

last time

handling page/protection faults

- process accesses virtual page that's not present?

- process writes virtual page that's not writeable?

- etc.

- no crash needed: can fix page table + return from exception

allocate-on-demand

- don't allocate in advance — slower startup, maybe wasted space

- page fault for missing page? allocate it, then return

copy-on-write

- don't copy for `fork()` in advance

- mark read-only until overwritten

- actually copy (and update page table) when/if written

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 (zero-indexed) 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
(Linux extension: can omit fd with special value of flags)

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**
(Linux extension: can omit fd with special value of flags)

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
(Linux extension: can omit fd with special value of flags)

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 one of:

MAP_SHARED — changing memory changes file and vice-versa

- multiple processes mmap same file: get same physical pages

- read()/write() must use same physical pages

- changes to memory (if writable) must be sent to disk eventually

MAP_PRIVATE — make a copy of data in file

- changes to memory do not change file

- almost as if copied during mmap call

- but probably actually copied using copy-on-write

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 one of:

MAP_SHARED — changing memory changes file and vice-versa

multiple processes mmap same file: get **same physical pages**

read()/write() must use **same physical pages**

changes to memory (if writable) must be sent to disk eventually

MAP_PRIVATE — make a copy of data in file

changes to memory do not change file

almost as if copied during mmap call

but probably actually copied using copy-on-write

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 one of:

MAP_SHARED — changing memory changes file and vice-versa
multiple processes mmap same file: get same physical pages
read()/write() must use same physical pages
changes to memory (if writable) must be sent to disk eventually

MAP_PRIVATE — make a copy of data in file
changes to memory do not change file
almost as if copied during mmap call
but probably actually copied using copy-on-write

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 one of:

MAP_SHARED — changing memory changes file and vice-versa

- multiple processes mmap same file: get same physical pages

- read()/write() must use same physical pages

- changes to memory (if writable) must be sent to disk eventually

MAP_PRIVATE — make a copy of data in file

- changes to memory do not change file

- almost **as if copied during mmap call**

- but probably actually copied using copy-on-write

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 one of:

MAP_SHARED — changing memory changes file and vice-versa

- multiple processes mmap same file: get same physical pages

- read()/write() must use same physical pages

- changes to memory (if writable) must be sent to disk eventually

MAP_PRIVATE — make a copy of data in file

- changes to memory do not change file

- almost as if copied during mmap call

- but probably actually copied using copy-on-write

mmap options (3)

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

flags, choose one of:

MAP_SHARED — changing memory changes file and vice-versa

MAP_PRIVATE — make a copy of data in file

...or'd with optional additional flags

Linux: **MAP_ANONYMOUS** — ignore fd, allocate empty space

trick: Linux tracks process's memory as list of mmap's

... 'normal' memory heap, just special case w/o file

and more (see manual page)

mmap options (4)

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

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

not mandatory unless MAP_FIXED is used (which is rare)

can pass NULL — “choose for me”

address chosen will be returned

MAP_FAILED (constant) on failure

mmap exercise

suppose `hello.txt` initially contains “foo”:

```
int fd = open("hello.txt", O_RDWR);
char *p1 = mmap(NULL, 3 /* size */,
                PROT_READ|PROT_WRITE,
                MAP_SHARED, fd, 0);
char *p2 = mmap(NULL, 3, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
char *p3 = mmap(NULL, 3, PROT_READ, MAP_SHARED, fd, 0);
p2[2] = 'b';
p1[2] = 'x'; p1[1] = 'i';
char buffer[3];
read(fd, buffer, 3);
printf("%3s/%3s/%3s\n", buffer, p2, p3);
```

What is the output? (Assume no failures.)

- A. foo/fob/foo D. fix/fob/fix
- B. fix/fob/foo E. fix/fob/fob
- C. fix/fix/fix F. something else

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]
ffffffff600000-ffffffff601000 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-xp 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c764e000-7f60c784e000 r-p 0001b000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c784e000-7f60c7852000 r-xp 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7852000-7f60c7854000 r-p 00000000 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]
ffffffff600000-ffffffff601000 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 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 96659129 /lib/x86_64-linux-gnu/libc-2.19
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
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19
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]
ffffffff600000-ffffffff601000 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 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c764e000-7f60c784e000 -p 001b0000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c784e000-7f60c7852000 r-p starting at offset 0 of the file /bin/cat -2.19
7f60c7852000-7f60c7854000 rw-p -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]
ffffffff600000-ffffffff601000 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-xp 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c764e000-7f60c784e000 r-p 001be000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c784e000-7f60c7852000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7852000-7f60c7854000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7854000-7f60c7859000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7859000-7f60c7a39000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a39000-7f60c7a7a000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7a000-7f60c7a7b000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7b000-7f60c7a7c000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659129 /lib/x86_64-linux-gnu/ld-2.19
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]
ffffffff600000-ffffffff601000 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

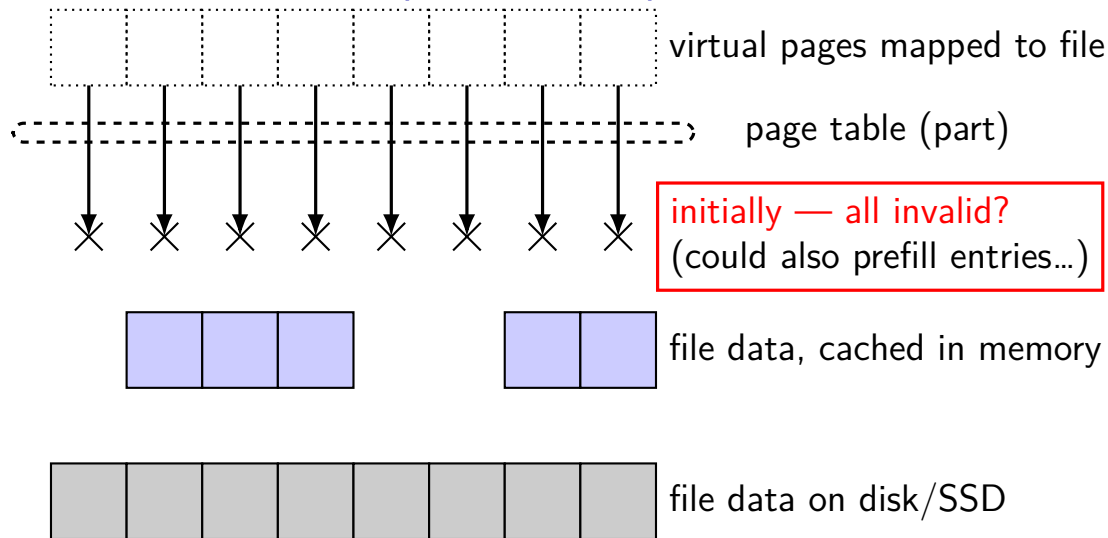
```
$ cat /proc/self/maps
```

```
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
7f60c7490000-7f60c764e000 r-xp 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c764e000-7f60c784e000 r-p 0001b000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c784e000-7f60c7852000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7852000-7f60c7854000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7854000-7f60c7859000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7859000-7f60c7a39000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a39000-7f60c7a7a000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7a000-7f60c7a7b000 r-p 00022000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19
7f60c7a7b000-7f60c7a7c000 r-p 00022000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19
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]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

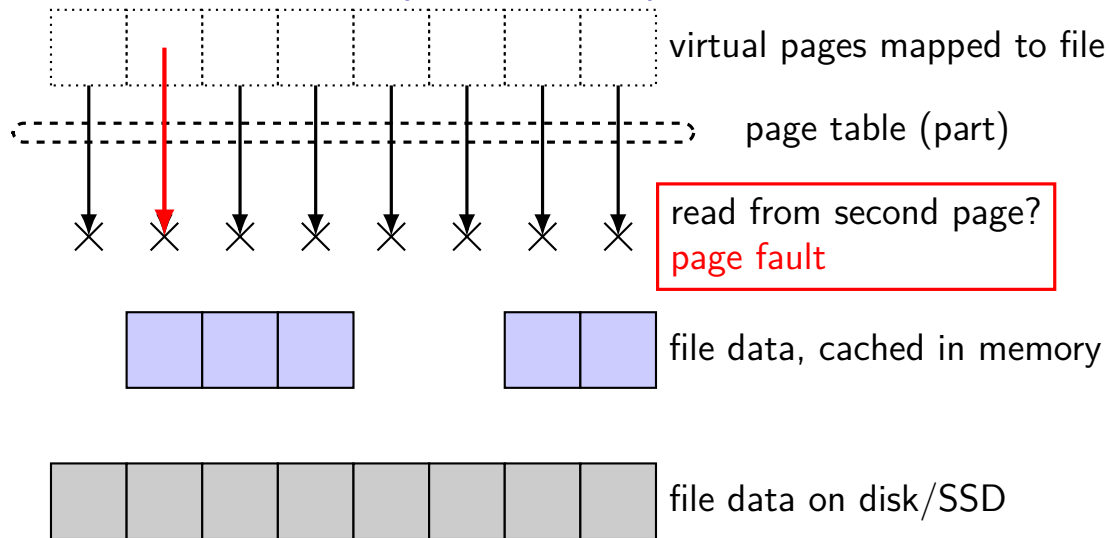
as if:

```
int fd = open("/bin/cat", O_RDONLY);
mmap(0x400000, 0xb000, PROT_READ | PROT_EXEC, MAP_PRIVATE, fd, 0x0);
```

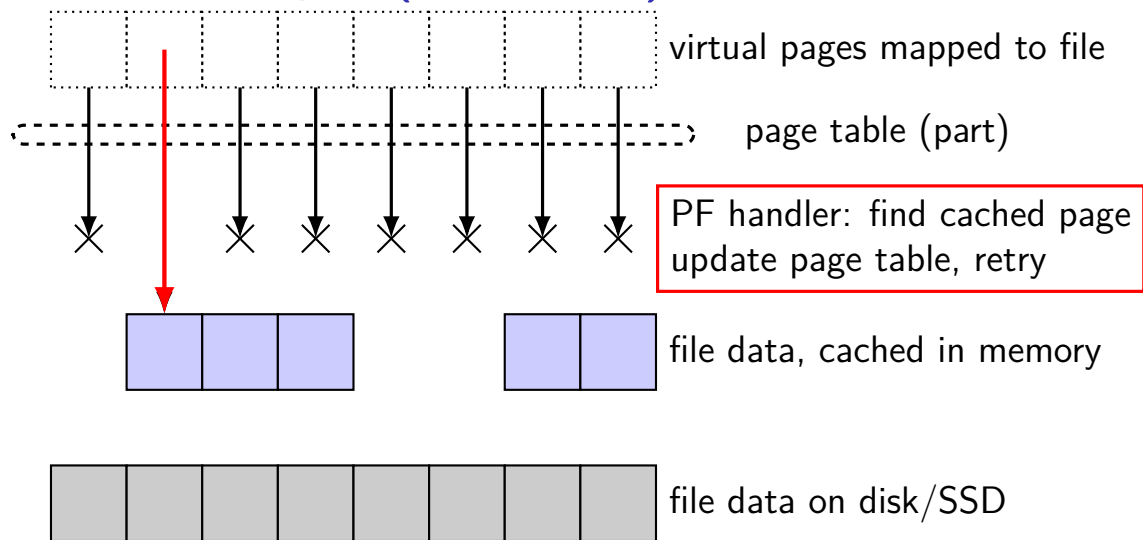
mapped pages (read-only)



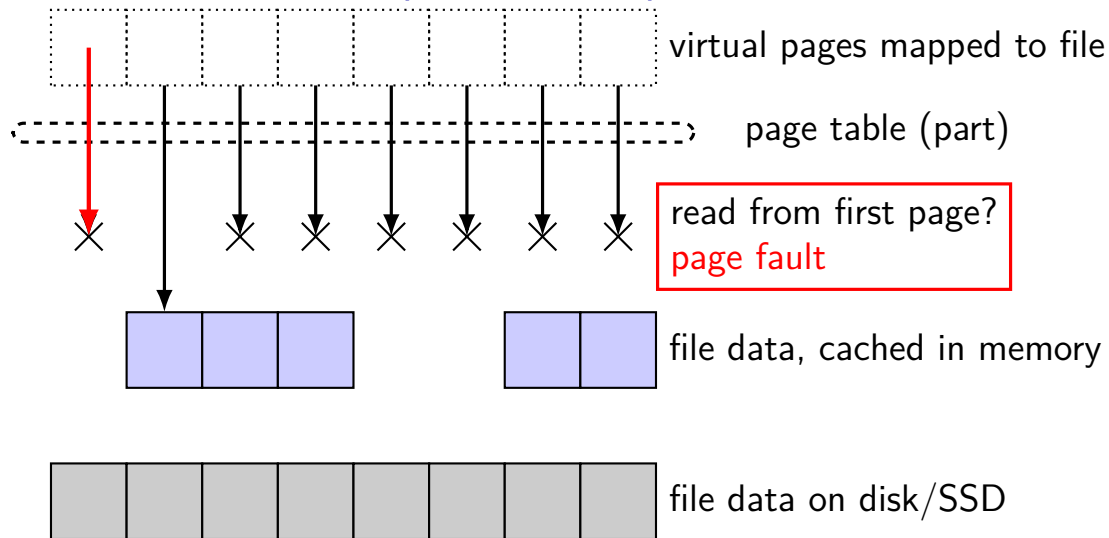
mapped pages (read-only)



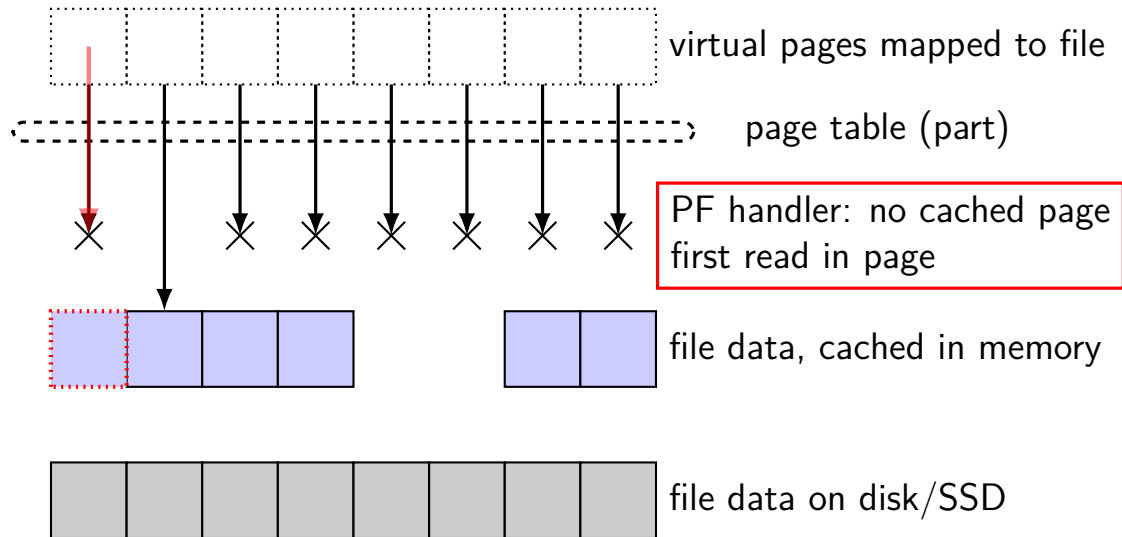
mapped pages (read-only)



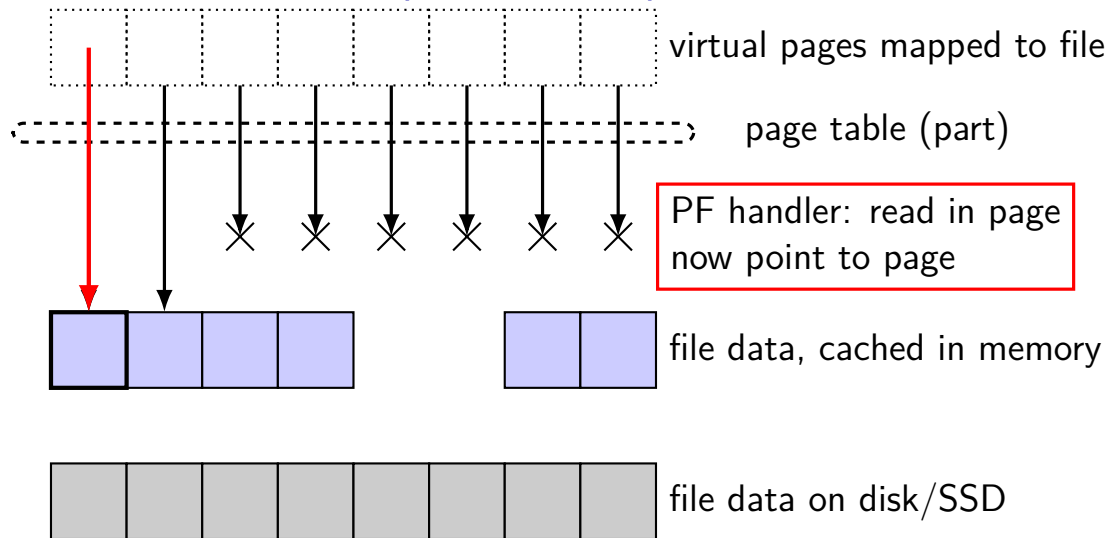
mapped pages (read-only)



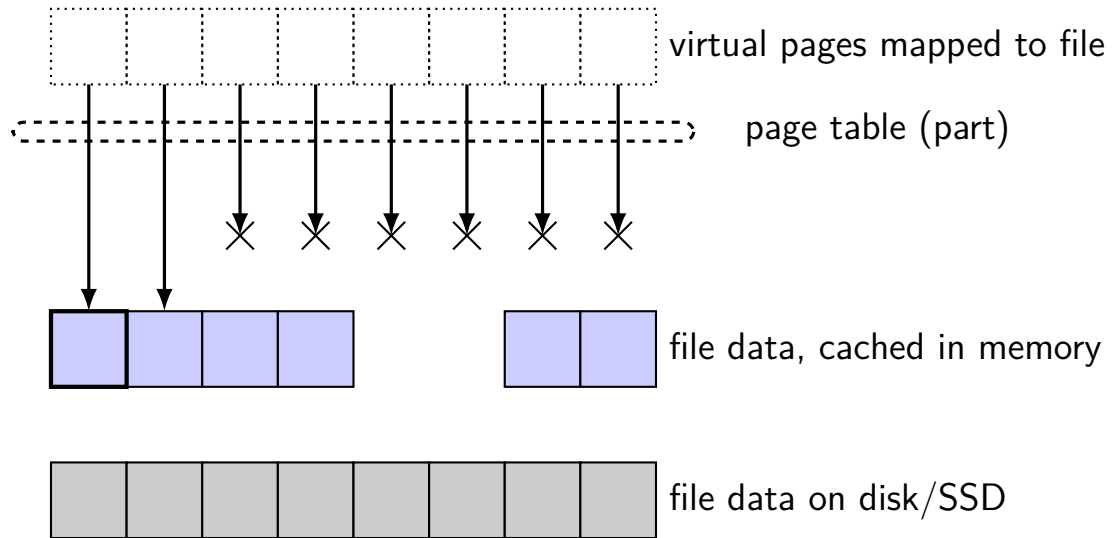
mapped pages (read-only)



mapped pages (read-only)



mapped pages (read-only)



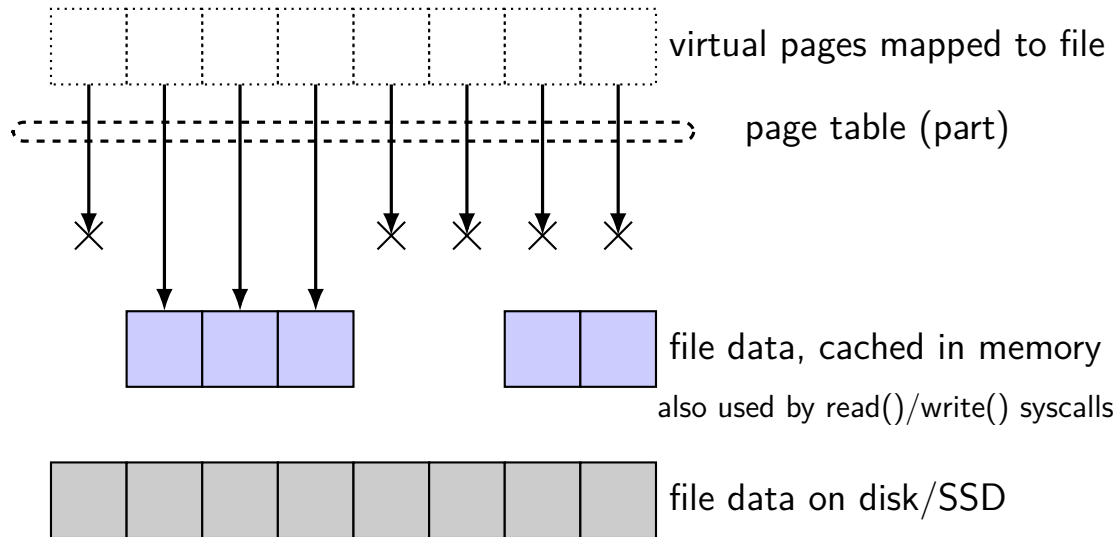
shared mmap

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

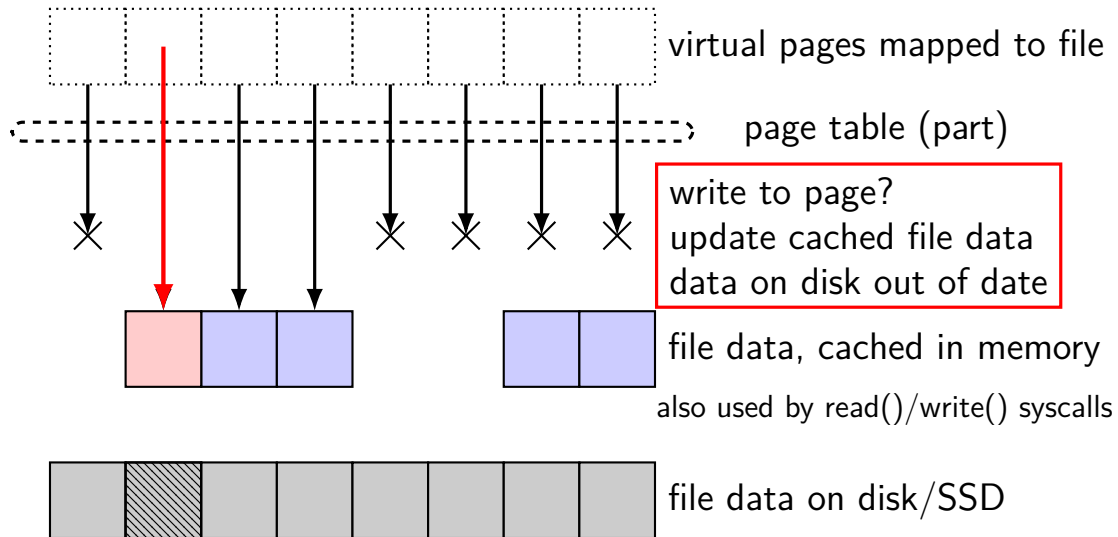
from /proc/PID/maps for this program:

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

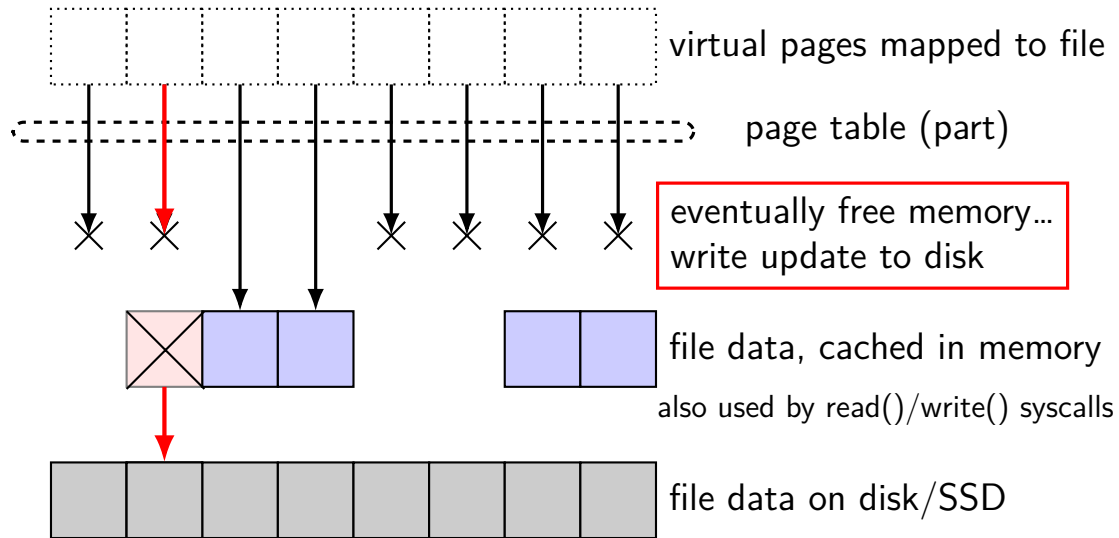
mapped pages (read/write, shared)



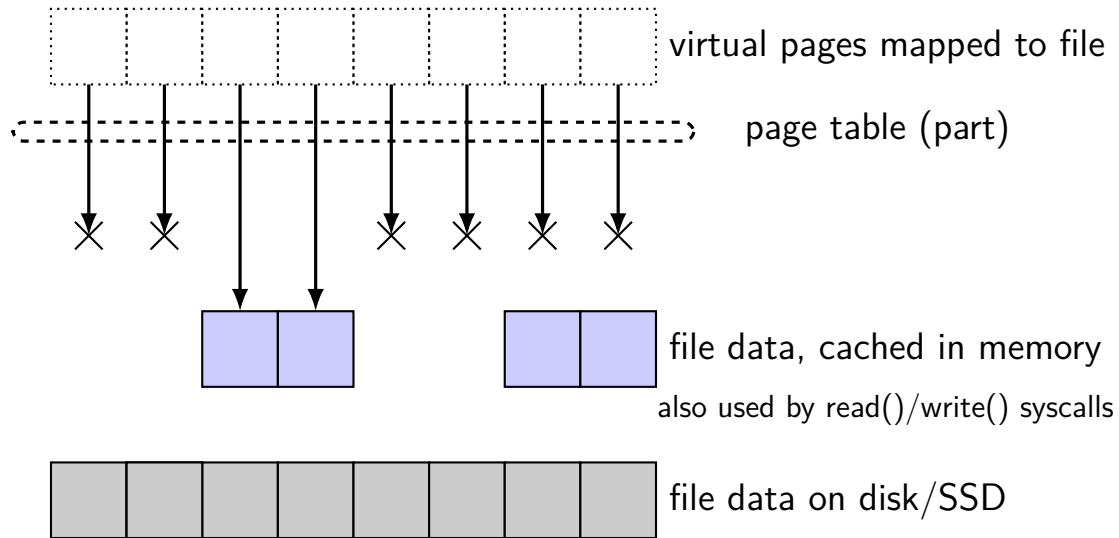
mapped pages (read/write, shared)



mapped pages (read/write, shared)



mapped pages (read/write, shared)



Linux maps

