last time

walkpgdir: find pointer to (second-level) PTE
- given virtual address (user would access)
- takes kernel pointer (virtual addr in top half) to first-level PT
- returns kernel pointer to entry

mappages(pgdir, VA, size, PA,...): set [VA, VA + size) to map to particular [PA, PA+size)
- calls walkpgdir() to access each second-level page table entry in range
- if range *partially* overlaps page, maps the whole page
- sets page table entry to present + points-to-specified PA
- stock xv6: assumes pages mapped exactly once

allockvm: make new page table (kernel part)

allocuvm: allocate user pages

kalloc/kfree: memory allocation in kernel (page from linked list)

mappages rounding note

mappages(pgdir, 0x4000, 0x1000, 0x50000, ...):

- sets VPN 0x4 (virtual addreses 0x4000-0x4FFF) to map to PPN 0x50 (physical addresses 0x50000-0x50FFF)

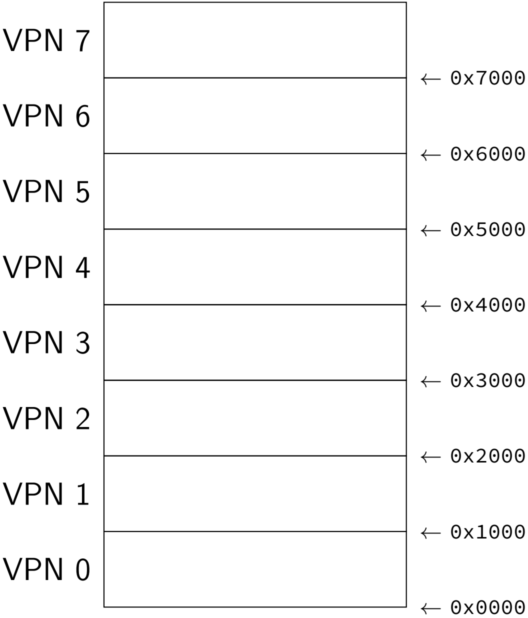
mappages(pgdir, 0x4000, 0x2000, 0x50000, ...):

- sets VPN 0x4 (virtual addreses 0x4000-0x4FFF) to map to PPN 0x50
- sets VPN 0x5 (virtual addreses 0x5000-0x5FFF) to map to PPN 0x50

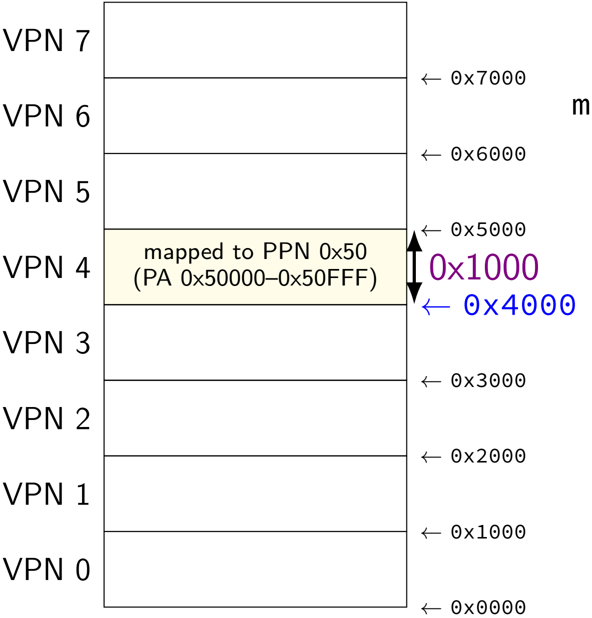
mappages(pgdir, 0x4200, 0x1000, 0x50800, ...):

- sets VPN 0x4 (virtual addreses 0x4000-0x4FFF) to map to PPN 0x50 (physical addresses 0x50000-0x50FFF)
- sets VPN 0x5 (virtual addreses 0x5000-0x5FFF) to map to PPN 0x50 (physical addresses 0x50000-0x50FFF)

mappages

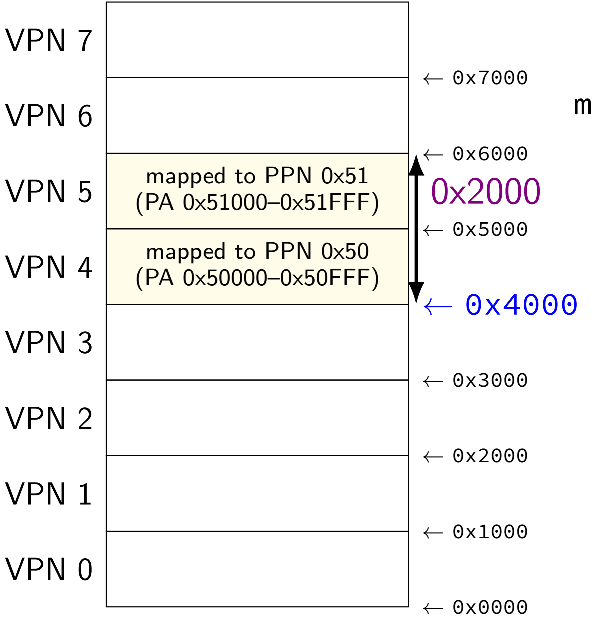

VPN 7
VPN 6
VPN 5
VPN 4
VPN 3
VPN 2
VPN 1
VPN 0
← 0x7000
← 0x6000
← 0x5000
← 0x4000
← 0x3000
← 0x2000
← 0x1000
← 0x0000

mappages



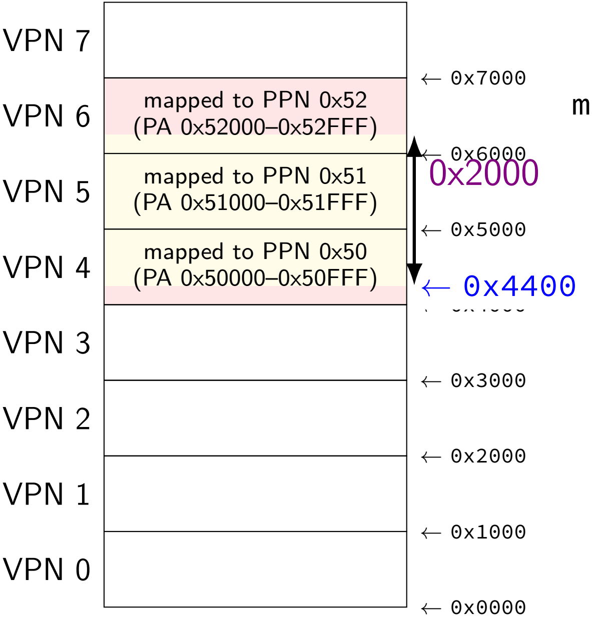

```
mappages(pgdir, 0x4000,
                0x1000, 0x50000, …)
```

