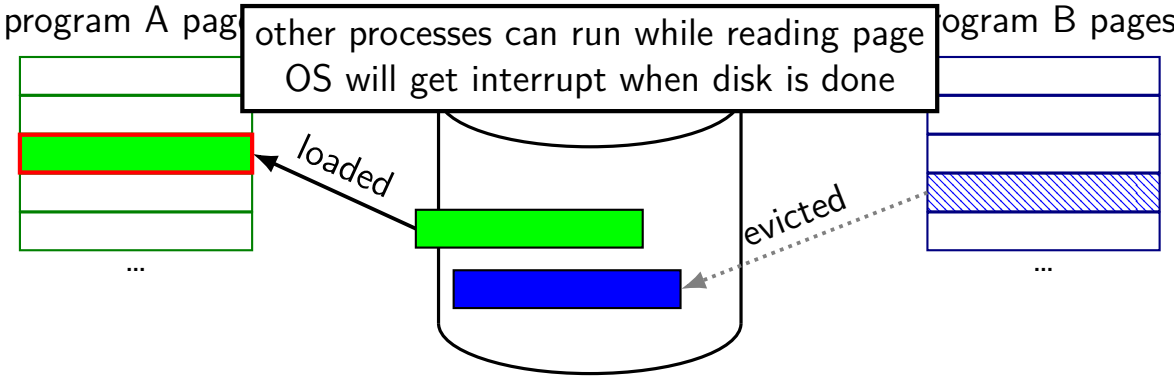


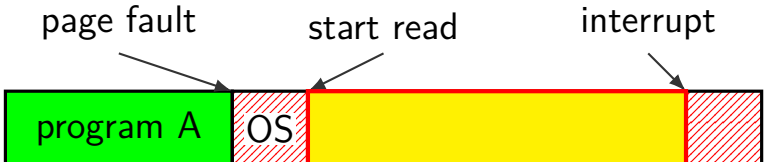
swapping timeline



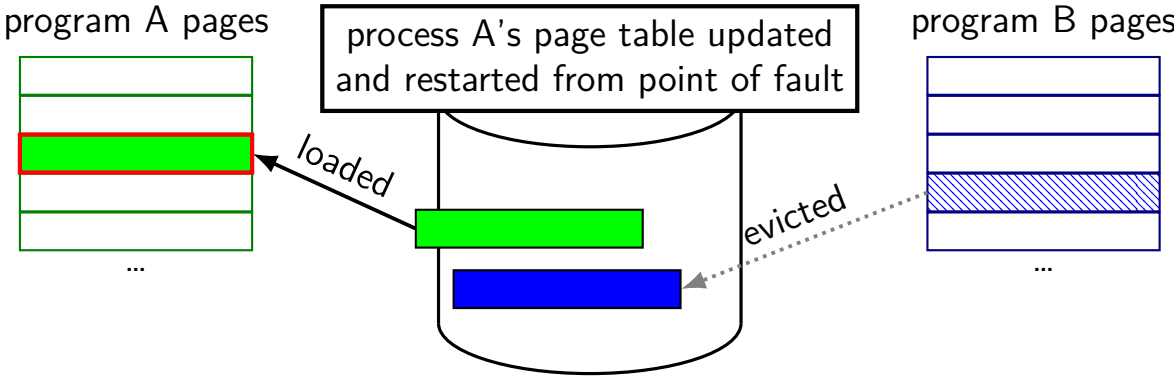


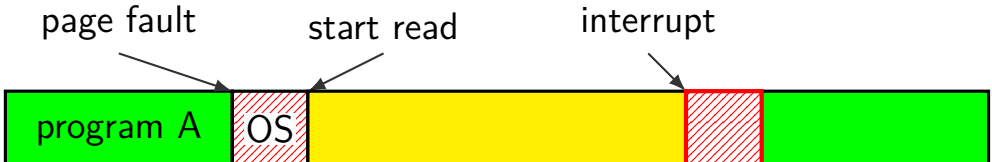
swapping timeline





swapping timeline





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")

x86: 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!)

replacement policy implemented in software

- a lot more room for fancy policies

usualy goal: least-recently-used approximation

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

second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

page 2 was at bottom of list
is not referenced
okay to use

	A	B						A	—	C	—	
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

page 1 was at bottom of list
reference — give second chance
moves to top of list
clear referenced bit

