

last time

deadlock definition

- shared resources
- hold and wait
- circular waiting

deadlock prevention

- never wait — abort attempt to get resource, revoke resource
- avoid circular waiting: consistent order to get resource

virtual memory: mapping program addresses to real addresses

program addresses = virtual

real addresses = physical

fixed-sized pages (xv6: 4KB)

lower bits of address: page offset

upper bits: (virtual or physical) page number (= VPN or PPN)

page table: table with row for each VPN w/ PPN + permissions

two-level page tables

exercise (1)

4096 ($= 2^{12}$) byte pages

4-byte page table entries — stored in memory

two-level table:

first 10 bits lookup in first level (“page directory”)

second 10 bits lookup in second level

exercise:

virtual address 0x12345678

base pointer 0x1000 (byte address)

first-level PTE *contents*: PPN 0x14; second-level PTE: PPN 0x15

address of 1st-level PTE? of second-level PTE?

exercise (2)

4096 ($= 2^{12}$) byte pages

4-byte page table entries — stored in memory

two-level table:

- first 10 bits lookup in first level (“page directory”)

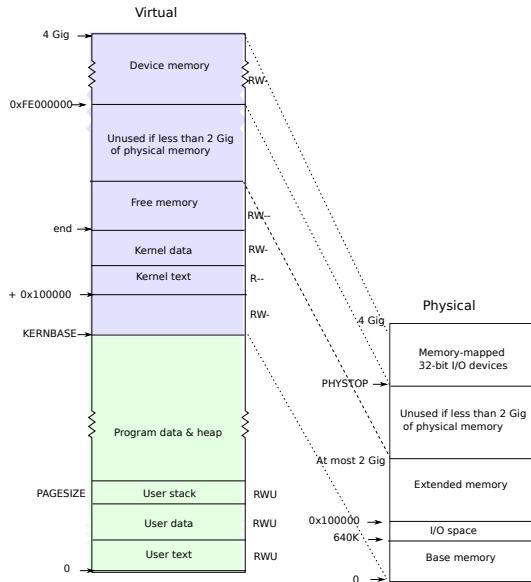
- second 10 bits lookup in second level

exercise: how big is...

- a process's x86-32 page tables with 1 valid 4K page?

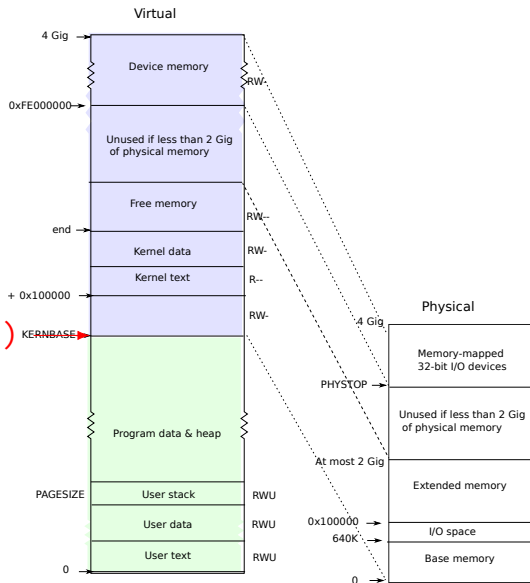
- a process's x86-32 page tables with all 4K pages populated?

xv6 memory layout

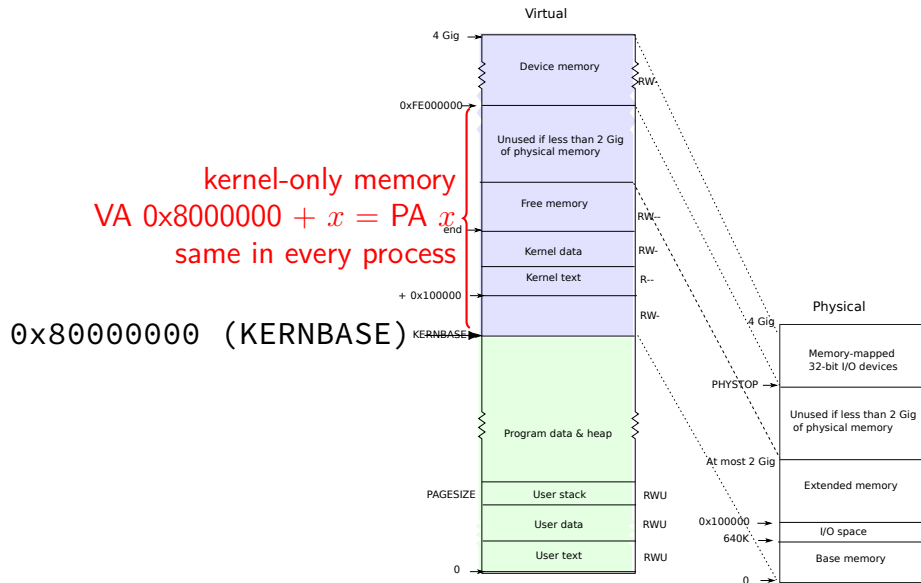


xv6 memory layout

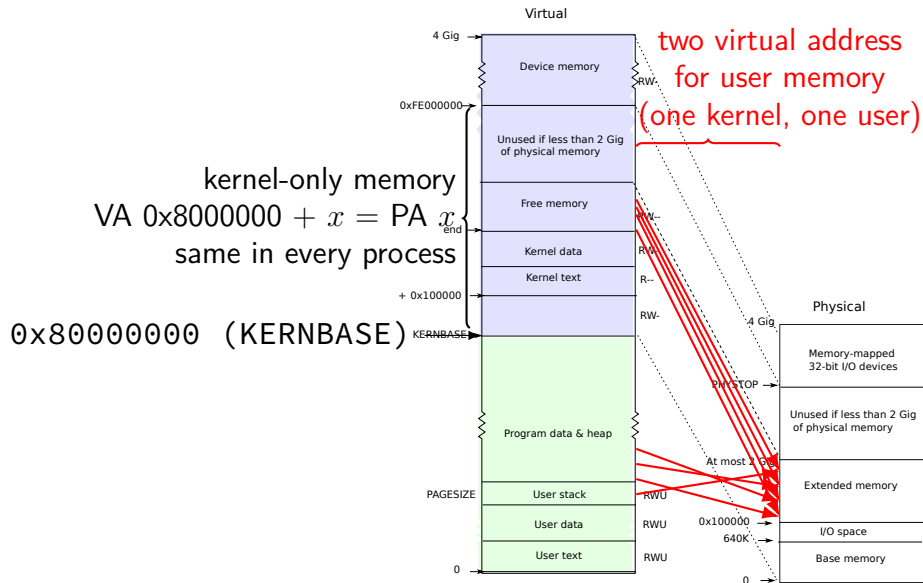
0x80000000 (KERNBASE)



