last time

xv6 memory layout

- kernel gets top half of virtual addresses
- 1:1 mapping (for convenience) to physical addresses

x86-32 page table format

- top 20-bits of physical address: physical page number
- top 20-bits of page table entry: physical page number
- trick: `addr | flags` = page table entry

walkpgdir: retrieve page table entry for virtual address

mappages: set range of page table entries

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`

`allockvm` — 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

`allocuvm` — allocate new user memory

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

`deallocuvm` — deallocate user memory

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`

`allockvm` — 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

`allocuvm` — allocate new user memory

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

`deallocuvm` — deallocate user memory

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`

`allockvm` — 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

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

`deallocuvm` — deallocate user memory

allocating user pages

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  ...
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0)
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
```

allocating user pages

allocate a new, zero page

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  ...
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0)
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
```

allocating user pages

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  ...
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0)
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
```

add page to second-level page table

allocating user pages

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  ...
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSI
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0)
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
```

this function used for initial allocation plus expanding heap on request

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`

`allockvm` — 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

`allocuvm` — allocate new user memory

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

`deallocuvm` — deallocate user memory

kalloc/kfree

kalloc/kfree — xv6's physical memory allocator

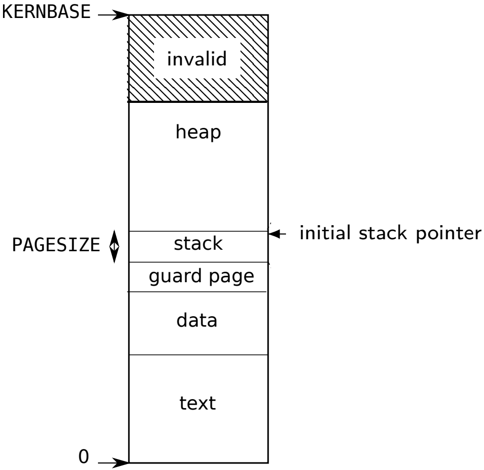
allocates/deallocates whole pages only

keep linked list of free pages
- list nodes — stored in corresponding free page itself
- kalloc — return first page in list
- kfree — add page to list

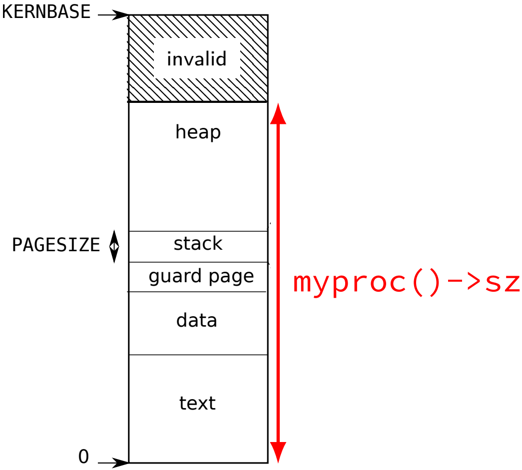
linked list created at boot

usuable memory fixed size (224MB)
- determined by PHYSTOP in `memlayout.h`

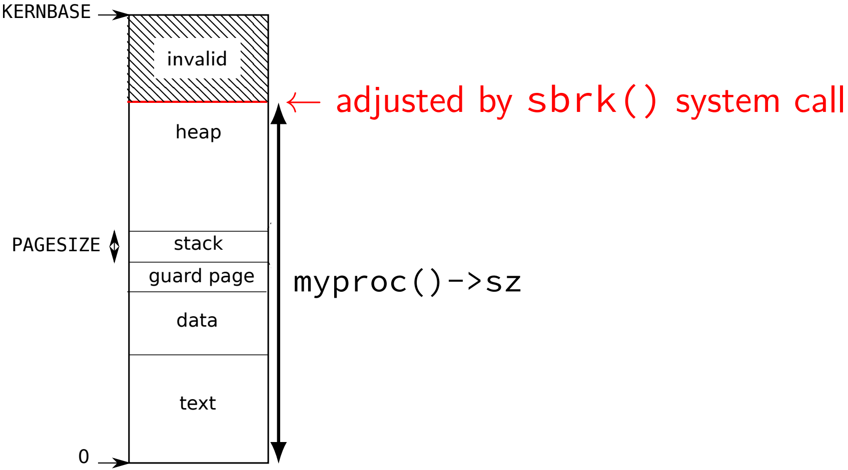
xv6 program memory



xv6 program memory



xv6 program memory



xv6 heap allocation

xv6: every process has a heap at the top of its address space

- yes, this is unlike Linux where heap is below stack

tracked in `struct proc` with `sz`

- = last valid address in process

position changed via `sbrk(amount)` system call

- sets `sz += amount`
- same call exists in Linux, etc. — but also others

sbrk

```
sys_sbrk()
{
  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

sbrk