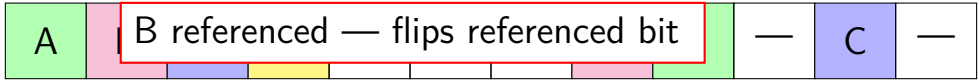
	A	B						A	—	C	—	
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

eventually page 1 gets to bottom of list again
but now not referenced — use

	A										C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

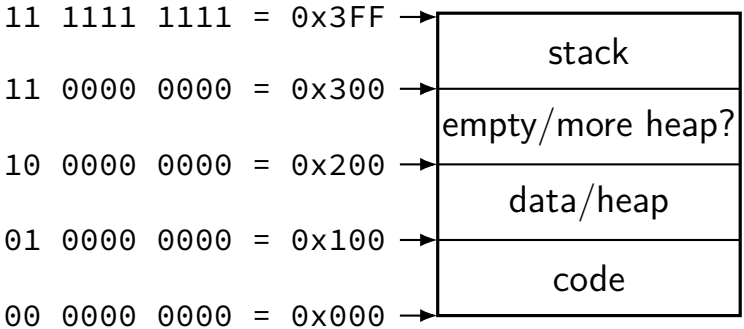


1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

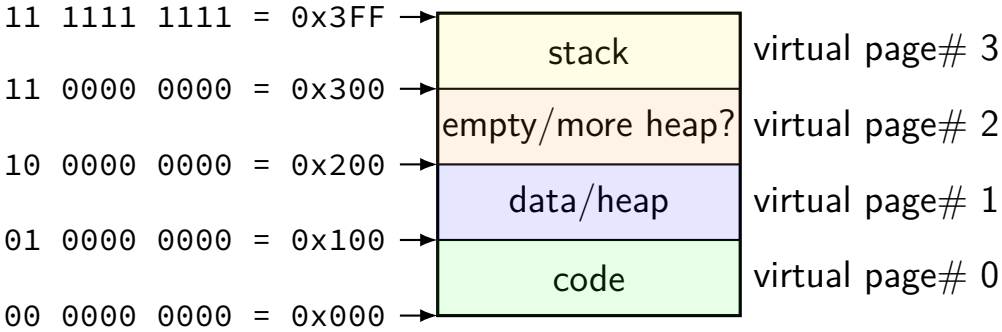
second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

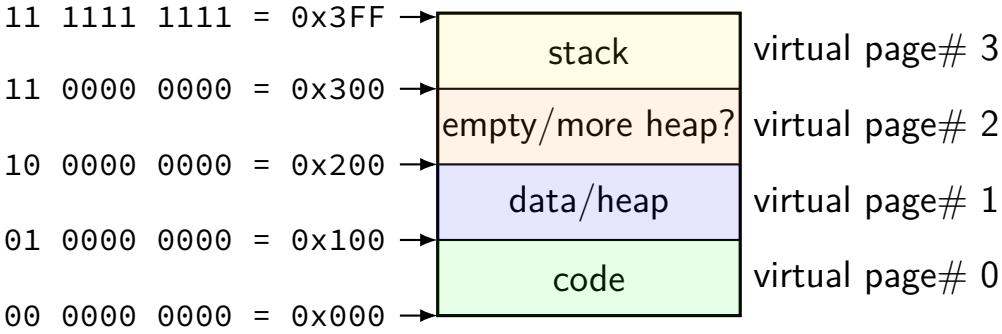
toy program memory



toy program memory

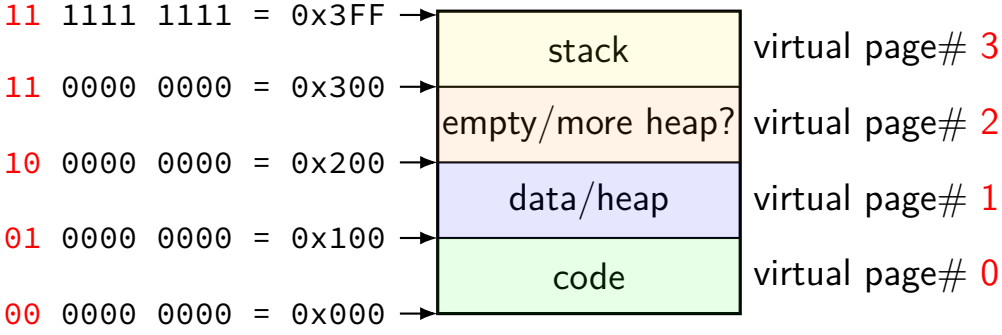


toy program memory



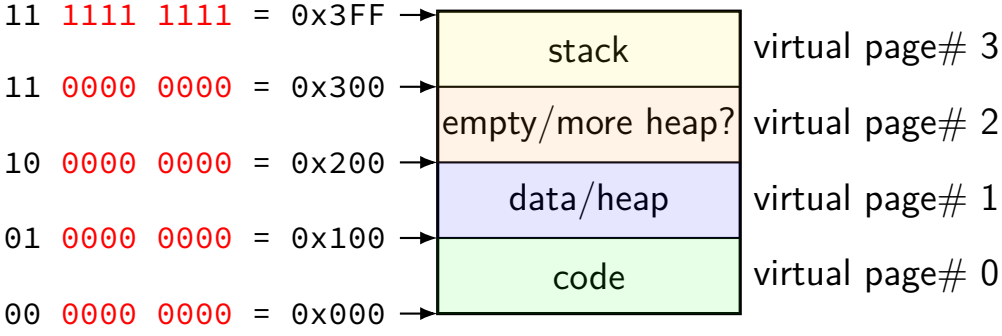
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



