# virtual memory 2

# changelog

23 March 2022: walkpgdir/mappages exercise: fix Q3 being cut off

25 March 2022: walkpgdir/mappages exercise: fix missing *s in explanation slide (not shown in lecutres)

27 March 2022: correct label of second second-level page table on quiz q 4 explanation

# last time

virtual addresses $\rightarrow$ physical addresses

dividing memory into pages

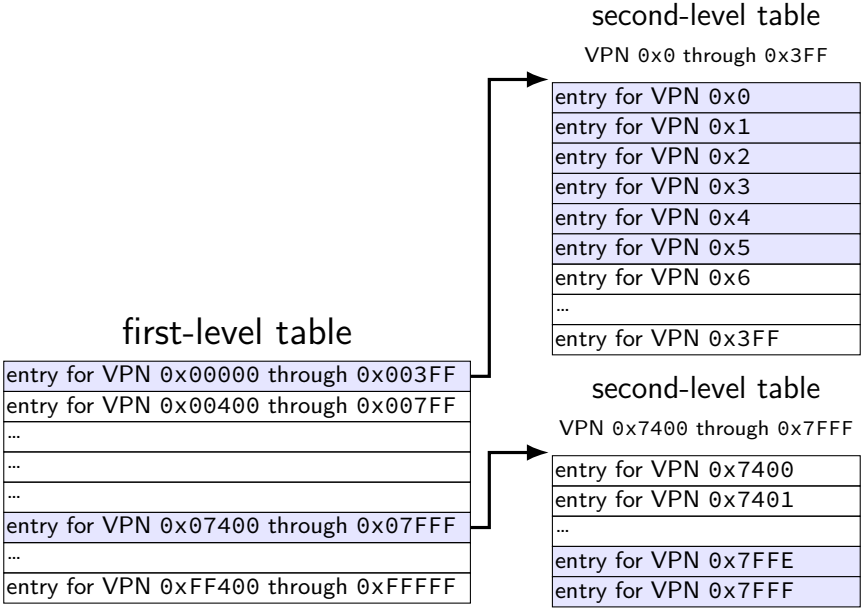
page tables: map virtual page # to physical page #

storing page tables in memory

two-level page tables

- each table: fixed-sized array of entries
- first part of virtual page number: first-level index
- second part of virtual page number: second-level index
- entries contain physical page numbers
- first-level, used to find second-level
- final level used to find actual code/data page

# Q4: page table layout (1)



# Q4: page table layout (2)

`0x0–0x3FFF`: virtual page numbers 0x0 through 0x3 (inclusive)

- first 10 bits: 0x0
- second 10 bits: 0x0 through 0x3

`0x4000–0x5FFF`: virtual page numbers 0x4 through 0x5

- first 10 bits: 0x0
- second 10 bits: 0x4 through 0x5

`0x7FFE000–0x7FFFFFF`: virtual page numbers 0x7FFE through 0x7FFF

- first 10 bits: 0x31
- second 10 bits: 0x3FE through 0x3FF

first-level table:

- entries at index 0x0 and 0x31

second-level table pointed to by index 0x0 of 1st level

second-level table pointed to by index 0x31 of 1st level

# Q5

for this lookup `0x4547` had VPN 0x4 and page offset 0x547.

- the second 10 bits of the 20-bit VPN are 0x4

second level page table entry = base address + 0x4 · 4 (bytes/entry) = 0x7010

- base address = physical address 0x7000 — physical page number 0x7

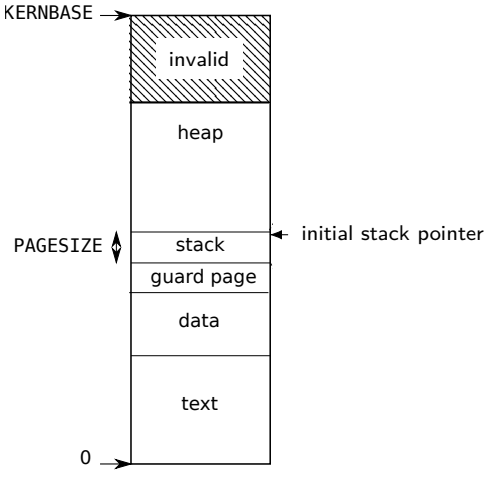
1st level page table entry
(20 bit page number of second-level table)(12 bits of flags)