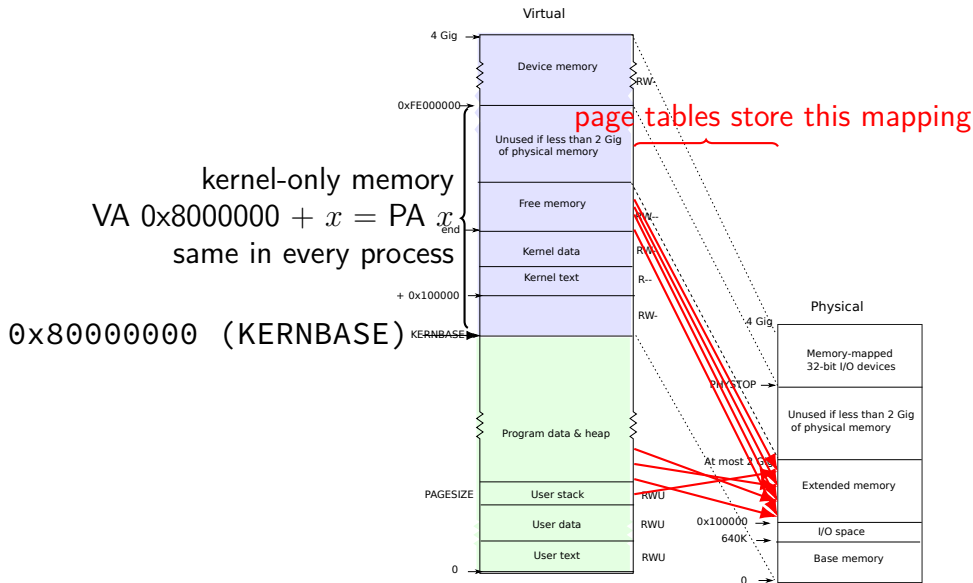
xv6 memory layout



xv6 memory layout



xv6 memory layout



xv6 kernel memory

virtual memory $>$ KERNBASE ($0x8000\ 0000$) is for kernel

always mapped as kernel-mode only

protection fault for user-mode programs to access

physical memory address 0 is mapped to $\text{KERNBASE}+0$

physical memory address N is mapped to $\text{KERNBASE}+N$

not done by hardware — just page table entries OS sets up on boot
very convenient for manipulating page tables with physical addresses

kernel code loaded into contiguous physical addresses

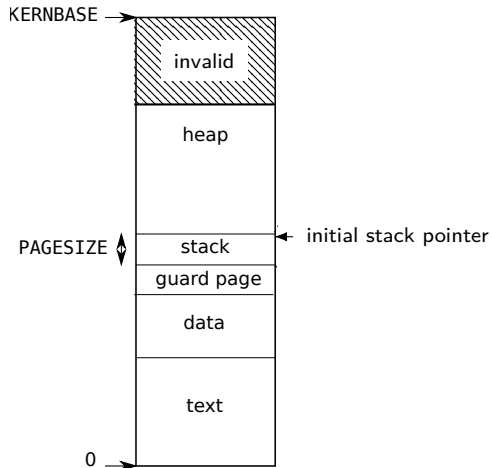
why two mappings?

program memory: layout programs expect
sized based on executable, heap allocations
uses any available memory

kernel code: access to all memory

kernel code: easy translation of physical to virtual addresses
e.g. page table setup: want to use particular physical addresses
no x86 instruction to read/write value using physical address only

xv6 program memory



guard page

1 page after stack

at lower addresses since stack grows towards lower addresses

marked as kernel-mode-only

idea: stack overflow → protection fault → kills program

xv6 types for paging (1)

virtual addresses: pointers (`void*`, etc.)

physical addresses: ints

P2V/V2P

V2P(x) (virtual to physical)

convert *kernel* address x to physical address

subtract KERNBASE (0x8000 0000)

assumes you pass a kernel address

have user address? need full page table lookup instead

P2V(x) (physical to virtual)

convert *physical* address x to kernel address

add KERNBASE (0x8000 0000)

xv6 convention: virtual addresses represented using pointers

xv6 convention: physical addresses represented using integers

P2V/V2P

V2P(x) (virtual to physical)

convert *kernel* address x to physical address

subtract KERNBASE (0x8000 0000)

assumes you pass a kernel address

have user address? need full page table lookup instead

P2V(x) (physical to virtual)

convert *physical* address x to kernel address

add KERNBASE (0x8000 0000)

xv6 convention: virtual addresses represented using pointers

xv6 convention: physical addresses represented using integers

P2V/V2P

V2P(x) (virtual to physical)

convert *kernel* address x to physical address

subtract KERNBASE (0x8000 0000)

assumes you pass a kernel address

have user address? need full page table lookup instead

P2V(x) (physical to virtual)

convert *physical* address x to kernel address

add KERNBASE (0x8000 0000)

xv6 convention: virtual addresses represented using pointers

xv6 convention: physical addresses represented using integers

xv6 types for paging (2)

x86-32 (as used by xv6) has 4-byte page table entries

page table entries, first-level: `pde_t`

- page directory entry
- alias for unsigned int

page table entries, second-level: `pte_t`

- page table entry
- alias for unsigned int

x86-32 page tables are 4096-byte *arrays of 1024 entries*

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	
Ignored																									<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 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											page table base register (CR3)											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											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											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																												P C D	PW T	Ignored			CR3																											
first-level page table entries																																																												
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																															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																															0	PTE: not present

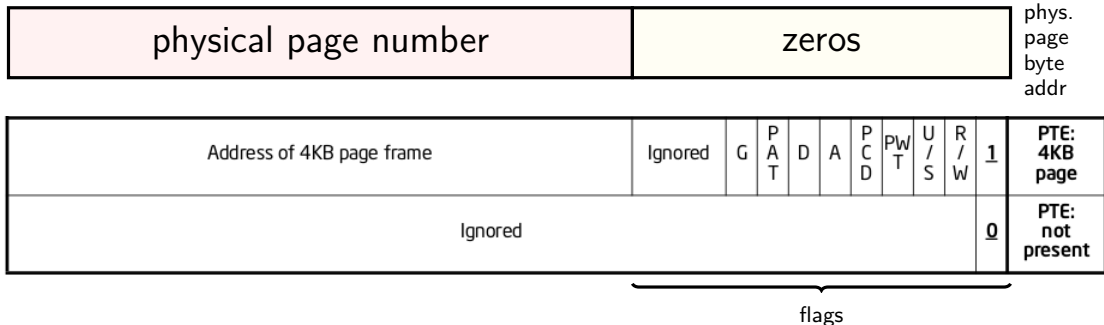
Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

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	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		
second-level page table entries																								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 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 */
```

xv6 page table-related functions

`kaalloc/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

xv6 page table-related functions

`kaalloc/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

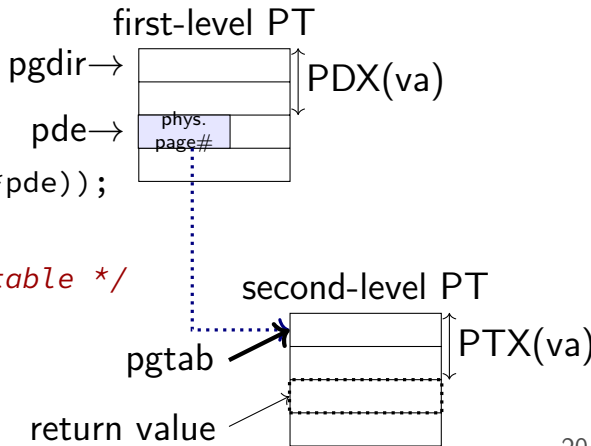
`allocvm` — allocate new user memory
setup user-accessible memory
allocate new second-level tables as needed

`deallocvm` — deallocate user memory

xv6: finding page table entries

```
// Return the address of the PTE in page table pgdir  
// that corresponds to virtual address va. If alloc!=0,  
// create any required page table pages.
```