mappages



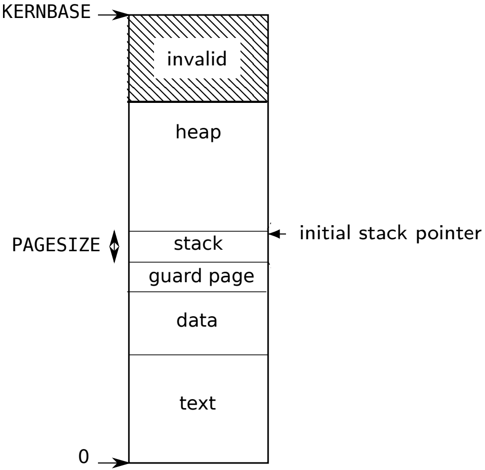

```
mappages(pgdir, 0x4000,
                0x2000, 0x50000, …)
```

mappages

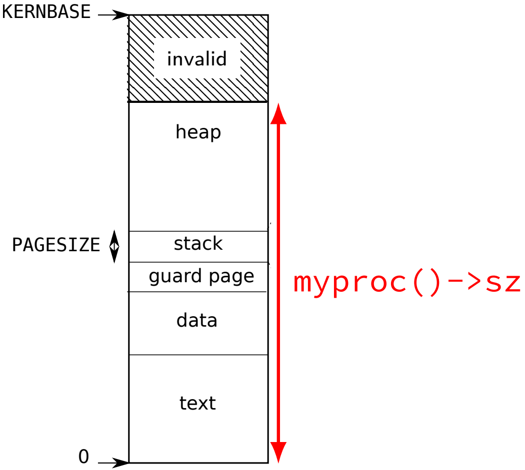



```
mappages(pgdir, 0x4400,
         0x2000, 0x50400, …)
```

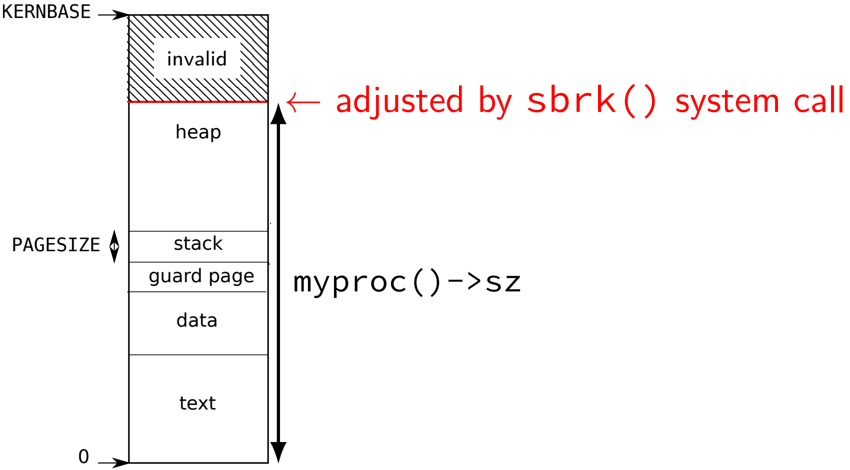
xv6 program memory



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

sz: current top of heap

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

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;
}
```

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

- processor relies on OS to know when cached PTEs change
- x86-32: can be done by reloading page table base register
- why growproc() calls switchvum()

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 crashin[g]

returning from pag[e fault]

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;
  }
}
```