20 bits are 0x7 — so must be `0x7???` where ??? are the flags

need present flag set: eliminate 0x7000, 0x7010

0x7007: present + writable + user-mode flags

# xv6 program memory



# xv6 page table-related functions

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

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

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

`mappages` — set range of page table entries

implementation: loop using `walkpgdir`

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

entries for `0x8000 0000` and up set

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

`allocuvm` — allocate new user memory

setup user-accessible memory

allocate new second-level tables as needed

`deallocuvm` — deallocate user memory

# xv6 page table-related functions

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

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

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

`mappages` — set range of page table entries

implementation: loop using `walkpgdir`

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

entries for `0x8000 0000` and up set

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

`allocuvm` — allocate new user memory

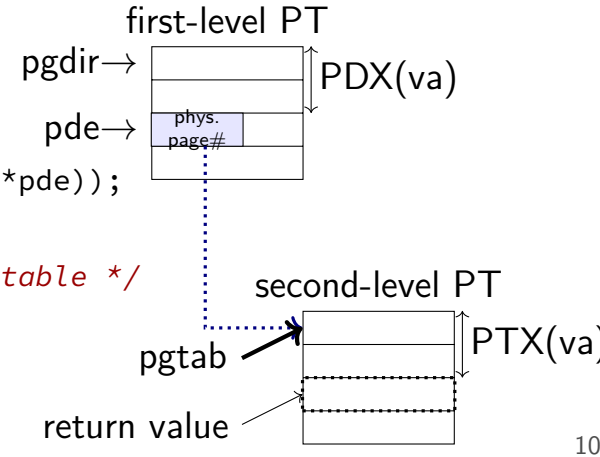
setup user-accessible memory

allocate new second-level tables as needed

`deallocuvm` — deallocate user memory

# xv6: finding page table entries

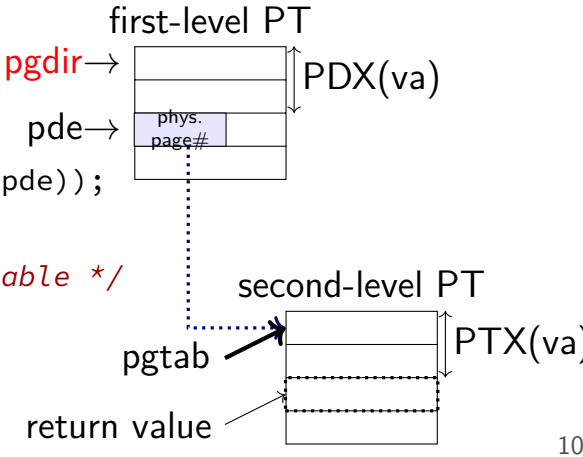
```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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')

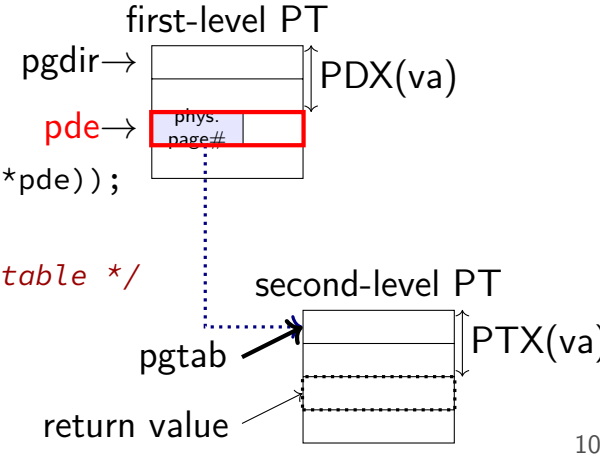
```
// Return t
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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) page table entry from first-level table ('page directory')

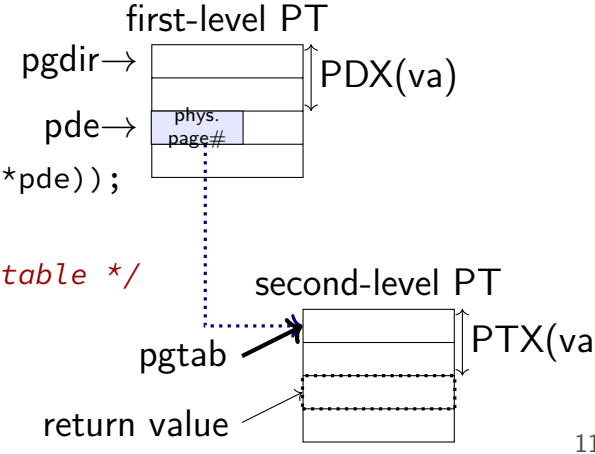
```
// Return the address of
// that corresponds to v
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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)];
}
```