```
static pte_t *  
walkpgdir(pde_t *pgdir, const void *va, int alloc)  
{  
    pde_t *pde;  
    pte_t *pgtab;  
  
    pde = &pgdir[PDX(va)];  
    if(*pde & PTE_P){  
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));  
    } else {  
        ... /* create new  
                second-level page table */  
    }  
    return &pgtab[PTX(va)];  
}
```



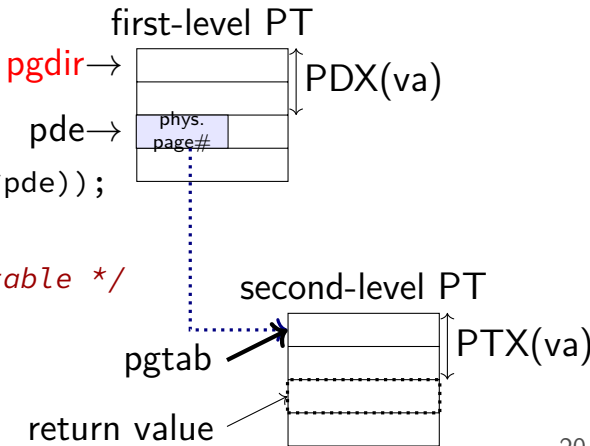
xv6: finding page table entries

`pgdir: pointer to first-level page table ('page directory')`

```
// Return the page table entry for virtual address va. If alloc!=0,  
// that corresponds to virtual address va. If alloc!=0,  
// create any required page table pages.
```

```
static pte_t *  
walkpgdir(pde_t *pgdir, const void *va, int alloc)
```

```
{  
    pde_t *pde;  
    pte_t *pgtab;  
  
    pde = &pgdir[PDX(va)];  
    if(*pde & PTE_P){  
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));  
    } else {  
        ... /* create new  
                second-level page table */  
    }  
    return &pgtab[PTX(va)];  
}
```

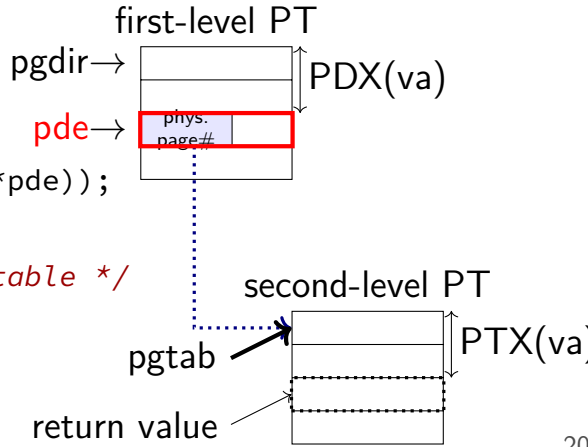


xv6: finding page table entries

```
// Return the address of  
// that corresponds to v  
// create any required page table pages.
```

retrieve (pointer to) page table entry from first-level table ('page directory')

```
static pte_t *  
walkpgdir(pde_t *pgdir, const void *va, int alloc)  
{  
    pde_t *pde;  
    pte_t *pgtab;  
  
    pde = &pgdir[PDX(va)];  
    if(*pde & PTE_P){  
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));  
    } else {  
        ... /* create new  
                second-level page table */  
    }  
    return &pgtab[PTX(va)];  
}
```

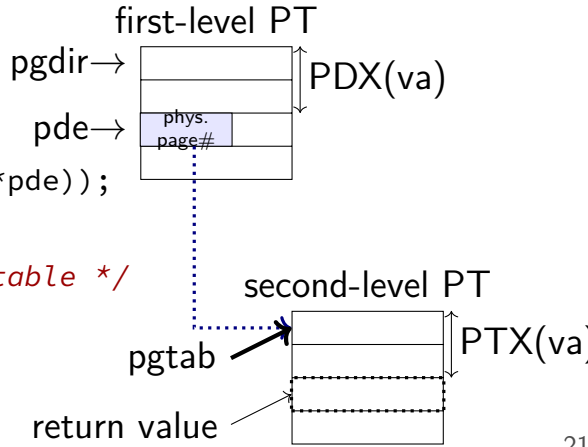


xv6: finding page table entries

```
// Return the address of  
// that corresponds to v  
// create any required p
```

check if first-level page table entry is valid
possibly create new second-level table +
update first-level table if it is not

```
static pte_t *  
walkpgdir(pde_t *pgdir, const void *va, int alloc)  
{  
    pde_t *pde;  
    pte_t *pgtab;  
  
    pde = &pgdir[PDX(va)];  
    if(*pde & PTE_P){  
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));  
    } else {  
        ... /* create new  
                second-level page table */  
    }  
    return &pgtab[PTX(va)];  
}
```

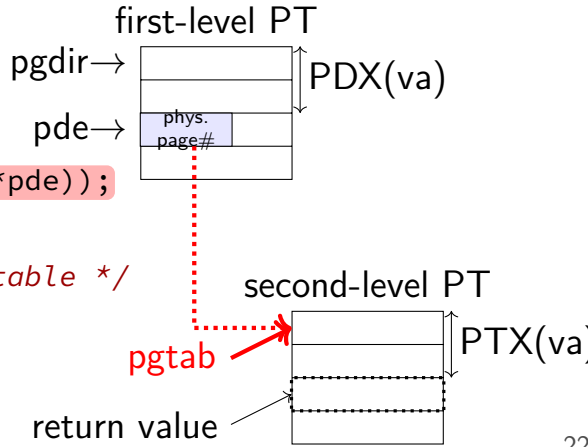


xv6: finding page table entries

retrieve location of second-level page table

```
// Return the address of  
// that corresponds to virtual address va. If alloc!=0,  
// create any required page table pages.
```

```
static pte_t *  
walkpgdir(pde_t *pgdir, const void *va, int alloc)  
{  
    pde_t *pde;  
    pte_t *pgtab;  
  
    pde = &pgdir[PDX(va)];  
    if(*pde & PTE_P){  
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));  
    } else {  
        ... /* create new  
                second-level page table */  
    }  
    return &pgtab[PTX(va)];  
}
```



xv6: finding page table entries

```
// Return the address of the physical page  
// that corresponds to the virtual address va  
// create any required page table pages.  
static pte_t *  
walkpgdir(pde_t *pgdir, const void *va, int alloc)
```

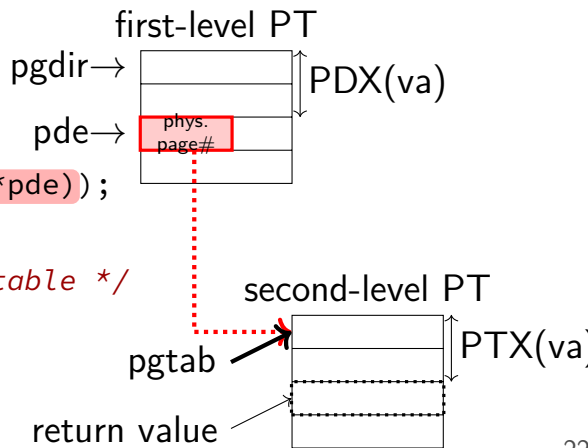
```
{  
    pde_t *pde;  
    pte_t *pgtab;
```

```
    pde = &pgdir[PDX(va)];  
    if(*pde & PTE_P){
```

```
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));  
    } else {  
        ... /* create new
```

