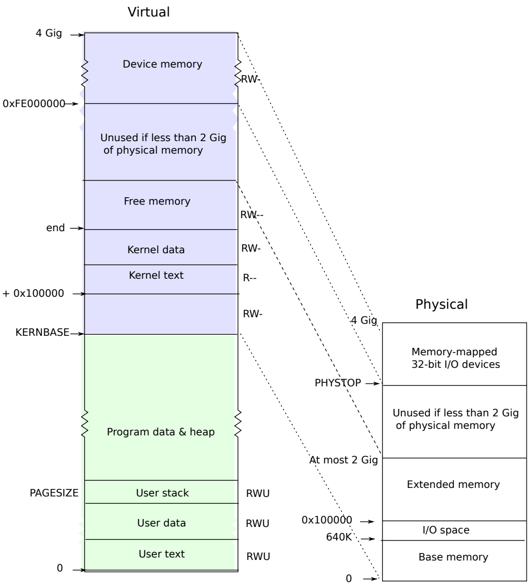
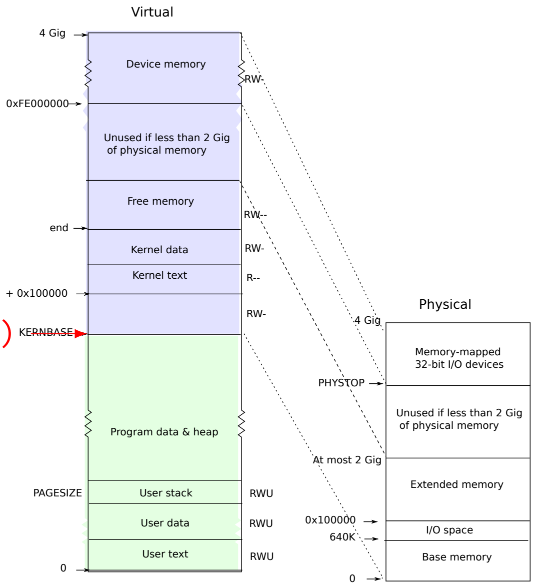
virtual memory 2

xv6 memory layout



xv6 memory layout

0x80000000 (KERNBASE)