# xv6: finding page table entries

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

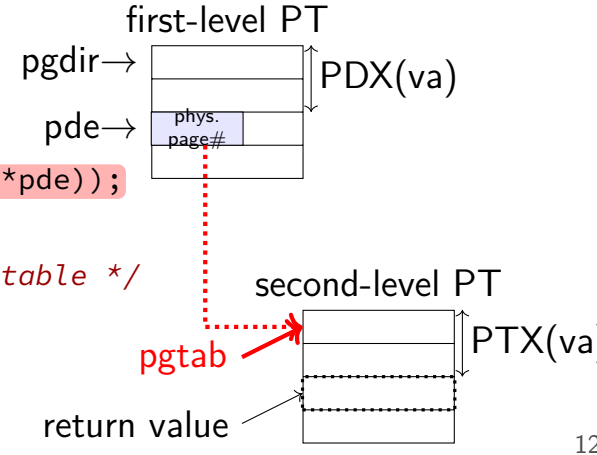
```
// Return the address of
// that corresponds to v
// create any required p
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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)];
}
```



# xv6: finding page table entries

retrieve location of second-level page table

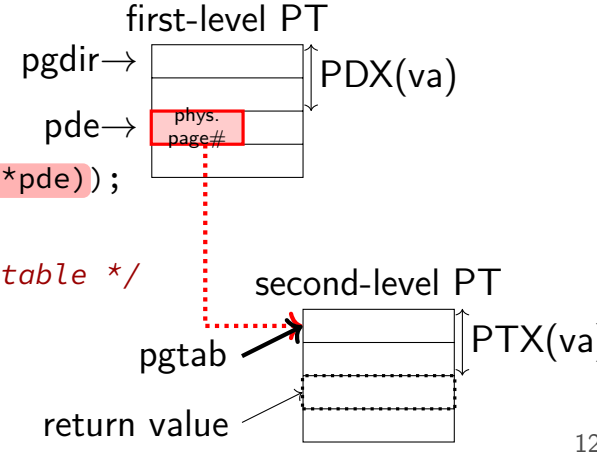
```
// Return the address o
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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

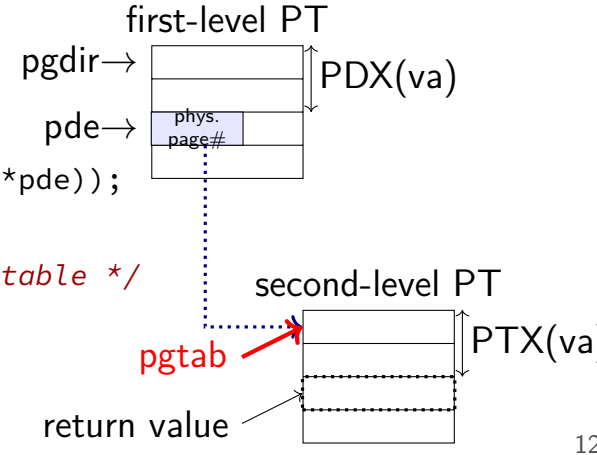
```
// Return the ad
// that correspo
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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

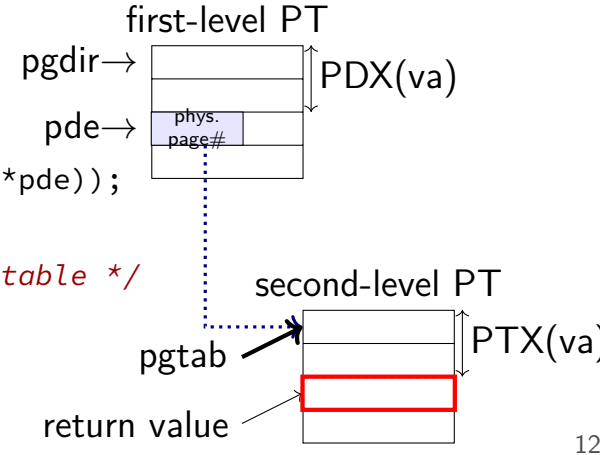
```
// Return the address
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  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

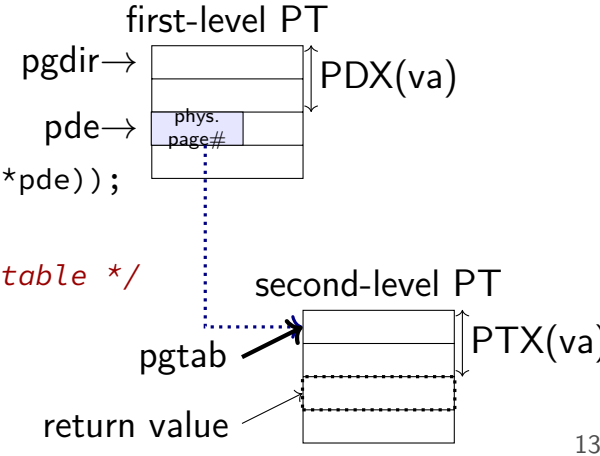
retrieve (pointer to) second-level page table entry from second-level table