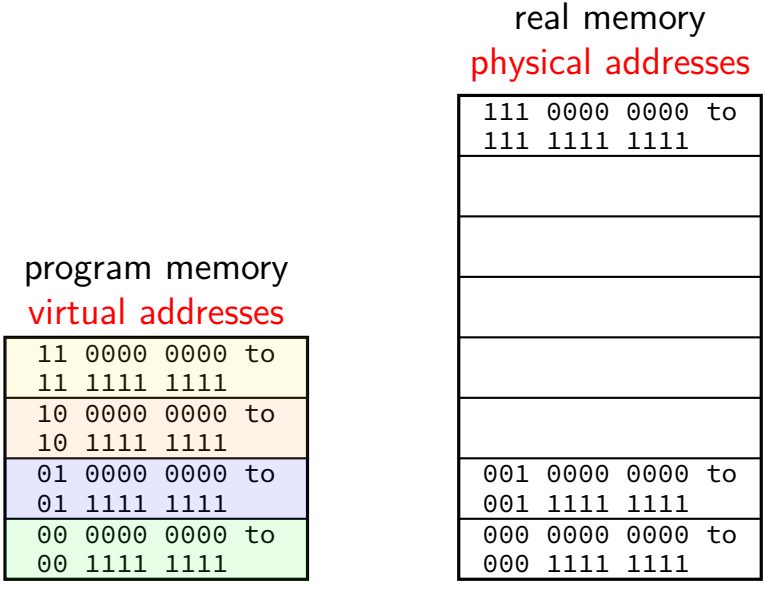
page number is upper bits of address
(because page size is power of two)