```
sys_sbrk()
{
  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

`sz`: current top of heap

sbrk

`sbrk(N)`: grow heap by N (shrink if negative)

```
sys_sbrk()
{
  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

sbrk

returns old top of heap (or -1 on out-of-memory)

```
sys_sbrk()
{
  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

growproc

```
growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();

  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

growproc

allocuvm — same function used to allocate initial space

maps pages for addresses `sz` to `sz + n`

calls `kalloc` to get each page

```
growproc(int
{
  uint sz;
  struct proc *curproc = myproc();

  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

xv6 page faults (now)

accessing page marked invalid (not-present) — triggers page fault

- xv6 now: default case in trap() function

xv6 page faults (now)

accessing page marked invalid (not-present) — triggers page fault

- xv6 now: default case in trap() function

```
/* in some user program: */
*((int*) 0x800444) = 1;
...
/* in trap() in trap.c: */
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
```

```
pid 4 processname: trap 14 err 6 on cpu 0 eip 0x1a addr 0x800444--kill proc
```

xv6 page faults (now)

accessing page marked invalid (not-present) — triggers page fault

- xv6 now: default case in trap() function

```
/* in some user program: */
*((int*) 0x800444) = 1;
...
/* in trap() in trap.c: */
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
```

```
pid 4 processname: trap 14 err 6 on cpu 0 eip 0x1a addr 0x800444--kill proc
```

trap 14 = T_PGFLT

special register CR2 contains faulting address

xv6 page faults (now)

accessing page marked invalid (not-present) — triggers page fault

- xv6 now: default case in trap() function

```
/* in some user program: */
*((int*) 0x800444) = 1;
...
/* in trap() in trap.c: */
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
```

```
pid 4 processname: trap 14 err 6 on cpu 0 eip 0x1a addr 0x800444--kill proc
```

trap 14 = T_PGFLT

special register CR2 contains faulting address

xv6: if one handled page faults

- alternative to crashing: update the page table and return
  - returning from page fault handler normally retries failing instruction
- “just in time” update of the process’s memory
  - example: don’t actually allocate memory until it’s needed

xv6: if one handled page faults

alternative to crashing: update the page table and return

- returning from page fault handler normally retries failing instruction

“just in time” update of the process’s memory

- example: don’t actually allocate memory until it’s needed

pseudocode for xv6 implementation (for trap())

```
if (tf->trapno == T_PGFLT) {
  void *address = (void *) rcr2();
  if (is_address_okay(myproc(), address)) {
      setup_page_table_entry_for(myproc(), address);
      // return from fault, retry access
  } else {
      // actual segfault, kill process
      cprintf("...");
      myproc()->killed = 1;
  }
}
```

xv6: if one handled page faults

alternative to crash

check *process control block* to see if access okay

- returning from page fault handler normally retries failing instruction

"just in time" update of the process's memory

- example: don't actually allocate memory until it's needed

pseudocode for xv6 implementation (for trap())

```
if (tf->trapno == T_PGFLT) {
  void *address = (void *) rcr2();
  if (is_address_okay(myproc(), address)) {
      setup_page_table_entry_for(myproc(), address);
      // return from fault, retry access
  } else {
      // actual segfault, kill process
      cprintf("...");
      myproc()->killed = 1;
  }
}
```

xv6: if one handled page faults

alternative to crashi

- returning from pag

if so, setup the page table so it works next time

that is, immediately after returning from fault

"just in time" update of the process's memory

- example: don't actually allocate memory until it's needed

pseudocode for xv6 implementation (for trap())

```
if (tf->trapno == T_PGFLT) {
  void *address = (void *) rcr2();
  if (is_address_okay(myproc(), address)) {
      setup_page_table_entry_for(myproc(), address);
      // return from fault, retry access
  } else {
      // actual segfault, kill process
      cprintf("...");
      myproc()->killed = 1;
  }
}
```

page table base register / TLBs

so far: just change page table entries

two missing tasks:

changing page table base register:

- xv6: `lcr3` — done as part of process context switch (`switchuvm`)

resetting processor's page table entry cache when page table entries change

- page table entry cache called the 'TLB' (translation lookaside buffer)
- x86-32: reloading page table base register
- processor relies on OS to know when cached PTEs change