```
// Return the addr
// that corresponc
// 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

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

```
  ...
  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;
  }
```

# xv6: creating second-level page tables

clear the new second-level page table
PTE = 0 → 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

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

```
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;
}
```

# xv6: setting last-level page entries

make sure it's not already set
in stock xv6: never change valid page table entry
in upcoming homework: this is not true

```
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;
}
```

# xv6: setting last-level page entries

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

```
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;
}
```

# xv6: setting last-level page entries

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

```
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;
}
```

# walkpgdir/mappages exercise

| VPN | user? | write? | PPN |
|---|---|---|---|
| 0x0 | yes | yes | 0x34 |
| 0x1 | no | no | (invalid) |
| 0x2 | yes | yes | 0x58 |
| 0x3 | no | no | (invalid) |
| 0x4 | no | no | (invalid) |
| … | … | … | … |
| 0x7FFFF | no | no | (invalid) |
| 0x80000 | no | yes | 0x0 |
| … | … | … | … |

effective value of **second-level** page table entries

Q1: result of `walkpgdir(this_pgdir, (void*)0x2, 0)`?

Q2a: result of `walkpgdir(this_pgdir, (void*)0x2345, 0)`?

Q2b: result of `walkpgdir(this_pgdir, (void*)0x23456, 1)`?

Q3: which entries change from `mappages(this_pgdir, (void*)0x3000, 0x2000, 0x55000, PTE_U|PTE_W)`?

# walkpgdir/mappages solutions

Q1: virtual address 0x2: virtual page number 0, page offset 0x2:
page table entry for VPN 0:
`PTE_ADDR(*value) == 0x34000`
(`0x34000` — physical address of first byte of page)

Q2a: virtual address 0x2345: virtual page number 0x2, page offset 0x345:
page table entry for VPN 2:
`PTE_ADDR(*value) == 0x58000`

Q2b: virtual address 0x23456: virtual page number 0x23, page offset 0x456:
page table entry for VPN 0x23:
`(*value & PTE_P) == 0` (invalid PTE)

Q3: virtual address 0x3000 (VPN 3, offset 0), plus 0x2000 bytes = 8KB = 2 pages
affects VPN 3, VPN 4 (two pages, starting at VPN 3)

# xv6 page table-related functions

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

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

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

`mappages` — set range of page table entries

implementation: loop using `walkpgdir`

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

entries for `0x8000 0000` and up set

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

`allocuvm` — allocate new user memory

setup user-accessible memory

allocate new second-level tables as needed

`deallocuvm` — deallocate user memory

# xv6 page table-related functions

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

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

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

`mappages` — set range of page table entries

implementation: loop using `walkpgdir`

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

entries for `0x8000 0000` and up set

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

`allocuvm` — allocate new user memory

setup user-accessible memory

allocate new second-level tables as needed

`deallocuvm` — deallocate user memory

# allocating user pages

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

# allocating user pages

```
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;
    }
  }
```

allocate a new, zero page

# allocating user pages

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

add page to second-level page table

# allocating user pages

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  ...
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += 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

# xv6 page table-related functions

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

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

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

`mappages` — set range of page table entries

implementation: loop using `walkpgdir`

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

entries for `0x8000 0000` and up set

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

`allocuvm` — allocate new user memory

setup user-accessible memory

allocate new second-level tables as needed

`deallocuvm` — deallocate user memory

# kalloc/kfree

kalloc/kfree — xv6's physical memory allocator

allocates/deallocates whole pages only

keep linked list of free pages

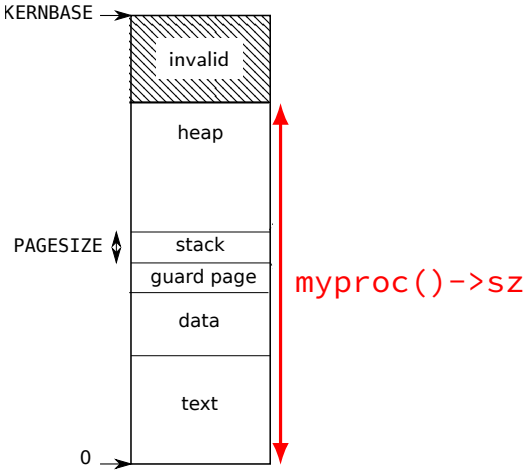
- list nodes — stored in corresponding free page itself
- kalloc — return first page in list
- kfree — add page to list