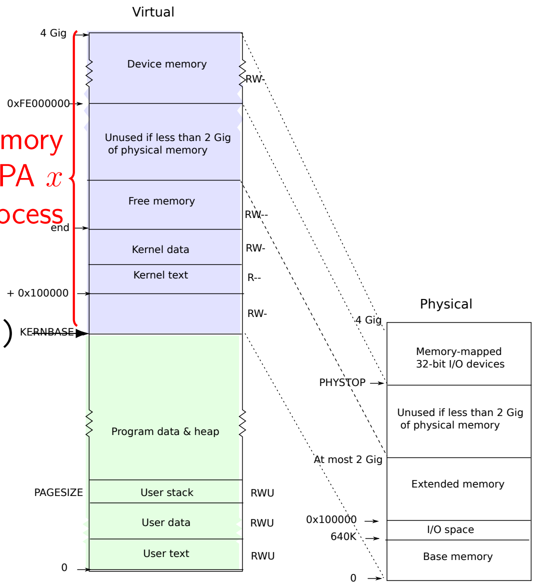
xv6 memory layout

kernel-only memory

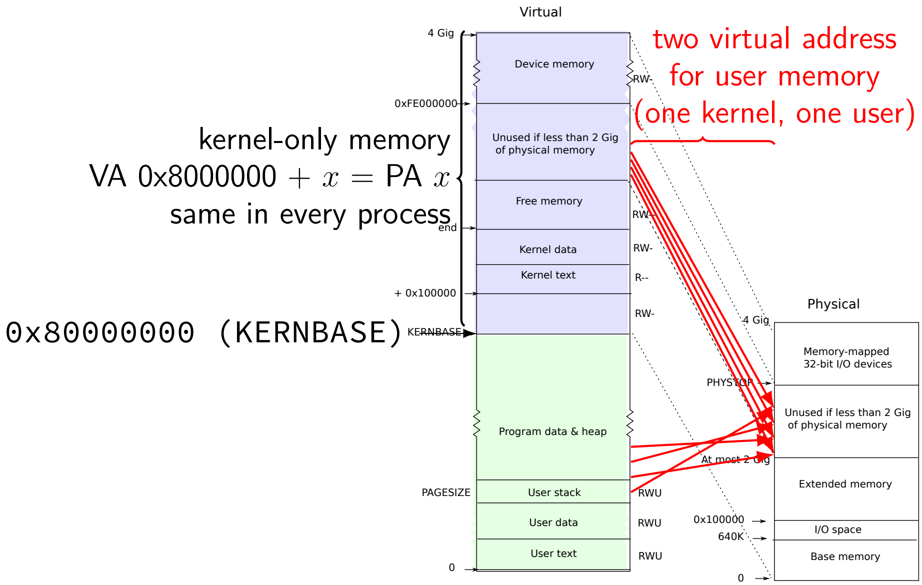
VA 0x8000000 $+ x =$ PA x

same in every process

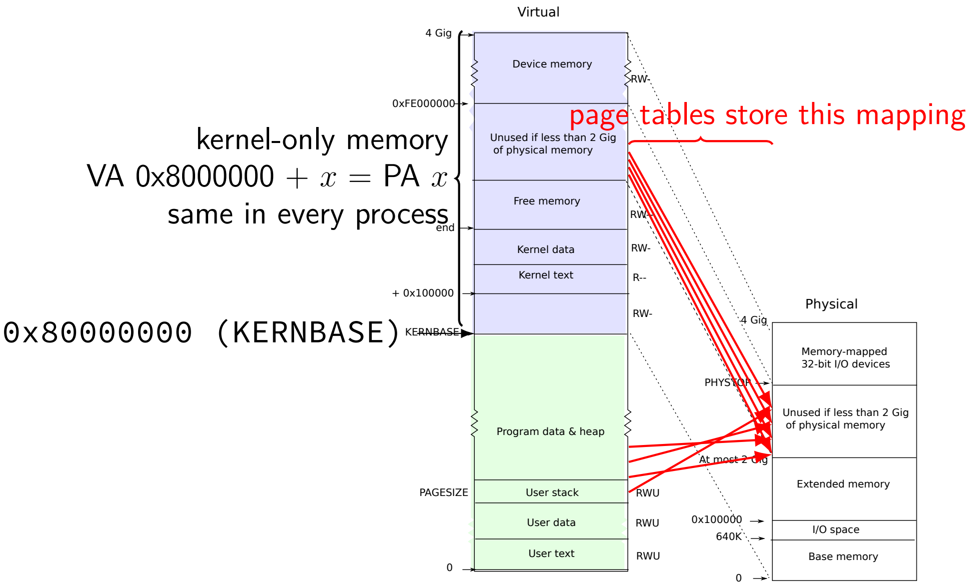
0x80000000 (KERNBASE)



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 KERNBASE$+0$

physical memory address N is mapped to 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

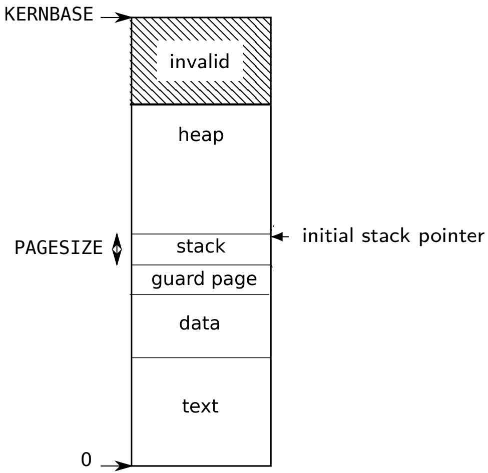
- sized based on executable, heap allocations
- uses any available memory

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

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
    ....
```

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							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

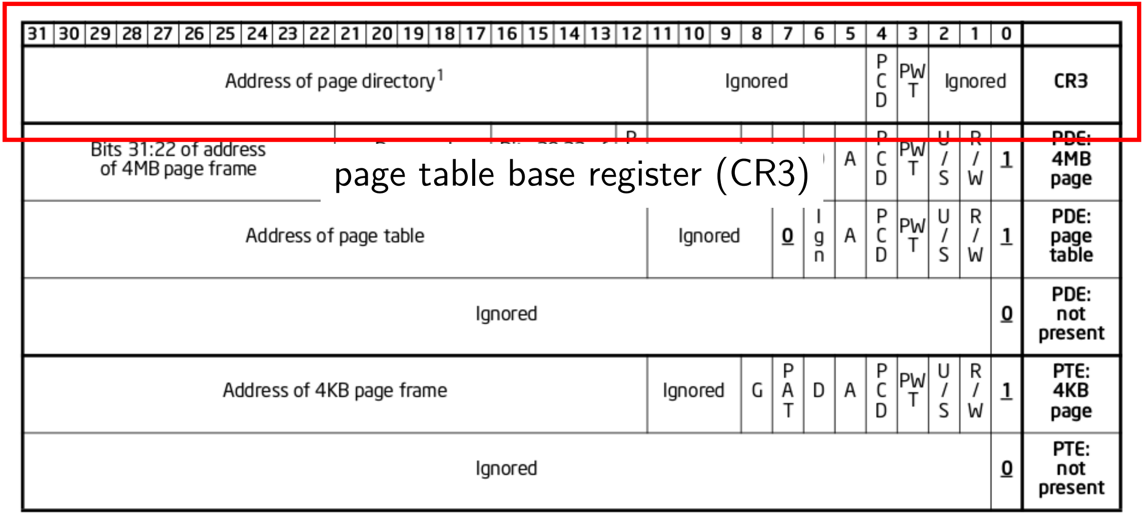


Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

first-level page table entries

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 pag																											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**

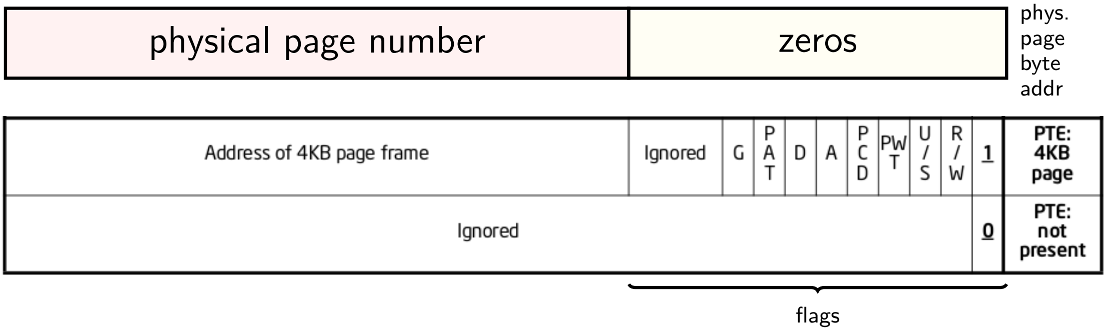
Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

<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 page directory[1]</td><td colspan="7">Ignored</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><u>1</u></td><td>D</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td><u>1</u></td><td>PDE: 4MB page</td></tr>
<tr><td colspan="20">Address of page table</td><td colspan="4">Ignored</td><td><u>0</u></td><td>Ign</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td><u>1</u></td><td>PDE: page table</td></tr>
<tr><td colspan="31">second-level page table entries</td><td><u>0</u></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><u>1</u></td><td>PTE: 4KB page</td></tr>
<tr><td colspan="31">Ignored</td><td><u>0</u></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 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 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 */
```