```
            second-level page table */
```

```
    }  
    return &pgtab[PTX(va)];  
}
```



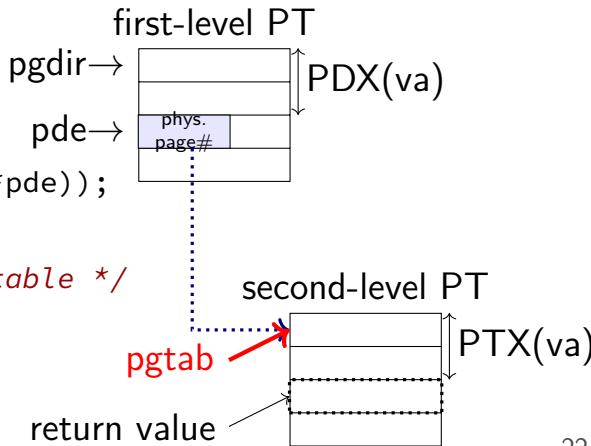
xv6: finding page table entries

convert page-table physical address to virtual

```
// Return the address  
// that corresponds to virtual address va. If alloc!=0,  
// create any required page table pages.
```

```
static pte_t *  
walkpgdir(pde_t *pgdir, const void *va, int alloc)
```

```
{  
    pde_t *pde;  
    pte_t *pgtab;  
  
    pde = &pgdir[PDX(va)];  
    if(*pde & PTE_P){  
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));  
    } else {  
        ... /* create new  
                second-level page table */  
    }  
    return &pgtab[PTX(va)];  
}
```



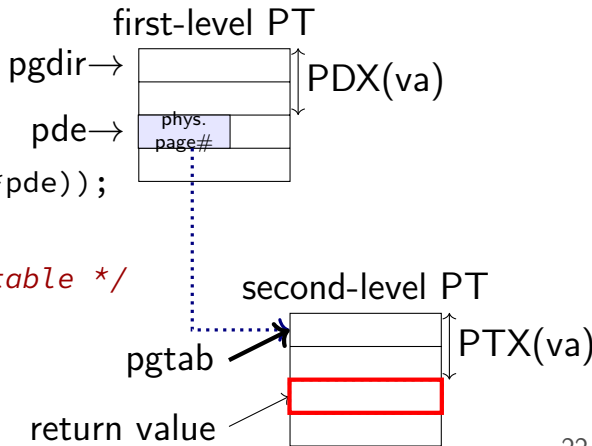
xv6: finding page table entries

```
// Return the address of the page table entry  
// that corresponds to the virtual address va.  
// create any required page table pages.
```

```
static pte_t *  
walkpgdir(pde_t *pgdir, const void *va, int alloc)
```

```
{  
    pde_t *pde;  
    pte_t *pgtab;  
  
    pde = &pgdir[PDX(va)];  
    if(*pde & PTE_P){  
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));  
    } else {  
        ... /* create new  
            second-level page table */  
    }  
    return &pgtab[PTX(va)];  
}
```

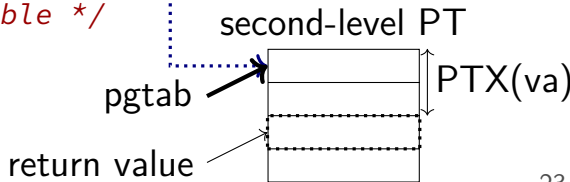
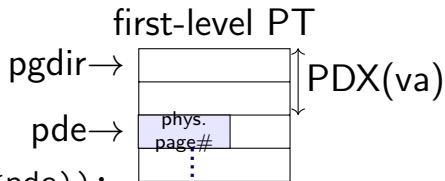
retrieve (pointer to) second-level page table entry
from second-level table



xv6: finding page table entries

```
// Return the address of the PTE in page table pgdir  
// that corresponds to virtual address va. If alloc!=0,  
// create any required page table pages.
```

```
static pte_t *  
walkpgdir(pde_t *pgdir, const void *va, int alloc)  
{  
    pde_t *pde;  
    pte_t *pgtab;  
  
    pde = &pgdir[PDX(va)];  
    if(*pde & PTE_P){  
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));  
    } else {  
        ... /* create new  
                second-level page table */  
    }  
    return &pgtab[PTX(va)];  
}
```



xv6: creating second-level page tables

```
...
if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
} else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
        return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

xv6: creating second-level page tables

```
...
if(*pde & PTE_P)
    pgtab = (pte_t*)kalloc();
} else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
        return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

return NULL if not trying to make new page table
otherwise use kalloc to allocate it
(and return NULL if that fails)

xv6: creating second-level page tables

clear the new second-level page table
PTE = 0 \rightarrow present = 0

```
...
if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
} else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
        return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

xv6: creating second-level page tables

```
...
if(*pde & PTE_P){
    pgtab = (pte_t*)
} else {
    if(!alloc || (p
        return 0;
// Make sure all those PTE_P bits are zero.
memset(pgtab, 0, PGSIZE);
// The permissions here are overly generous, but they can
// be further restricted by the permissions in the page table
// entries, if necessary.
*pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

create a first-level page entry
with physical address of second-level page table
P for “present” (valid)
W for “writable”
U for “user-mode” (in addition to kernel)

xv6: creating second-level page tables

```
...
if(*pde & PTE_P){
    pgtab = (pte_t*)
} else {
    if(!alloc || (p
        return 0;
// Make sure all those PTE_P bits are zero.
memset(pgtab, 0, PGSIZE);
// The permissions here are overly generous, but they can
// be further restricted by the permissions in the page table
// entries, if necessary.
*pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

create a first-level page entry

with **physical address of second-level page table**

P for “present” (valid)

W for “writable”

U for “user-mode” (in addition to kernel)

xv6: creating second-level page tables

```
if(*pde & PTE_P){
    pgtab = (pte_t*)
} else {
    if(!alloc || (p
        return 0;
// Make sure all those PTE_P bits are zero.
memset(pgtab, 0, PGSIZE);
// The permissions here are overly generous, but they can
// be further restricted by the permissions in the page table
// entries, if necessary.
*pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

create a first-level page entry

with physical address of second-level page table

P for “present” (valid)

W for “writable”

U for “user-mode” (in addition to kernel)

aside: permissions

xv6: sets first-level page table entries with all permissions

...but second-level entries can override

xv6 page table-related functions

`kaalloc/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

xv6: setting last-level page entries

```
static int loop for a = va to va + size and pa = pa to pa + size
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
    char *a, *last; pte_t *pte;

    a = (char*)PGROUNDDOWN((uint)va);
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for(;;){
        if((pte = walkpgdir(pgdir, a, 1)) == 0)
            return -1;
        if(*pte & PTE_P)
            panic("remap");
        *pte = pa | perm | PTE_P;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
```

xv6: setting last-level page entries

```
static int
mappages(pde_t *pgdir, void *va, uint size)
{
    char *a, *last; pte_t *pte;

    a = (char*)PGROUNDDOWN((uint)va);
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for(;;){
        if((pte = walkpgdir(pgdir, a, 1)) == 0)
            return -1;
        if(*pte & PTE_P)
            panic("remap");
        *pte = pa | perm | PTE_P;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
```

for each virtual page in range:
get its page table entry
(or fail if out of memory)

xv6: setting last-level page entries

```
static int
mappages(pde_t *pgdir, void *va, int size)
{
    char *a, *last;

    a = (char*)PGROUNDDOWN((uint)va);
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for(;;){
        if((pte = walkpgdir(pgdir, a, 1)) == 0)
            return -1;
        if(*pte & PTE_P)
            panic("remap");
        *pte = pa | perm | PTE_P;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
```

make sure it's not already set

in stock xv6: never change valid page table entry

in upcoming homework: this is not true

xv6: setting last-level page entries