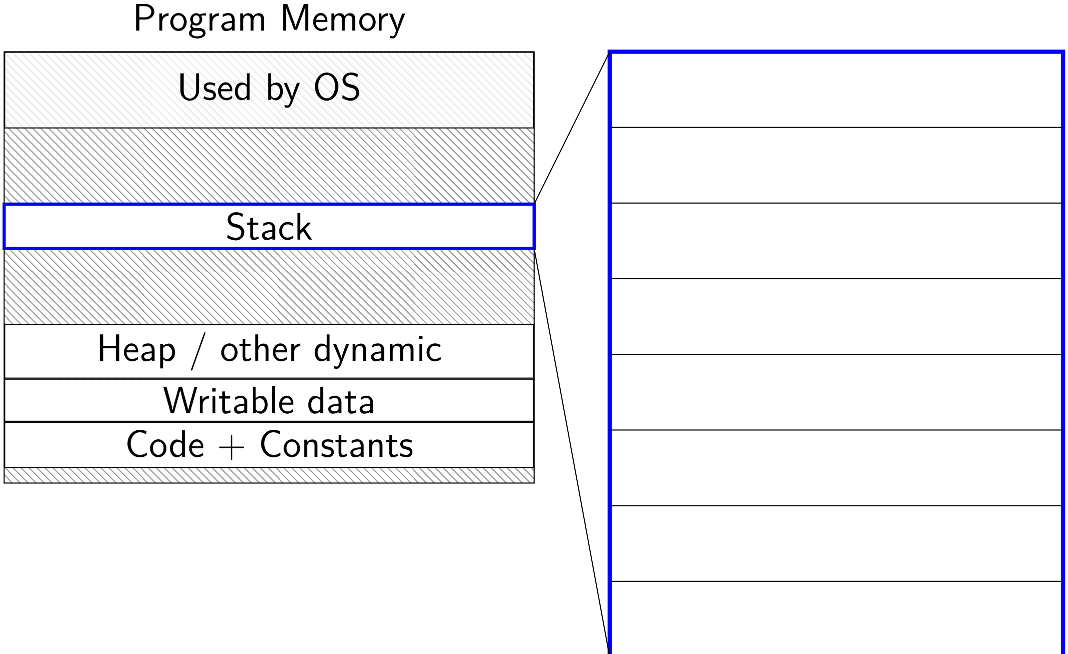
page fault tricks

OS can do all sorts of ‘tricks’ with page tables

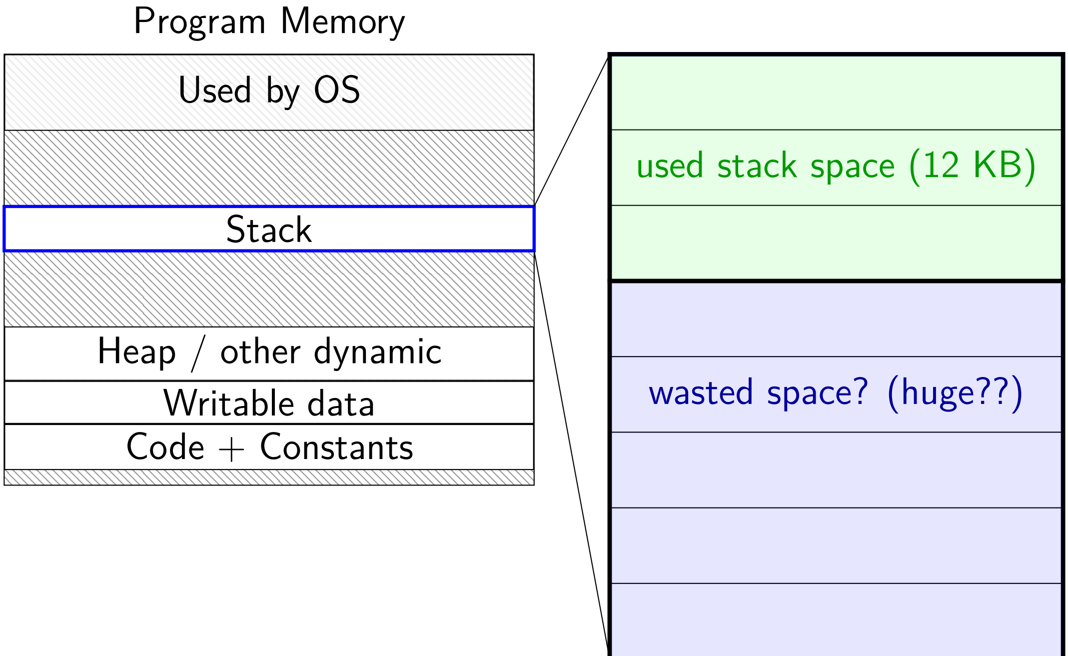
key idea: what processes *think* they have in memory != their actual memory