```
$ cat /proc/self/maps
```

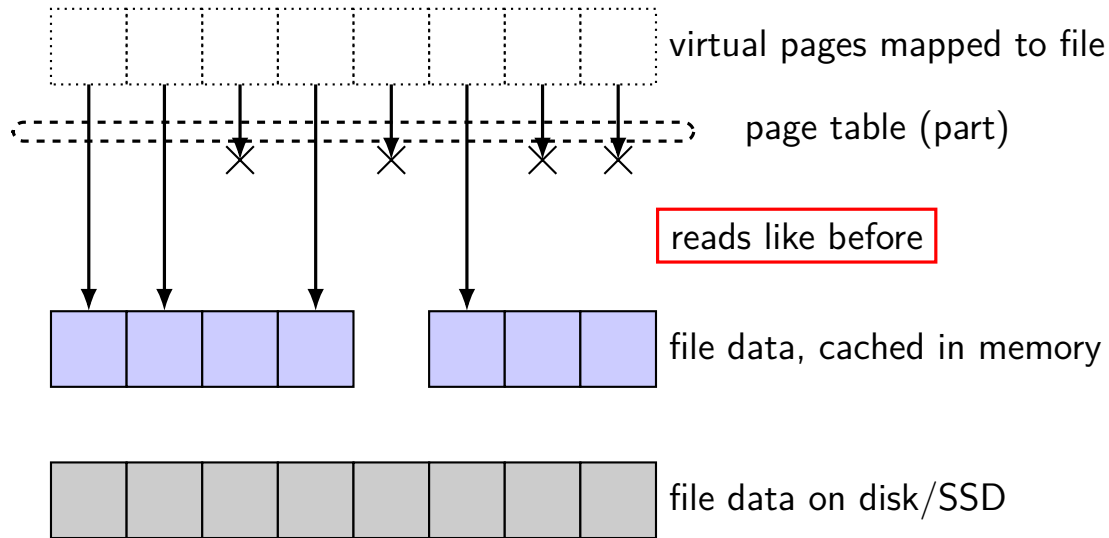
```
00400000-0040b000 r-xp 00000000 08:01 48328831 /bin/cat
0060a000-0060b000 r-p 0000a000 08:01 48328831 /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 48328831 /bin/cat
01974000-01995000 rw-p 00000000 00:00 0 [heap]
7f60c718b000-7f60c7490000 r-p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
7f60c7490000-7f60c764e000 r-xp 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c764e000-7f60c784e000 r-p 001be000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c784e000-7f60c7852000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7852000-7f60c7854000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7854000-7f60c7859000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7859000-7f60c7a39000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a39000-7f60c7a7a000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7a000-7f60c7a7b000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7b000-7f60c7a7c000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7c000-7f60c7a7d000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7d000-7f60c7a7e000 rw-p 00000000 00:00 0 [stack]
7ffc5d2b2000-7ffc5d2d3000 rw-p 00000000 00:00 0 [vvar]
7ffc5d3b0000-7ffc5d3b3000 r-p 00000000 00:00 0 [vdso]
7ffc5d3b3000-7ffc5d3b5000 r-xp 00000000 00:00 0 [vsyscall]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

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

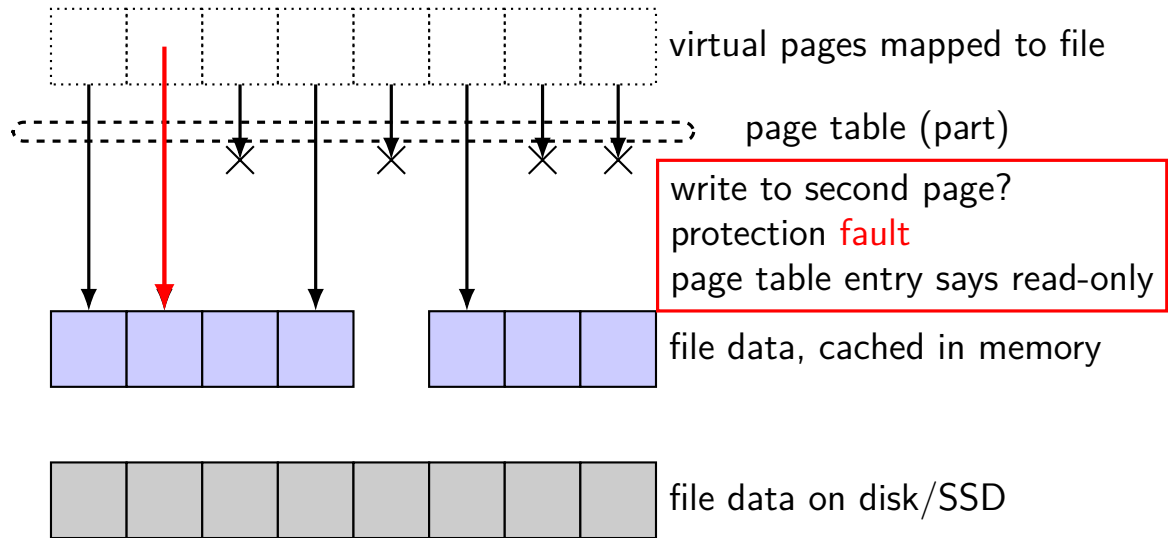
```
int fd = open("/bin/cat", O_RDONLY);
mmap(0x60b000, 0x1000, PROT_READ | PROT_WRITE,
      MAP_PRIVATE, fd, 0xb000);
```

(aside: probably used for global variables)

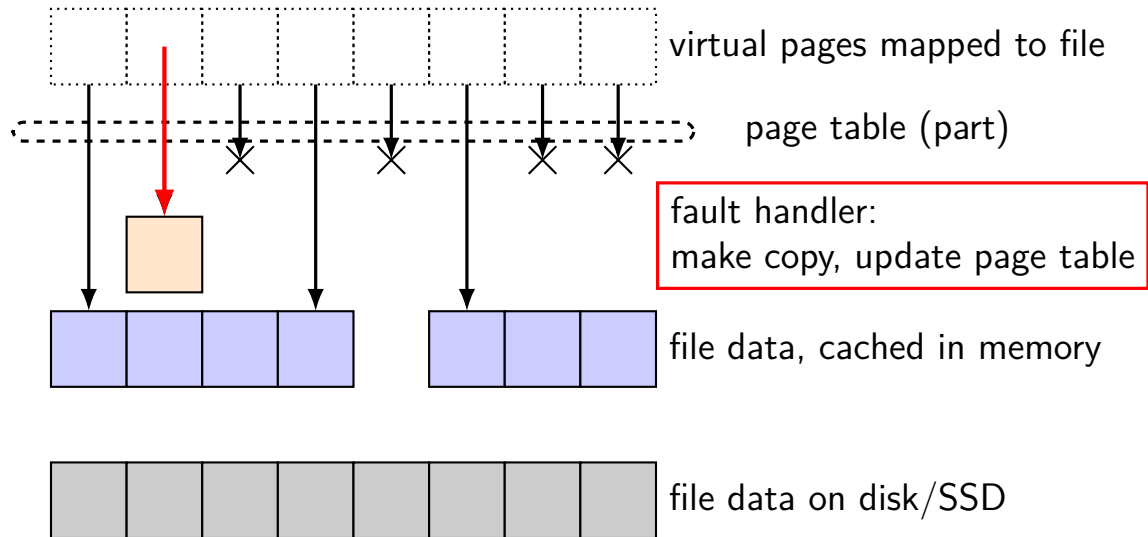
mapped pages (copy-on-write)



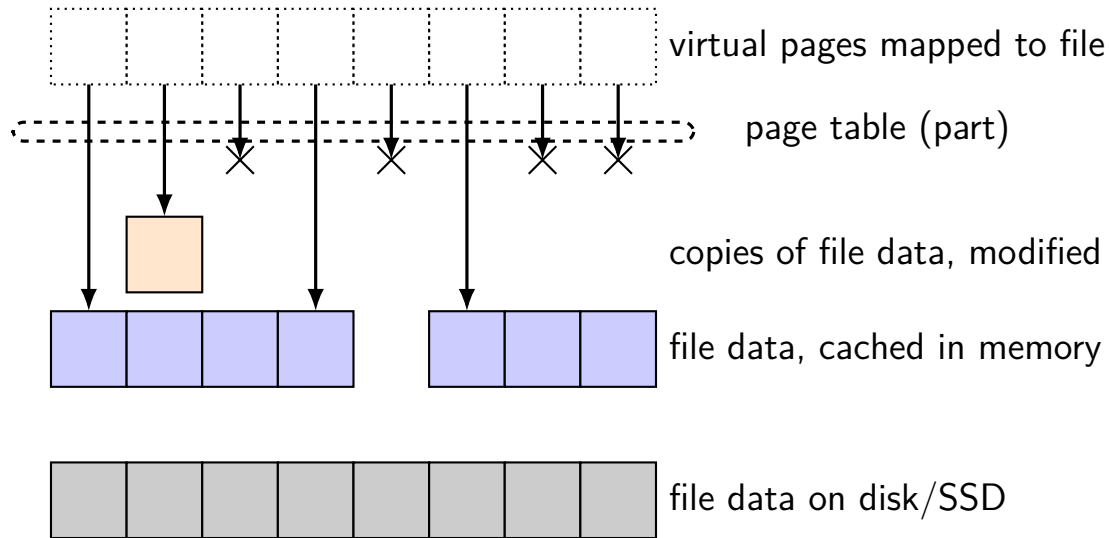
mapped pages (copy-on-write)



mapped pages (copy-on-write)



mapped pages (copy-on-write)



maps counting

4KB (0x1000 byte) pages

virtual 0x10000–0x1FFFF (64KB) → “foo.dat” bytes
0–0x0FFFF

map setup private (copy-on-write)

bytes 0–0x3FFF and 0x5000–0x6FFF cached in memory

program reads addresses 0x13800–0x15800

then, program overwrites addresses 0x14800–0x15100

assume: program page table filled in on demand only

smarter OS would probably proactively fill in multiple pages

maps counting

4KB (0x1000 byte) pages

virtual 0x10000–0x1FFFF (64KB) → “foo.dat” bytes
0–0x0FFFF

map setup private (copy-on-write)

bytes 0–0x3FFF and 0x5000–0x6FFF cached in memory

program reads addresses 0x13800–0x15800

then, program overwrites addresses 0x14800–0x15100

assume: program page table filled in on demand only

smarter OS would probably proactively fill in multiple pages

question: how much exceptions (page/protection faults)?

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 -2.19
7f60c7852000-7f60c7854000 rw-p -2.19
7f60c7854000-7f60c7859000 rw-p
7f60c7859000-7f60c787c000 r-xp 2.19.s
7f60c7a39000-7f60c7a3b000 rw-p
7f60c7a7a000-7f60c7a7b000 rw-p
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]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

heap — no corresponding file
allocated using `sbrk()`
but can get same effect with `mmap()` call

Linux maps

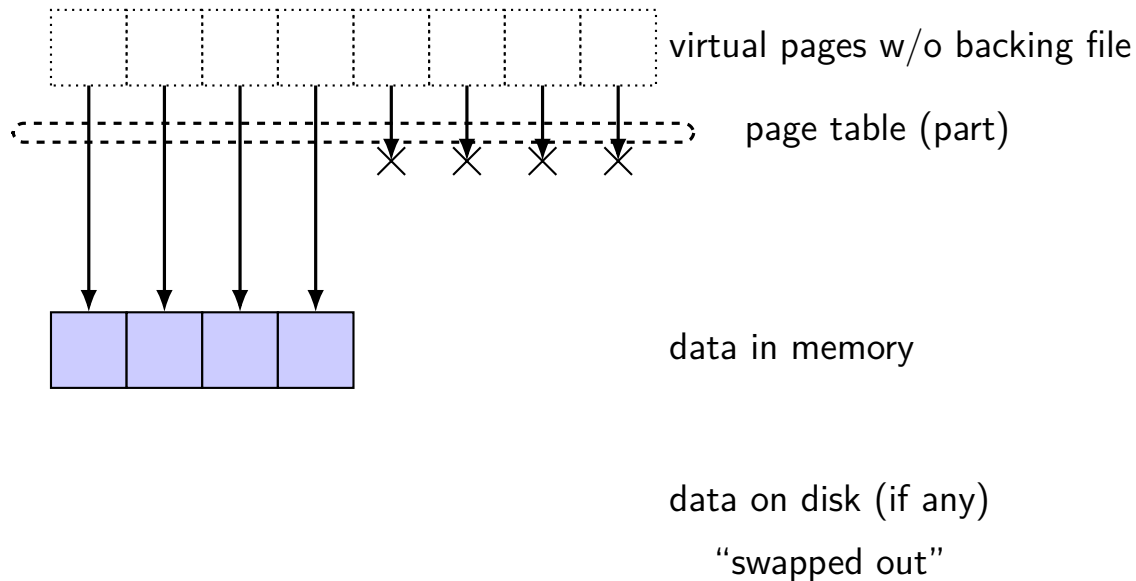
```
$ cat /proc/self/maps
```

```

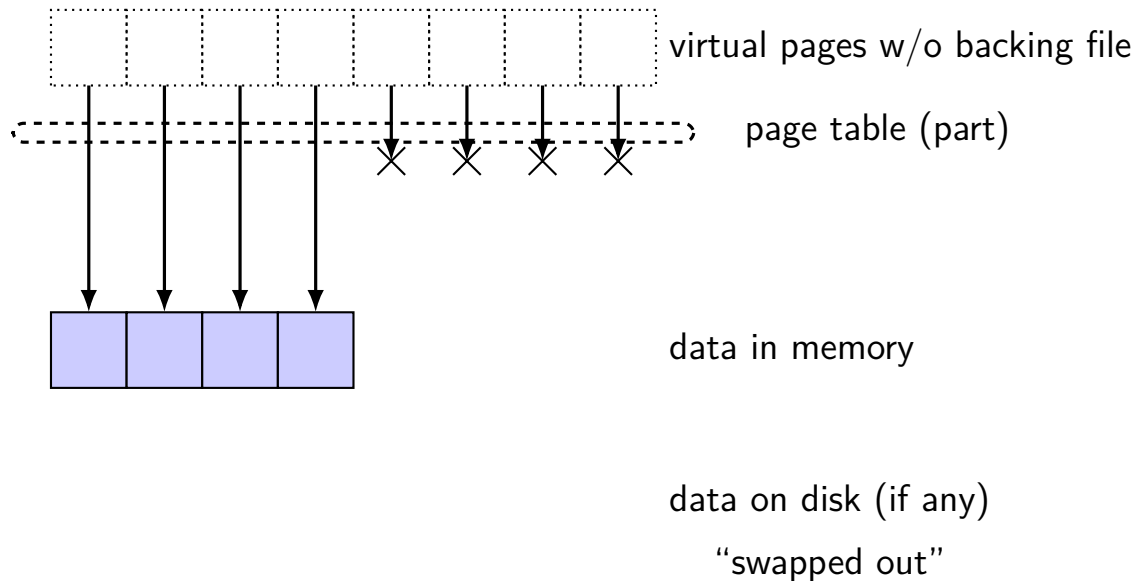
00400000-0040b000 r-xp 00000000 08:01 48328831 /bin/cat
0060a000-0060b000 r-p 0000a000 08:01 48328831 /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 48328831 /bin/cat
01974000-01995000 rw-p 00000000 00:00 0 [heap]
7f60c718b000-7f60c7490000 r-p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
7f60c7490000-7f60c764e000 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
as if:
mmap(..., 0x5000, PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_ANONYMOUS /* = no file */, ...);
7ffc5d3b3000-7ffc5d3b5000 r-xp 00000000 00:00 0 [vdso]
ffffffffffff600000-ffffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