```
static int  
mappages(pde  
{  
    char *a, *  
  
    a = (char*)PGROUNDDOWN((uint)va);  
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);  
    for(;;){  
        if((pte = walkpgdir(pgdir, a, 1)) == 0)  
            return -1;  
        if(*pte & PTE_P)  
            panic("remap");  
        *pte = pa | perm | PTE_P;  
        if(a == last)  
            break;  
        a += PGSIZE;  
        pa += PGSIZE;  
    }  
    return 0;  
}
```

set page table entry to valid value
pointing to physical page at pa
with specified permission bits (write and/or user-mode)
and P for present

```
*pte = pa | perm | PTE_P;
```

xv6: setting last-level page entries

```
static int
mappages(pde_t *pgdir, void *va)
{
    char *a, *last; pte_t *pte;

    a = (char*)PGROUNDDOWN((uint)va);
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for(;;){
        if((pte = walkpgdir(pgdir, a, 1)) == 0)
            return -1;
        if(*pte & PTE_P)
            panic("remap");
        *pte = pa | perm | PTE_P;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
```

advance to next physical page (pa)
and next virtual page (va)

xv6 page table-related functions

`kaalloc/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

xv6: setting process page tables (exec())

exec step 1: create new page table with kernel mappings

setupkvm()

(recall: kernel mappings — high addresses)

exec step 2a: allocate memory for executable pages

allocvm() in loop

new physical pages chosen by kalloc()

exec step 2b: load executable pages from executable file

loadvm() in a loop

copy from disk into newly allocated pages (in loadvm())

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

xv6: setting process page tables (exec())

exec step 1: create new page table with kernel mappings

setupkvm()

(recall: kernel mappings — high addresses)

exec step 2a: allocate memory for executable pages

allocvm() in loop

new physical pages chosen by kalloc()

exec step 2b: load executable pages from executable file

loadvm() in a loop

copy from disk into newly allocated pages (in loadvm())

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

create new page table (setupkvm())

use `kalloc()` to allocate first-level table

call `mappages()` (several times) for kernel mappings

(hard-coded lists of calls to make to `mappages()`)

xv6: setting process page tables (exec())

exec step 1: create new page table with kernel mappings

setupkvm()

(recall: kernel mappings — high addresses)

exec step 2a: **allocate memory for executable pages**

allocvm() in loop

new physical pages chosen by kalloc()

exec step 2b: load executable pages from executable file

loadvm() in a loop

copy from disk into newly allocated pages (in loadvm())

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

reading executables (headers)

xv6 executables contain **program headers**: list of
virtual address range (to allocate)
corresponding location in executable file
permissions (read/write/etc.?)

exec() iterates over the list, and foreach header:

allocvm() — allocate memory, setting up page table entries

loadvm() — copy from executable file into newly allocated memory

reading executables (headers)

xv6 executables contain program headers: list of
virtual address range (to allocate)
corresponding location in executable file
permissions (read/write/etc.?)

exec() iterates over the list, and foreach header:

allocvm() — allocate memory, setting up page table entries

loadvm() — copy from executable file into newly allocated 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

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz, void **pa) {
{
    ...
    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;
    }
}
```

allocate a new, zero page

allocating user pages

```
allocuvm(pde_t *pgdir, uint o
{
    ...
    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
{
    ...
    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;
        }
    }
}
```

this function used for initial allocation plus expanding heap on request

reading executables (headers)

xv6 executables contain program headers: list of
virtual address range (to allocate)
corresponding location in executable file
permissions (read/write/etc.?)

exec() iterates over the list, and foreach header:

allocvm() — allocate memory, setting up page table entries

loadvm() — copy from executable file into newly allocated memory

loadvm()

`loadvm(pgdir, address, file, offset, sz)`

for each virtual page between address and address + sz:

find the physical address of that page (`walkpgdir()`)

find the kernel address for that physical address (`P2V()`)

copy from disk into that kernel address

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

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

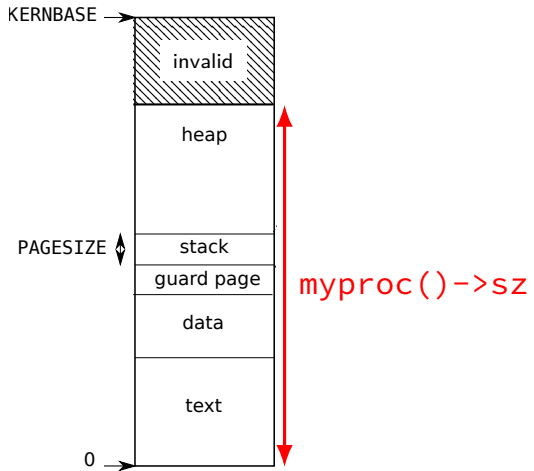
- kfree — add page to list

linked list created at boot

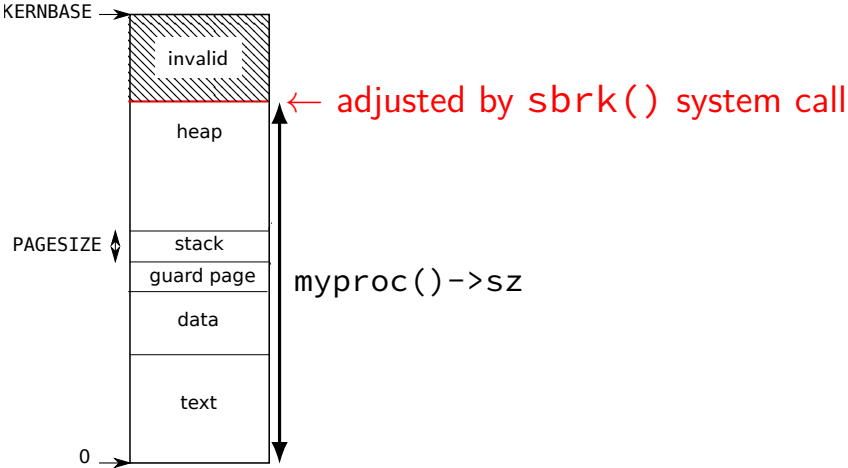
usable memory fixed size (224MB)