OS fixes disagreement from page fault handler

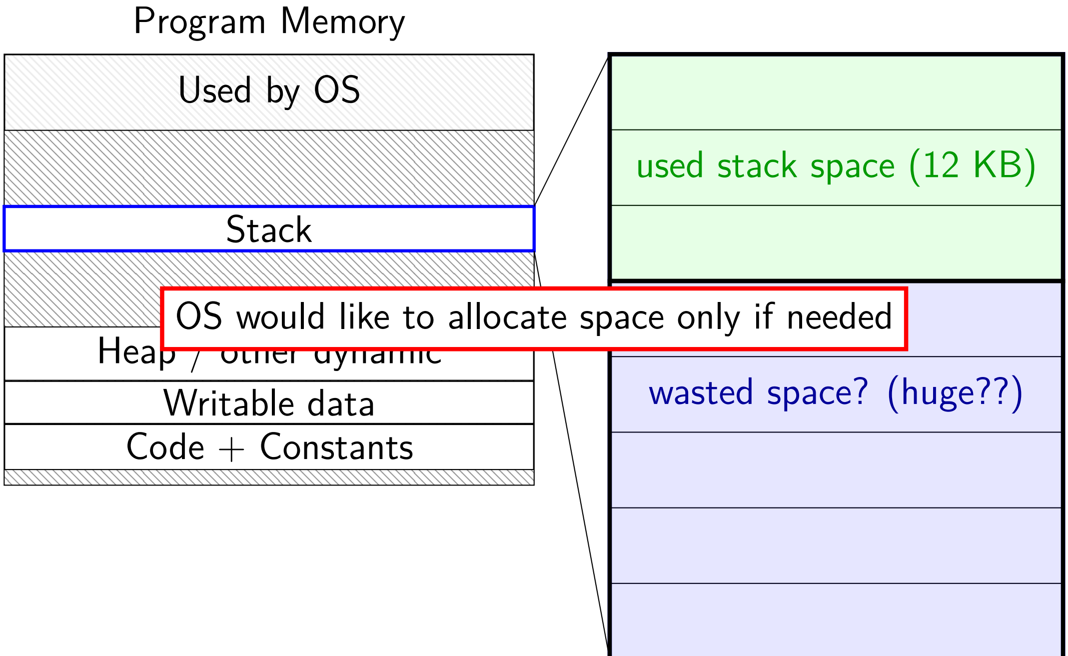
space on demand



space on demand



space on demand



OS would like to allocate space only if needed

allocating space on demand

`%rsp = 0x7FFFC000`

```
...
// requires more stack space
A: pushq %rbx

B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

VPN	valid?	physical page
...	...	...
0x7FFFB	0	---
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
...	...	...

allocating space on demand

`%rsp = 0x7FFFC000`

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

page fault!

VPN	valid?	physical page
…	…	…
0x7FFFB	0	---
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
…	…	…

pushq triggers exception
hardware says "accessing address `0x7FFFBFF8`"
OS looks up what's should be there — "stack"

allocating space on demand

`%rsp = 0x7FFFC000`

```
...
// requires more stack space
A: pushq %rbx

B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

restarted

VPN	valid?	physical page
…	…	…
0x7FFFB	1	0x200D8
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
…	…	…

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

space on demand really

common for OSes to allocate a lot space on demand

- sometimes new heap allocations
- sometimes global variables that are initially zero

benefit: malloc/new and starting processes is faster

also, similar strategy used to load programs on demand

- (more on this later)

future assigment: add allocate heap on demand in xv6

exercise

```
void foo() {
    char array[1024 * 128];
    for (int i = 0; i < 1024 * 128; i += 1024 * 16)
        array[i] = 100;
    }
}
```

4096-byte pages, stack allocated on demand, compiler optimizations don't omit the stores to or allocation of `array`, the compiler doesn't initialize `array`, and the stack pointer is initially a multiple of 4096.

How much physical memory is allocated for `array`?