toy program memory

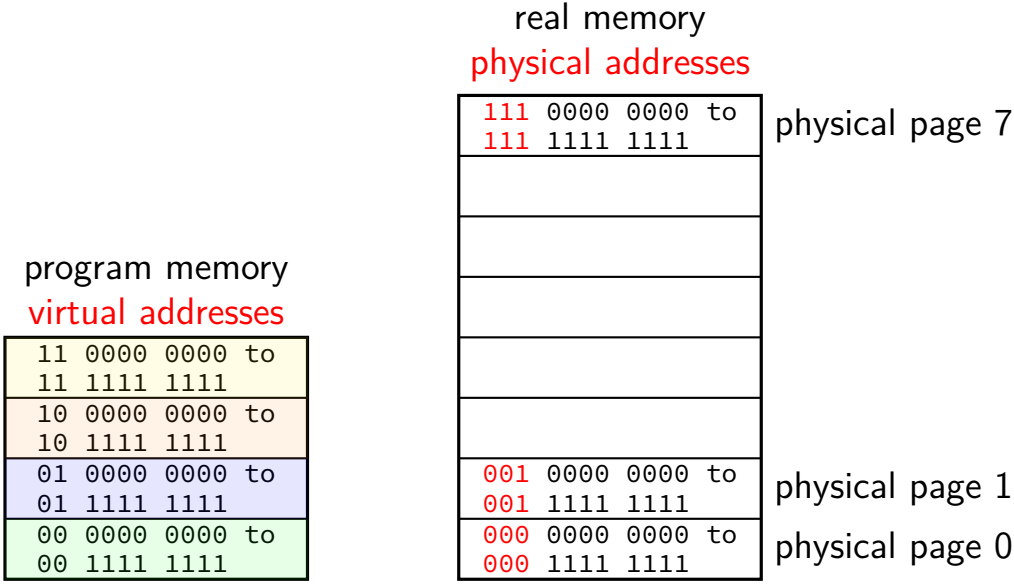


rest of address is called page offset

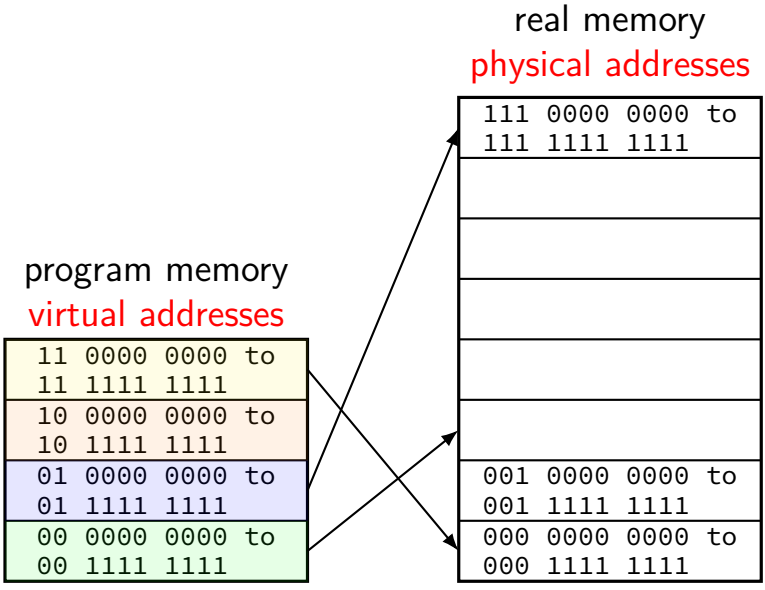
toy physical memory



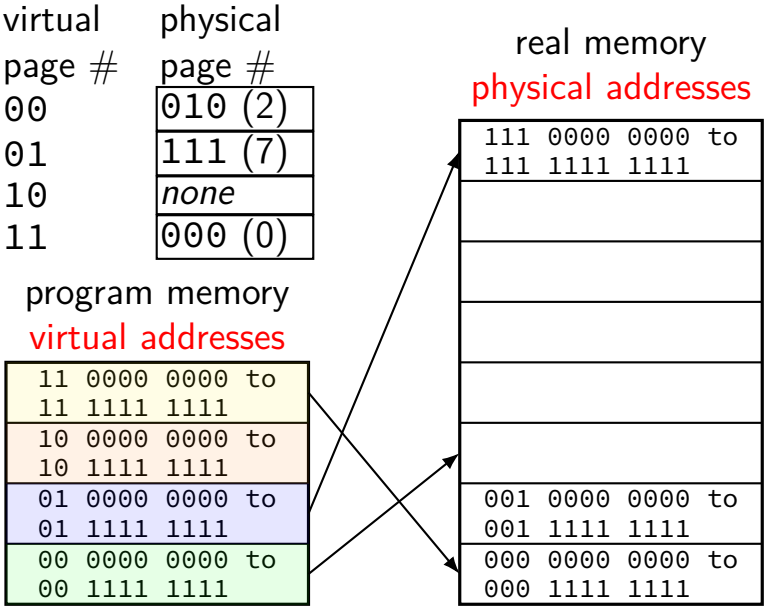
toy physical memory