- determined by PHYSTOP in memlayout.h

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

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

allocuvm — same function used to allocate initial space
maps pages for addresses SZ to SZ + n
calls kalloc to get each page

```
    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 crashing
returning from page 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

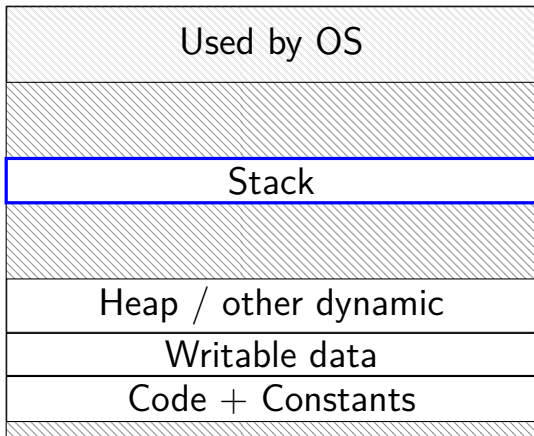
OS can do all sorts of 'tricks' with page tables

key idea: what processes *think* they have in memory \neq their actual memory

OS fixes disagreement from page fault handler

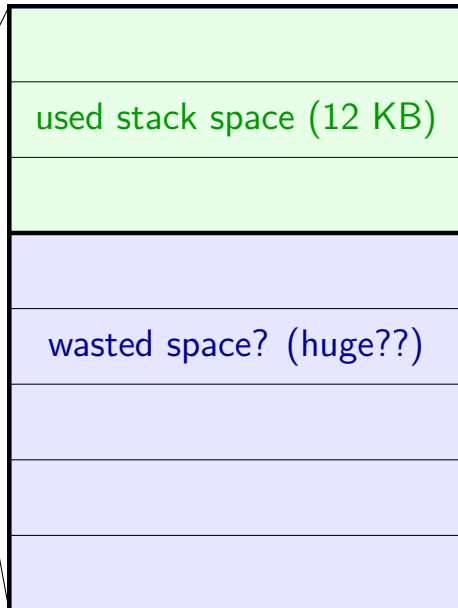
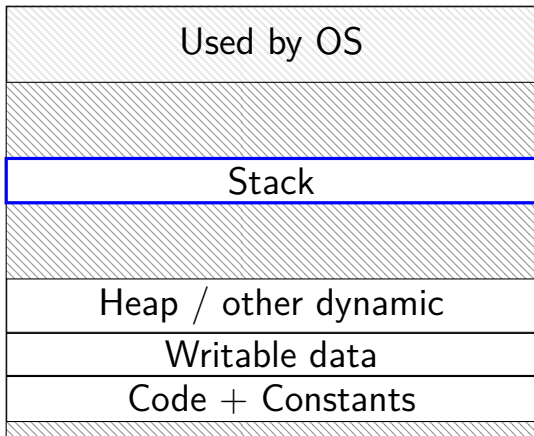
space on demand

Program Memory



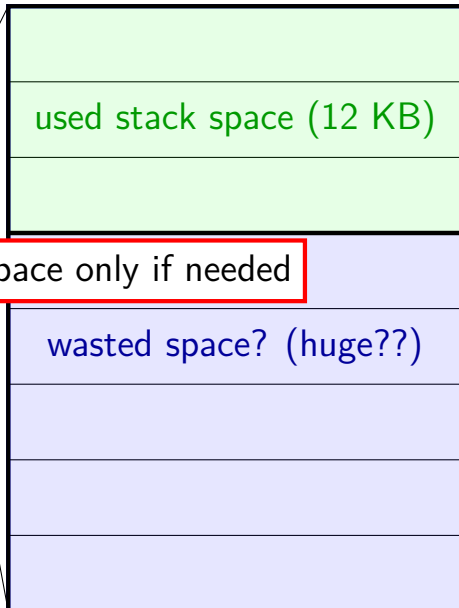
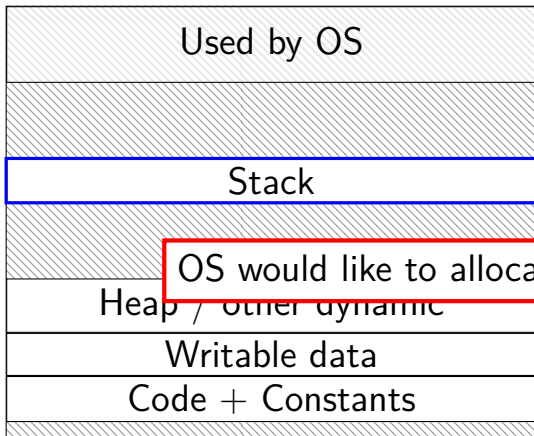
space on demand

Program Memory



space on demand

Program Memory



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

```
...  
0x7FFFB  
0x7FFFC  
0x7FFFD  
0x7FFFE  
0x7FFFF  
...
```

valid? physical
page

valid?	physical page
...	...
0	---
1	0x200DF
1	0x12340
1	0x12347
1	0x12345
...	...

allocating space on demand

`%rsp = 0x7FFFC000`

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

VPN

```
...  
0x7FFFB  
0x7FFFC  
0x7FFFD  
0x7FFFE  
0x7FFFF  
...
```

valid? physical
page

valid?	physical page
...	...
0	---
1	0x200DF
1	0x12340
1	0x12347
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 restarted  
B: movq 8(%rcx), %rbx  
C: addq %rbx, %rax  
...
```

VPN	valid?	physical page
...
<code>0x7FFFB</code>	<code>1</code>	<code>0x200D8</code>
<code>0x7FFFC</code>	<code>1</code>	<code>0x200DF</code>
<code>0x7FFFD</code>	<code>1</code>	<code>0x12340</code>
<code>0x7FFFE</code>	<code>1</code>	<code>0x12347</code>
<code>0x7FFFF</code>	<code>1</code>	<code>0x12345</code>
...

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 assignment: 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()`

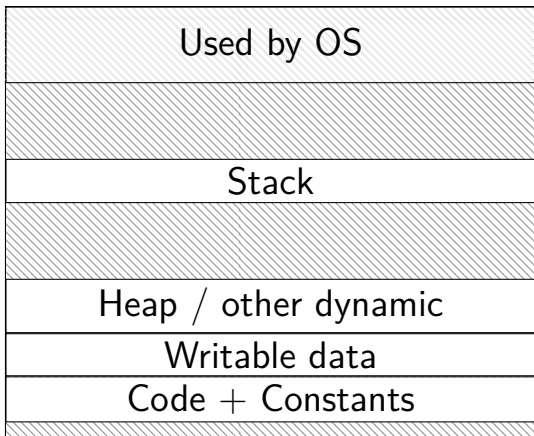
creates a **copy** of an entire program!

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

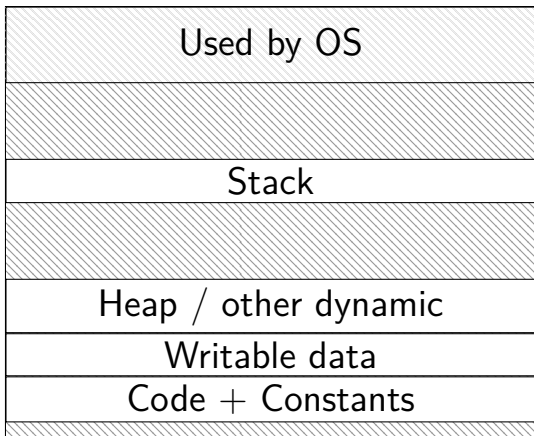
how isn't this really slow?

do we really need a complete copy?

bash

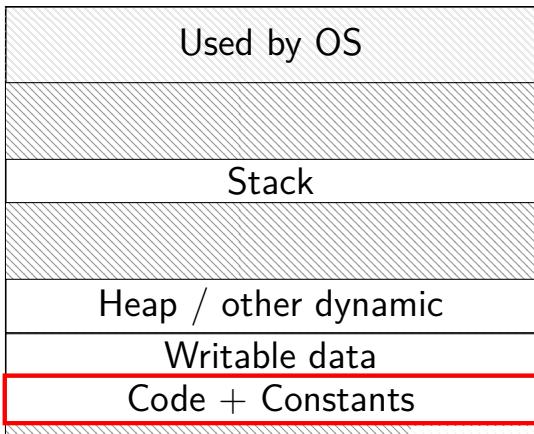


new copy of bash

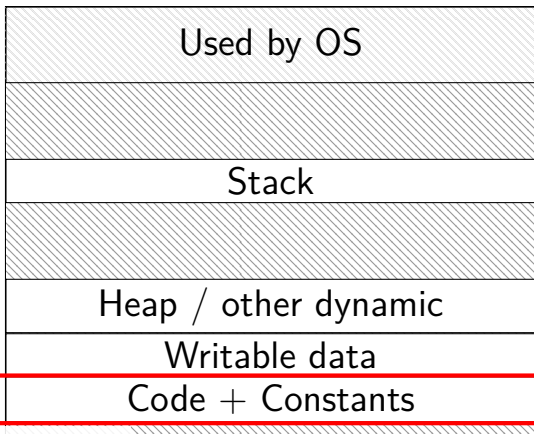


do we really need a complete copy?