linked list created at boot

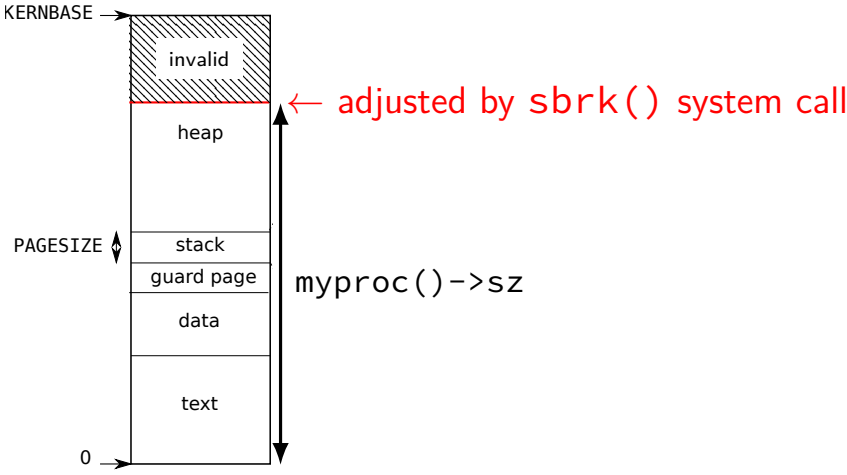
usuable memory fixed size (224MB)

- determined by PHYSTOP in `memlayout.h`

# xv6 program memory



# xv6 program memory



# xv6 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
    returning from pag

if so, setup the page table so it works next time
that is, immediately after returning from fault

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

pseudocode for xv6 implementation (for trap())

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

# backup slides

# x86-32 page table entries

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

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

# x86-32 page table entries

page table base register (CR3)

| 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 | | | | | | | | | | | | | | | | | | | | | | | | | | 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