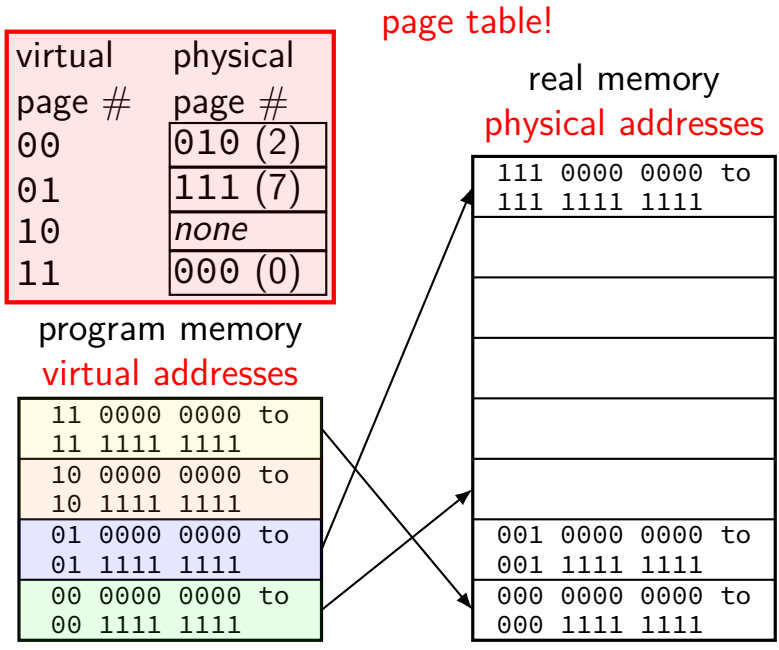
toy physical memory



toy physical memory



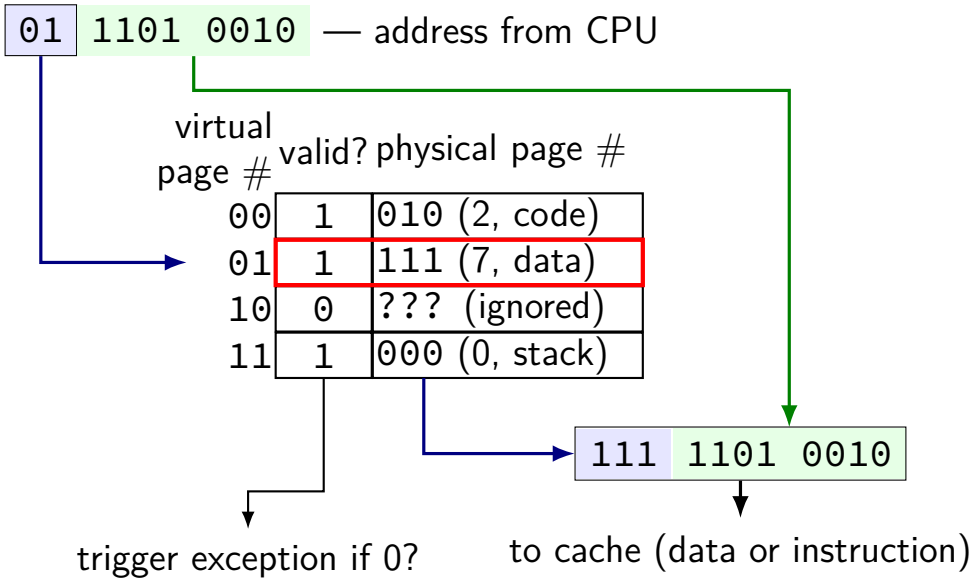
toy physical memory



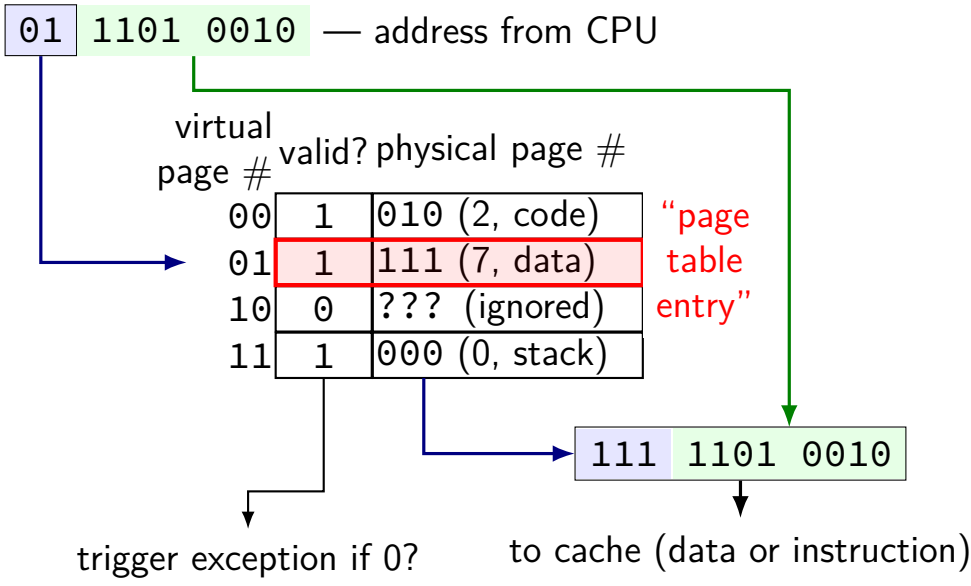
toy page table lookup

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