bash



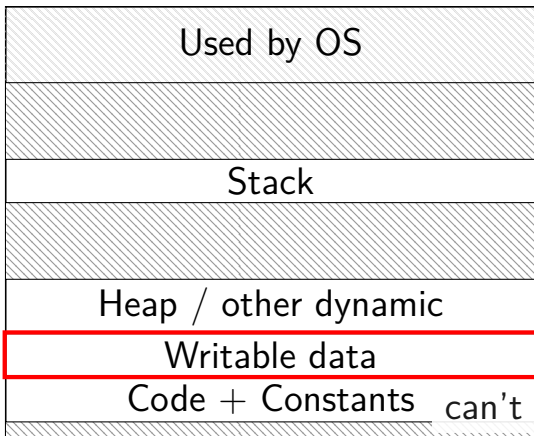
new copy of bash



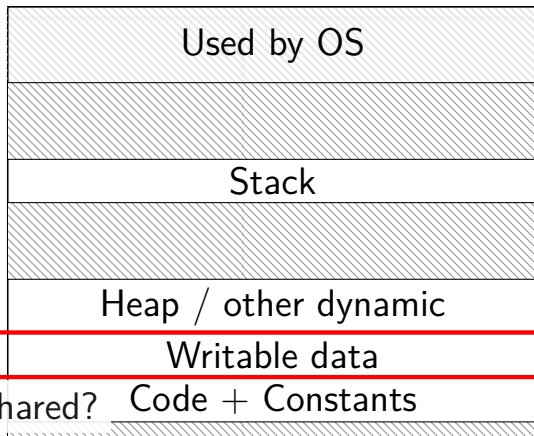
shared as read-only

do we really need a complete copy?

bash



new copy of bash



can't be shared?

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 writable

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
- B. fix/fob/foo
- C. fix/fix/fix
- D. fix/fob/fix
- E. fix/fob/fob
- 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 001be000 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-xp 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 0x4000000-0x40b0000

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 private = copy-on-write (if writeable)
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]
```

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 001be000 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
7f60c7854000-7f60c7854000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7859000-7f60c7859000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a39000-7f60c7a39000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7a000-7f60c7a7a000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7b000-7f60c7a7b000 r-p 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.19
7f60c7a7c000-7f60c7a7c000 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]
```

device major number 8

device minor number 1

inode 48328831

more on what this means when we talk about filesystems

backup slides

sketch: implementing mmap

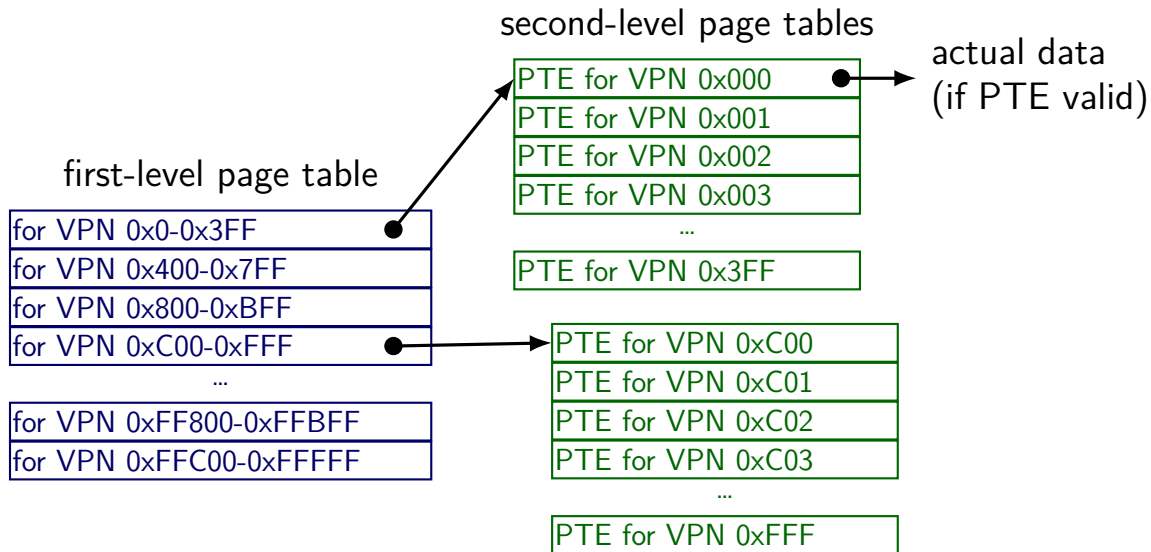
access mapped file for first time, read from disk
(like swapping when memory was swapped out)

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

two-level page tables

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



two-level page tables

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

x86-32: arrays of 2^{10} 32-bit
page table entries

first-level page table

for VPN 0x0-0x3FF
for VPN 0x400-0x7FF
for VPN 0x800-0xBFF
for VPN 0xC00-0xFFF
...
for VPN 0xFF800-0xFFBFF
for VPN 0xFFC00-0xFFFFF

second-level page tables

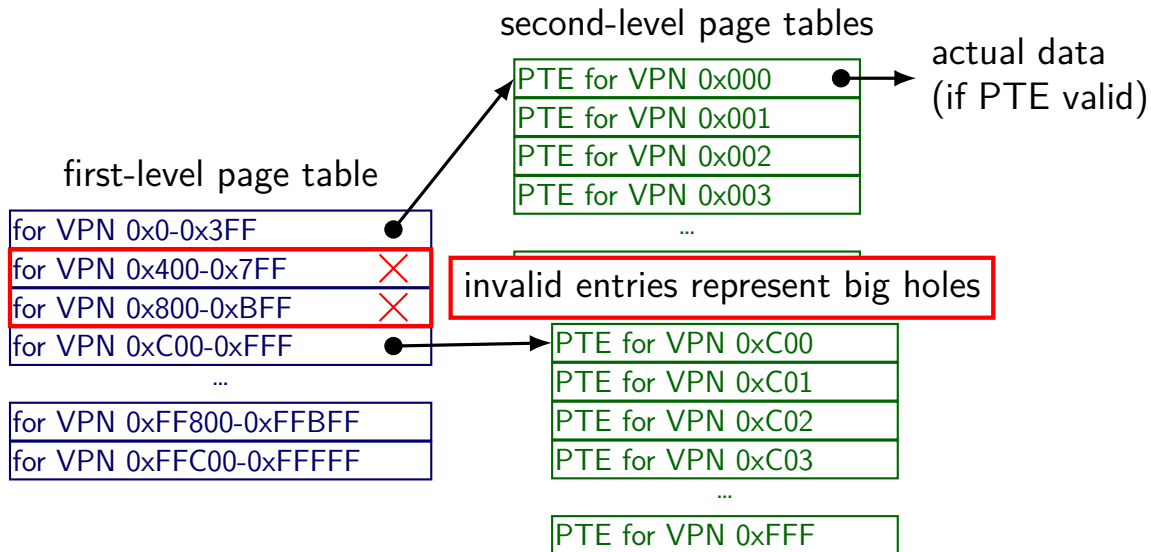
PTE for VPN 0x000
PTE for VPN 0x001
PTE for VPN 0x002
PTE for VPN 0x003
...
PTE for VPN 0x3FF