```

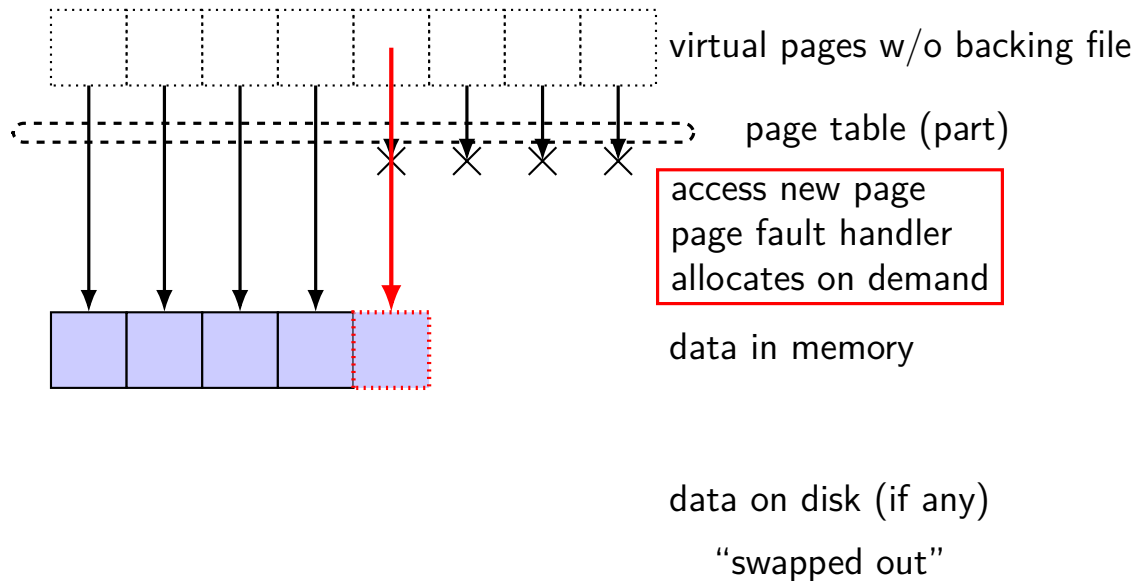
mapped pages (no backing file)



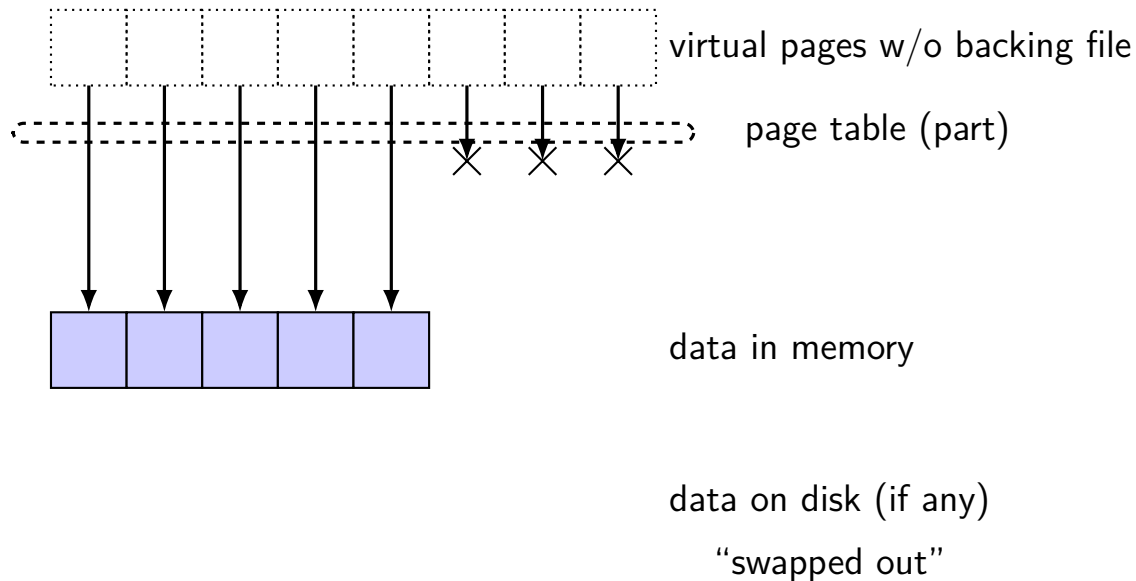
mapped pages (no backing file)



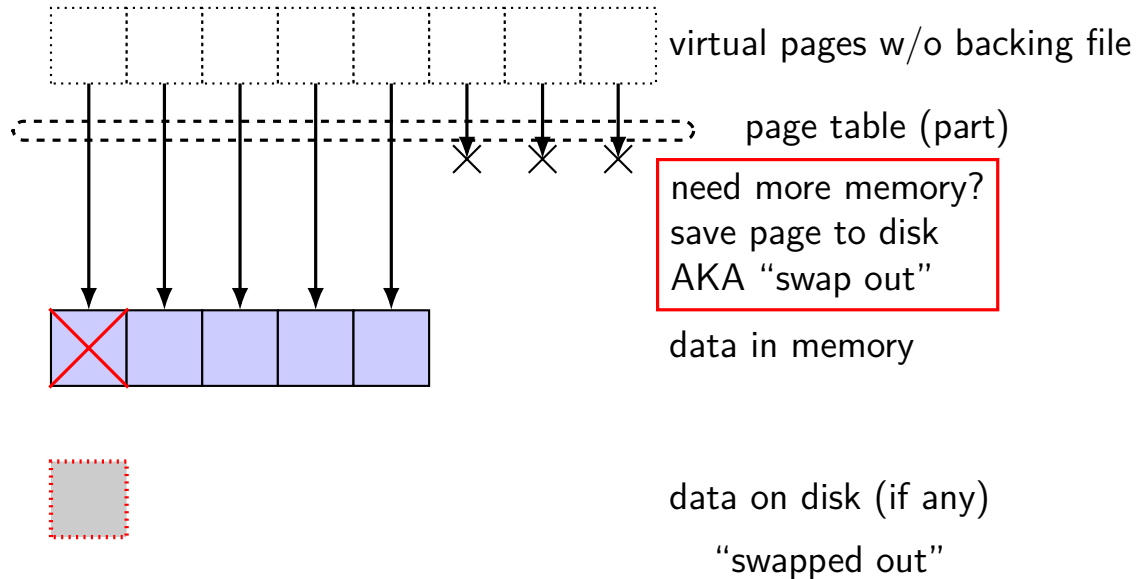
mapped pages (no backing file)



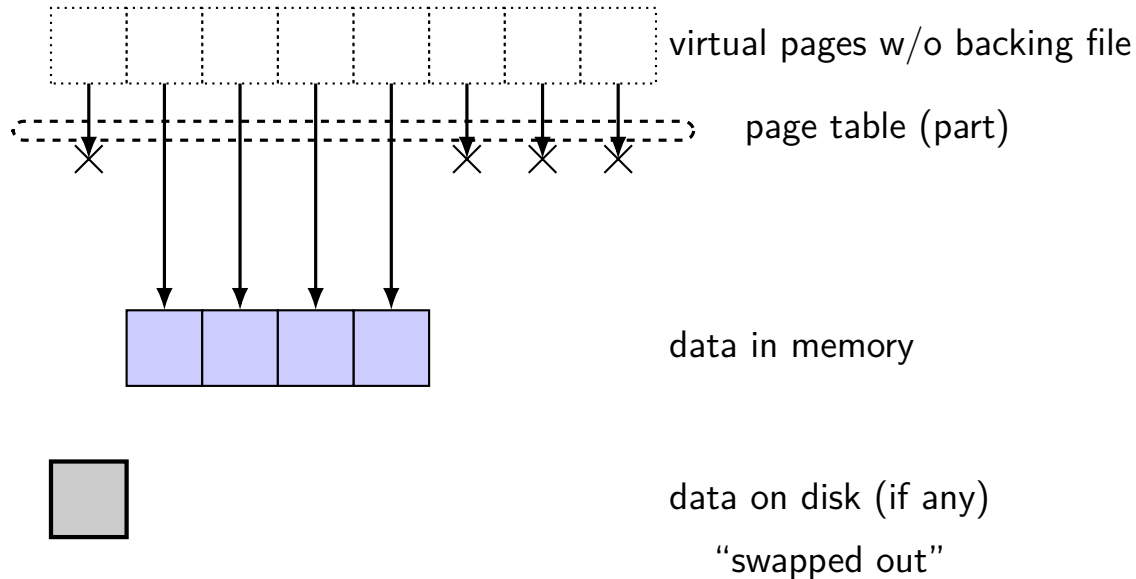
mapped pages (no backing file)



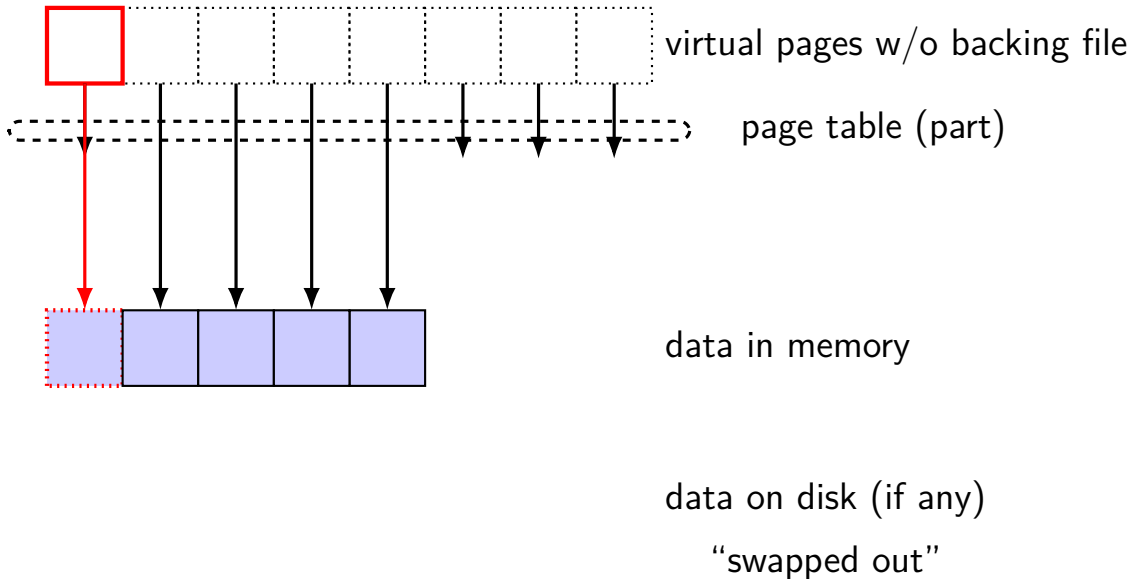
mapped pages (no backing file)



mapped pages (no backing file)



mapped pages (no backing file)