toy page table lookup

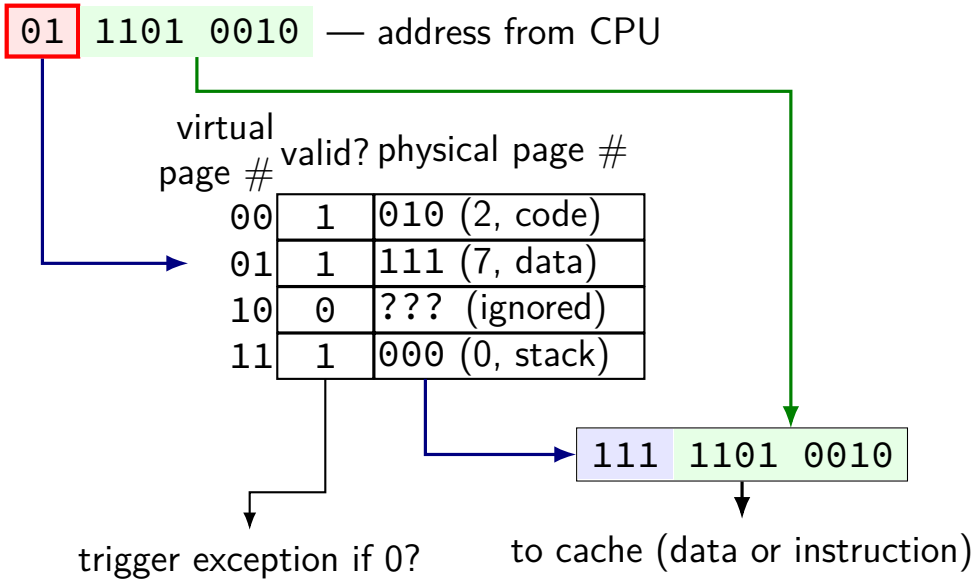


toy page table lookup

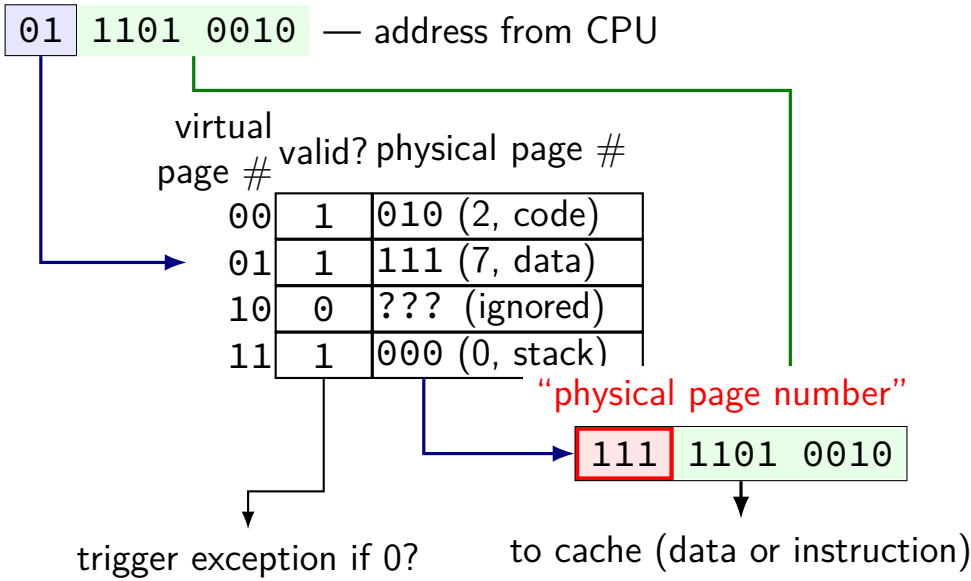


toy page table lookup

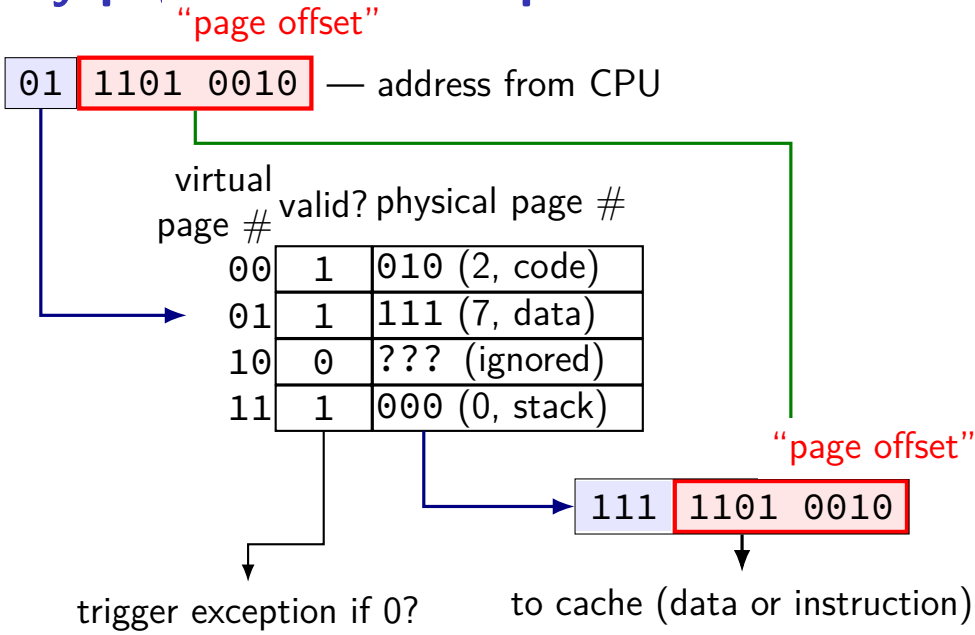
"virtual page number"