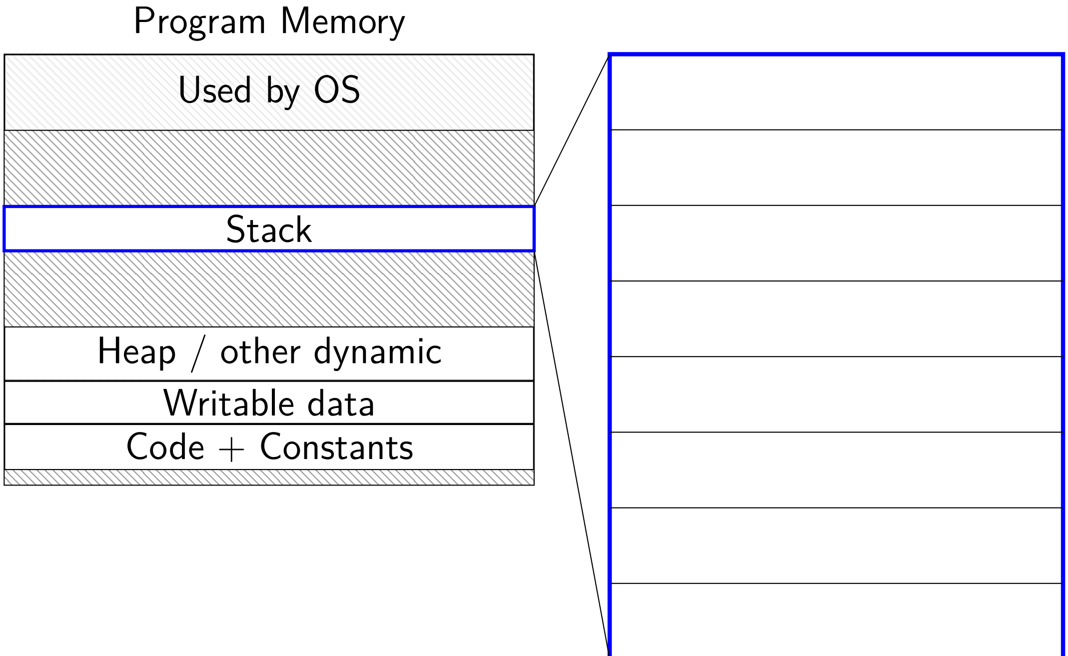
page fault tricks

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

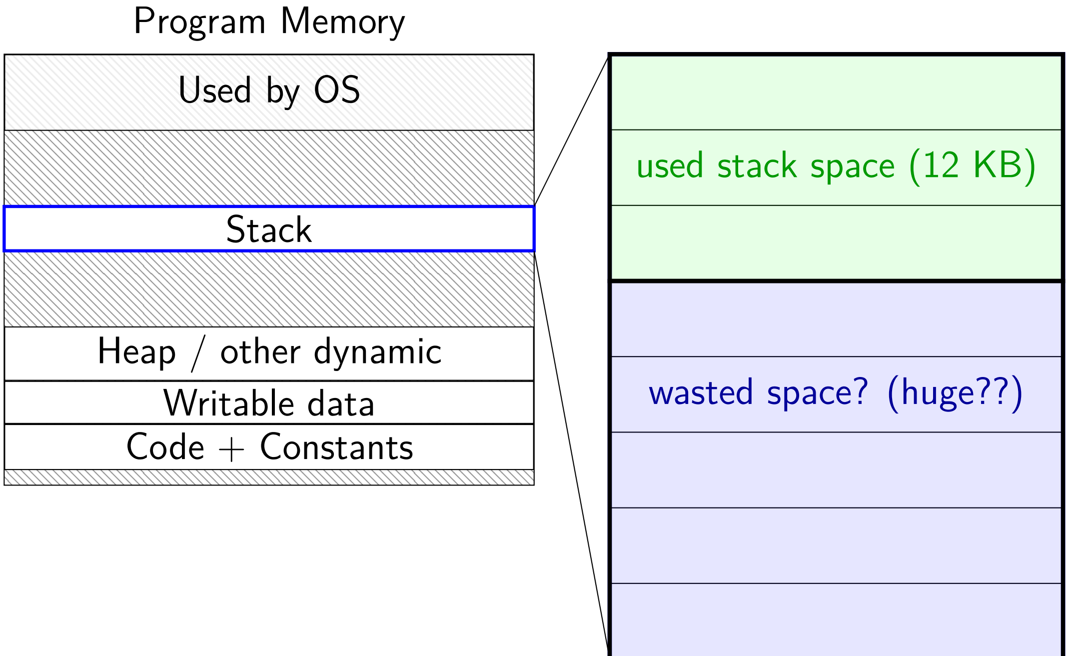
key idea: what processes *think* they have in memory $!=$ their actual memory

OS fixes disagreement from page fault handler

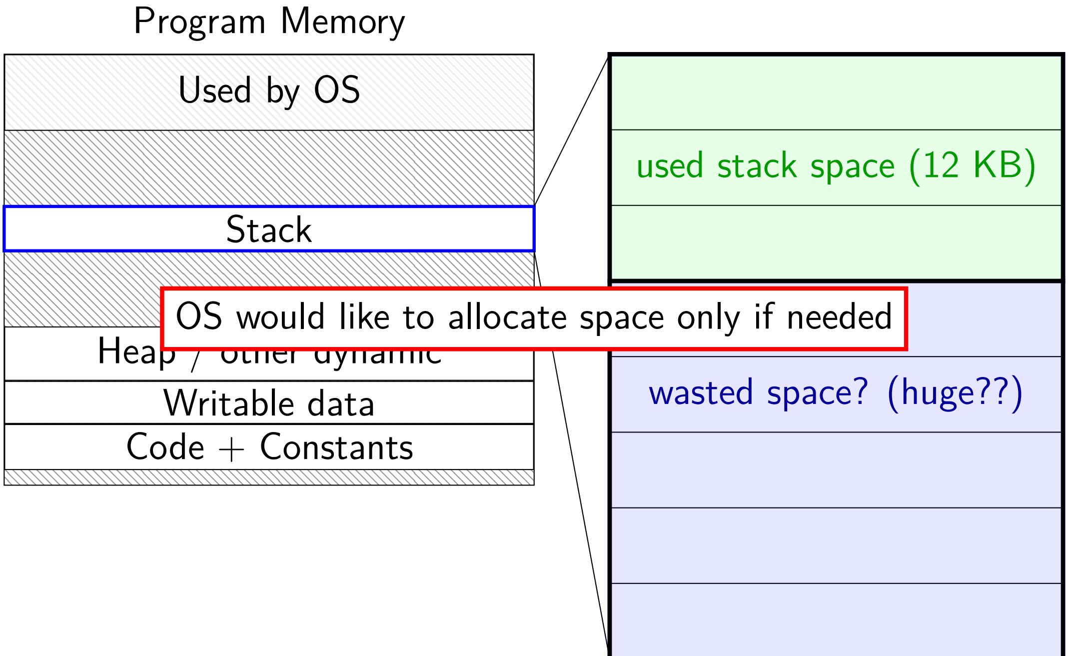
space on demand



space on demand



space on demand



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

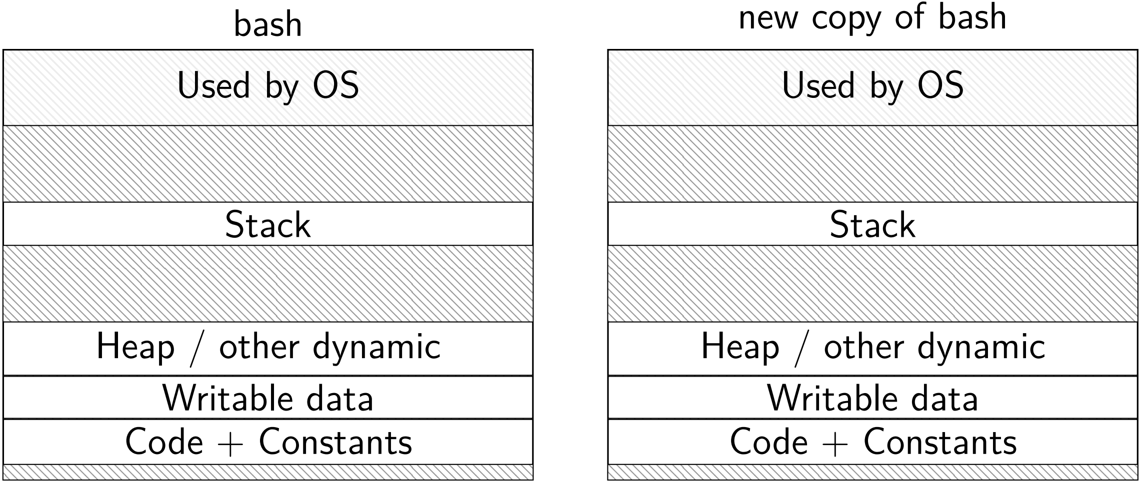
recall : `fork()`

creates a copy of an entire program!