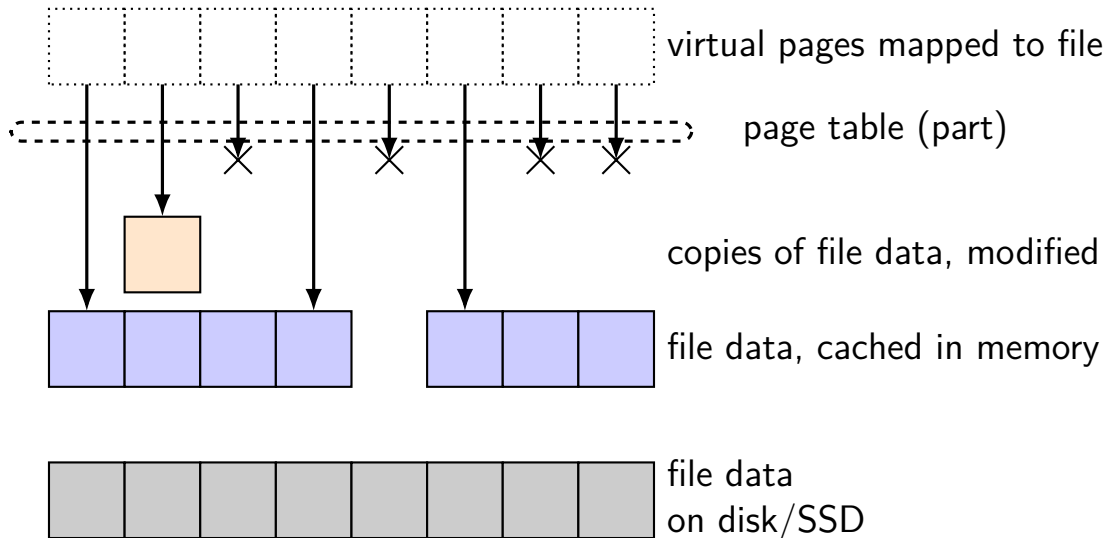


Linux maps

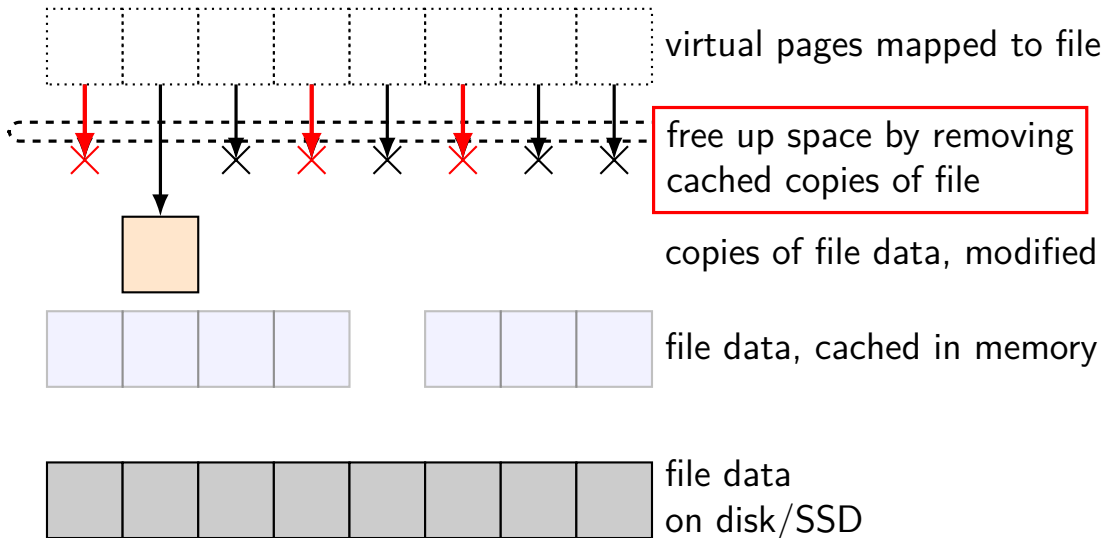
```
$ cat /proc/self/maps
```

```
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]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

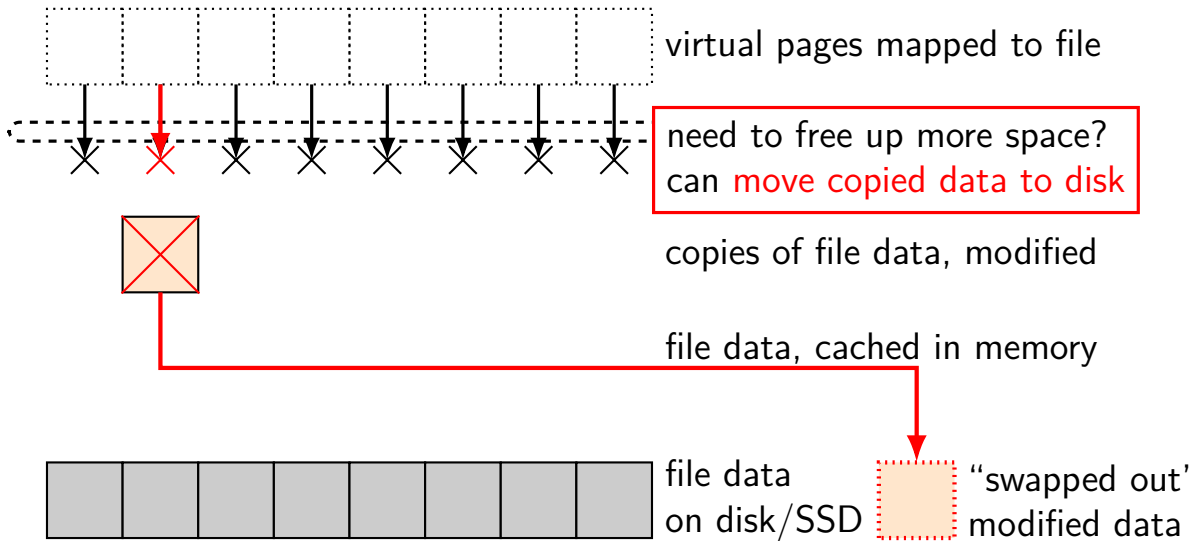
swapping with copy-on-write



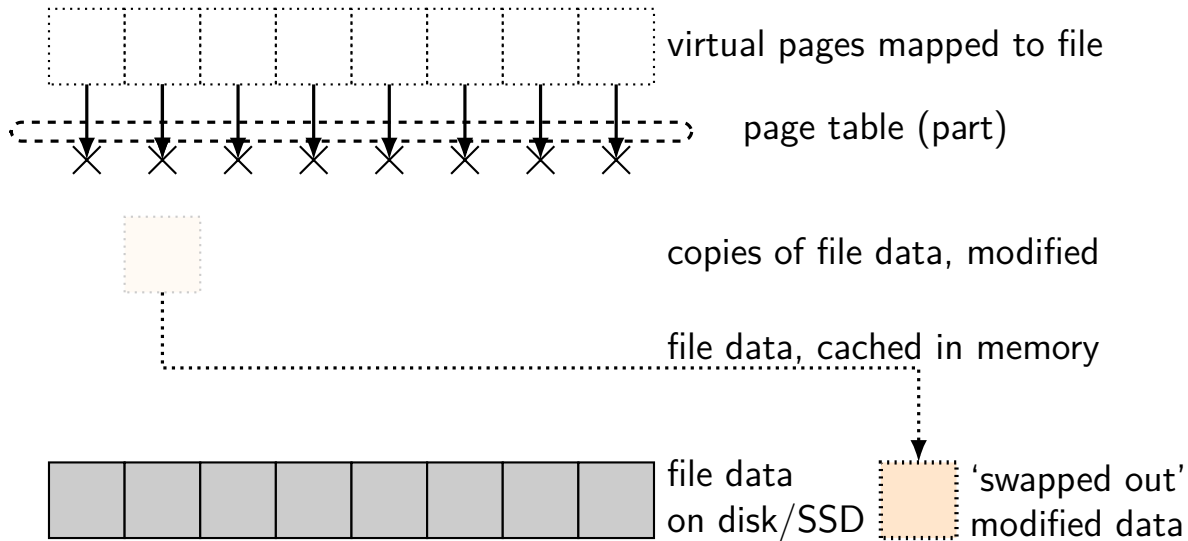
swapping with copy-on-write



swapping with copy-on-write



swapping with copy-on-write



the page cache

memory is a cache for disk

files and 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 'temporarily' kept in faster storage

possibly being used by one or more processes?

possibly part of a file on disk being read/written?

possibly both

goal: manage this cache intelligently

the page cache

memory is a cache for disk

files and 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 'temporarily' kept in faster storage

possibly being used by one or more processes?

possibly part of a file on disk being read/written?

possibly both

goal: manage this cache intelligently

the page cache

memory is a cache for disk

files and 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 'temporarily' kept in faster storage
possibly being used by one or more processes?
possibly part of a file on disk being read/written?
possibly both

goal: manage this cache intelligently

the page cache

memory is a cache for disk

files and 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 'temporarily' kept in faster storage

possibly being used by one or more processes?

possibly part of a file on disk being read/written?

possibly both

goal: manage this cache intelligently

page cache components [text]

mapping: virtual address or file+offset \rightarrow physical page

- handle cache hits

find backing location based on virtual address/file+offset

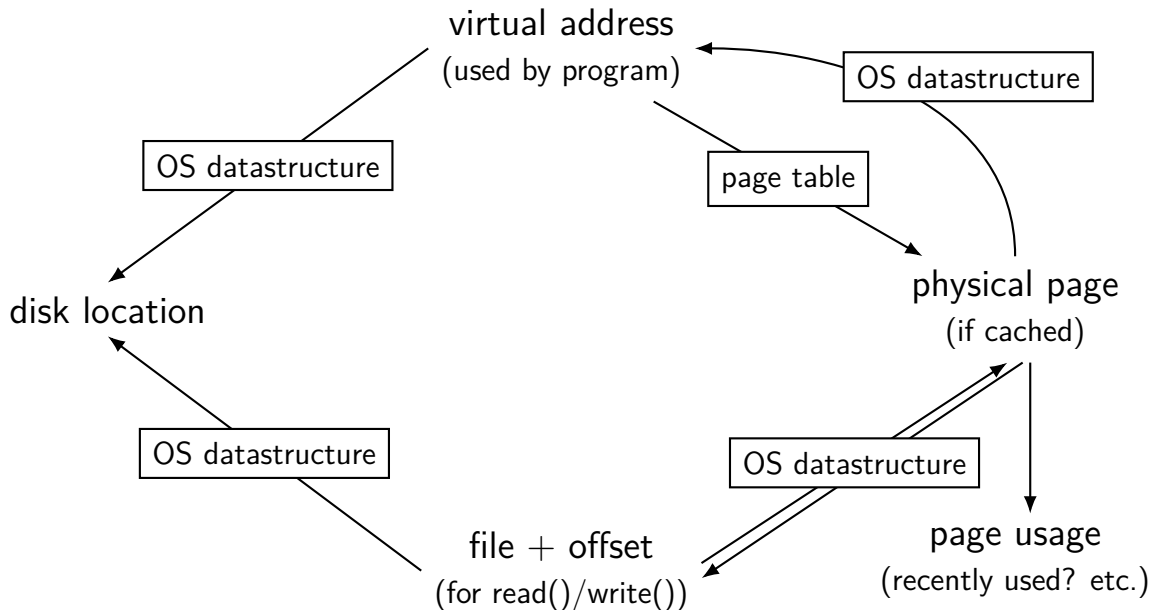
- handle cache misses

track information about each physical page

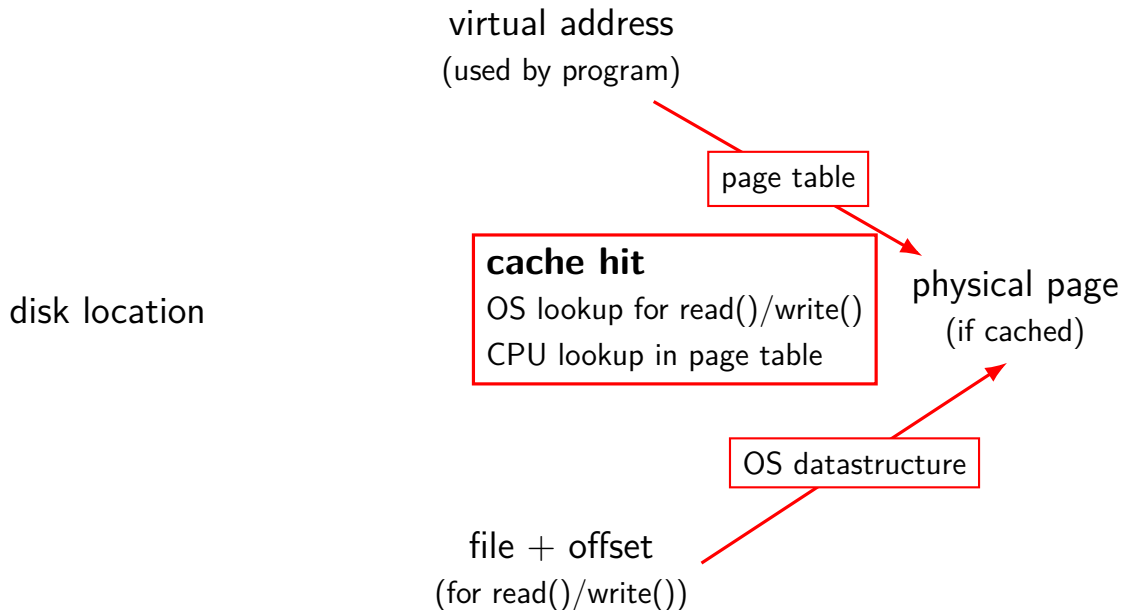
- handle page allocation

- handle cache eviction

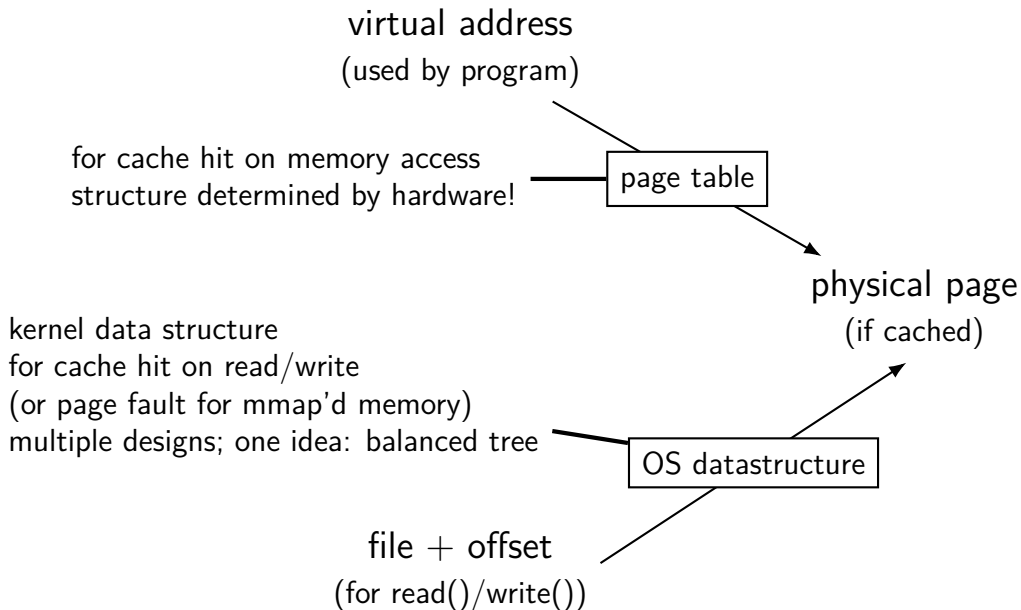
page cache components



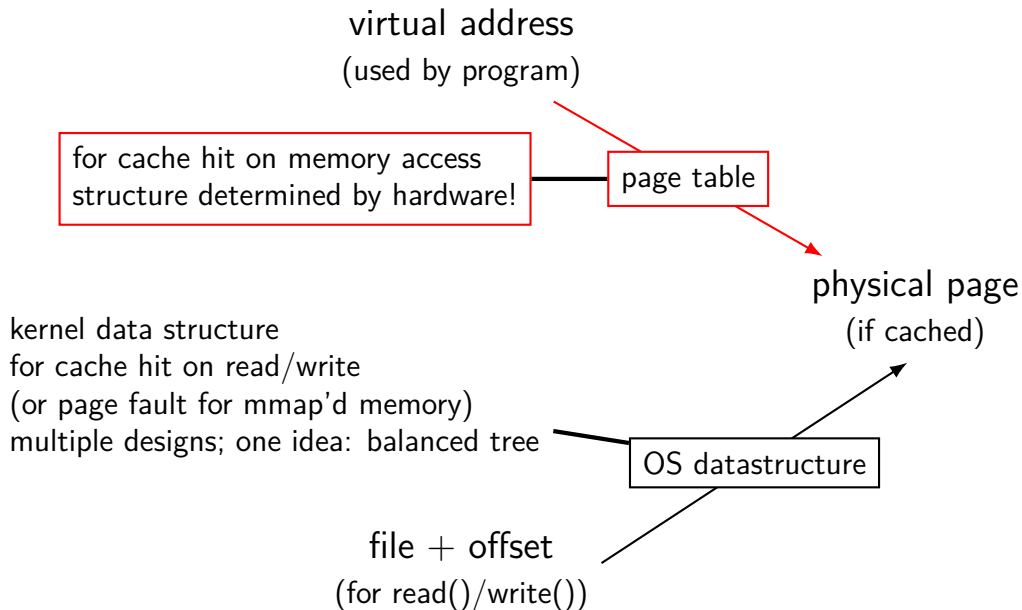
page cache components



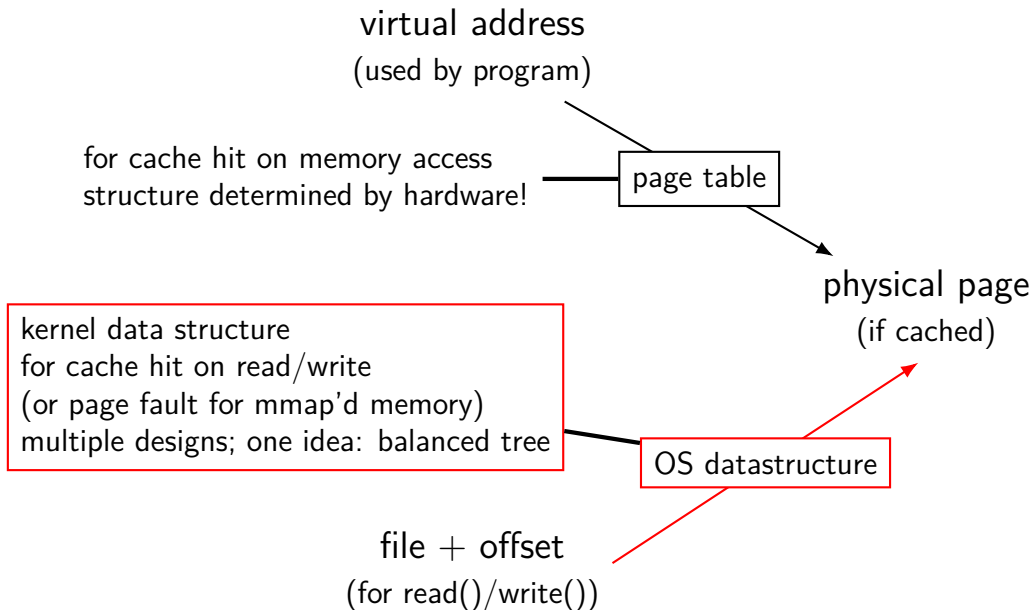
virtual addr/file offset to physical page



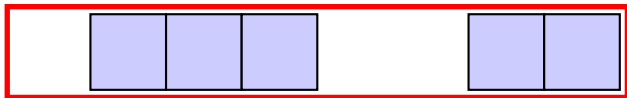
virtual addr/file offset to physical page



virtual addr/file offset to physical page



mapped pages (read/write, shared)



file data, cached in memory



file data on disk/SSD

page allocation = “page replacement”

assumed common case: every physical page in use

because we're caching a ton of stuff
(actually true in practice? sometimes...)

then: allocating new physical page →
replacing what a current physical page is used for

so page allocation = *page replacement*

page replacement

step 1: *evict* a page to free a physical page

case 1: there's an unused page, just use that (easy)

case 2: need to remove whatever what's in that page (more work)

step 2: load new, more important in its place

needs some way of knowing location of data

page replacement

step 1: *evict* a page to free a physical page

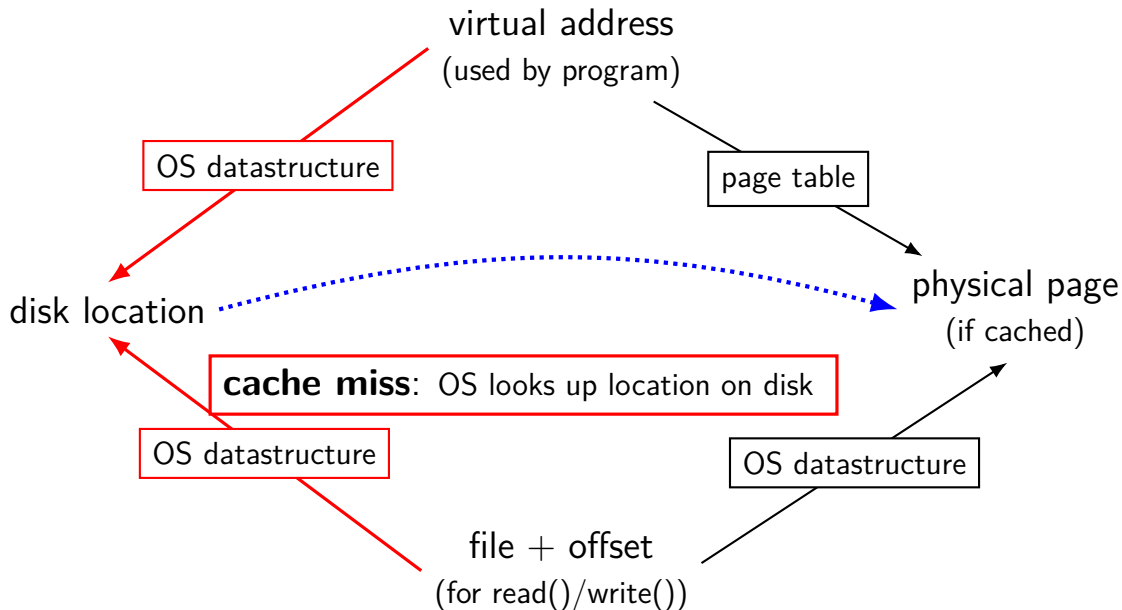
case 1: there's an unused page, just use that (easy)

case 2: need to remove whatever what's in that page (more work)

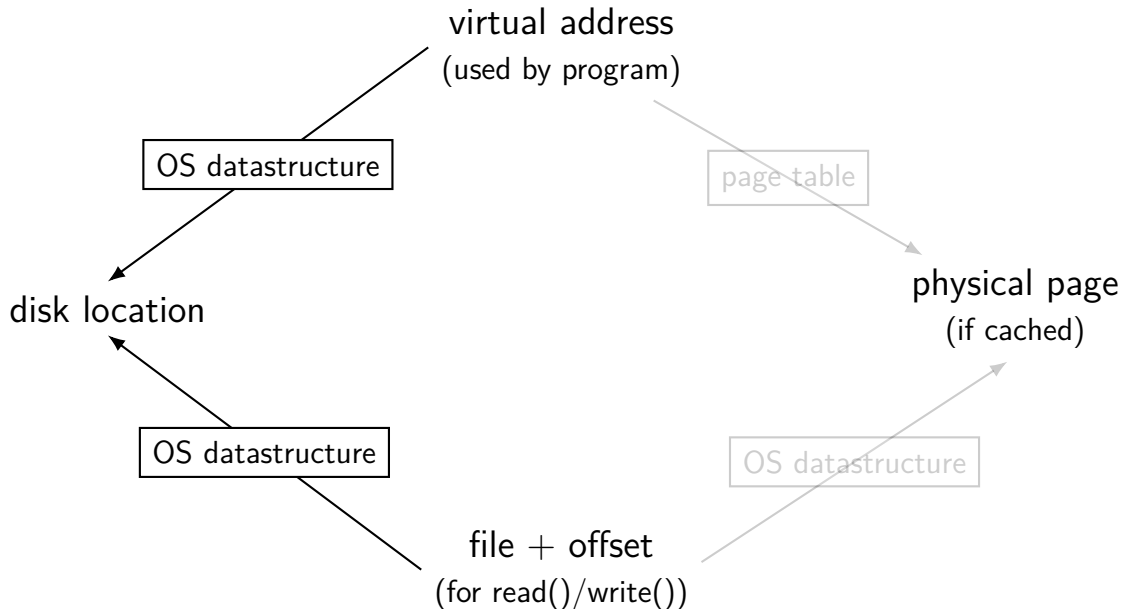
step 2: load new, more important in its place

needs some way of knowing location of data

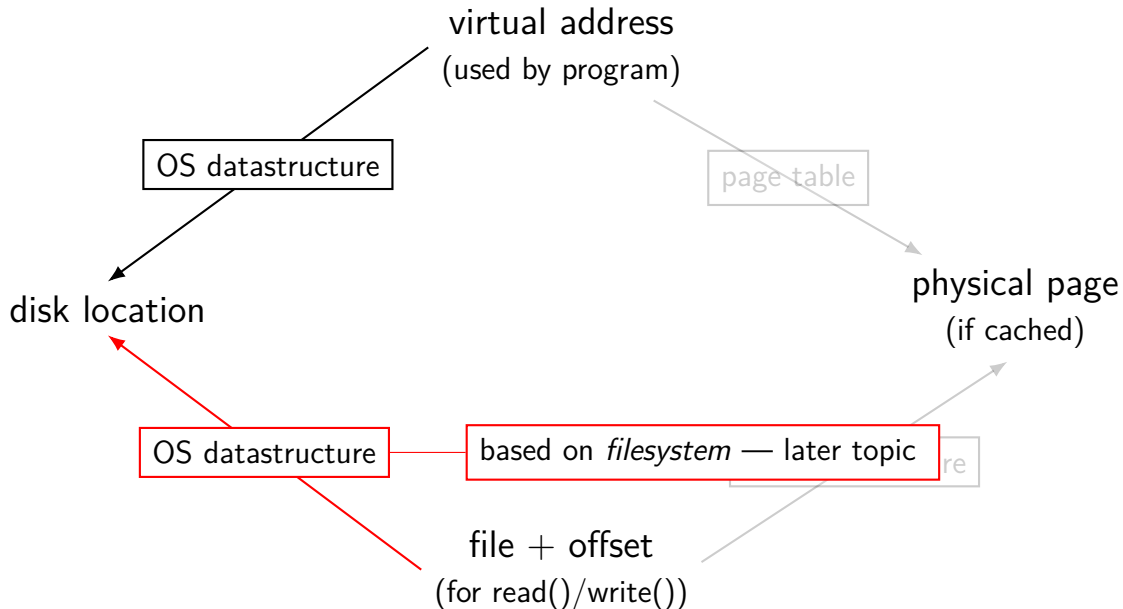
page cache components



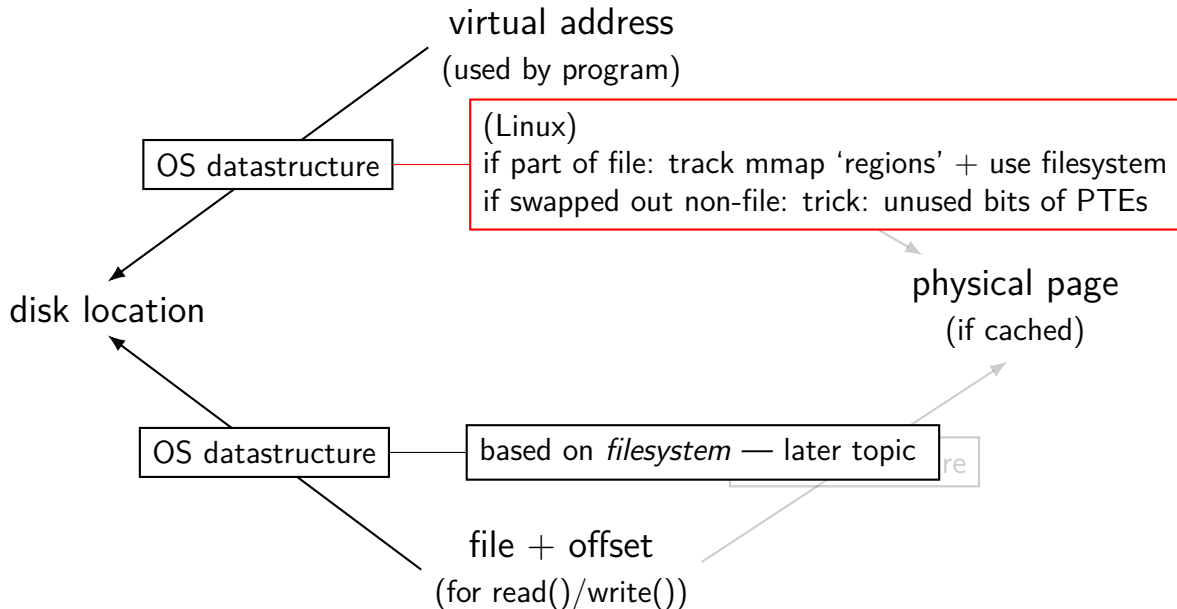
virtual address/file offset \rightarrow location on disk



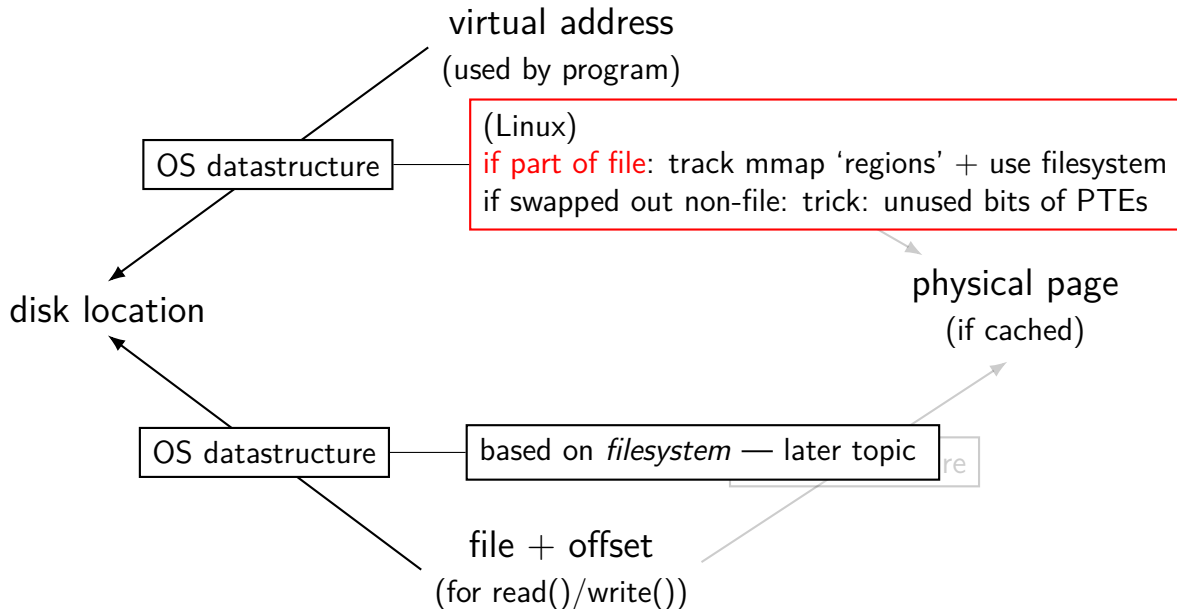
virtual address/file offset \rightarrow location on disk



virtual address/file offset \rightarrow location on disk



virtual address/file offset \rightarrow location on disk



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
```

PCB contains 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 (e.g. heap after fork) ...

page replacement

step 1: *evict* a page to free a physical page

case 1: there's an unused page, just use that (easy)

case 2: need to remove whatever what's in that page (more work)

step 2: load new, more important in its place

needs some way of knowing location of data

evicting a page

remove victim page from page table, etc.

- every page table it is referenced by

- every list of file pages

- ...