xv6 page table-related functions

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

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

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

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

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

`deallocuvm` — deallocate user memory

xv6 page table-related functions

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

`walkpgdir` — get pointer to second-level page table entry

  …to check it/make it valid/invalid/point somewhere/etc.

`mappages` — set range of page table entries

  implementation: loop using `walkpgdir`

`allockvm` — create new set of page tables, set kernel (high) part

  entries for `0x8000 0000` and up set

  allocate new first-level table plus several second-level tables

`allocuvm` — allocate new user memory

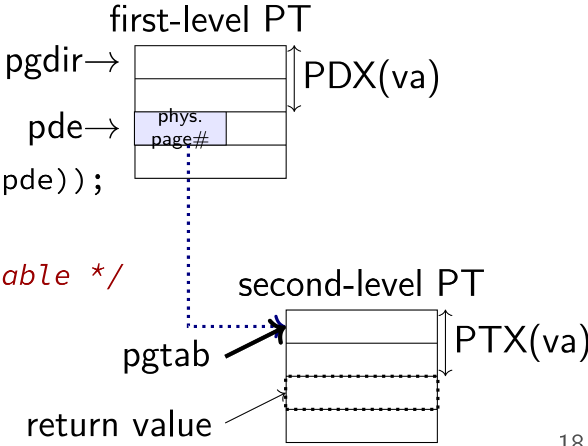
  setup user-accessible memory

  allocate new second-level tables as needed

`deallocuvm` — deallocate user memory

xv6: finding page table entries

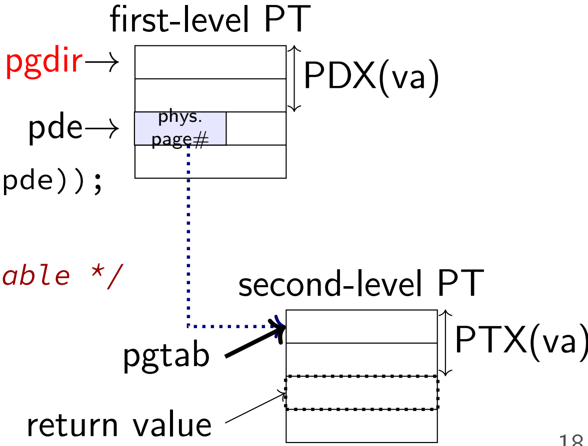
```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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')

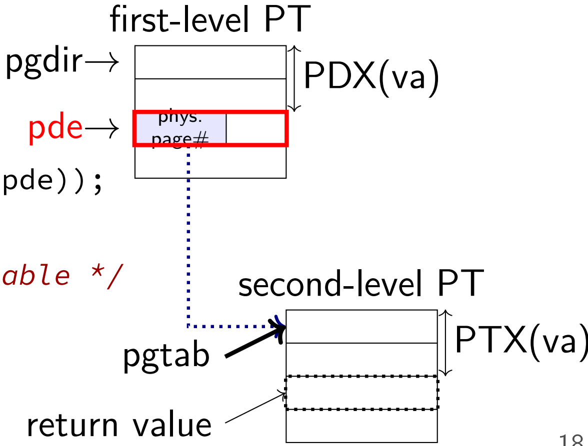
```
// Return th
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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.
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)];
}
```

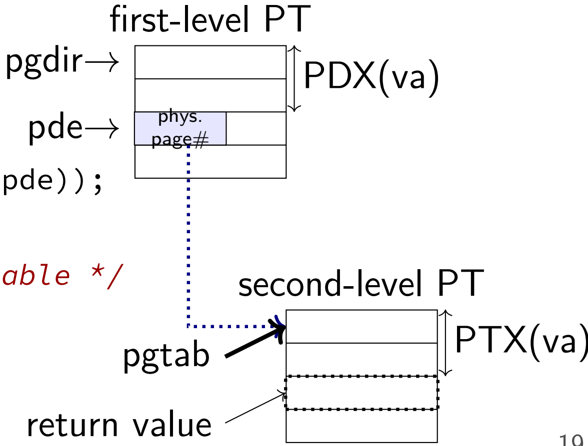
retrieve (pointer to) page table entry from first-level table (‘page directory’)



xv6: finding page table entries