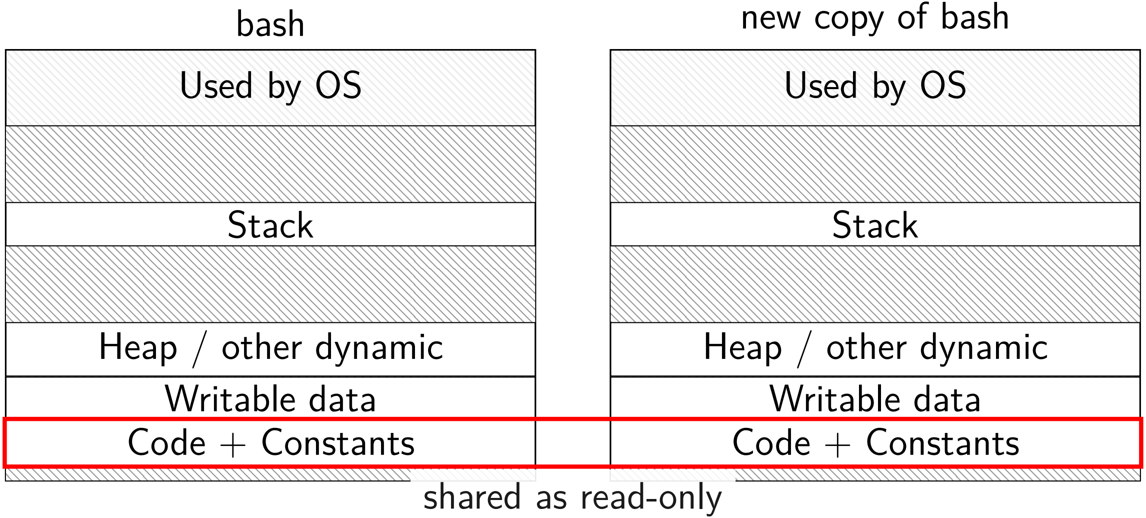
(usually, the copy then calls `execve` — replaces itself with another program)

how isn't this really slow?

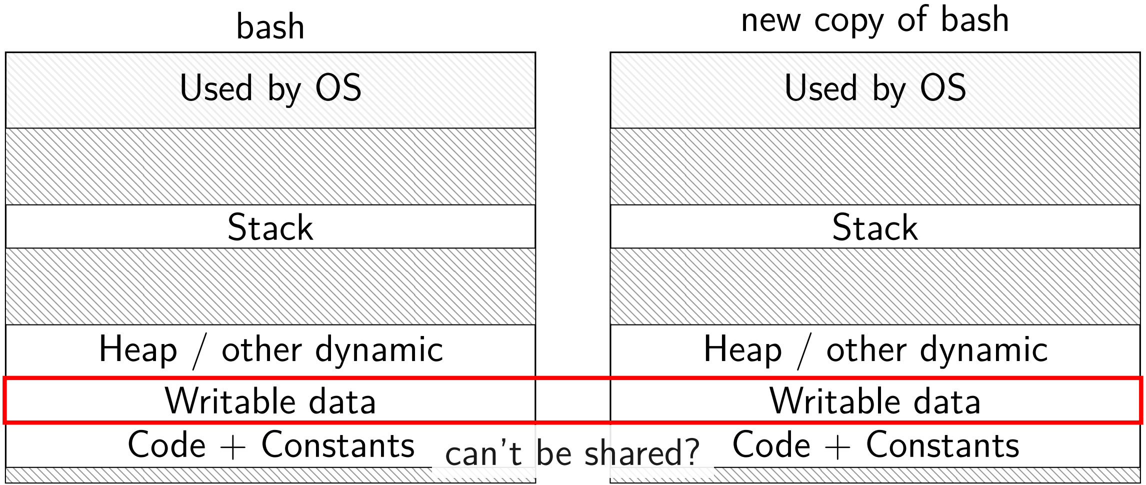
do we really need a complete copy?



do we really need a complete copy?



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
```

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							PCD	PWT	Ignored			**CR3**
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	**PDE: 4MB page**
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**
Ignored																															0	**PDE: not present**
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	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

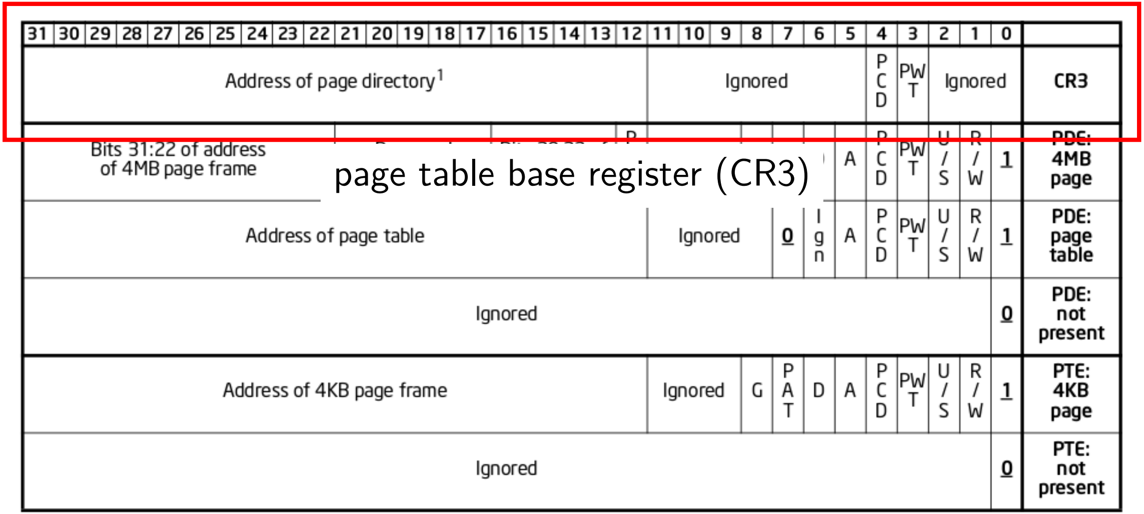


Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

first-level page table entries

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td colspan="20">Address of pag</td><td colspan="7"></td><td>PCD</td><td>PWT</td><td colspan="3">Ignored</td><td>CR3</td></tr>
<tr><td colspan="10">Bits 31:22 of address of 4MB page frame</td><td colspan="5">Reserved (must be 0)</td><td colspan="4">Bits 39:32 of address[2]</td><td>PAT</td><td colspan="3">Ignored</td><td>G</td><td>1</td><td>D</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PDE: 4MB page</td></tr>
<tr><td colspan="20">Address of page table</td><td colspan="4">Ignored</td><td>0</td><td>Ign</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PDE: page table</td></tr>
<tr><td colspan="31">Ignored</td><td>0</td><td>PDE: not present</td></tr>
<tr><td colspan="20">Address of 4KB page frame</td><td colspan="3">Ignored</td><td>G</td><td>PAT</td><td>D</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PTE: 4KB page</td></tr>
<tr><td colspan="31">Ignored</td><td>0</td><td>PTE: not present</td></tr>
</table>

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							PCD	PWT	Ignored			**CR3**
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	**PDE: 4MB page**
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**
second-level page table entries																															0	**PDE: not present**
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	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 entry v addresses

physical page number	zeros	phys. page byte addr

Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored										0	**PTE: not present**

flags

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 overriden in next level)\n");
  }
  if (top_level_pte & PTE_U) {
    cprintf("is user-accessible (may be overriden 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 overriden in next level)\n");
  }
  if (top_level_pte & PTE_U) {
    cprintf("is user-accessible (may be overriden 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 overriden in next level)\n");
  }
  if (top_level_pte & PTE_U) {
    cprintf("is user-accessible (may be overriden 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 overriden in next level)\n");
  }
  if (top_level_pte & PTE_U) {
    cprintf("is user-accessible (may be overriden 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 overriden in next level)\n");
  }
  if (top_level_pte & PTE_U) {
    cprintf("is user-accessible (may be overriden 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)

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)