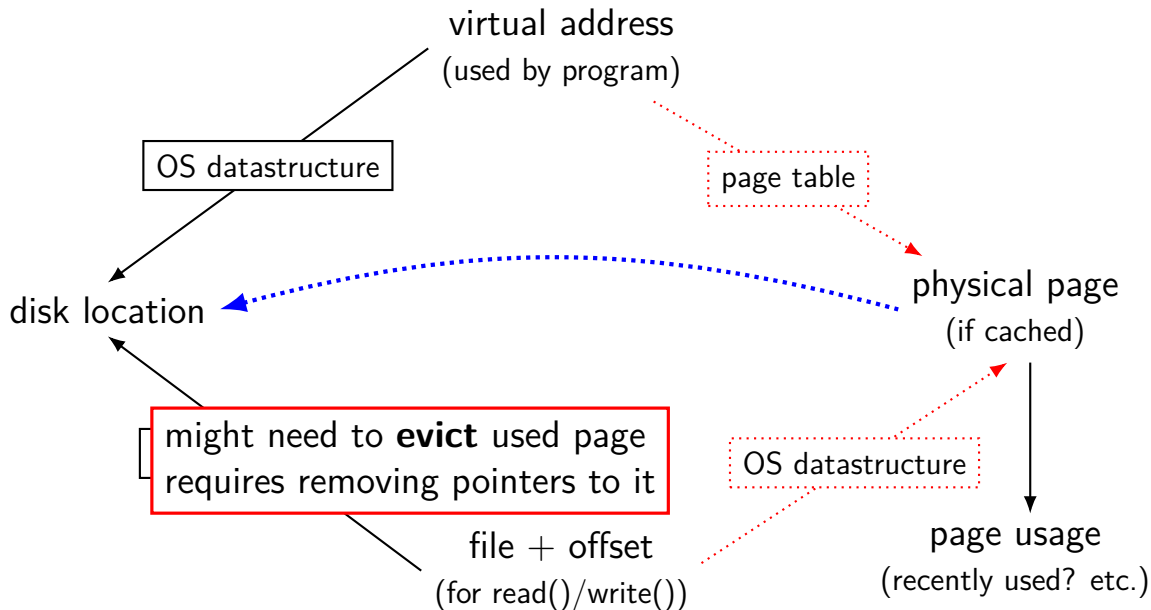
if needed, save victim page to disk

going to require:

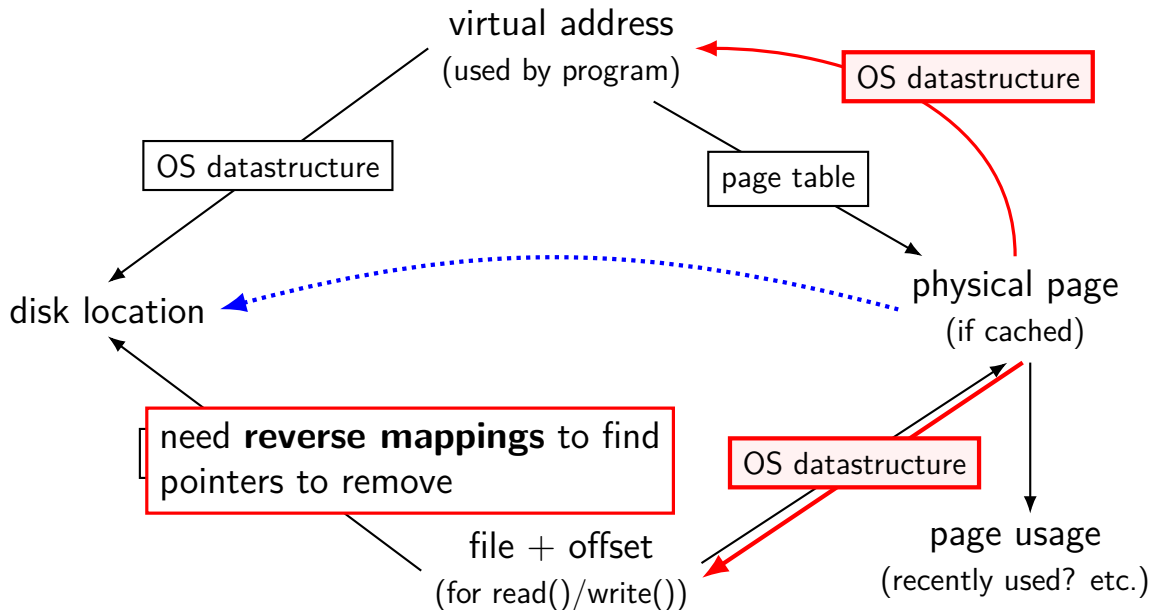
way to find page tables, etc. using page

way to detect whether it needs to be saved to disk

page cache components



page cache components



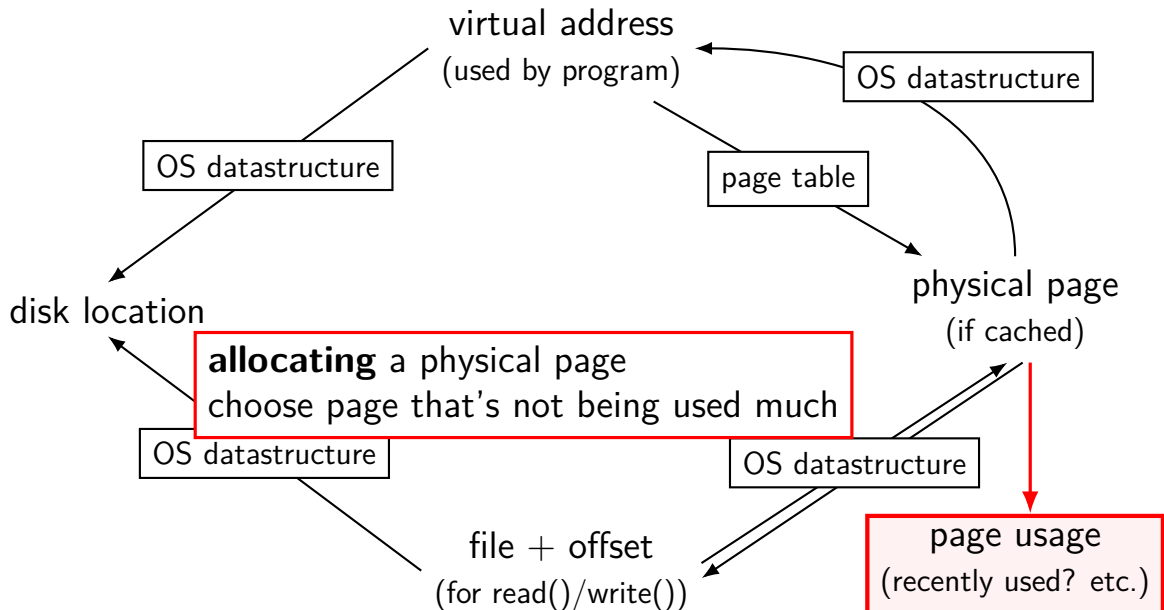
tracking physical pages: finding mappings

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

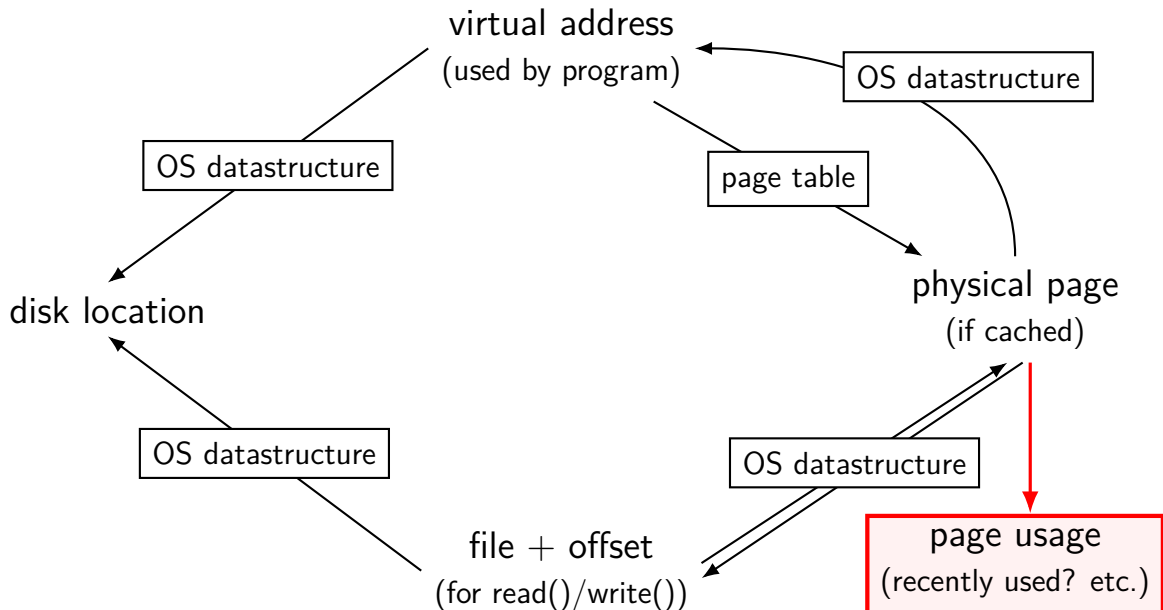
need to track where every page is used!

common solution: structure for every physical page with info about every cached file/page table using page

page cache components



page cache components



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

can save modified pages to disk in the background

we will get back to this later

for now — only access/evict on demand

optimizing for hit-rate

assuming:

- we only bring in pages on demand (no reading in advance)
- we only care about maximizing cache hits

best possible page replacement algorithm: Belady's MIN

replace the page in memory accessed **furthest in the future**
(never accessed again = infinitely far in the future)

optimizing for hit-rate

assuming:

- we only bring in pages on demand (no reading in advance)
- we only care about maximizing cache hits

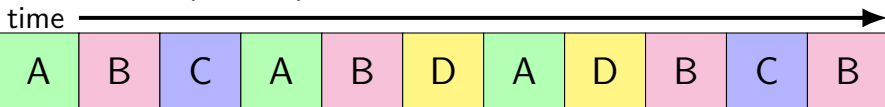
best possible page replacement algorithm: Belady's MIN

replace the page in memory accessed **furthest in the future**
(never accessed again = infinitely far in the future)

impossible to implement in practice, but...

Belady's MIN

referenced (virtual) pages:



phys. page#	A	B	C	A	B	D	A	D	B	C	B
1	A										
2		B									
3			C								

Belady's MIN

referenced (virtual) pages:

	time →										
phys. page#	A	B	C	A	B	D	A	D	B	C	B
1	A										
2		B									
3			C				D				

A next accessed in 1 time unit

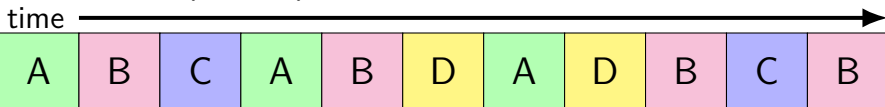
B next accessed in 3 time units

C next accessed in 4 time units

choose to replace C

Belady's MIN

referenced (virtual) pages:



phys. page#	1	2	3	4	5	6	7	8	9	10
1	A									
2		B								
3			C			D				

Belady's MIN

referenced (virtual) pages:

phys. page#	time →										
	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

A next accessed in ∞ time units

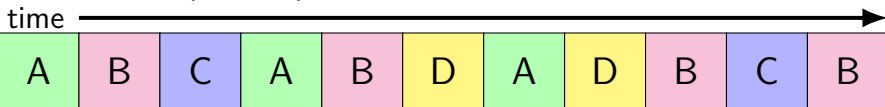
B next accessed in 1 time units

D next accessed in ∞ time units

choose to replace A or D (equally good)

Belady's MIN

referenced (virtual) pages:



phys. page#	1	2	3	4	5	6	7	8	9	10	11
1	A										C
2		B									
3			C			D					

Belady's MIN exercise

referenced (virtual) pages:

phys. page#	time →										
	A	B	C	D	B	B	A	C	A	D	C
1	A										
2		B									
3			C								

exercise: What does this access to D replace? (A, B, or C?)

practically optimizing for hit-rate

recall?: locality assumption

temporal locality: things accessed now will be accessed again soon

(for now: not concerned about spatial locality)

more possible policies: least recently used or least frequently used

practically optimizing for hit-rate

recall?: locality assumption

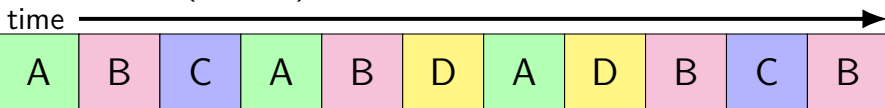
temporal locality: things accessed now will be accessed again soon

(for now: not concerned about spatial locality)

more possible policies: **least recently used** or least frequently used

least recently used (the good case)

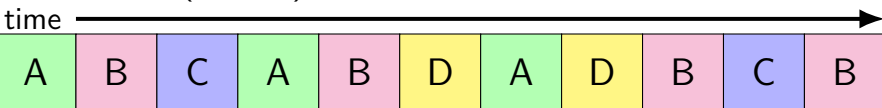
referenced (virtual) pages:



phys. page#	A	B	C	A	B	D	A	D	B	C	B
1	A										
2		B									
3			C								

least recently used (the good case)

referenced (virtual) pages:



phys. page#	1	2	3	4	5	6	7
1	A						
2		B					
3			C				D

A *last* accessed 2 time units ago

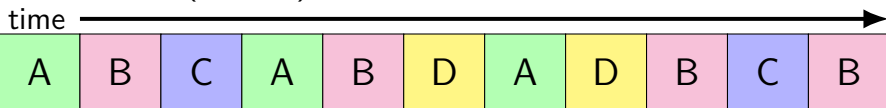
B *last* accessed 1 time unit ago

C *last* accessed 3 time units ago

choose to replace C

least recently used (the good case)

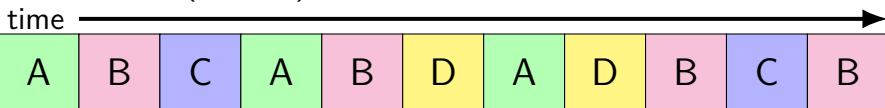
referenced (virtual) pages:



phys. page#	1	2	3							
1	A									
2		B								
3			C			D				

least recently used (the good case)

referenced (virtual) pages:



phys. page#	1	2	3								
1	A									C	
2		B									
3			C			D					

A *last* accessed in 3 time units ago

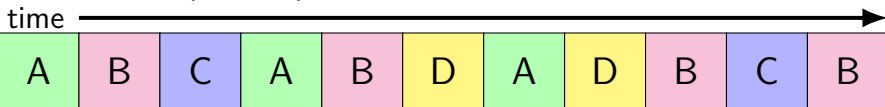
B *last* accessed in 1 time unit ago

D *last* accessed in 2 time units ago

choose to replace A

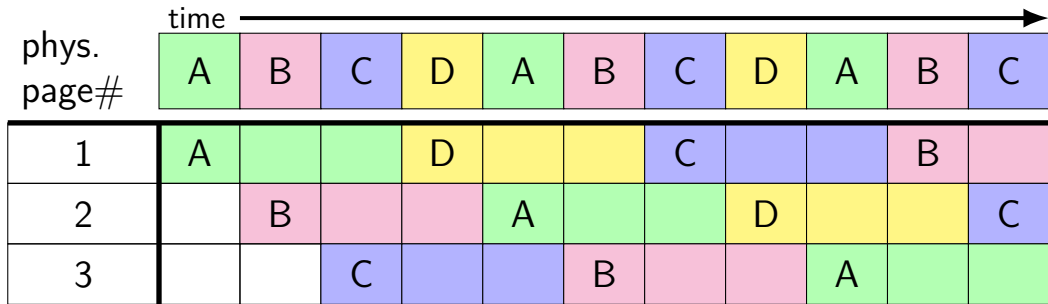
least recently used (the good case)

referenced (virtual) pages:



phys. page#	1	2	3	4	5	6	7	8	9	10	11
1	A									C	
2		B									
3			C			D					

least recently used (the worst case)



least recently used (the worst case)

	time →										
phys. page#	A	B	C	D	A	B	C	D	A	B	C
1	A			D			C			B	
2		B			A			D			C
3			C			B			A		

8 replacements with LRU

versus 3 replacements with MIN:

1	A									B	
2		B					C				
3			C	D							

least recently used (exercise) [intro]

A	B	A	D	C	B	D	B	C	D	A
---	---	---	---	---	---	---	---	---	---	---

1											
2											
3											

least recently used (exercise)

A	B	A	D	C	B	D	B	C	D	A
---	---	---	---	---	---	---	---	---	---	---

1	A	A	A	A							
2		B	B	B							
3				D							

pure LRU implementation

implementing LRU in software

maintain doubly-linked list of all physical pages

whenever a page is accessed:

- remove page from linked list, then
- add page to head of list

whenever a page needs to be replaced:

- remove a page from the tail of the linked list, then
- evict that page from all page tables (and anything else)
- and use that page for whatever needs to be loaded

pure LRU implementation

implementing LRU in software

maintain doubly-linked list of all physical pages

whenever a page is accessed:

remove page from linked list, then

add page

need to run code on every access
probably 100+x slowdown?

whenever a page needs to be evicted:

remove a page from the tail of the linked list, then

evict that page from all page tables (and anything else)

and use that page for whatever needs to be loaded

so, what's practical

probably won't implement LRU — too slow

what can we practically do?

tools for tracking accesses

approximating LRU = “was this accessed recently”?

don't need to detect all accesses, only one recent one

“was this accessed since we started looking a few seconds ago?”

tools for tracking accesses

approximating LRU = “was this accessed recently”?

don't need to detect all accesses, only one recent one

“was this accessed since we started looking a few seconds ago?”

ways to detect accesses AKA references:

mark page invalid, if page fault happens make valid and record
'accessed/referenced'

'accessed' or 'referenced' bit set by HW (on x86, but not everywhere)

tools for tracking accesses

approximating LRU = “was this accessed recently”?

don't need to detect all accesses, only one recent one

“was this accessed since we started looking a few seconds ago?”

ways to detect accesses AKA references:

mark page invalid, if page fault happens make valid and record
'accessed/referenced'

'accessed' or 'referenced' bit set by HW (on x86, but not everywhere)

tools for tracking accesses

approximating LRU = “was this accessed recently”?

don't need to detect all accesses, only one recent one

“was this accessed since we started looking a few seconds ago?”

ways to detect accesses AKA references:

mark page invalid, if page fault happens make valid and record
'accessed/referenced'

'accessed' or 'referenced' bit set by HW (on x86, but not everywhere)

tools for tracking accesses

approximating LRU = “was this accessed recently”?

don't need to detect all accesses, only one recent one

“was this accessed since we started looking a few seconds ago?”

ways to detect accesses AKA references:

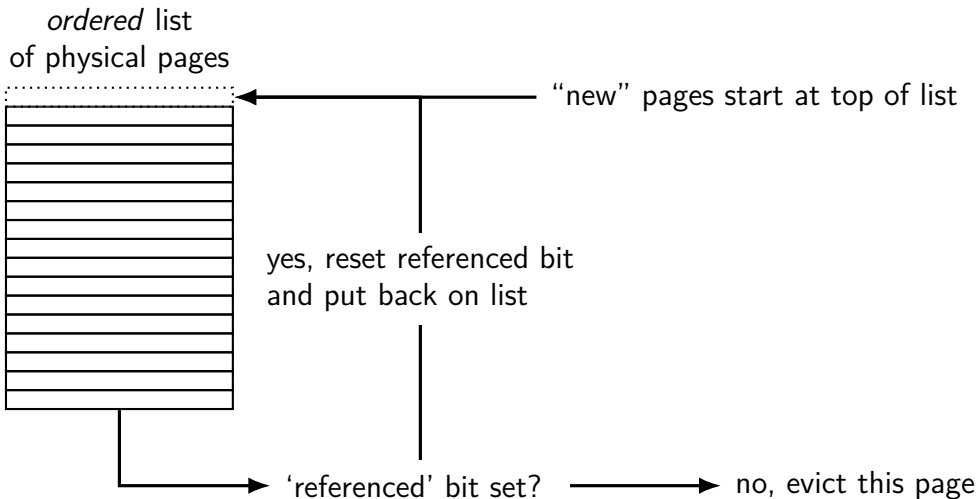
mark page invalid, if page fault happens make valid and record
'accessed/referenced'

'accessed' or 'referenced' bit set by HW (on x86, but not everywhere)

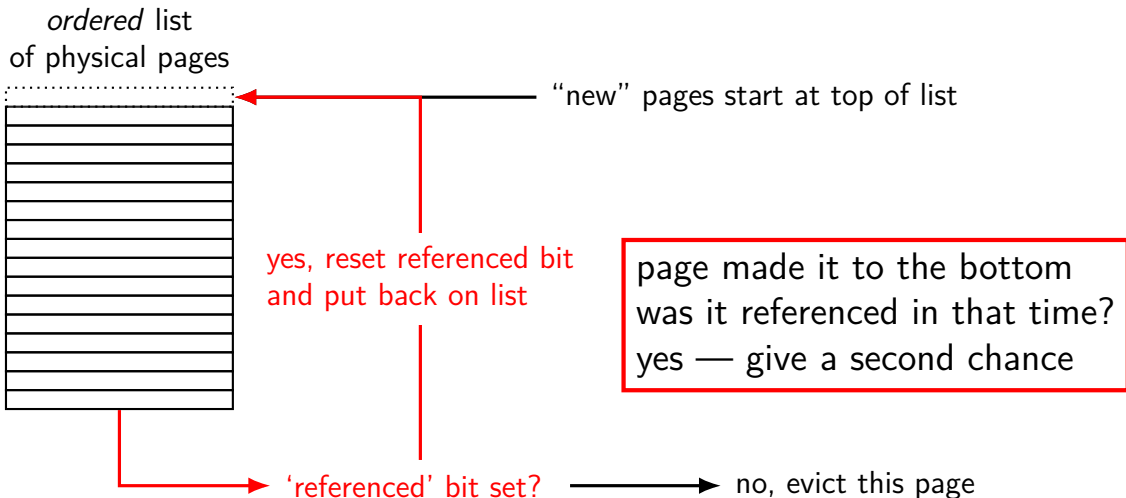
same idea applies for detecting writes

to know whether replaced page needs to be saved to disk
called “dirty” bit instead of accessed/referenced bit