```
// Return the address of
// that corresponds to v
// create any required p
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)];
}
```

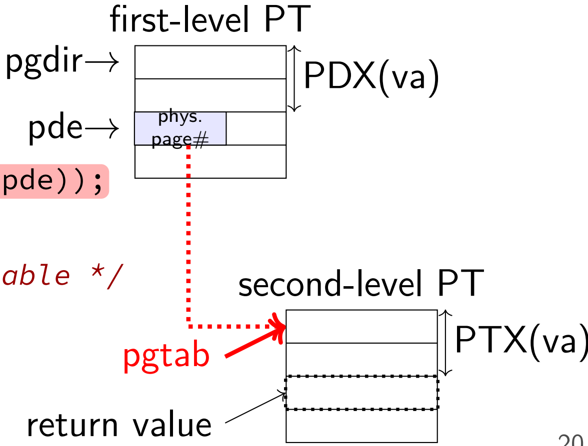
check if first-level page table entry is valid
possibly create new second-level table +
update first-level table if it is not



xv6: finding page table entries

retrieve location of second-level page table

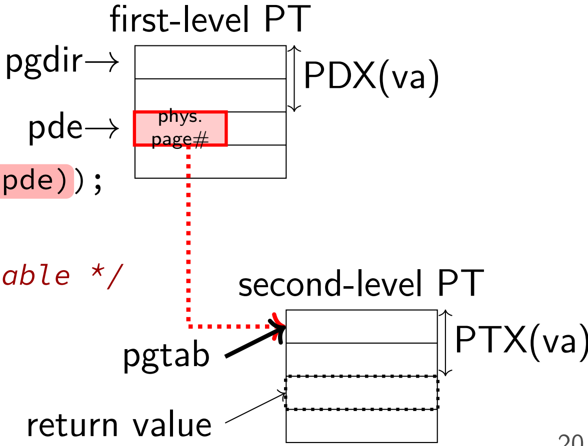
```
// Return the address o
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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

PTE_ADDR(*pde) — return physical page address from page table entry

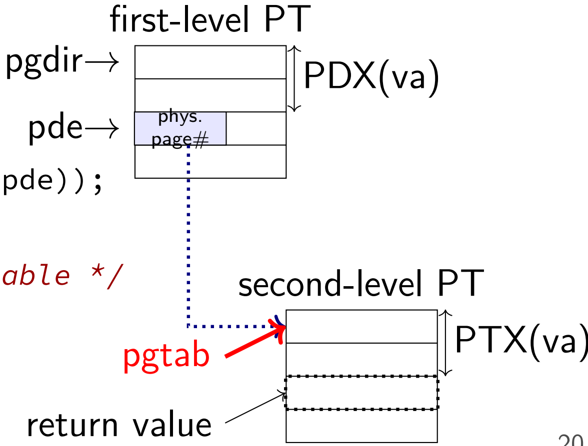
```
// Return the ad
// that correspo
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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

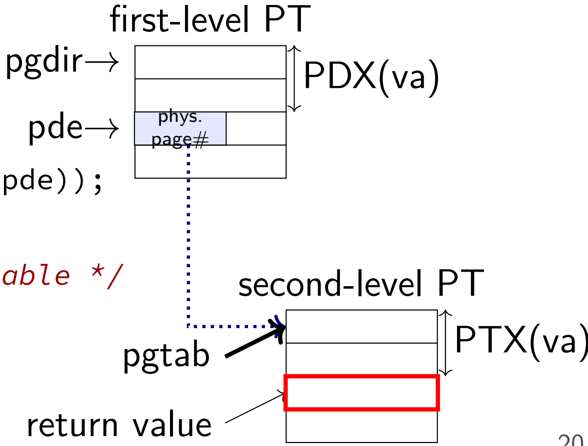
```
// Return the address
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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

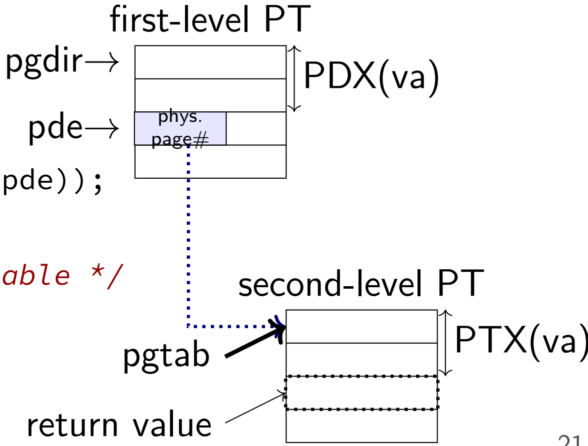
retrieve (pointer to) second-level page table entry from second-level table

```
// Return the addr
// that correspond
// 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 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_
} 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 permissons

…but second-level entries can override

xv6 page table-related functions

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

`walkpgdir` — get pointer to second-level page table entry

- …to check it/make it valid/invalid/point somewhere/etc.

`mappages` — set range of page table entries

- implementation: loop using `walkpgdir`

`allockvm` — create new set of page tables, set kernel (high) part

- entries for `0x8000 0000` and up set
- allocate new first-level table plus several second-level tables

`allocuvm` — allocate new user memory

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

`deallocuvm` — deallocate user memory

xv6: setting last-level page entries

loop for a = va to va + size and pa = pa to pa + size

```
static int
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, ui
{
  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 *pg
{
  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

xv6: setting last-level page entries

```
static int
mappages(pde_t *pgdir, void *v
{
  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

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

`walkpgdir` — get pointer to second-level page table entry

- ...to check it/make it valid/invalid/point somewhere/etc.

`mappages` — set range of page table entries

- implementation: loop using `walkpgdir`

`allockvm` — create new set of page tables, set kernel (high) part

- entries for `0x8000 0000` and up set
- allocate new first-level table plus several second-level tables

`allocuvm` — allocate new user memory

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

`deallocuvm` — deallocate user memory

xv6: 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

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

exec step 2b: load executable pages from executable file

- `loaduvm()` in a loop
- copy from disk into newly allocated pages (in `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

- `setupkvm()`
- (recall: kernel mappings — high addresses)

exec step 2a: allocate memory for executable pages

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

exec step 2b: load executable pages from executable file

- `loaduvm()` in a loop
- copy from disk into newly allocated pages (in `loaduvm()`)

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

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

exec step 2b: load executable pages from executable file

- `loaduvm()` in a loop
- copy from disk into newly allocated pages (in `loaduvm()`)

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

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:

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