```
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)

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

```
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;
}
```

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

> `sz` — top of heap of new program
> name of the field in `struct proc`

loading user pages from executable

```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uin
{
  ...
  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

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

```
loaduvm(pde_t *pgdir, char *addr                              , uin
{
  ...
  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

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

```
loaduvm(pde_t *pgdir, ch                                        , uin
{
  ...
  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

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

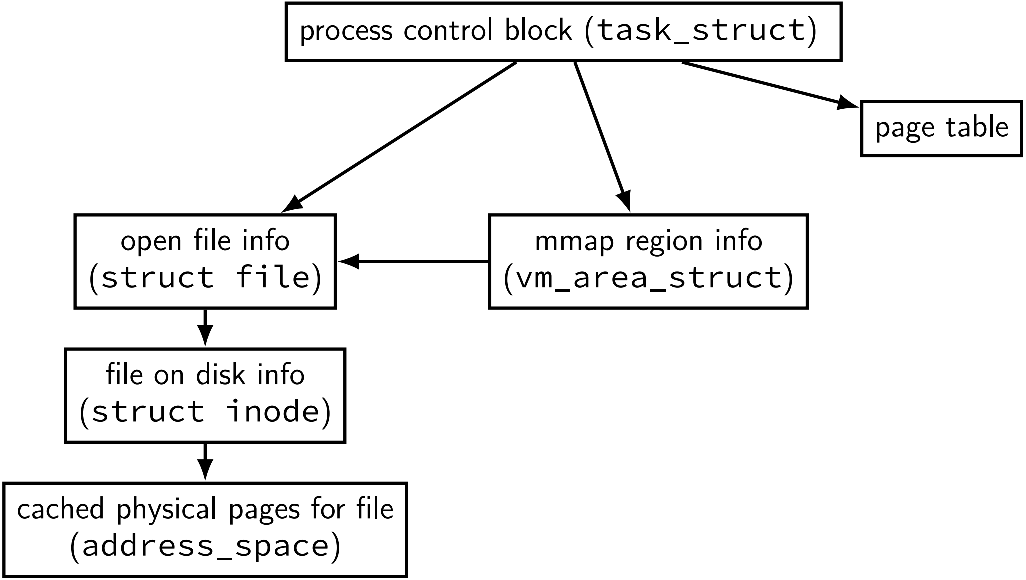
```
loaduvm(pde_t *pgdir                                            , uir
{
  ...
  for(i = 0; i < sz;
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip,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

loading user pages from executable

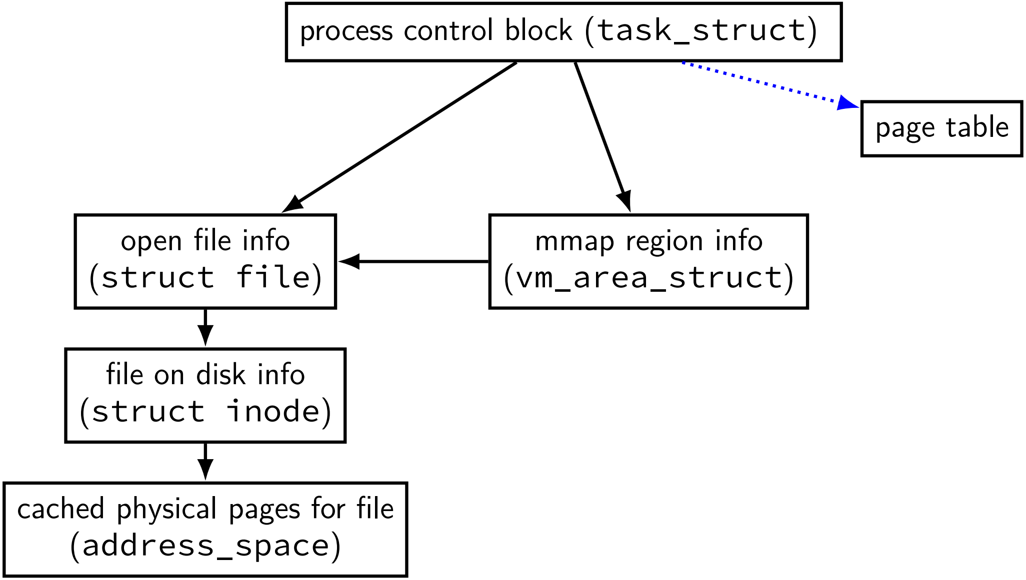
copy from file (represented by `struct inode`) into memory
P2V(pa) — mapping of physical addresss in kernel memory

```
loaduv                                                                    , uir
{
  ...
  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;
}
```