A. 16 bytes
B. 64 bytes
C. 128 bytes
D. 4096 bytes $(4 \cdot 1024)$
E. 16384 bytes $(16 \cdot 1024)$
F. 32768 bytes $(32 \cdot 1024)$
G. 131072 bytes $(128 \cdot 1024)$
H. depends on cache block size
I. something else?

fast copies

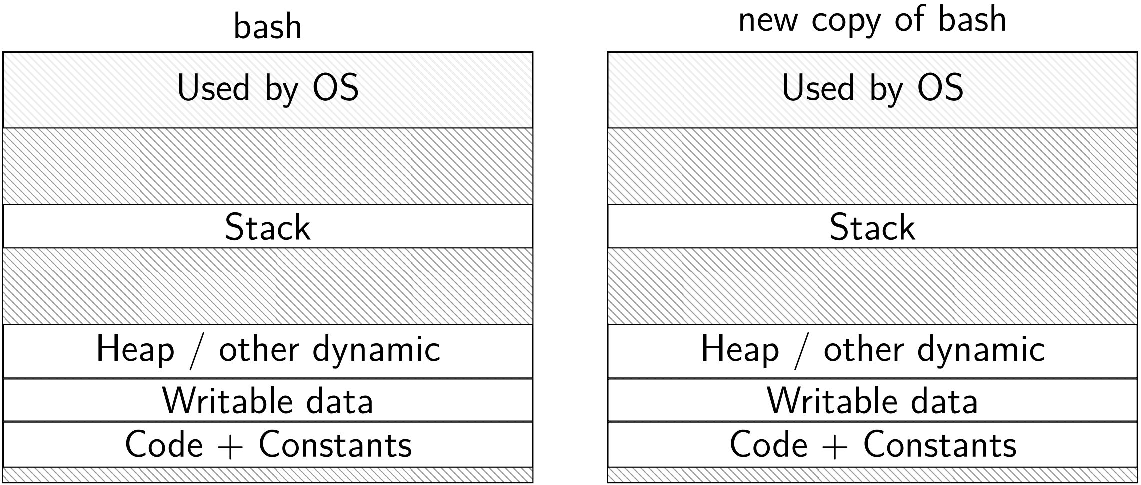
recall : `fork()`

creates a copy of an entire program!

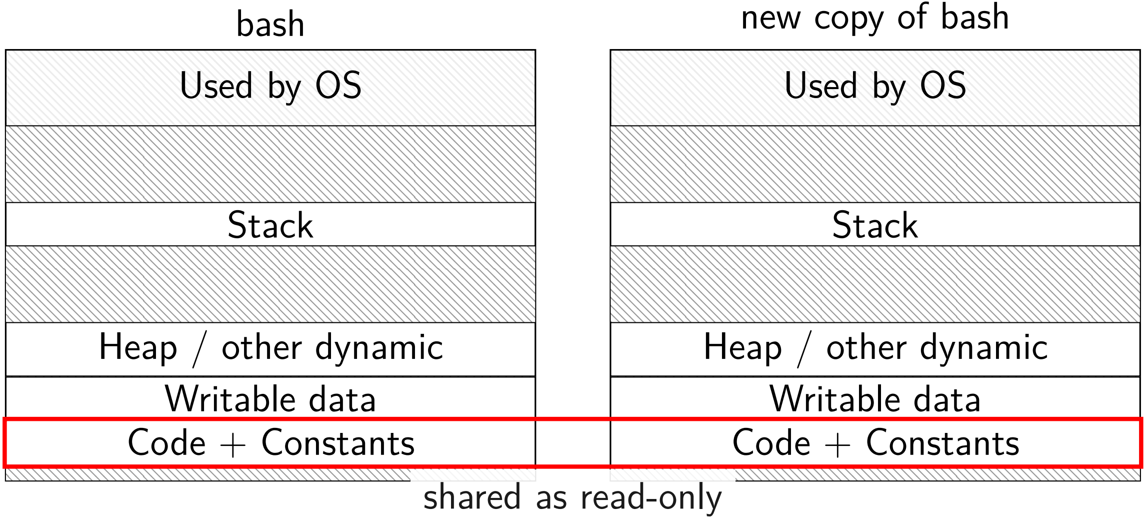
(usually, the copy then calls `execve` — replaces itself with another program)

how isn't this really slow?

do we really need a complete copy?



do we really need a complete copy?