toy page table lookup

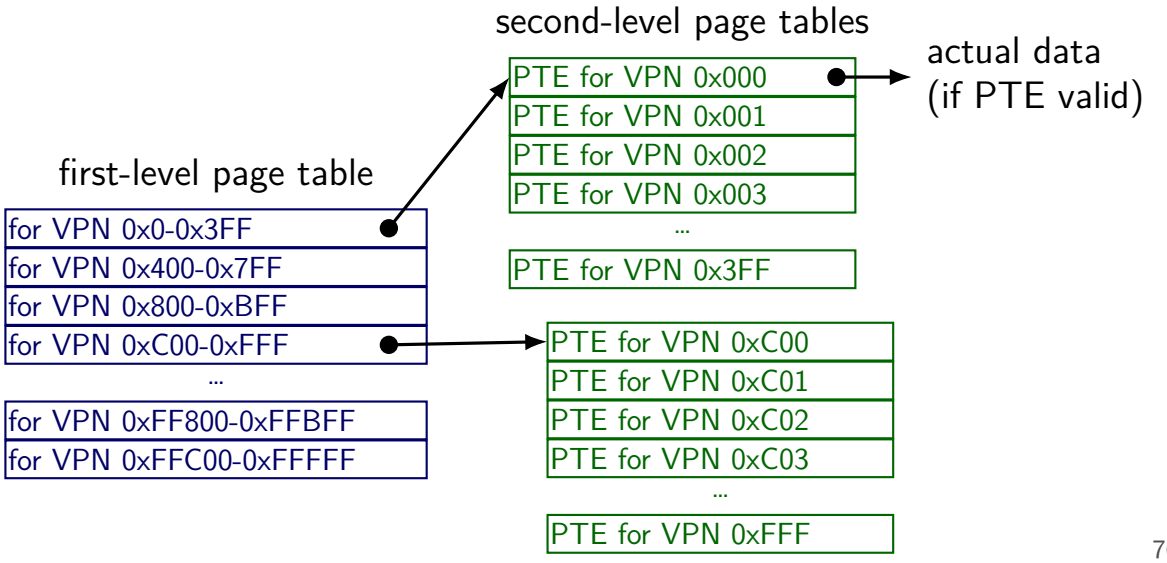


toy page table lookup



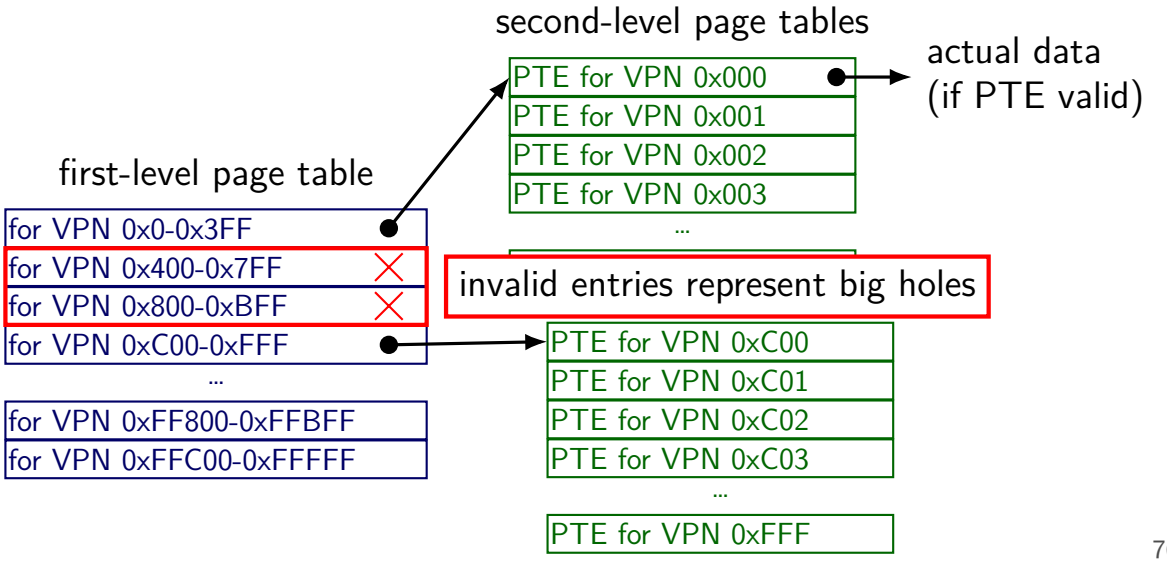
two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table



two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table



two-level page tables

two-level page table: 2^{20} pages total; 2^{10} entries per table

first-level pag

for VPN 0x0-0x3FF

for VPN 0x400-0x7F

for VPN 0x800-0xBF

for VPN 0xC00-0xFF

...