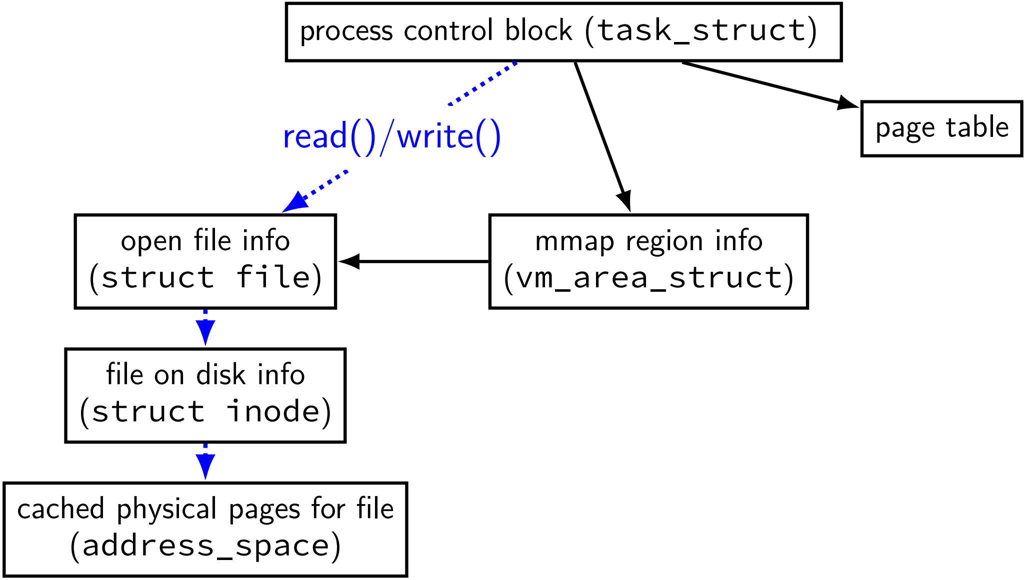
Linux: forward mapping



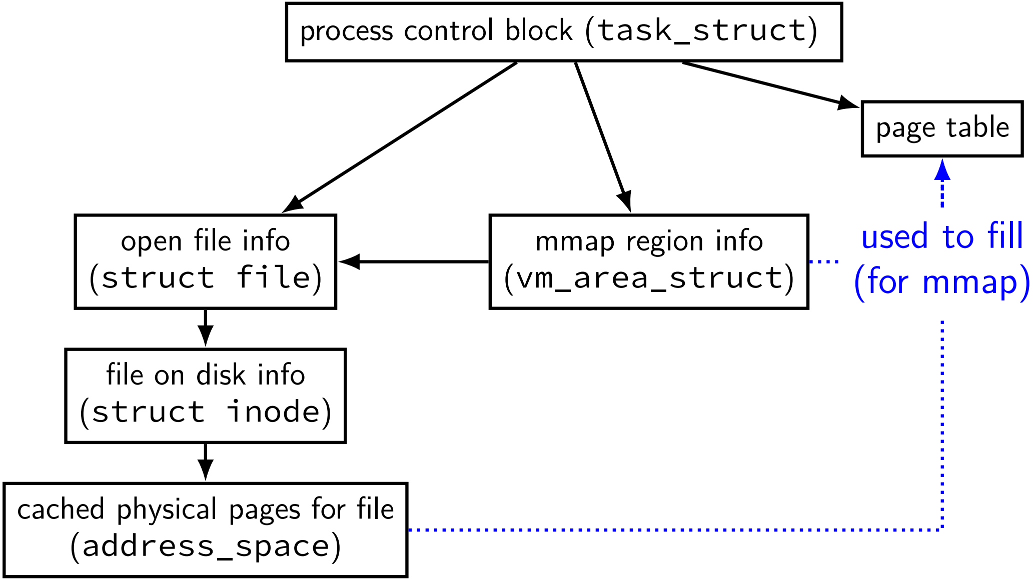
Linux: forward mapping



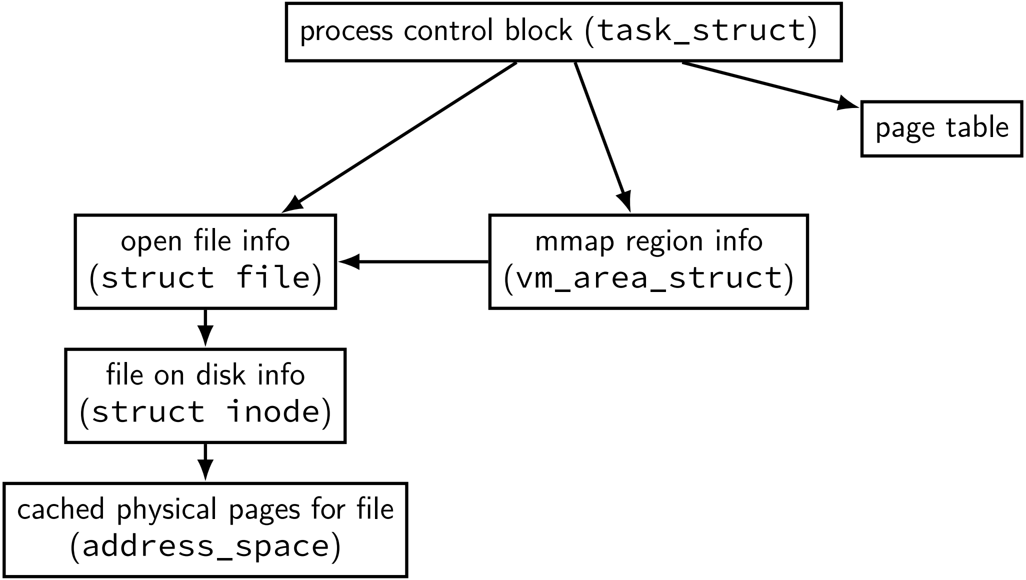
Linux: forward mapping



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

- `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

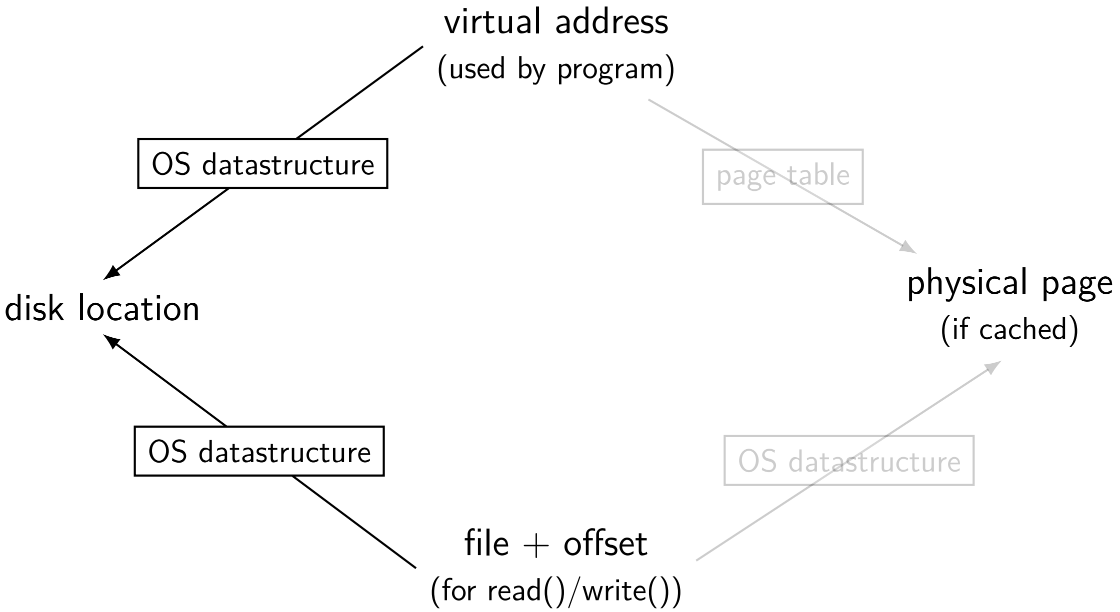