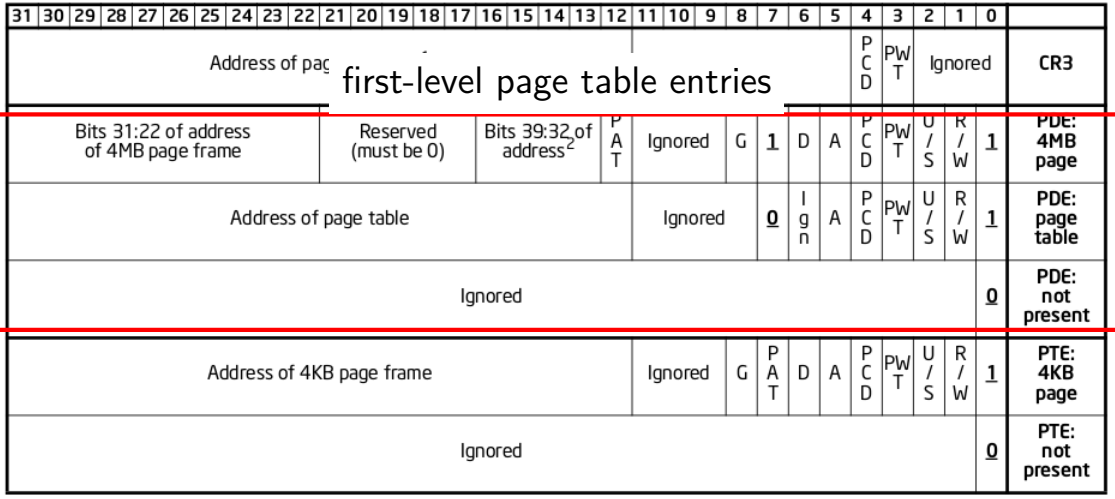


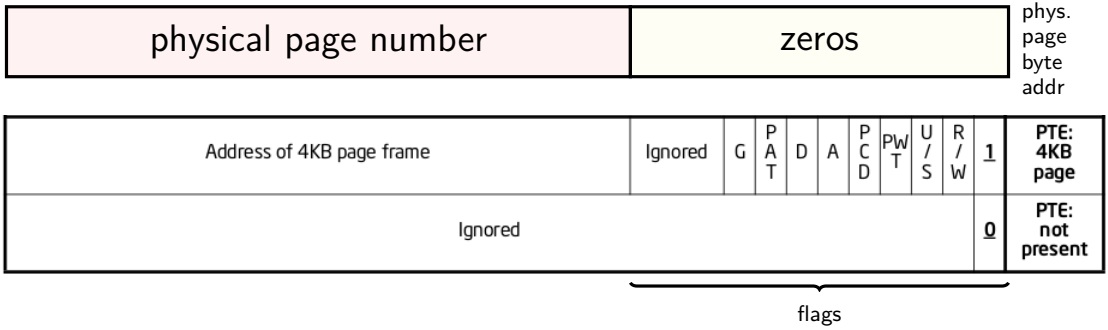

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

# x86-32 page table entries

| 31–12 | 11–9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Address of page directory[1] | Ignored | | | | | PCD | PWT | Ignored | | | CR3 |
| Bits 31:22 of address of 4MB page frame; Reserved (must be 0); Bits 39:32 of address[2]; PAT | Ignored | G | 1 | D | A | PCD | PWT | U/S | R/W | 1 | PDE: 4MB page |
| Address of page table | Ignored | | 0 | Ign | A | PCD | PWT | U/S | R/W | 1 | PDE: page table |
| second-level page table entries | | | | | | | | | | 0 | PDE: not present |
| Address of 4KB page frame | Ignored | G | PAT | D | A | PCD | PWT | U/S | R/W | 1 | PTE: 4KB page |
| Ignored | | | | | | | | | | 0 | PTE: not present |

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

# x86-32 page table entry v addresses




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));
}
```

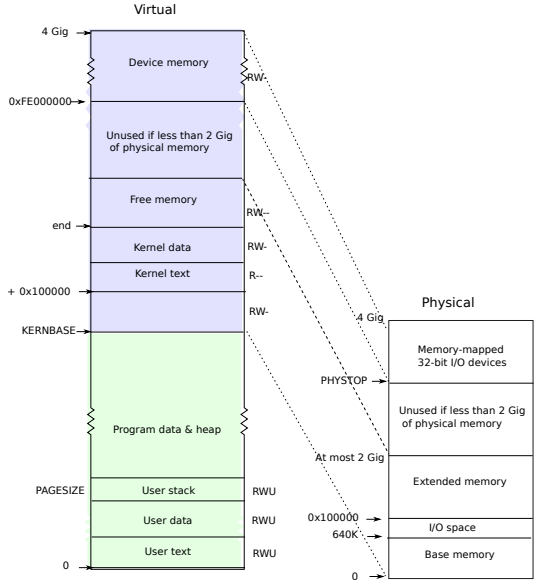
# xv6: extracting top-level page table entry

```
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

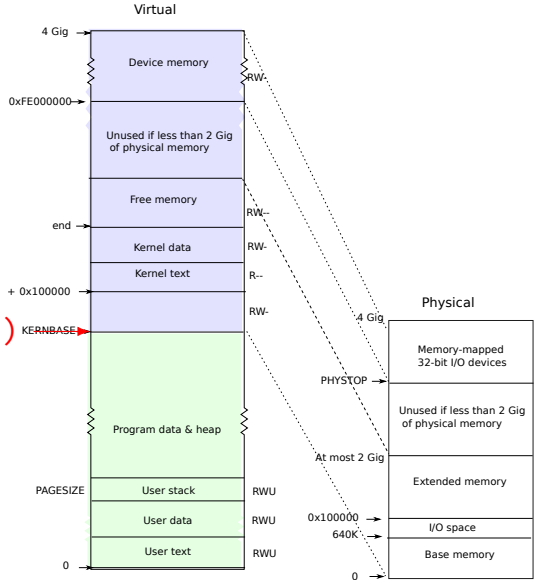
```
pde_t *some_page_table;  // if top-level table
pte_t *some_page_table;  // if next-level table
...
...
some_page_table[index] =
    PTE_P | PTE_W | PTE_U | base_physical_address;
/* P = present; W = writable; U = user-mode accessible */
```