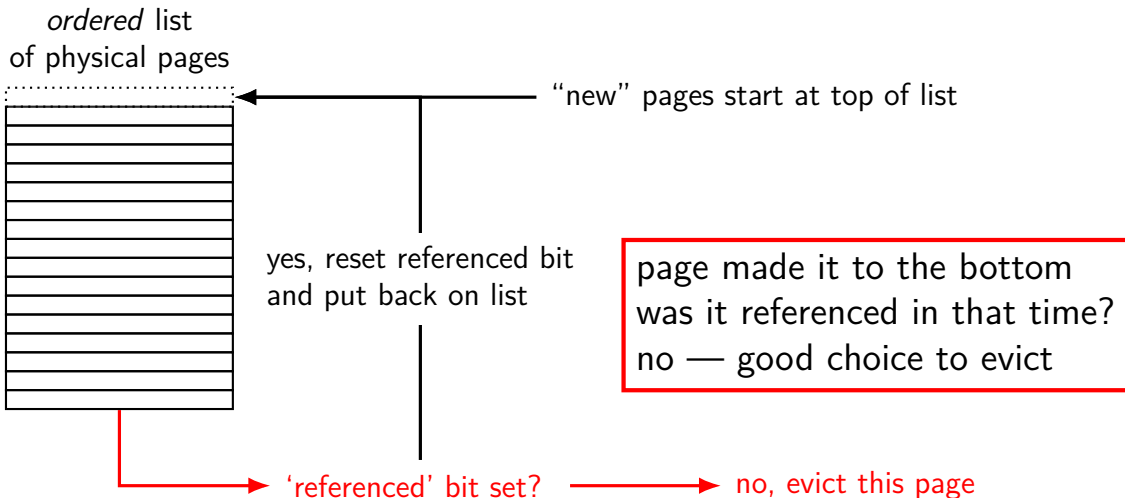
approximating LRU: second chance



approximating LRU: second chance



approximating LRU: second chance



second chance example (0)

	A		B		C	
--	---	--	---	--	---	--

1		A					
2				B			
3						C	

page list							
last added	3NR	1NR	*1R	2NR	*2R	3NR	*3R
—	2NR	3NR	3NR	1R	1R	2R	2R
end of list	1NR	2NR	2NR	3NR	3NR	1R	1R

second chance example (0)

	A		B
--	---	--	---

place A in physical page 1
accessed right after → becomes referenced

1		A					
2				B			
3						C	

page list							
last added	3NR	1NR	*1R	2NR	*2R	3NR	*3R
—	2NR	3NR	3NR	1R	1R	2R	2R
end of list	1NR	2NR	2NR	3NR	3NR	1R	1R

second chance example (0)

	A		B
--	---	--	---

place B in physical page 2
accessed right after → becomes referenced

1		A					
2				B			
3						C	

page list							
last added	3NR	1NR	*1R	2NR	*2R	3NR	*3R
—	2NR	3NR	3NR	1R	1R	2R	2R
end of list	1NR	2NR	2NR	3NR	3NR	1R	1R

second chance example (0)

future slides:
going to skip writing
these intermediate steps
(just for space)

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

second chance example (1)

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

second chance example (1)

place A in page 1

not referenced on return from page fault handler

immediately referenced by program when page fault handler returns

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

second chance example (1)

		page 2 was at bottom of list is not referenced okay to use						—	B
1	A						D		
2		B							
3			C			C			
page list									
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	

second chance example (1)

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

second chance example (1)

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

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

second chance example (1)

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

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

second chance example (1)

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

second chance example: exercise (1)

A	B	C	D	—	—	—	B	A
---	---	---	---	---	---	---	---	---

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

exercise: What does this access to A replace? (D, B, or C?)
 what is at end of list after? (PP 1, 2, or 3?)

second chance example: exercise (2)

A	B	C	D	—	—	—	B	A	—	C
---	---	---	---	---	---	---	---	---	---	---

1	A						D				?
2		B									?
3			C			C				A	?

page list										
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R

second chance example: exercise (2)

A	B	C	D	—	—	—	B	A	—	C
---	---	---	---	---	---	---	---	---	---	---

1	A						D				?
2		B									?
3			C			C				A	?

page list											
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	

exercise: What does this access to C replace? (D, B, or A?)
 what is at end of list after? (PP 1, 2, or 3?)

second chance cons

performs poorly with big memories...

may need to scan through lots of pages to find unaccessed

likely to count accesses from a long time ago

want some variation to tune its sensitivity

second chance cons

performs poorly with big memories...

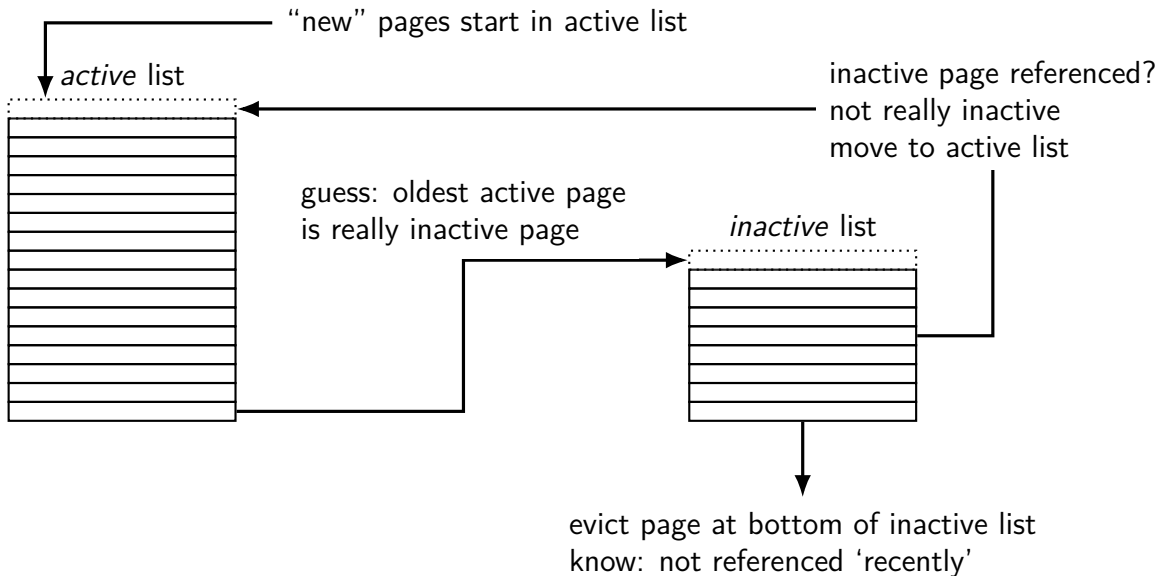
may need to scan through lots of pages to find unaccessed

likely to count accesses from a long time ago

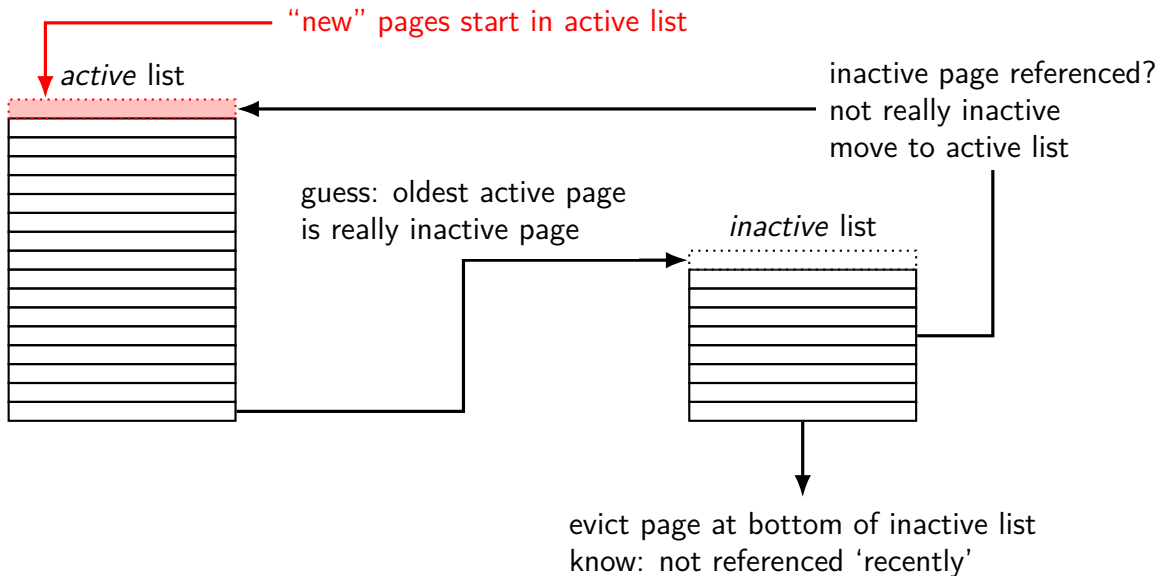
want some variation to tune its sensitivity

one idea: smaller list of pages to scan for accesses

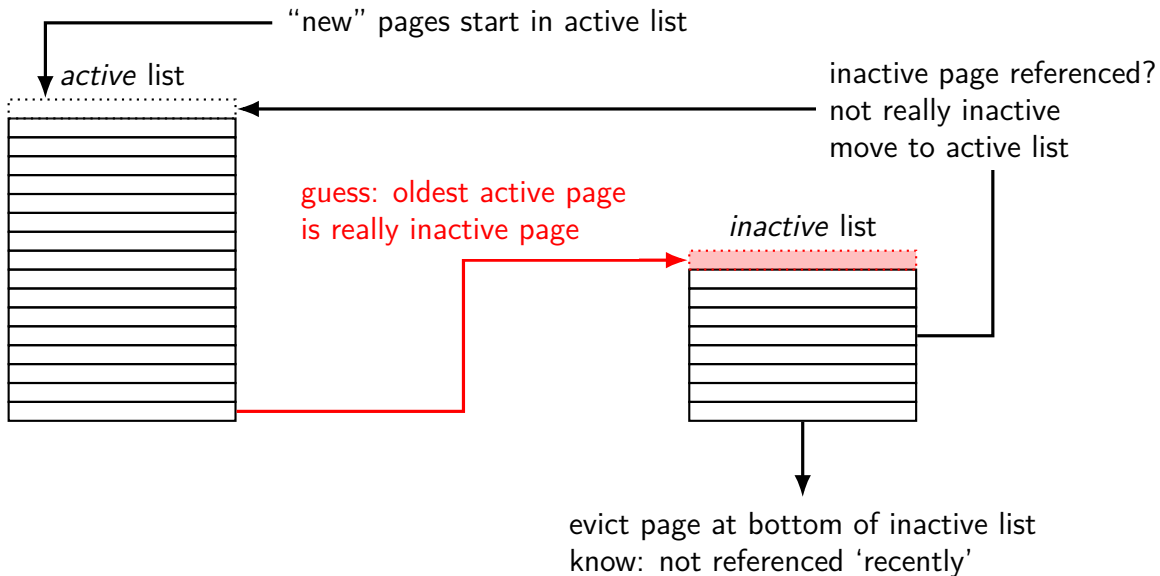
approximating LRU: SEQ



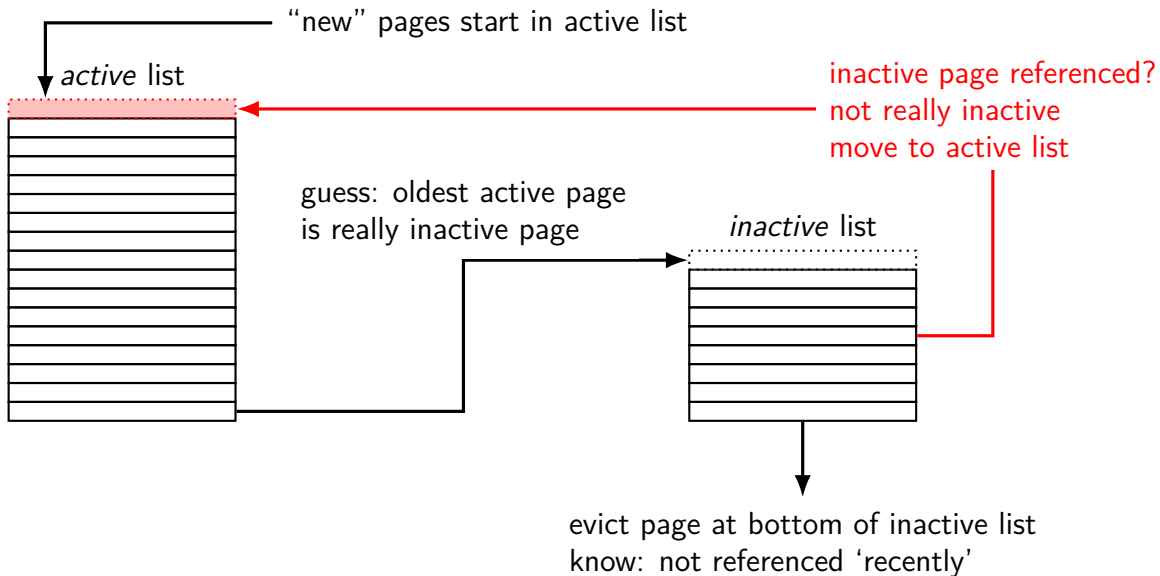
approximating LRU: SEQ



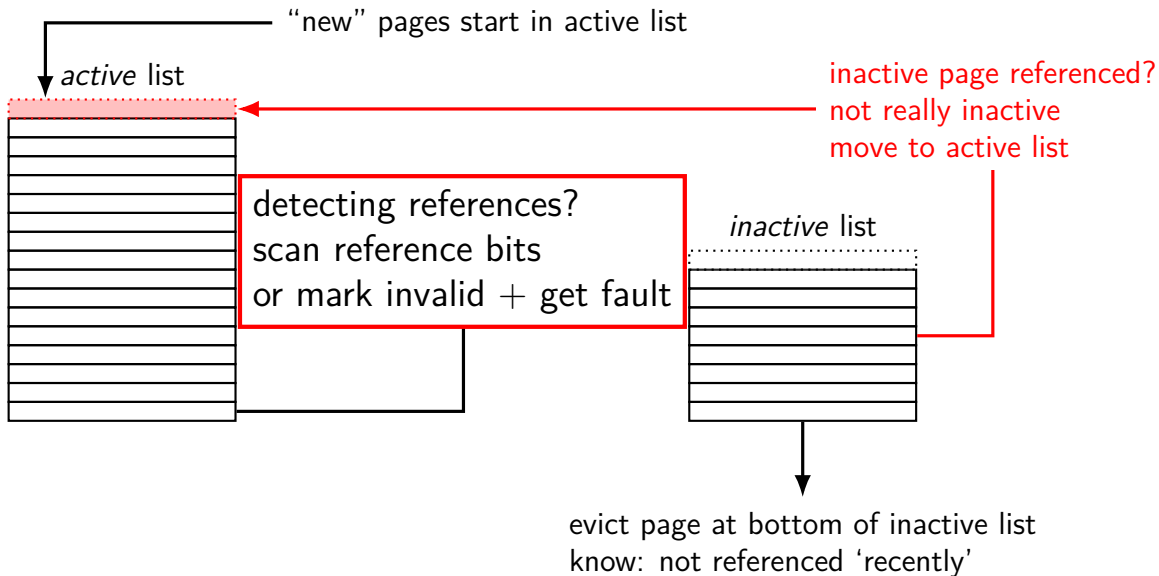
approximating LRU: SEQ



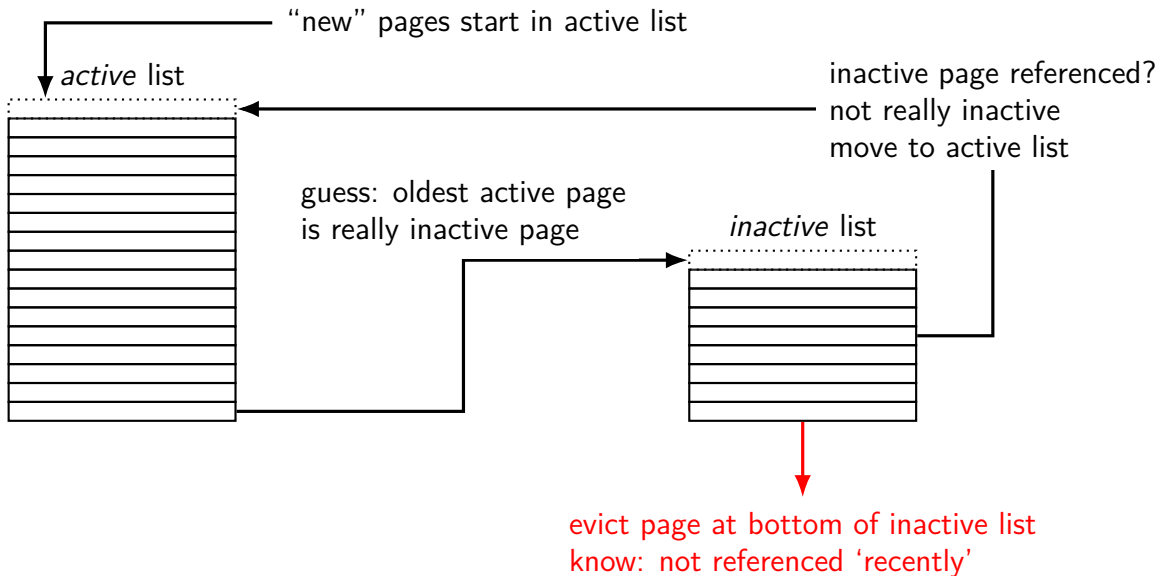
approximating LRU: SEQ



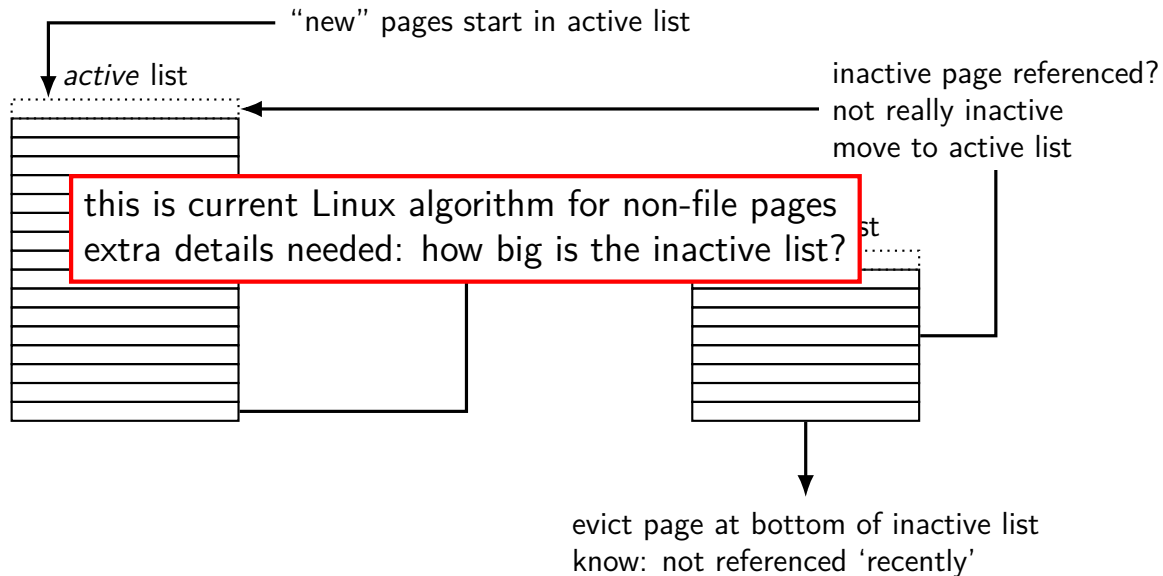
approximating LRU: SEQ



approximating LRU: SEQ



approximating LRU: SEQ



tracking usage: CLOCK (view 1)

ordered list
of physical pages

page #4: last referenced bits: Y Y Y...
page #5: last referenced bits: N N N...
page #6: last referenced bits: N Y Y...
page #7: last referenced bits: Y N Y...
page #8: last referenced bits: Y Y N...
page #1: last referenced bits: Y Y Y...
page #2: last referenced bits: N N N...
page #3: last referenced bits: Y Y N...

periodically:

take page from bottom of list

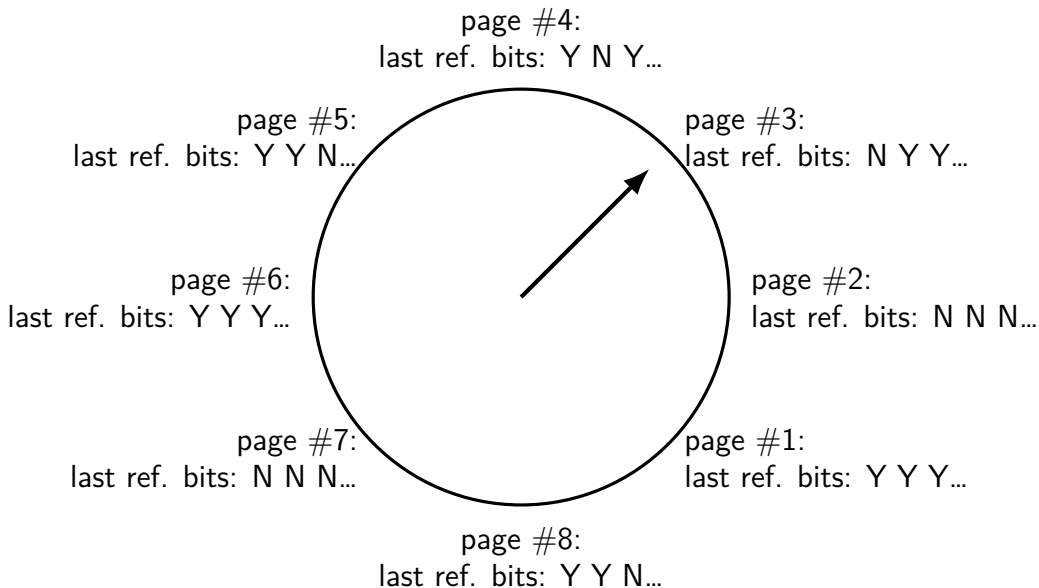
record current referenced bit

clear reference bit for next pass

add to top of list



tracking usage: CLOCK (view 2)



problems with LRU

question: when does LRU perform poorly?

exercise: which of these is LRU bad for?

code in a text editor for handling out-of-disk-space errors

initial values of the shell's global variables

on a desktop, long movies that are too big to fit in memory and played from beginning to end

on web server, long movies that are too big to fit in memory and frequently downloaded by clients

files that are parsed when loaded and overwritten when saved

on web server, frequently requested HTML files

solution for LRU being bad?

one idea that Linux uses:

for *file data*, use different replacement policy

tries to avoid keeping around file data accessed only once

CLOCK-Pro: special casing for one-use pages

by default, Linux tries to handle scanning of files

one read of file data — e.g. play a video, load file into memory

basic idea: **delay considering pages active until second access**

second access = second scan of accessed bits/etc.

single scans of file won't "pollute" cache

without this change: reading large files slows down other programs

recently read part of large file steals space from active programs

being proactive

previous assumption: load on demand

why is something loaded?