HDD reads and writes: milliseconds to tens of milliseconds

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

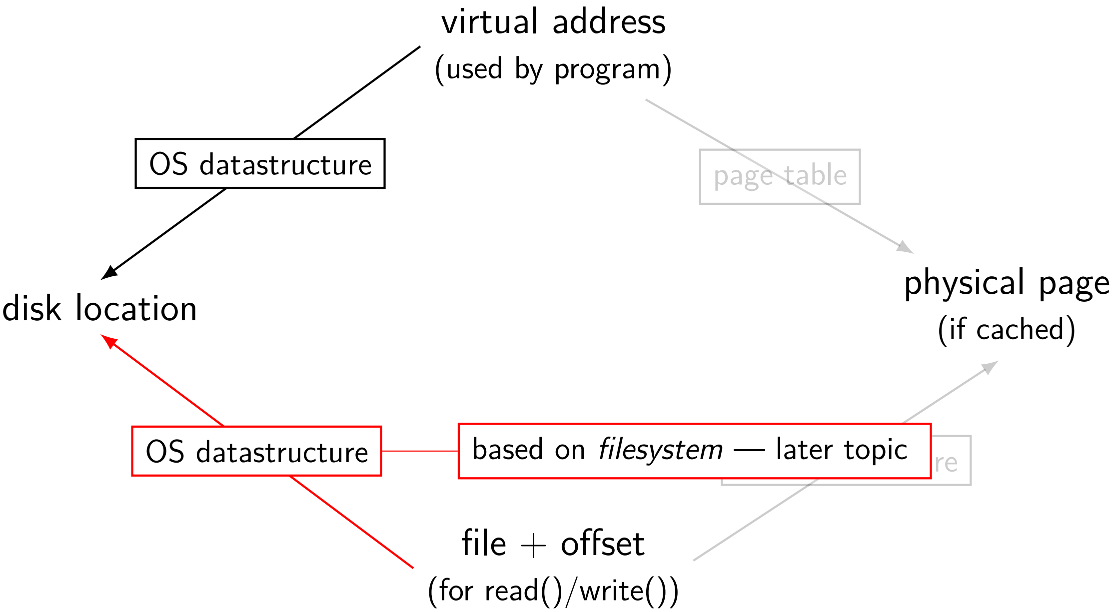
SSD writes and writes: hundreds of microseconds

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

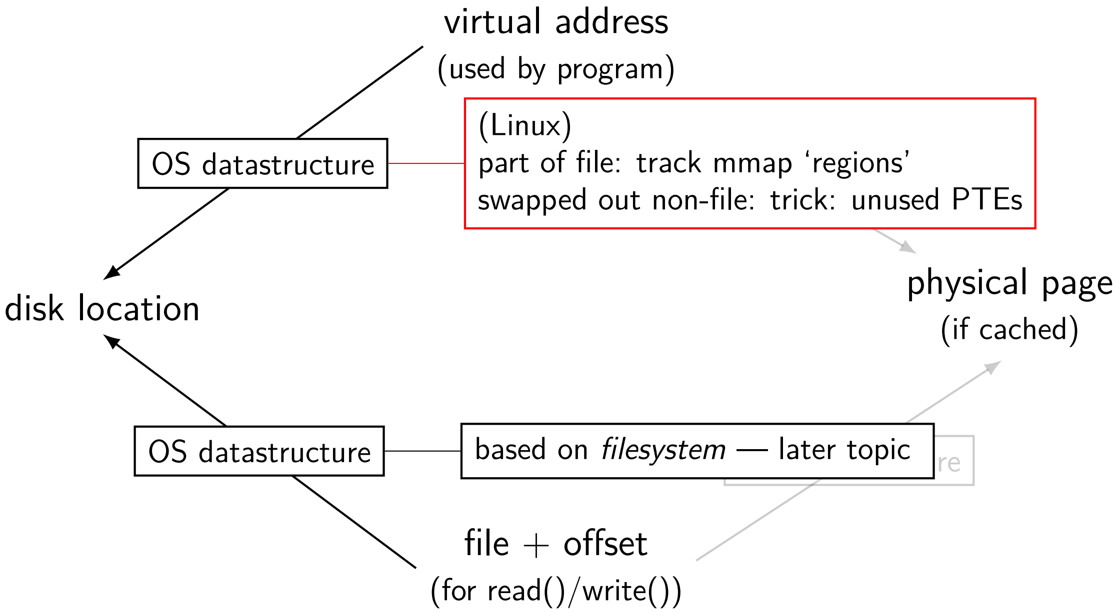
virtual address/file offset → location on disk