bash

Used by OS

Stack

Heap / other dynamic

Writable data

Code + Constants

new copy of bash

Used by OS

Stack

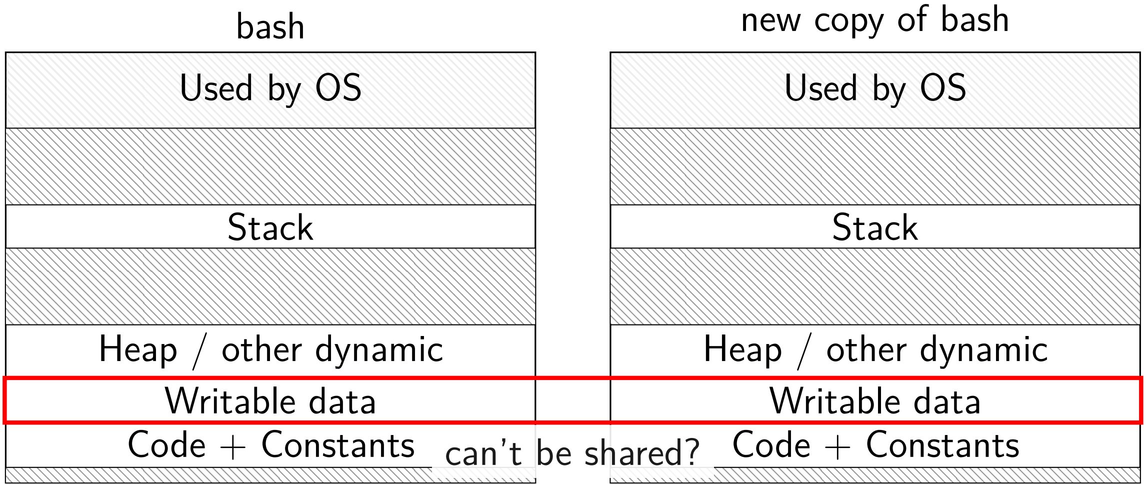
Heap / other dynamic

Writable data

Code + Constants

shared as read-only

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

exercise

Process with 4KB pages has this memory layout:

addresses	use
`0x0000-0x0FFF`	inaccessible
`0x1000-0x2FFF`	code (read-only)
`0x3000-0x3FFF`	global variables (read/write)
`0x4000-0x5FFF`	heap (read/write)
`0x6000-0xEFFF`	inaccessible
`0xF000-0xFFFF`	stack (read/write)

Process calls `fork()`, then child overwrites a 128-byte heap array and modifies an 8-byte variable on the stack.

After this, on a system with copy-on-write, how many physical pages must be allocated so both child+parent processes can read any accessible memory without a page fault?

xv6: adding space on demand

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

xv6 tracks "end of heap" (now just for `sbrk()`)

adding allocate on demand logic for the heap:

on sbrk(): don't change page table right away

on page fault

- case 1: if address $\geq$ sz: out of bounds: kill process
- case 2: otherwise, allocate page containing address, return from trap

versus more complicated OSes

typical desktop/server:
range of valid addresses is not just 0 to maximum

need some more complicated data structure to represent

copy-on write cases

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

- kill program instead of making it writable

fault from trying to write read-only page:

case 1: multiple process's page table entries refer to it

- copy the page
- replace read-only page table entry to point to copy

case 2: only one page table entry refers to it

- make it writeable

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 additonal 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)