```
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;
```

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, uin
{
  ...
  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

loaduvm()

`loaduvm(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`

`allockvm` — create new set of page tables, set kernel (high) part

- entries for `0x8000 0000` and up set
- allocate new first-level table plus several second-level tables

`allocuvm` — allocate new user memory

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

`deallocuvm` — deallocate user memory

kalloc/kfree

kalloc/kfree — xv6's physical memory allocator

allocates/deallocates whole pages only

keep linked list of free pages

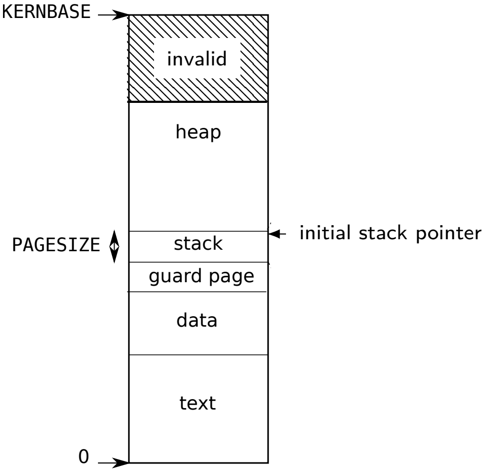
- list nodes — stored in corresponding free page itself
- kalloc — return first page in list
- kfree — add page to list

linked list created at boot

usuable memory fixed size (224MB)

- determined by PHYSTOP in `memlayout.h`

xv6 program memory



guard page

1 page after stack

- at lower addresses since stack grows towards lower addresses

marked as kernel-mode-only

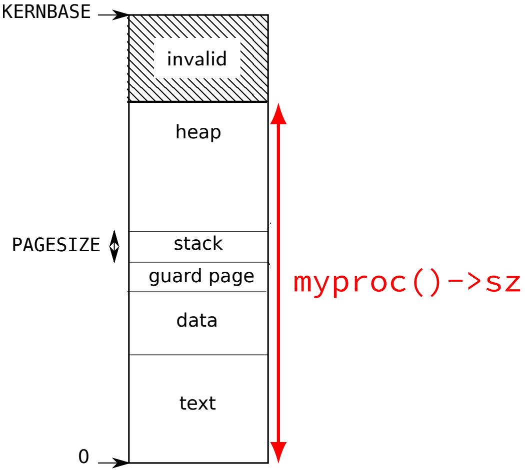
idea: stack overflow → protection fault → kills program

skipping the guard page

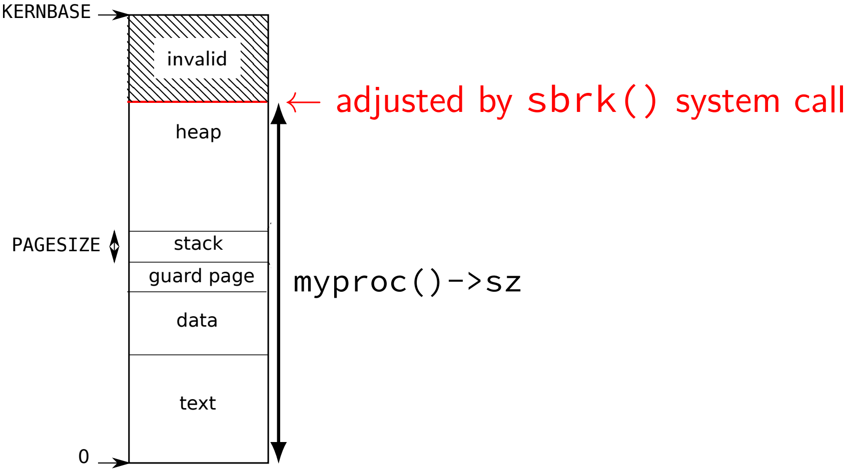
```
void example() {
    int array[2000];
    array[0] = 1000;
    ...
}
```

```
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
    ....
```

xv6 program memory



xv6 program memory



xv6 heap allocation

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

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

tracked in `struct proc` with `sz`

- = last valid address in process

position changed via `sbrk(amount)` system call

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

sbrk

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

sbrk

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

sz: current top of heap

sbrk

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

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

sbrk

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

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

growproc

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

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

growproc

allocuvm — same function used to allocate initial space

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

calls `kalloc` to get each page

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

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

xv6 page faults (now)

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

- xv6 now: default case in trap() function

xv6 page faults (now)

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

xv6 now: default case in trap() function

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

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

xv6 page faults (now)

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

xv6 now: default case in trap() function

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

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

trap 14 = T_PGFLT

special register CR2 contains faulting address

xv6 page faults (now)

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

xv6 now: default case in trap() function

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

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

trap 14 = T_PGFLT

special register CR2 contains faulting address

xv6: if one handled page faults

alternative to crashing: update the page table and return
- returning from page fault handler normally retries failing instruction

"just in time" update of the process's memory
- example: don't actually allocate memory until it's needed

xv6: if one handled page faults

alternative to crashing: update the page table and return

- returning from page fault handler normally retries failing instruction

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

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

pseudocode for xv6 implementation (for trap())

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

xv6: if one handled page faults

alternative to crash

check *process control block* to see if access okay

returning from page fault handler normally retries failing instruction

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

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

pseudocode for xv6 implementation (for trap())

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

xv6: if one handled page faults

alternative to crashin[g]

returning from pa[g]

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;
  }
}
```