# xv6 memory layout

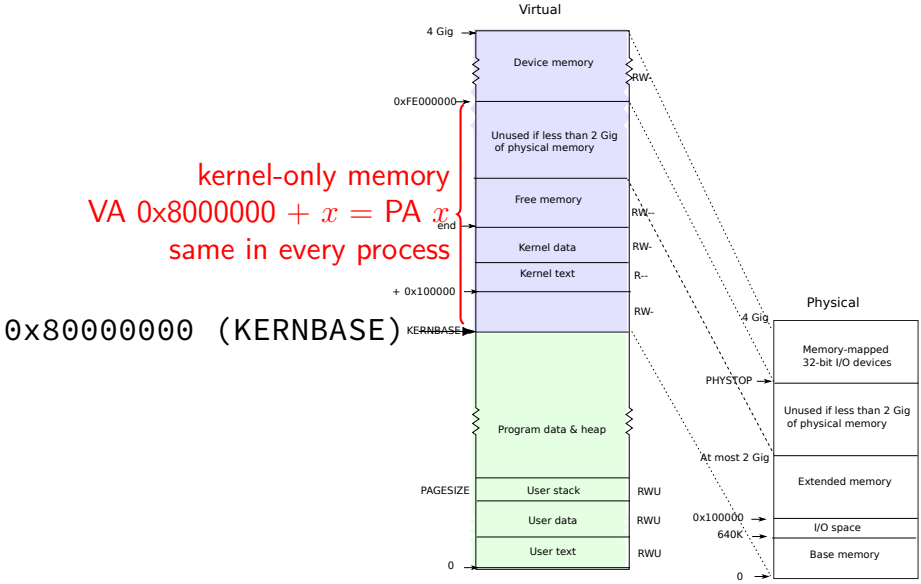


# xv6 memory layout

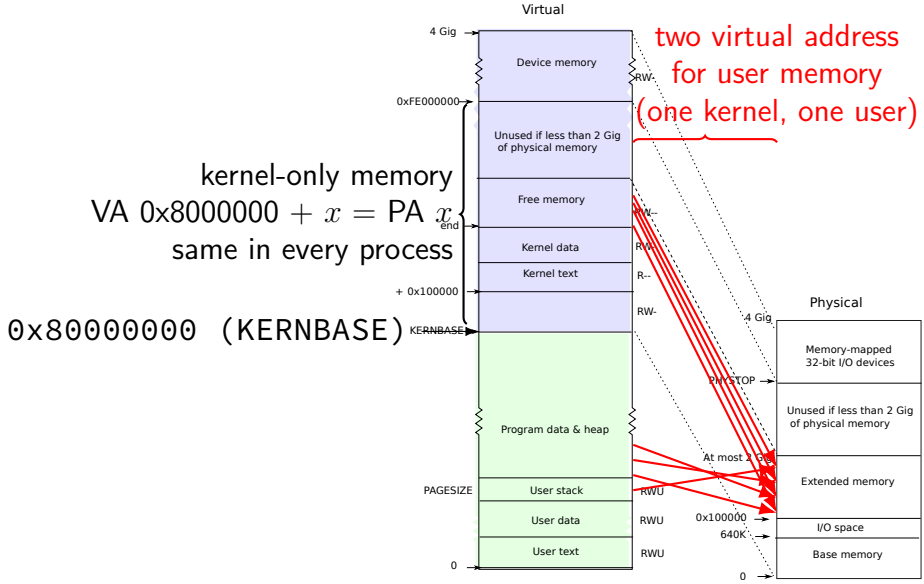
0x80000000 (KERNBASE)



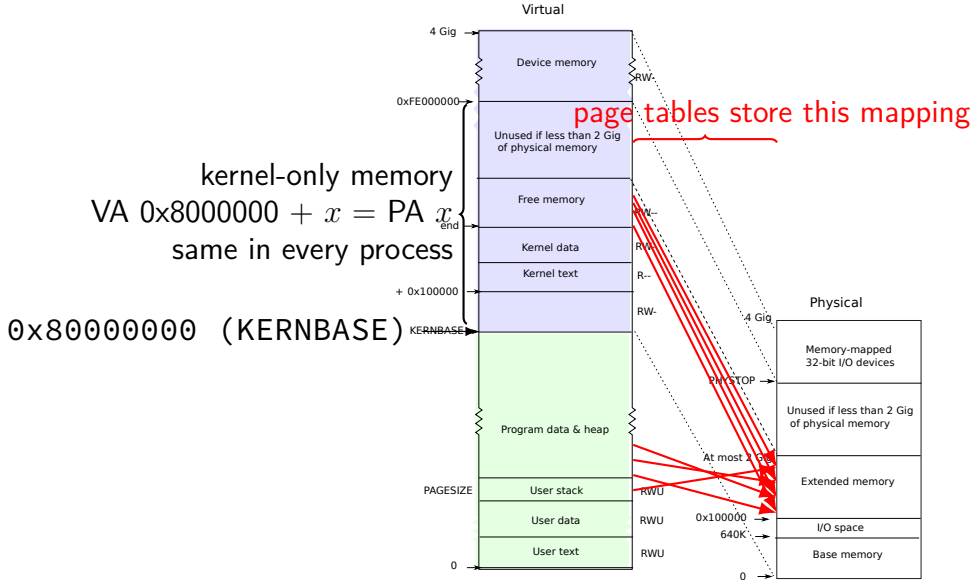
# xv6 memory layout



# 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 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

# skipping the guard page

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

---

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

# create new page table (kernel mappings)

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

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

# create new page table (kernel mappings)

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

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

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

# create new page table (kernel mappings)

initialize to 0 — every page invalid

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

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

# create new page table (kernel mappings)

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

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

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

# create new page table (kernel mappings)

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

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

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

# reading executables (headers)

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

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

# reading executables (headers)

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

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

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

# reading executables (headers)

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

`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;
```

# loading user pages from executable

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

# loading user pages from executable