- page fault

- maybe because application starts

can we do better?

readahead

program accesses page 4 of a file, page 5, page 6. What's next?

readahead

program accesses page 4 of a file, page 5, page 6. What's next?

page 7 — idea: guess this

on page fault, does it look like contiguous accesses?

called **readahead**

readahead implementation ideas?

which of these is probably best?

- (a) when there's a page fault requiring reading page X of a file from disk, read pages X and $X + 1$
- (b) when there's a page fault requiring reading page $X > 200$ of a file from disk, read the rest of the file
- (c) when page fault occurs for page X of a file, read pages X through $X + 200$ and proactively add all to the current program's page table
- (d) when page fault occurs for page X of a file, read pages X through $X + 200$ but don't place pages $X + 1$ through $X + 200$ in the page table yet

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

when to start reads?

how much to readahead?

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern

- not enough to look at misses!

- want to check when readahead pages are used — keep up with program

when to start reads?

how much to readahead?

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern

- not enough to look at misses!

- want to check when readahead pages are used — keep up with program

when to start reads?

- takes some time to read in data — well before needed

how much to readahead?

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern

- not enough to look at misses!

- want to check when readahead pages are used — keep up with program

when to start reads?

- takes some time to read in data — well before needed

how much to readahead?

- if too much: evict other stuff programs need

- if too little: won't keep up with program

- if too little: won't make efficient use of HDD/SSD/etc.

being less lazy elsewhere

showed OS: proactively reading in pages

can also proactively free pages (faster replacement)

and proactively write out pages 'dirty' pages

- save time writing later

- avoid data loss on power failure

page cache/replacement summary

program memory + files — swapped to disk, cached in memory

mostly, assume temporal locality

- least recently used variants

special cases for non-LRU-friendly patterns (e.g. scans)

- maybe more we haven't discussed?

being proactive (writeback early, readahead, pre-evicted pages)

missing: handling non-miss-rate goals?

backup slides

x86-32 page table entries

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Address of page directory ¹																				Ignored					P C D	PW T	Ignored			CR3			
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address ²		P A T	Ignored		G	1	D	A	P C D	PW T	U / S	R / W	1	PDE: 4MB page				
Address of page table																				Ignored					0	I g n	A	P C D	PW T	U / S	R / W	1	PDE: page table
Ignored																													0	PDE: not present			
Address of 4KB page frame																				Ignored		G	P A T	D	A	P C D	PW T	U / S	R / W	1	PTE: 4KB page		
Ignored																													0	PTE: not present			

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory ¹												Ignored								P C D	PW T	Ignored					CR3					
Bits 31:22 of address of 4MB page frame												Ignored								P C D	PW T	U / S	R / W	1			PDE: 4MB page					
Address of page table												Ignored				0	I g n	A	P C D	PW T	U / S	R / W	1			PDE: page table						
Ignored																										0			PDE: not present			
Address of 4KB page frame												Ignored				G	P A T	D	A	P C D	PW T	U / S	R / W	1			PTE: 4KB page					
Ignored																												0			PTE: not present	

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				P C D		P W T		Ignored			CR3					
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)		Bits 39:32 of address ²				P A T	Ignored		G	1	D	A	P C D	P W T	U / S	R / W	1	PDE: 4MB page				
Address of page table																Ignored		0	I g n	A	P C D	P W T	U / S	R / W	1	PDE: page table						
Ignored																												0	PDE: not present			
Address of 4KB page frame																Ignored		G	P A T	D	A	P C D	P W T	U / S	R / W	1	PTE: 4KB page					
Ignored																												0	PTE: not present			

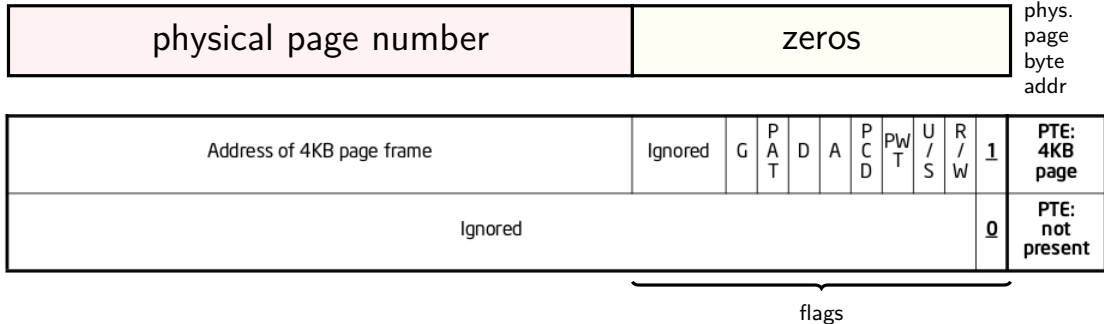
Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory ¹																Ignored				P C D	PW T	Ignored			CR3							
Bits 31:22 of address of 4MB page frame						Reserved (must be 0)				Bits 39:32 of address ²				P A T	Ignored	G	<u>1</u>	D	A	P C D	PW T	U / S	R / W	<u>1</u>	PDE: 4MB page							
Address of page table																Ignored				<u>0</u>	I g n	A	P C D	PW T	U / S	R / W	<u>1</u>	PDE: page table				
second-level page table entries																												<u>0</u>	PDE: not present			
Address of 4KB page frame																Ignored	G	P A T	D	A	P C D	PW T	U / S	R / W	<u>1</u>	PTE: 4KB page						
Ignored																												<u>0</u>	PTE: not present			

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entry v addresses



trick: page table entry with lower bits zeroed =
physical *byte* address of corresponding page
page # is address of page (2^{12} byte units)

makes constructing page table entries simpler:
physicalAddress | flagsBits

x86-32 pagetables: page table entries

xv6 header: mmu.h

```
// Page table/directory entry flags.
#define PTE_P           0x001    // Present
#define PTE_W           0x002    // Writeable
#define PTE_U           0x004    // User
#define PTE_PWT         0x008    // Write-Through
#define PTE_PCD         0x010    // Cache-Disable
#define PTE_A           0x020    // Accessed
#define PTE_D           0x040    // Dirty
#define PTE_PS          0x080    // Page Size
#define PTE_MBZ         0x180    // Bits must be zero

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
#define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
```


xv6: extracting top-level page table entry

```
void output_top_level_pte_for(struct proc *p, void *address) {
    pde_t *top_level_page_table = p->pgdir;
    // PDX = Page Directory index
    // next level uses PTX(....)
    int index_into_pgdir = PDX(address);
    pde_t top_level_pte = top_level_page_table[index_into_pgdir];
    cprintf("top level PT for %x in PID %d\n", address, p->pid);
    if (top_level_pte & PTE_P) {
        cprintf("is present (valid)\n");
    }
    if (top_level_pte & PTE_W) {
        cprintf("is writable (may be overridden in next level)\n");
    }
    if (top_level_pte & PTE_U) {
        cprintf("is user-accessible (may be overridden in next level)\n");
    }
    cprintf("has base address %x\n", PTE_ADDR(top_level_pte));
}
```

xv6: extracting top-level page table entry

```
void output_top_level_pte_for(struct proc *p, void *address) {
    pde_t *top_level_page_table = p->pgdir;
    // PDX = Page Directory index
    // next level uses PTX(....)
    int index_into_pgdir = PDX(address);
    pde_t top_level_pte = top_level_page_table[index_into_pgdir];
    cprintf("top level PT for %x in PID %d\n", address, p->pid);
    if (top_level_pte & PTE_P) {
        cprintf("is present (valid)\n");
    }
    if (top_level_pte & PTE_W) {
        cprintf("is writable (may be overridden in next level)\n");
    }
    if (top_level_pte & PTE_U) {
        cprintf("is user-accessible (may be overridden in next level)\n");
    }
    cprintf("has base address %x\n", PTE_ADDR(top_level_pte));
}
```

xv6: extracting top-level page table entry

```
void output_top_level_pte_for(struct proc *p, void *address) {
    pde_t *top_level_page_table = p->pgdir;
    // PDX = Page Directory index
    // next level uses PTX(...)
    int index_into_pgdir = PDX(address);
    pde_t top_level_pte = top_level_page_table[index_into_pgdir];
    cprintf("top level PT for %x in PID %d\n", address, p->pid);
    if (top_level_pte & PTE_P) {
        cprintf("is present (valid)\n");
    }
    if (top_level_pte & PTE_W) {
        cprintf("is writable (may be overridden in next level)\n");
    }
    if (top_level_pte & PTE_U) {
        cprintf("is user-accessible (may be overridden in next level)\n");
    }
    cprintf("has base address %x\n", PTE_ADDR(top_level_pte));
}
```

xv6: extracting top-level page table entry

```
void output_top_level_pte_for(struct proc *p, void *address) {
    pde_t *top_level_page_table = p->pgdir;
    // PDX = Page Directory index
    // next level uses PTX(....)
    int index_into_pgdir = PDX(address);
    pde_t top_level_pte = top_level_page_table[index_into_pgdir];
    cprintf("top level PT for %x in PID %d\n", address, p->pid);
    if (top_level_pte & PTE_P) {
        cprintf("is present (valid)\n");
    }
    if (top_level_pte & PTE_W) {
        cprintf("is writable (may be overridden in next level)\n");
    }
    if (top_level_pte & PTE_U) {
        cprintf("is user-accessible (may be overridden in next level)\n");
    }
    cprintf("has base address %x\n", PTE_ADDR(top_level_pte));
}
```

xv6: extracting top-level page table entry

```
void output_top_level_pte_for(struct proc *p, void *address) {
    pde_t *top_level_page_table = p->pgdir;
    // PDX = Page Directory index
    // next level uses PTX(....)
    int index_into_pgdir = PDX(address);
    pde_t top_level_pte = top_level_page_table[index_into_pgdir];
    cprintf("top level PT for %x in PID %d\n", address, p->pid);
    if (top_level_pte & PTE_P) {
        cprintf("is present (valid)\n");
    }
    if (top_level_pte & PTE_W) {
        cprintf("is writable (may be overridden in next level)\n");
    }
    if (top_level_pte & PTE_U) {
        cprintf("is user-accessible (may be overridden in next level)\n");
    }
    cprintf("has base address %x\n", PTE_ADDR(top_level_pte));
}
```

xv6: manually setting page table entry

```
pde_t *some_page_table; // if top-level table
pte_t *some_page_table; // if next-level table
...
...
some_page_table[index] =
    PTE_P | PTE_W | PTE_U | base_physical_address;
/* P = present; W = writable; U = user-mode accessible */
```

skipping the guard page

```
void example() {  
    int array[2000];  
    array[0] = 1000;  
    ...  
}
```

example:

```
    subl    $8024, %esp // allocate 8024 bytes on stack  
    movl    $1000, 12(%esp) // write near bottom of allocation  
                                // goes beyond guard page  
                                // since not all of array init'd  
    ....
```

create new page table (kernel mappings)

```
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;

    if((pgdir = (pde_t*)kalloc()) == 0)
        return 0;
    memset(pgdir, 0, PGSIZE);
    if (P2V(PHYSTOP) > (void*)DEVSPACE)
        panic("PHYSTOP too high");
    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
        if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                    (uint)k->phys_start, k->perm) < 0) {
            freevm(pgdir);
            return 0;
        }
    return pgdir;
}
```


create new page table (kernel mappings)

allocate first-level page table
("page directory")

```
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;

    if((pgdir = (pde_t*)kalloc()) == 0)
        return 0;
    memset(pgdir, 0, PGSIZE);
    if (P2V(PHYSTOP) > (void*)DEVSPACE)
        panic("PHYSTOP too high");
    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
        if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                    (uint)k->phys_start, k->perm) < 0) {
            freevm(pgdir);
            return 0;
        }
    return pgdir;
}
```

create new page table (kernel mappings)

initialize to 0 — every page invalid

```
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;

    if((pgdir = (pde_t*)kalloc()) == 0)
        return 0;
    memset(pgdir, 0, PGSIZE);
    if (P2V(PHYSTOP) > (void*)DEVSPACE)
        panic("PHYSTOP too high");
    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
        if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                    (uint)k->phys_start, k->perm) < 0) {
            freevm(pgdir);
            return 0;
        }
    return pgdir;
}
```

create new page table (kernel mappings)

```
pde_t*  
setupkvm(void)  
{  
    pde_t *pgdir;  
    struct kmap *k;
```

iterate through list of kernel-space mappings
for everything above address 0x8000 0000
(hard-coded table including flag bits, etc.
because some addresses need different flags
and not all physical addresses are usable)

```
    if((pgdir = (pde_t*)kalloc(0)) == 0)  
        return 0;  
    memset(pgdir, 0, PGSIZE);  
    if (P2V(PHYSTOP) > (void*)DEVSPACE)  
        panic("PHYSTOP too high");  
    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)  
        if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,  
                    (uint)k->phys_start, k->perm) < 0) {  
            freevm(pgdir);  
            return 0;  
        }  
    return pgdir;  
}
```

create new page table (kernel mappings)

```
pde_t*  
setupkvm(void)  
{
```

on failure (no space for new second-level page tales)
free everything

```
    pde_t *pgdir;  
    struct kmap *k;  
  
    if((pgdir = (pde_t*)kalloc()) == 0)  
        return 0;  
    memset(pgdir, 0, PGSIZE);  
    if (P2V(PHYSTOP) > (void*)DEVSPACE)  
        panic("PHYSTOP too high");  
    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)  
        if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,  
                    (uint)k->phys_start, k->perm) < 0) {  
            freevm(pgdir);  
            return 0;  
        }  
    return pgdir;  
}
```

reading executables (headers)

xv6 executables contain list of sections to load, represented by:

```
struct proghdr {  
    uint type;           /* <-- debugging-only or not? */  
    uint off;            /* <-- location in file */  
    uint vaddr;          /* <-- location in memory */  
    uint paddr;          /* <-- confusing ignored field */  
    uint filesz;         /* <-- amount to load */  
    uint memsz;          /* <-- amount to allocate */  
    uint flags;          /* <-- readable/writeable (ignored) */  
    uint align;  
};
```

reading executables (headers)

xv6 executables contain list of sections to load, represented by:

```
struct proghdr {  
    uint type;           /* <-- debugging-only or not? */  
    uint off;            /* <-- location in file */  
    uint vaddr;           /* <-- location in memory */  
    uint paddr;          /* <-- confusing ignored field */  
    uint filesz;          /* <-- amount to load */  
    uint memsz;           /* <-- amount to allocate */  
    uint flags;           /* <-- readable/writeable (ignored) */  
    uint align;  
};  
  
...  
if((sz = allocvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)  
    goto bad;  
...  
if(loadvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)  
    goto bad;
```

reading executables (headers)

xv6 executables contain list of sections to load, represented by:

```
struct proghdr {
    uint type;
    uint off;
    uint vaddr;           /* <-- location in memory */
    uint paddr;           /* <-- confusing ignored field */
    uint filesz;          /* <-- amount to load */
    uint memsz;           /* <-- amount to allocate */
    uint flags;           /* <-- readable/writable (ignored) */
    uint align;
};

...
if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;

...
if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
```

loading user pages from executable

```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
    ...
    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
            panic("loaduvm: address should exist");
        pa = PTE_ADDR(*pte);
        if(sz - i < PGSIZE)
            n = sz - i;
        else
            n = PGSIZE;
        if(readi(ip, P2V(pa), offset+i, n) != n)
            return -1;
    }
    return 0;
}
```


loading user pages from executable

```
loadvm(pde_t *pgdir, char *addr, uintr_t *ui)
{
    ...
    for(i = 0; i < sz; i += PGSIZE)
        if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
            panic("loadvm: address should exist");
        pa = PTE_ADDR(*pte);
        if(sz - i < PGSIZE)
            n = sz - i;
        else
            n = PGSIZE;
        if(readi(ip, P2V(pa), offset+i, n) != n)
            return -1;
    }
    return 0;
}
```

get page table entry being loaded
already allocated earlier
look up address to load into

loading user pages from executable

```
loadvm(pde_t *pgdir, ch  
{
```

get physical address from page table entry
convert back to (kernel) virtual address
for read from disk

```
    ...  
    for(i = 0; i < sz; i += PGSIZE){  
        if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)  
            panic("loadvm: address should exist");
```

```
        pa = PTE_ADDR(*pte);
```

```
        if(sz - i < PGSIZE)
```

```
            n = sz - i;
```

```
        else
```

```
            n = PGSIZE;
```

```
        if(readi(ip, P2V(pa), offset+i, n) != n)
```

```
            return -1;
```

```
    }
```

```
    return 0;
```

```
}
```

loading user pages from executable

```
loaduvm(pde_t *pgdir, void *addr, uintr_t *ui, off_t offset, int n)
{
    ...
    for(i = 0; i < sz; i += PGSIZE) {
        if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
            panic("loaduvm: address should exist");
        pa = PTE_ADDR(*pte);
        if(sz - i < PGSIZE)
            n = sz - i;
        else
            n = PGSIZE;
        if(readi(ip, P2V(pa), offset+i, n) != n)
            return -1;
    }
    return 0;
}
```

exercise: why don't we just use `addr` directly?
(instead of turning it into a physical address,
then into a virtual address again)

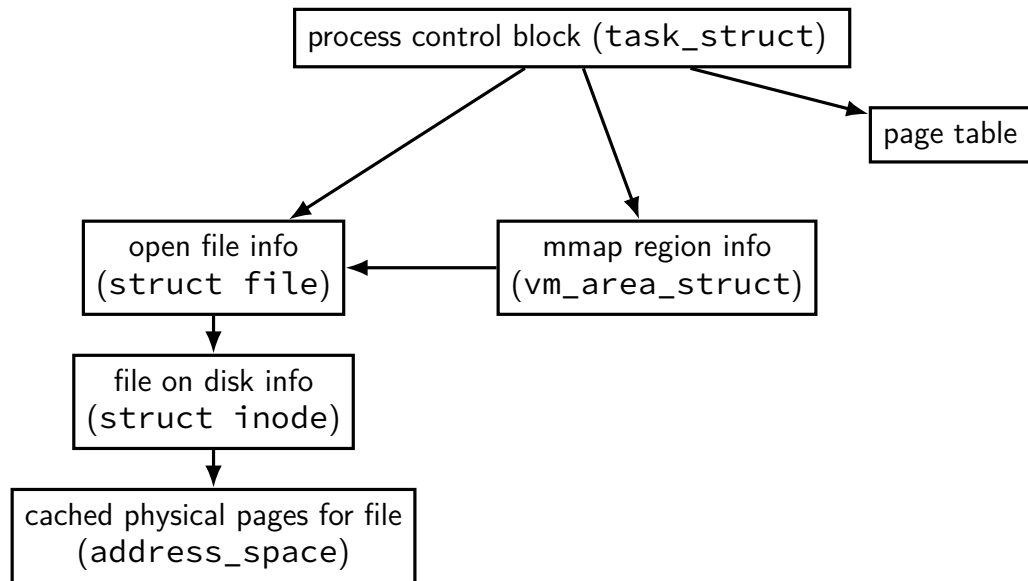
loading user pages from executable

copy from file (represented by struct inode) into memory, using
P2V(pa) — mapping of physical addresss in kernel memory

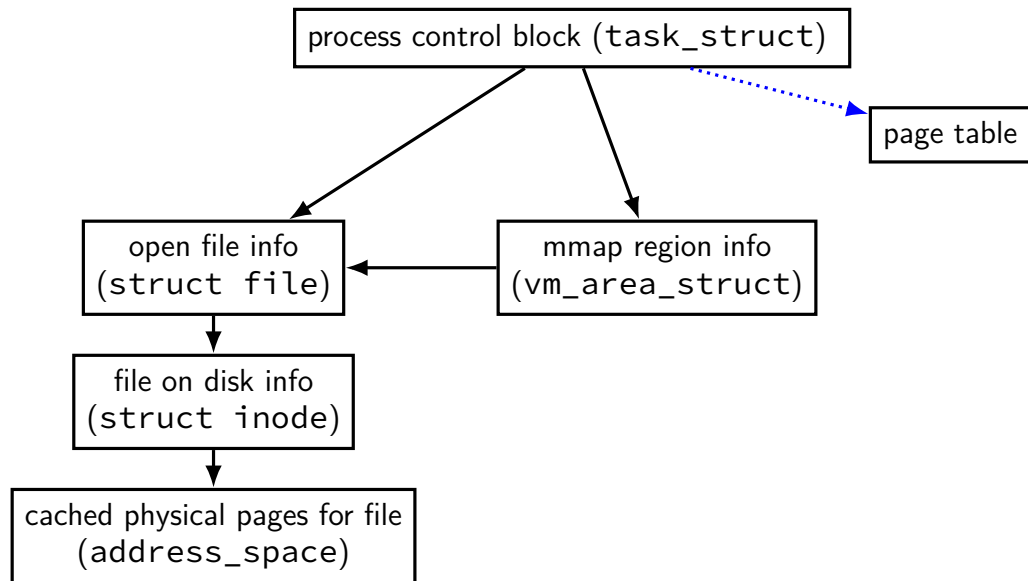
```
loadvm  
{
```

```
...  
for(i = 0; i < sz; i += PGSIZE){  
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)  
        panic("loadvm: address should exist");  
    pa = PTE_ADDR(*pte);  
    if(sz - i < PGSIZE)  
        n = sz - i;  
    else  
        n = PGSIZE;  
    if(readi(ip, P2V(pa), offset+i, n) != n)  
        return -1;  
}  
return 0;  
}
```