for VPN 0xFF800-0x

for VPN 0xFFC00-0xFFFFF

first-level page table

VPN range	valid	user?	write?	physical page # (of next page table)
0x0–0x3FF	1	1	1	0x22343
0x400–0x7FF	0	0	1	0x00000
0x800–0xBFF	0	0	0	0x00000
0xC00–0xFFF	1	1	0	0x33454
0x1000–0x13FF	1	1	0	0xFF043
...	...	...	...	...
0xFFC00–0xFFFFF	1	1	0	0xFF045

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

two-level page table: 2^{20} pages total; 2^{10} entries per table

first-level pag

for VPN 0x0-0x3FF
for VPN 0x400-0x7F
for VPN 0x800-0xBF
for VPN 0xC00-0xFF
...
for VPN 0xFF800-0x
for VPN 0xFFC00-0xFFFFF

first-level page table

VPN range	valid	user?	write?	physical page # (of next page table)
0x0–0x3FF	1	1	1	0x22343
0x400–0x7FF	0	0	1	0x00000
0x800–0xBFF	0	0	0	0x00000
0xC00–0xFFF	1	1	0	0x33454
0x1000–0x13FF	1	1	0	0xFF043
...	...	...	...	...
0xFFC00–0xFFFFF	1	1	0	0xFF045

PTE for VPN 0xC03
...
PTE for VPN 0xFFF

two-level page tables

two-level page table: 2^{20} pages total; 2^{10} entries per table

first-level pag

for VPN 0x0-0x3FF

for VPN 0x400-0x7F

for VPN 0x800-0xBF

for VPN 0xC00-0xFF

...

for VPN 0xFF800-0x

for VPN 0xFFC00-0xFFFFF

first-level page table

VPN range	valid	user?	write?	physical page # (of next page table)
0x0–0x3FF	1	1	1	0x22343
0x400–0x7FF	0	0	1	0x00000
0x800–0xBFF	0	0	0	0x00000
0xC00–0xFFF	1	1	0	0x33454
0x1000–0x13FF	1	1	0	0xFF043
...	...	...	...	...
0xFFC00–0xFFFFF	1	1	0	0xFF045

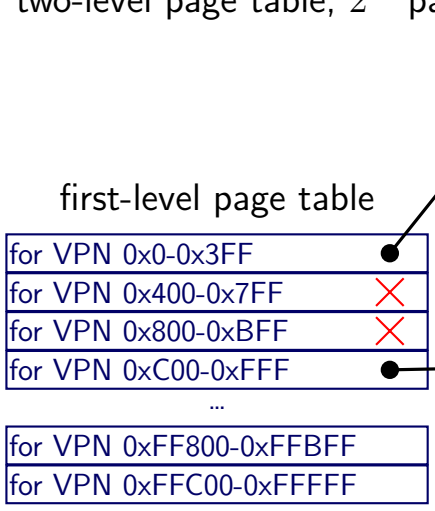
PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table



a second-level page table

VPN	valid	user?	write?	physical page # (of data)
0xC00	1	1	0	0x42443
0xC01	1	1	0	0x4A9DE
0xC02	1	1	0	0x5C001
0xC03	0	0	0	0x00000
0xC04	1	1	0	0x6C223
...	...	...	...	...
0xFFF	0	0	0	0x00000

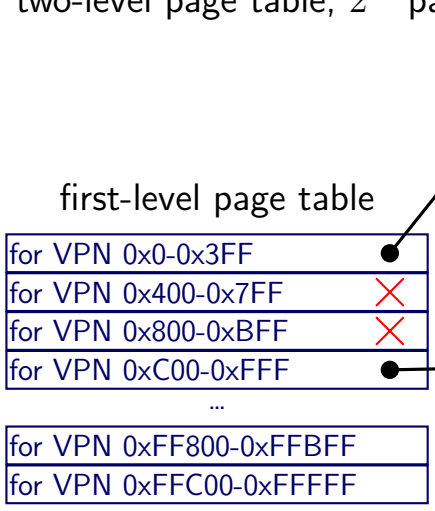
PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table



a second-level page table

VPN	valid	user?	write?	physical page # (of data)
0xC00	1	1	0	0x42443
0xC01	1	1	0	0x4A9DE
0xC02	1	1	0	0x5C001
0xC03	0	0	0	0x00000
0xC04	1	1	0	0x6C223
...	...	...	...	...
0xFFF	0	0	0	0x00000

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table