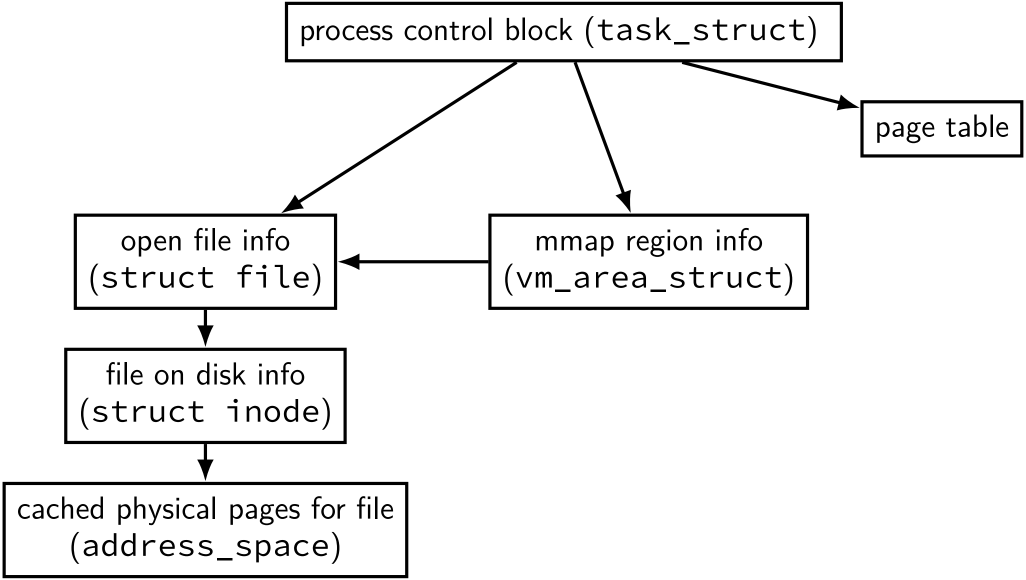
```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

`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

backup slides

Linux: forward mapping



process control block (`task_struct`)

page table

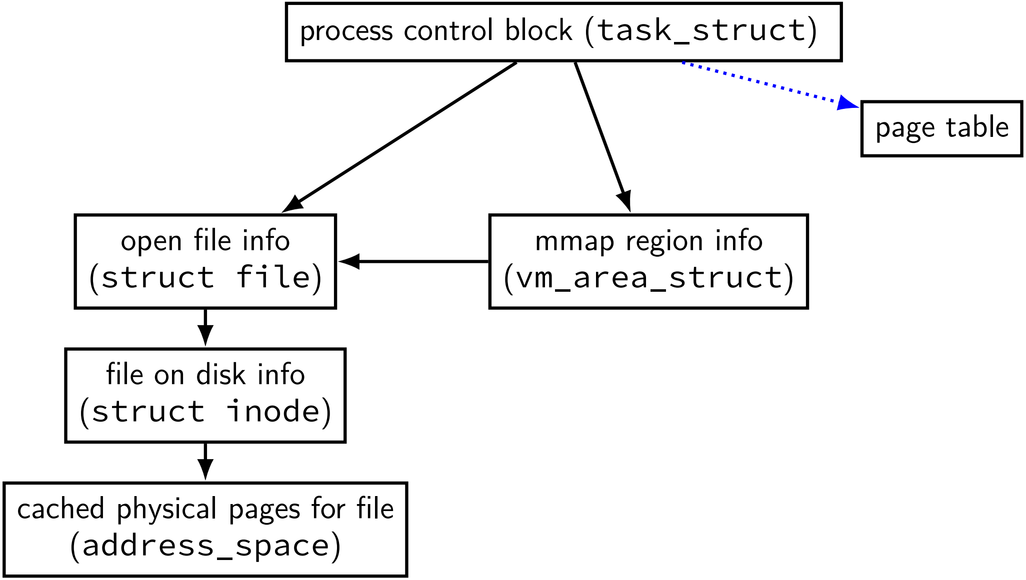
open file info (`struct file`)

mmap region info (`vm_area_struct`)

file on disk info (`struct inode`)

cached physical pages for file (`address_space`)

Linux: forward mapping



process control block (`task_struct`)

page table

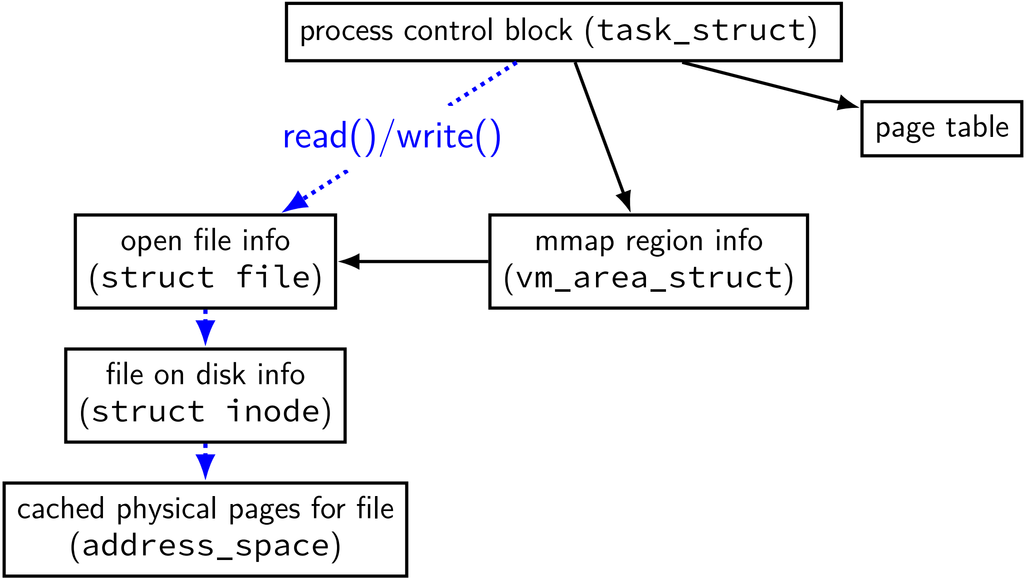
open file info (`struct file`)

mmap region info (`vm_area_struct`)

file on disk info (`struct inode`)

cached physical pages for file (`address_space`)

Linux: forward mapping



process control block (`task_struct`)

page table

read()/write()

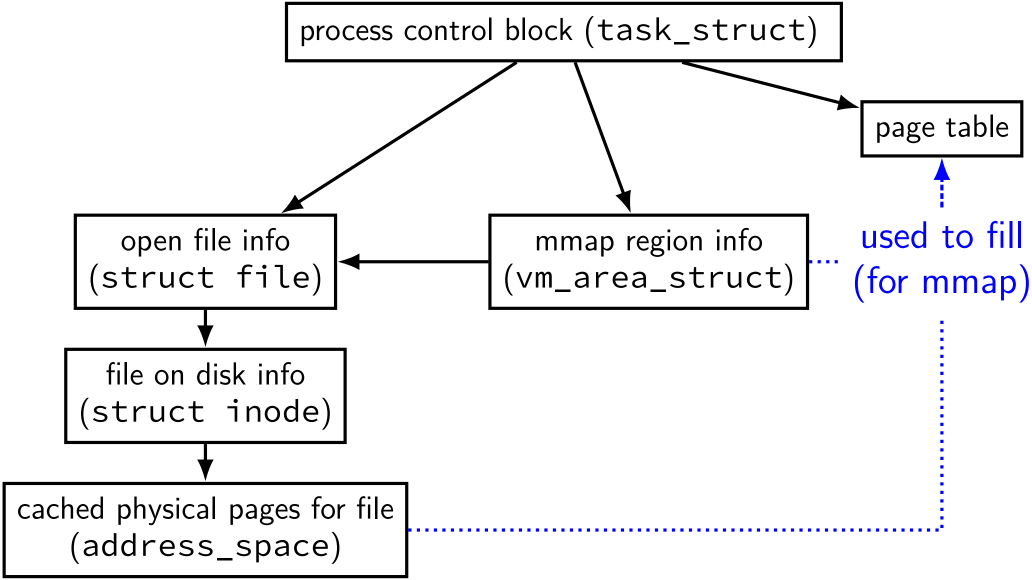
open file info (`struct file`)