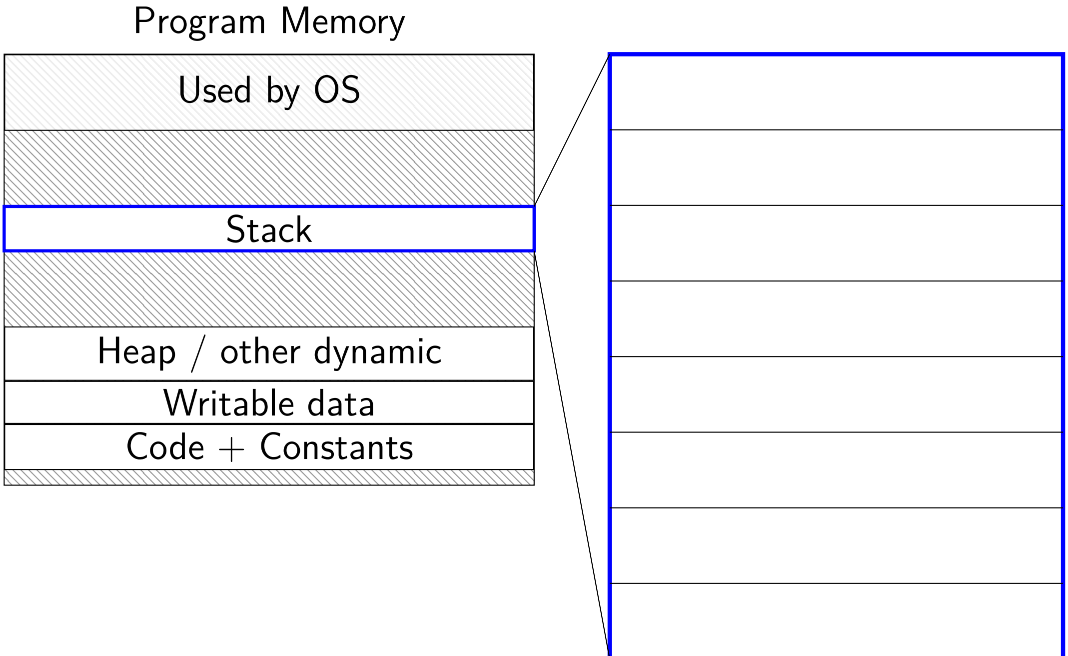
page fault tricks

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

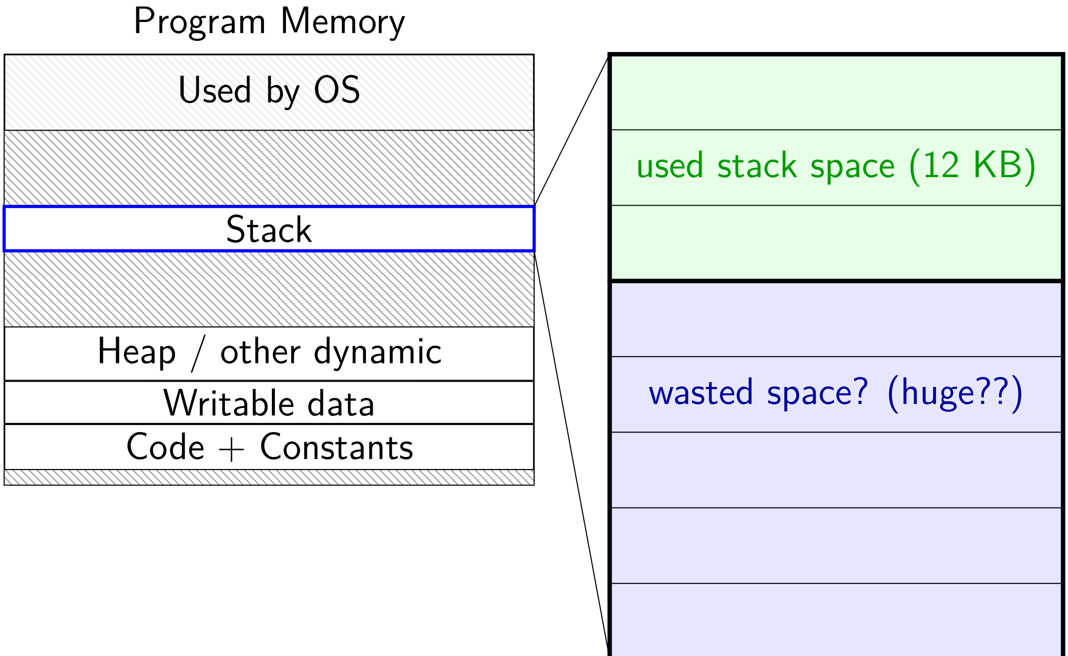
key idea: what processes *think* they have in memory != their actual memory