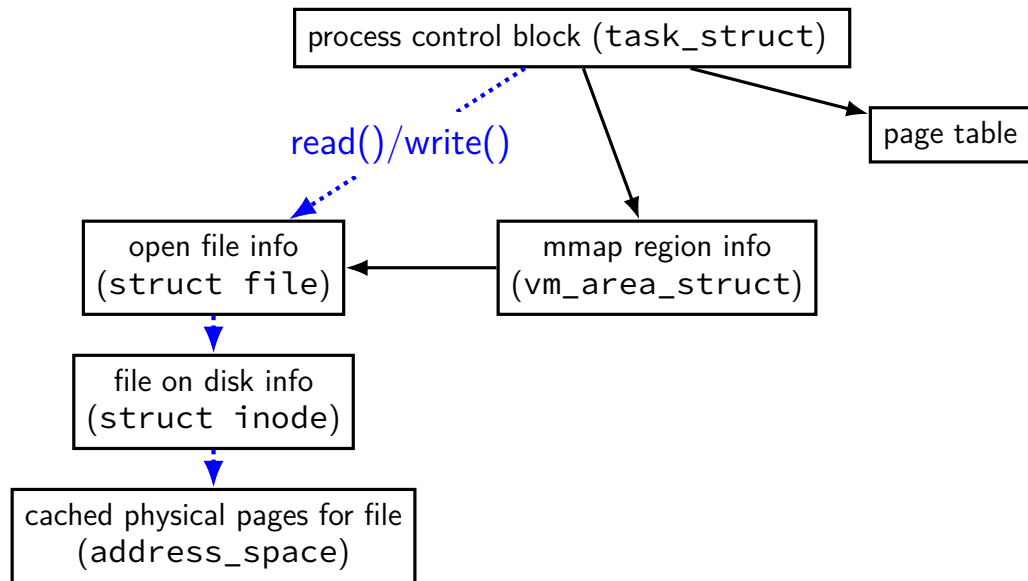
Linux: forward mapping



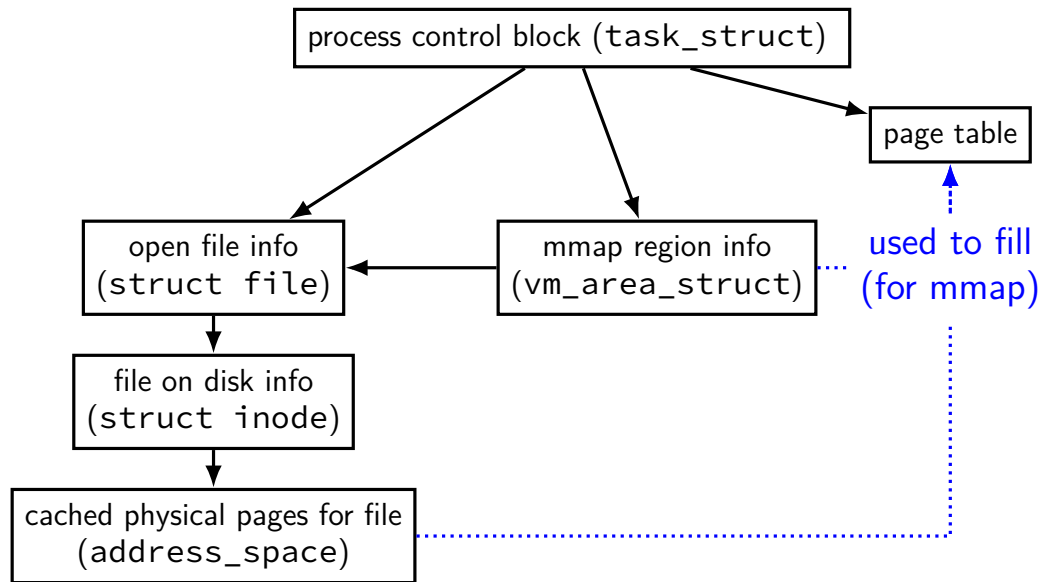
Linux: forward mapping



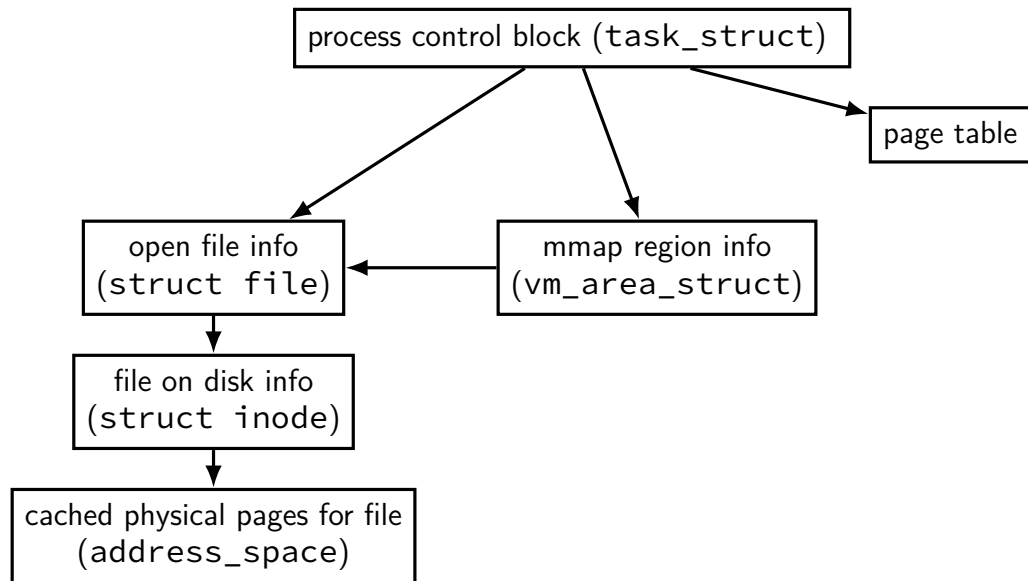
Linux: forward mapping



Linux: forward mapping



Linux: forward mapping



sketch: implementing mmap

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

write “mapped” memory, write to disk eventually
need to detect whether writes happened
usually hardware support: dirty bit

extra detail: other processes should see changes
all accesses to file use **same physical memory**
how? OS tracks copies of files in memory

xv6: setting process page tables (exec())

exec step 1: create new page table with kernel mappings
done in `setupkvm()`, which calls `mappages()`

exec step 2a: allocate memory for executable pages
`allocuvvm()` in loop
new physical pages chosen by `kalloc()`

exec step 2b: load from executable file
copying from executable file implemented by `loaduvvm()`

exec step 3: allocate pages for heap, stack (`allocuvvm()` calls)

xv6: setting process page tables (exec())

exec step 1: create new page table with kernel mappings
done in `setupkvm()`, which calls `mappages()`

exec step 2a: **allocate memory for executable pages**
`allocuvvm()` in loop
new physical pages chosen by `kalloc()`

exec step 2b: load from executable file
copying from executable file implemented by `loaduvvm()`

exec step 3: allocate pages for heap, stack (`allocuvvm()` calls)

minor and major faults

minor page fault

- page is already in memory (“page cache”)
- just fill in page table entry

major page fault

- page not already in memory (“page cache”)
- need to allocate space
- possibly need to read data from disk/etc.

Linux: reporting minor/major faults

```
$ /usr/bin/time --verbose some-command
  Command being timed: "some-command"
  User time (seconds): 18.15
  System time (seconds): 0.35
  Percent of CPU this job got: 94%
  Elapsed (wall clock) time (h:mm:ss or m:ss): 0:19.57
...
  Maximum resident set size (kbytes): 749820
  Average resident set size (kbytes): 0
  Major (requiring I/O) page faults: 0
  Minor (reclaiming a frame) page faults: 230166
  Voluntary context switches: 1423
  Involuntary context switches: 53
  Swaps: 0
...
  Exit status: 0
```

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 reads and writes: hundreds of microseconds

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

HDD/SDDs are slow

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

- minimum size: 512 bytes

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

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

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

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

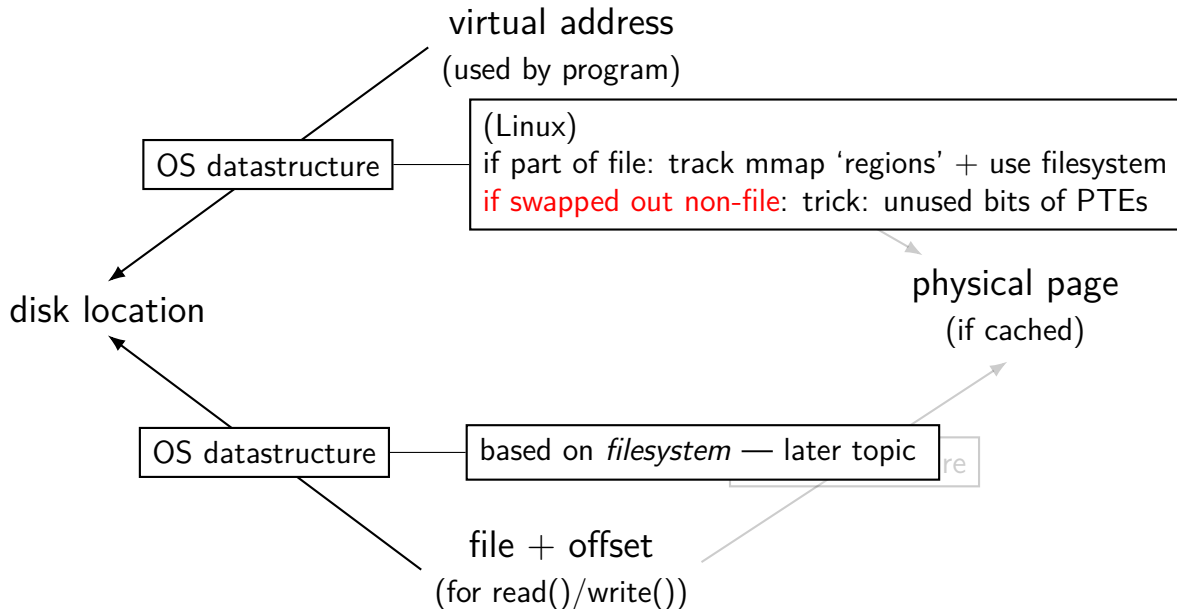
- minimum size: 512 bytes

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

SSD reads and writes: hundreds of microseconds

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

virtual address/file offset \rightarrow location on disk



Linux: tracking swapped out pages

need to lookup **location on disk**

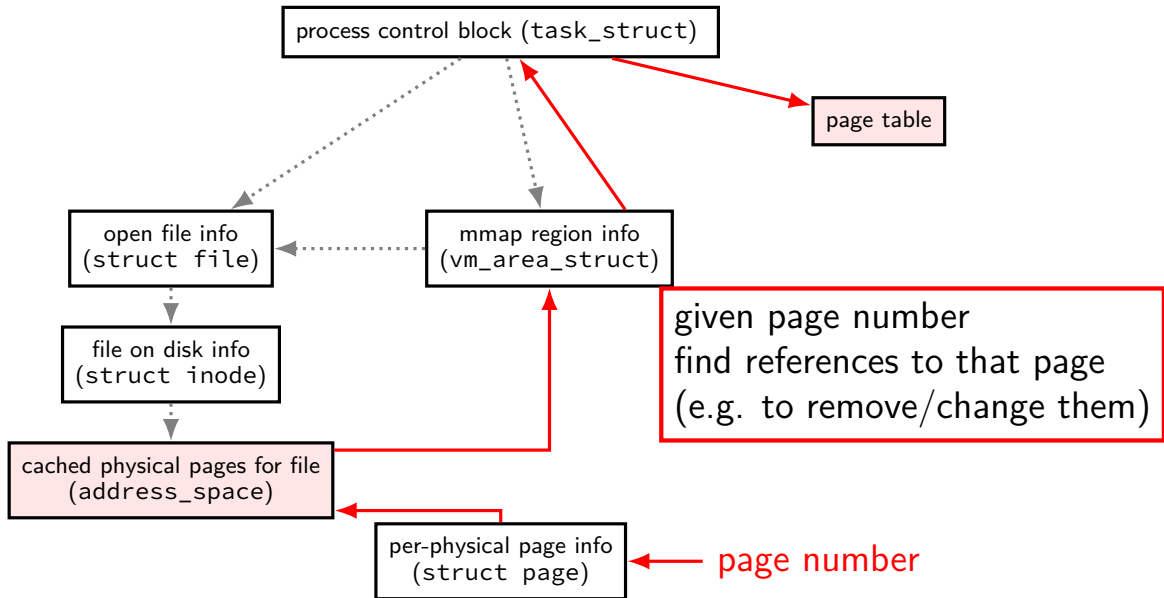
potentially one location for every virtual page

trick: store location in “ignored” part of **page table entry**
instead of physical page #, permission bits, etc., store offset on disk

Address of 4KB page frame	Ignored	G	P A T	D	A	P C D	PW T	U / S	R / W	<u>1</u>	PTE: 4KB page
Ignored										<u>0</u>	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Linux: reverse mapping (file pages)



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, ...)

predicting the future?

can't really...

look for common patterns

working set intuition

say we're executing a loop

what memory does this require?

code for the loop

code for functions called in the loop
and functions they call

data structures used by the loop and functions called in it, etc.

only uses a subset of the program's memory

the working set model

one common pattern: **working sets**

at any time, program is using a **subset of its memory**

...called its *working set*

rest of memory is inactive

...until program switches to different working set

working sets and running many programs

give each program its working set

...and, to run as much as possible, not much more
inactive — won't be used

working sets and running many programs

give each program its working set

...and, to run as much as possible, not much more

inactive — won't be used

replacement policy: identify working sets \approx recently used data

replace anything that's not in in it

cache size versus miss rate

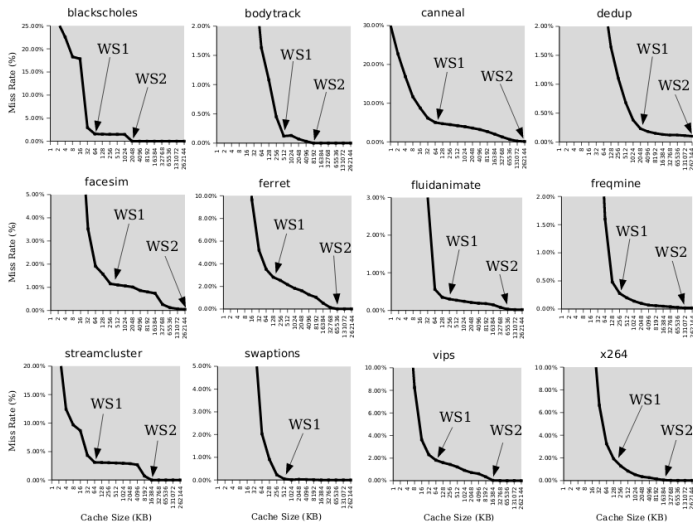


Figure 3: Miss rates versus cache size. Data assumes a shared 4-way associative cache with 64 byte lines. WS1 and WS2 refer to important working sets which we analyze in more detail in Table 2. Cache requirements of PARSEC benchmark programs can reach hundreds of megabytes.

estimating working sets

working set \approx what's been used recently

except when program switching working sets

so, what a program recently used \approx working set

can use this idea to estimate working set (from list of memory accesses)

estimating working sets

working set \approx what's been used recently

except when program switching working sets

so, what a program recently used \approx working set

can use this idea to estimate working set (from list of memory accesses)

recording accesses

goal: “check is this physical page still being used?”

software support: temporarily mark page table invalid

use resulting page fault to detect “yes”

hardware support: accessed bits in page tables

hardware sets to 1 when accessed

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...
0x00123	0	0	...	0x4442
...

OS page info

PPN	last known access?	...
...
0x04442	(never)	...
...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

...

(OS exception's handler)

...

oops! page fault

processor does lookup

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...
0x00123	0	0	...	0x4442
...

OS page info

PPN	last known access?	...
...
0x04442	(never)	...
...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx  
mov 0x123789, %ecx  
...  
...  
mov 0x123300, %ecx
```

the kernel

```
...  
(OS exception's handler)  
...
```

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...
0x00123	1	0	...	0x4442
...

update page info: +
mark present

OS page info

PPN	last known access?	...
...
0x04442	at time X	...
...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx  
mov 0x123789, %ecx  
...  
...  
mov 0x123300, %ecx
```

the kernel

```
...  
(OS exception's handler)  
...
```

processor does lookup

no page fault, not recorded in OS info

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...
0x00123	1	0	...	0x4442
...

OS page info

PPN	last known access?	...
...
0x04442	at time X	...
...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

processor does lookup

no page fault, not recorded in OS info

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...
0x00123	1	0	...	0x4442
...

OS page info

PPN	last known access?	...
...
0x04442	at time X	...
...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

OS clears present bit
to check for next access

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...
0x00123	1	0	...	0x4442
...

OS page info

PPN	last known access?	...
...
0x04442	at time X	...
...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx  
mov 0x123789, %ecx  
...  
...  
mov 0x123300, %ecx
```

the kernel

```
...  
(OS exception's handler)  
...
```

OS clears present bit
to check for next access

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...
0x00123	0	0	...	0x4442
...

OS page info

PPN	last known access?	...
...
0x04442	at time X	...
...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx  
mov 0x123789, %ecx  
...  
...  
mov 0x123300, %ecx
```

processor does lookup

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...
0x00123	0	0	...	0x4442
...

the kernel

...

(OS exception's handler)

...

oops! page fault

OS page info

PPN	last known access?	...
...
0x04442	at time X	...
...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx  
mov 0x123789, %ecx  
...  
...  
mov 0x123300, %ecx
```

the kernel

```
...  
(OS exception's handler)  
...
```

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...
0x00123	1	0	...	0x4442
...

update page info: +
mark present

OS page info

PPN	last known access?	...
...
0x04442	at time Y	...
...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00123	1	0	0	...	0x4442
...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx  
mov 0x123789, %ecx  
...  
...  
mov 0x123300, %ecx
```

the kernel

```
...  
(OS exception's handler)  
...
```

processor does lookup
sets accessed bit to 1

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00123	1	0	0	...	0x4442
...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

processor does lookup
sets accessed bit to 1

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00123	1	1	0	...	0x4442
...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```


the kernel

```
...
(OS exception's handler)
...
```

processor does lookup

keeps access bit set to 1

page table for program 1



VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00123	1	1	0	...	0x4442
...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```


the kernel

```
...
(OS exception's handler)
...
```

processor does lookup

keeps access bit set to 1

page table for program 1



VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00123	1	1	0	...	0x4442
...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00123	1	1	0	...	0x4442
...

OS reads + records +
clears access bit

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00123	1	0	0	...	0x4442
...

OS reads + records +
clears access bit

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx  
mov 0x123789, %ecx  
...  
...  
mov 0x123300, %ecx
```

the kernel

```
...  
(OS exception's handler)  
...
```

processor does lookup
sets accessed bit to 1 (again)

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00123	1	0	0	...	0x4442
...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

processor does lookup
sets accessed bit to 1 (again)

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00123	1	1	0	...	0x4442
...

accessed bits: multiple processes

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00123	1	0	0	...	0x4442
...

page table for program 2

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...
0x00483	1	1	0	...	0x4442
...

OS needs to clear+check
all accessed bits
for the physical page

dirty bits

“was this part of the mmap'd file changed?”

“is the old swapped copy still up to date?”

software support: temporarily mark read-only

hardware support: ***dirty bit*** set by hardware

same idea as accessed bit, but only changed on writes

x86-32 accessed and dirty bit

Address of 4KB page frame	Ignored	G	P A T	D	A	P C D	PW T	U / S	R / W	1	PTE: 4KB page
Ignored										0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

A: accessed — processor sets to 1 when PTE used

used = for read or write or execute

likely implementation: part of loading PTE into TLB

D: dirty — processor sets to 1 when PTE is used for write

lazy replacement?

so far: don't do anything special **until memory is full**

only then is there a reason to writeback pages or evict pages

lazy replacement?

so far: don't do anything special **until memory is full**

only then is there a reason to writeback pages or evict pages

but real OSes are more proactive

non-lazy writeback

what happens when a computer loses power

how much data can you lose?

if we never run out of memory...all of it?

no changed data written back

solution: track or scan for dirty pages and writeback

example goals:

lose no more than 90 seconds of data

force writeback at file close

...

non-lazy eviction

so far — allocating memory involves evicting pages

hopefully pages that haven't been used a long time anyways

non-lazy eviction

so far — allocating memory involves evicting pages

hopefully pages that haven't been used a long time anyways

alternative: evict earlier “in the background”

“free”: probably have some idle processor time anyways

allocation = remove already evicted page from linked list
(instead of changing page tables, file cache info, etc.)

xv6 page table-related functions

`kalloc/kfree` — allocate physical page, return kernel address

`walkpgdir` — get pointer to second-level page table entry
...to check it/make it valid/invalid/point somewhere/etc.

`mappages` — set range of page table entries
implementation: loop using `walkpgdir`

`allocvm` — create new set of page tables, set kernel (high) part
entries for `0x8000 0000` and up set
allocate new first-level table plus several second-level tables

`allocvm` — allocate new user memory
setup user-accessible memory
allocate new second-level tables as needed

`deallocvm` — deallocate user memory