mmap region info (`vm_area_struct`)

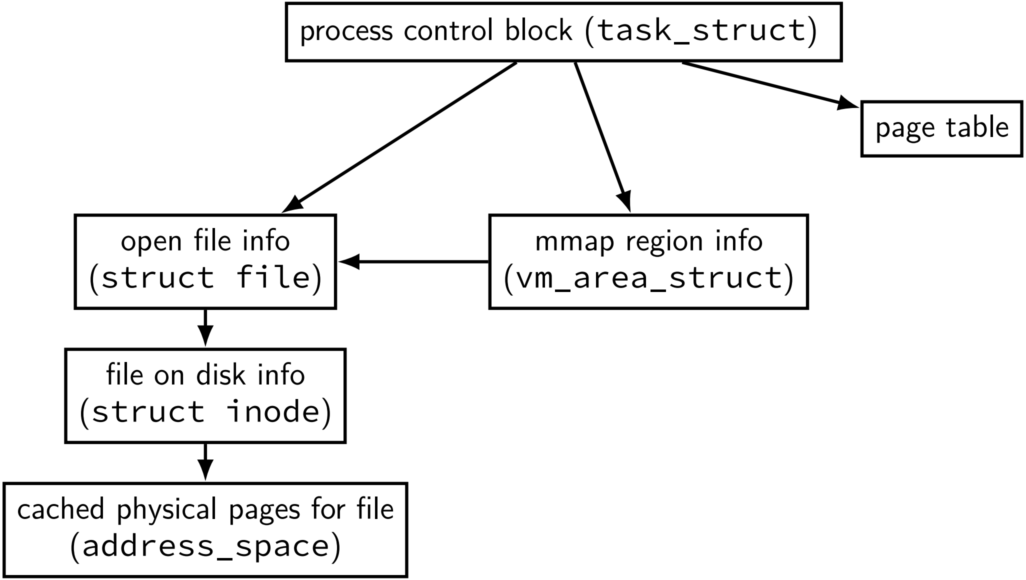
file on disk info (`struct inode`)

cached physical pages for file (`address_space`)

Linux: forward mapping



Linux: forward mapping



process control block (`task_struct`)

page table

open file info (`struct file`)

mmap region info (`vm_area_struct`)

file on disk info (`struct inode`)

cached physical pages for file (`address_space`)

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

- `allocuvm()` in loop
- new physical pages chosen by `kalloc()`

exec step 2b: load from executable file

- copying from executable file implemented by `loaduvm()`

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

- `allocuvm()` in loop
- new physical pages chosen by `kalloc()`

exec step 2b: load from executable file

- copying from executable file implemented by `loaduvm()`

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