OS fixes disagreement from page fault handler

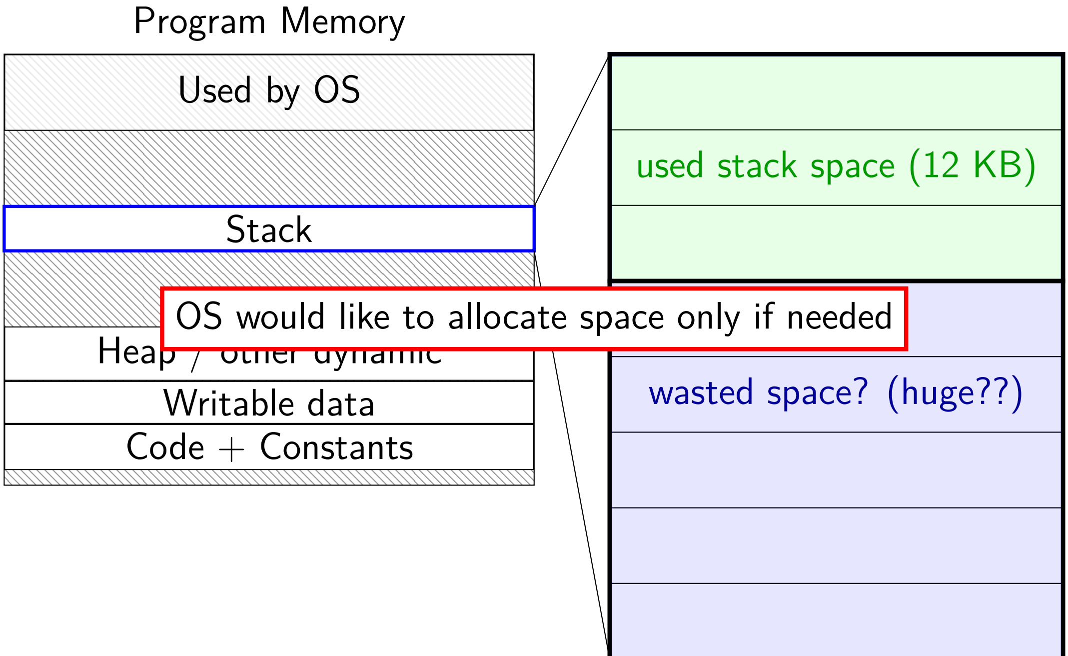
space on demand



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

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

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: if address $\geq$ sz
- kill process — out of bounds

on page fault: if address $<$ sz
- find virtual page number of address
- allocate page of memory, add to page table
- return from interrupt

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

fast copies

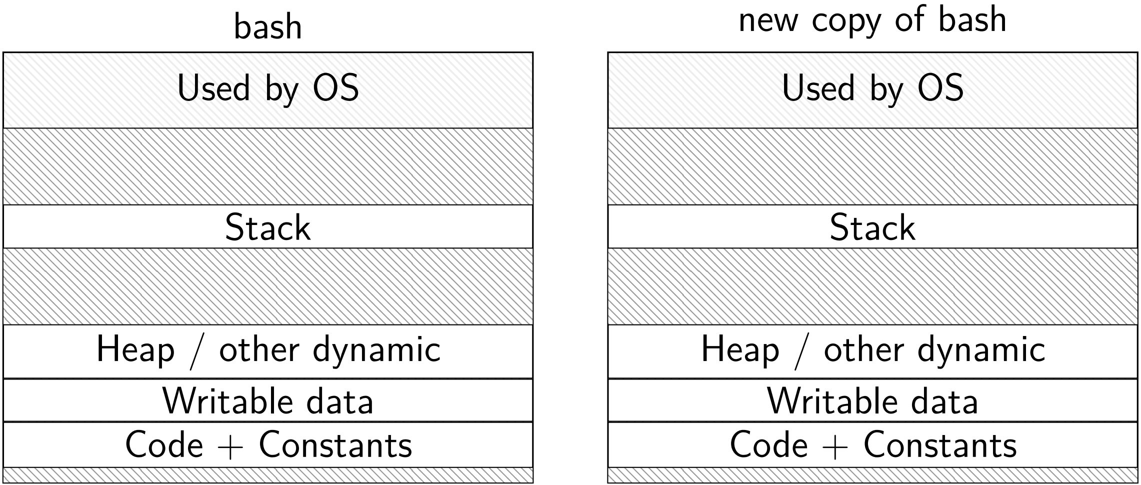
recall : `fork()`

creates a copy of an entire program!

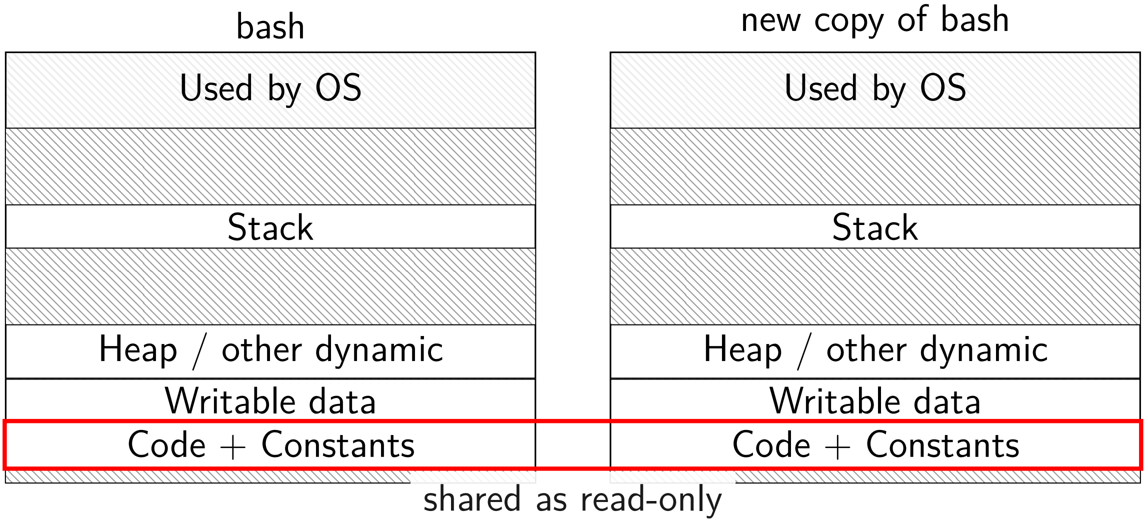
(usually, the copy then calls `execve` — replaces itself with another program)

how isn't this really slow?