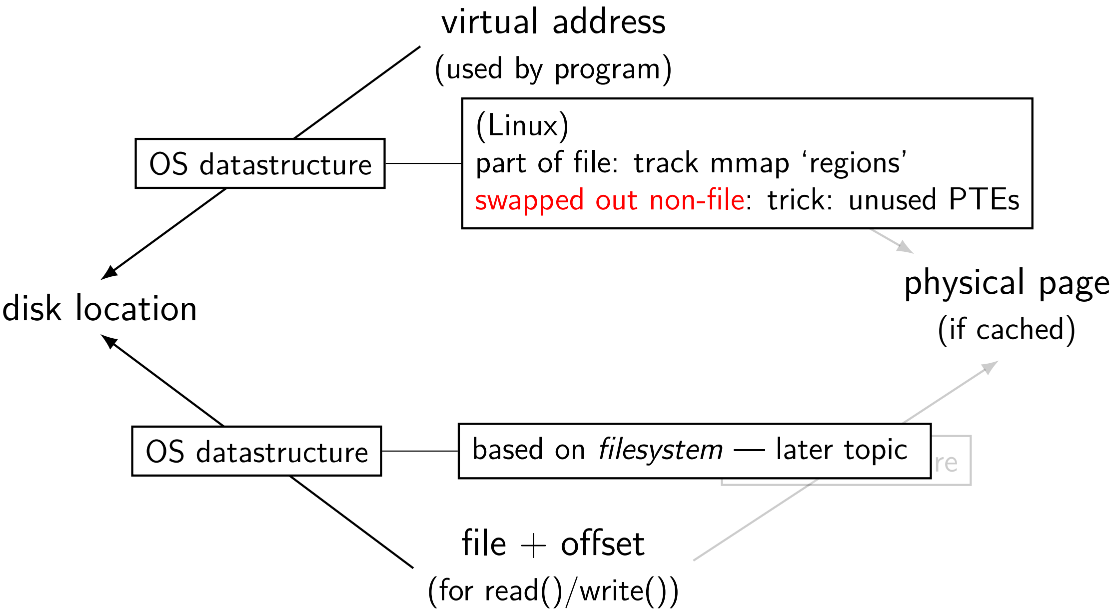
virtual address/file offset → location on disk



virtual address/file offset → location on disk



virtual address/file offset → location on disk



Linux: tracking swapped out pages

need to lookup location on disk

potentially one location for every virtual page

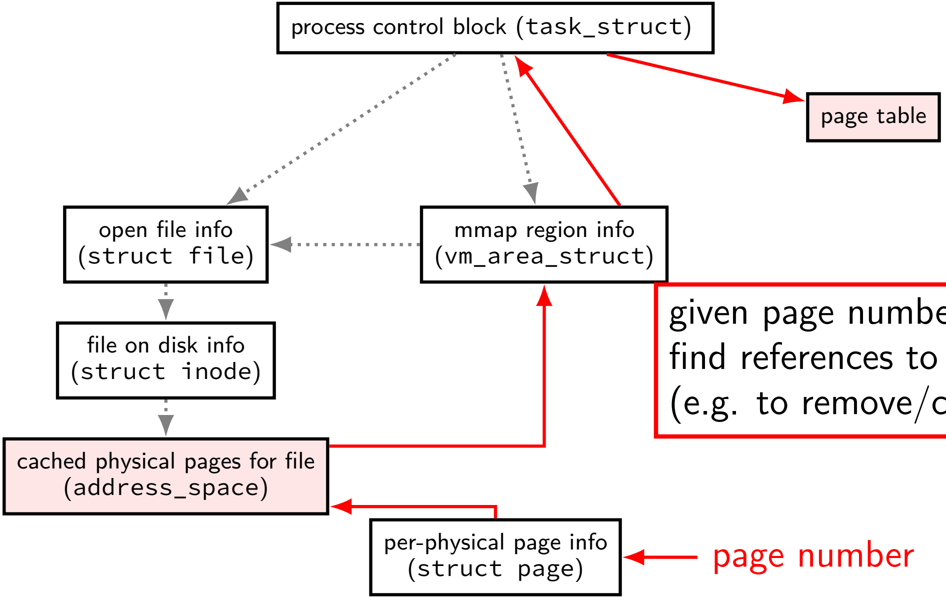
trick: store location in "ignored" part of page table entry

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

Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	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

Linux: reverse mapping (file pages)



given page number
find references to that page
(e.g. to remove/change them)

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

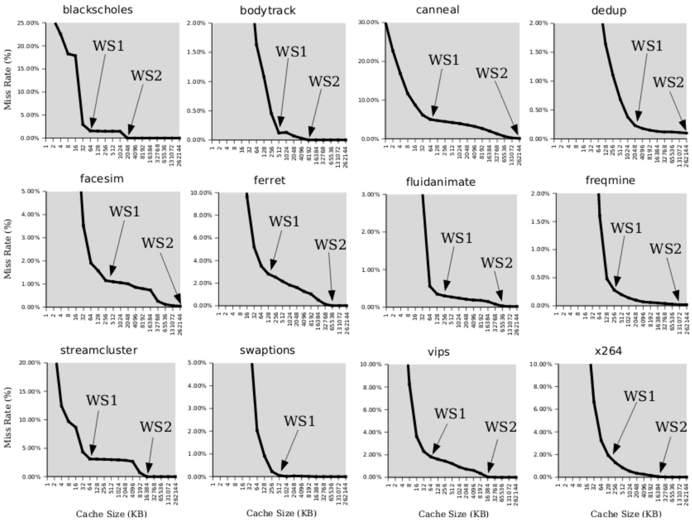


Figure 3: Miss rates versus cache size. Data assumes a shared 4-way associative cache with 64 byte lines. WS1 and WS2 refer to important working sets which we analyze in more detail in Table 2. Cache requirements of PARSEC benchmark programs can reach hundreds of megabytes.

Bienia et al, "The PARSEC Benchmark Suite: Characterization and Architectural Implications"

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)

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

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.

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)

…

update page info +

mark present

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)
…

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

the kernel

…
(OS exception's handler)
…

oops! page fault

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
```

the kernel

…
(OS exception's handler)
…

update page info +
mark present

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 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
```

processor does lookup

sets accessed bit to 1

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
```

processor does lookup

sets accessed bit to 1

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
…	…	…	…	…	…

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)
…

OS reads + records + clears access bit

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)
...

OS reads + records + clears access bit

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	0	0	…	0x4442
…	…	…	…	…	…

accessed bit usage (hardware support)

program 1

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

processor does lookup

sets accessed bit to 1 (again)

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
…	…	…	…	…	…

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	PAT	D	A	PCD	PWT	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: acccessed — 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`

`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