PTE for VPN 0xC00
PTE for VPN 0xC01
PTE for VPN 0xC02
PTE for VPN 0xC03
...
PTE for VPN 0xFFF

actual data
(if PTE valid)

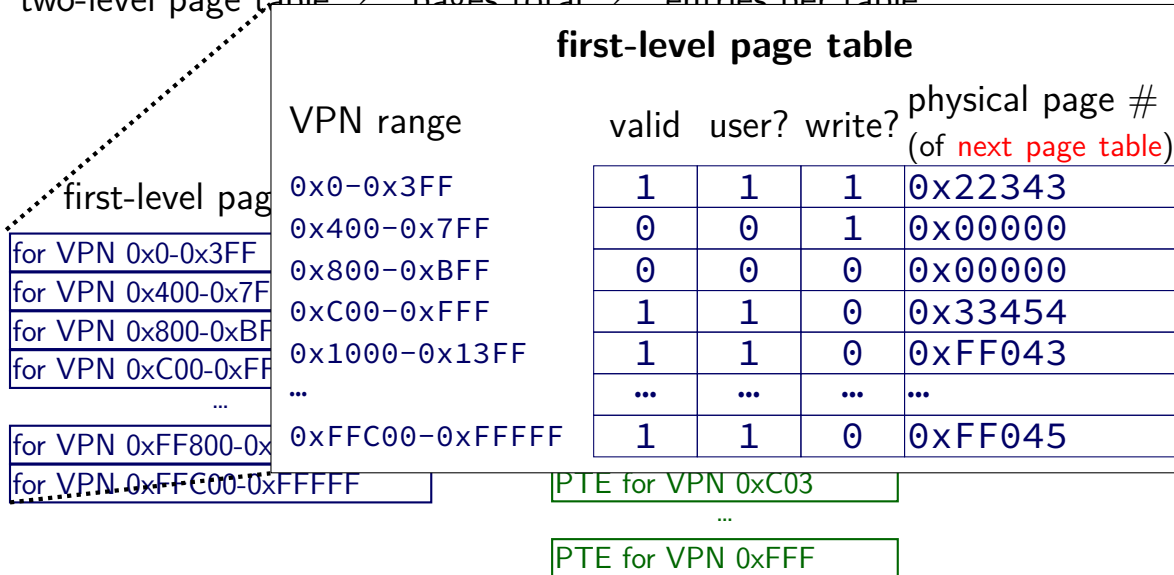
two-level page tables

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



two-level page tables

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



two-level page tables

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

	VPN range	valid	user?	write?	physical page # (of next page table)
first-level page table for VPN 0x0-0x3FF for VPN 0x400-0x7FF for VPN 0x800-0xBF for VPN 0xC00-0xFF ... for VPN 0xFF800-0x for VPN 0xFFC00-0xFFFF	0x0-0x3FF	1	1	1	0x22343
	0x400-0x7FF	0	0	1	0x00000
	0x800-0xBFF	0	0	0	0x00000
	0xC00-0xFFF	1	1	0	0x33454
	0x1000-0x13FF	1	1	0	0xFF043

	0xFFC00-0xFFFF	1	1	0	0xFF045
	...				
	...				
	...				

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

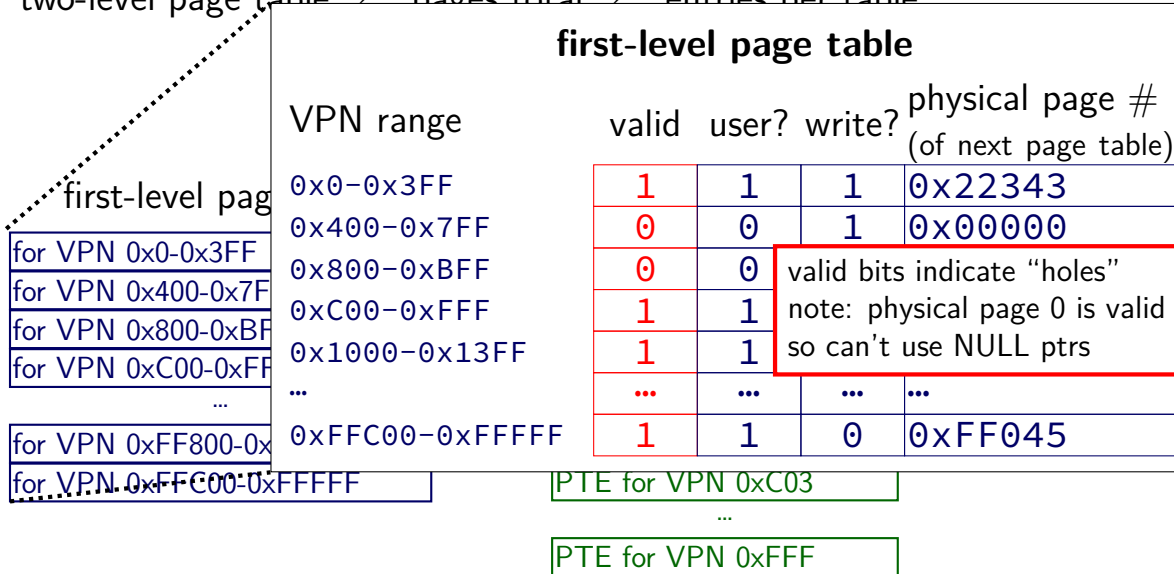
two-level page table: 2^{20} pages total: 2^{10} entries per table

	VPN range	valid	user?	write?	physical page # (of next page table)
first-level page table for VPN 0x0-0x3FF for VPN 0x400-0x7FF for VPN 0x800-0xBF for VPN 0xC00-0xFF	0x0-0x3FF	1	1	1	0x22343
	0x400-0x7FF	0	0	1	0x00000
	0x800-0xBFF	0	0	0	0x00000
	0xC00-0xFF	0	0	0	0x33454
	0x1000-0x13FF	0	0	0	0xFF043
...	
for VPN 0xFF800-0xFF	0xFFC00-0xFFFF	1	1	0	0xFF045
for VPN 0xFFC00-0xFFFF		PTE for VPN 0xC03			
		...			
		PTE for VPN 0xFFF			

pointers to page tables
(arrays of PTEs)
but using page number
(not byte number)

two-level page tables

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



two-level page tables

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

first-level page table

for VPN 0x0-0x3FF	●
for VPN 0x400-0x7FF	✗
for VPN 0x800-0xBFF	✗
for VPN 0xC00-0xFFF	●
...	
for VPN 0xFF800-0xFFBFF	
for VPN 0xFFC00-0xFFFFF	

a second-level page table

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

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

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

first-level page table

for VPN 0x0-0x3FF	●
for VPN 0x400-0x7FF	✗
for VPN 0x800-0xBFF	✗
for VPN 0xC00-0xFFF	●
...	
for VPN 0xFF800-0xFFBFF	
for VPN 0xFFC00-0xFFFFF	

a second-level page table

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

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page table naming

what the page table base register points to:

first-level page table

top-level page table

page directory (Intel's term, used in xv6 code)

what first-level page table entries point to

second-level page table

page table (Intel's term, used in xv6 code)

 I'll avoid using this term unqualified...

 but Intel manuals/xv6 do not