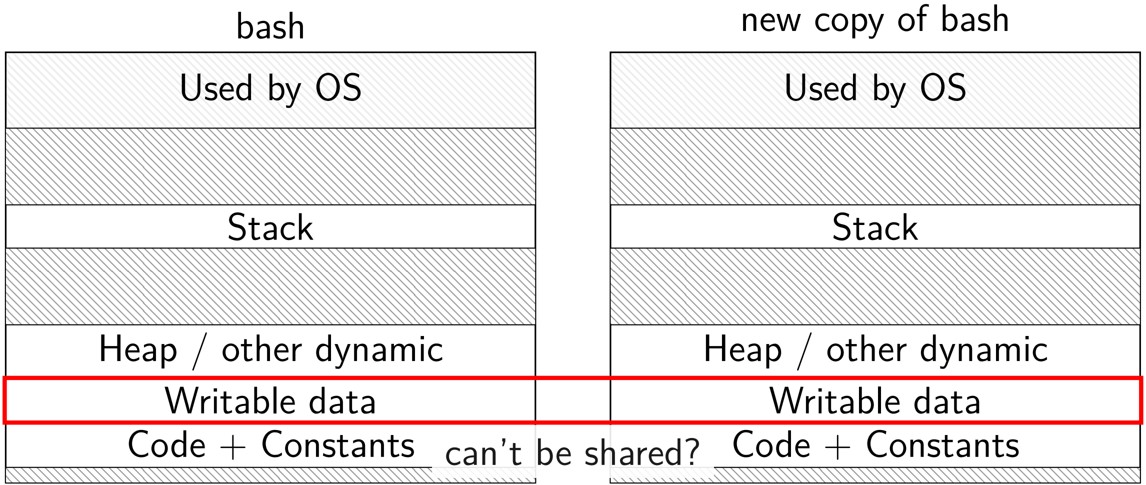
do we really need a complete copy?



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

copy-on write cases

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

- kill program instead of making it writable

trying to write read-only page and…

only one page table entry refers to it

- make it writeable
- return from fault

multiple process's page table entries refer to it

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

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?

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?

page cache components [text]

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

- handle cache hits

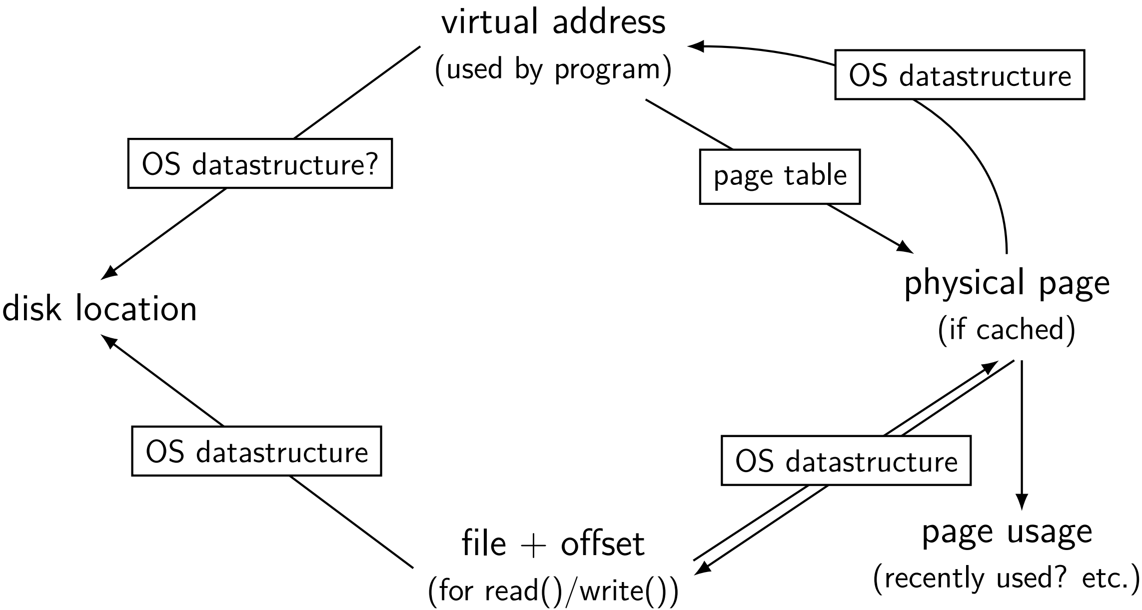
find backing location based on virtual address/file+offset

- handle cache misses

track information about each physical page

- handle page allocation
- handle cache eviction

page cache components



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)

```
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;
}
```

allocate first-level page table (“page directory”)

create new page table (kernel mappings)

initialize to 0 — every page invalid

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

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

create new page table (kernel mappings)

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

  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;
}
```

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)

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;
}
```

loading user pages from executable

```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, 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;
}
```

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

```
loaduvm(pde_t *pgdir, ch                                            , 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;
}
```

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

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                                 , 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

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

```
loaduv                                                    , 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;
}
```