```
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;
}
```

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

# loading user pages from executable

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

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

# loading user pages from executable

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

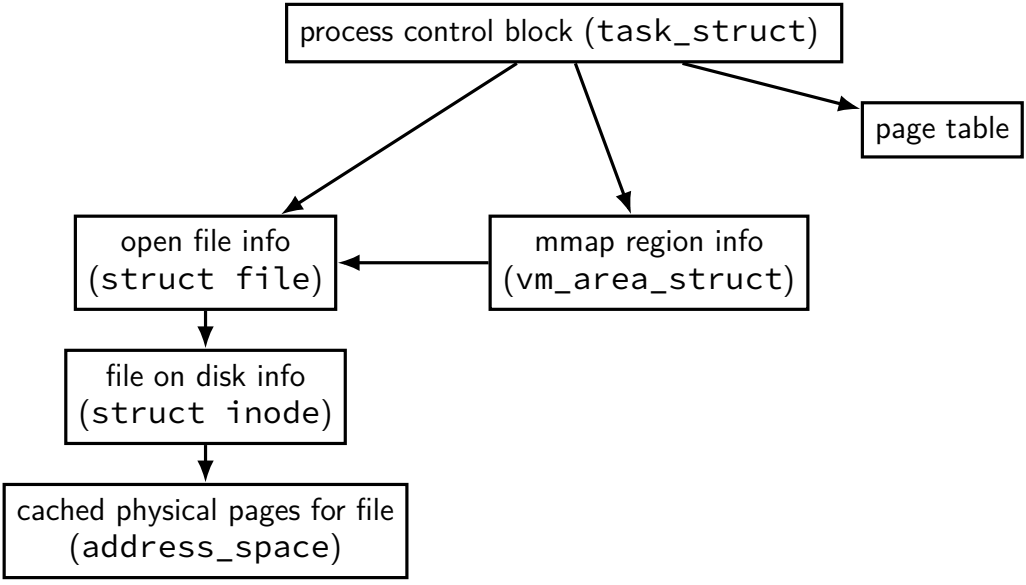
```
loaduvm(pde_t *pgdir                                    , uin
{
  ...
  for(i = 0; i < sz;
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip,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

# loading user pages from executable

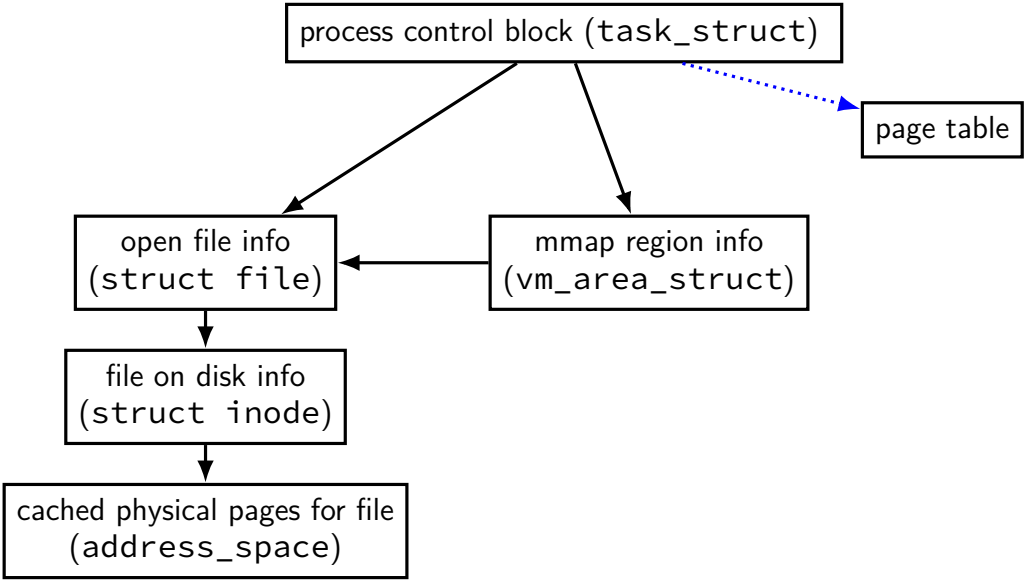
copy from file (represented by `struct inode`) into memory
P2V(pa) — mapping of physical addresss in kernel memory

```
loaduv                                                    , 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;
}
```

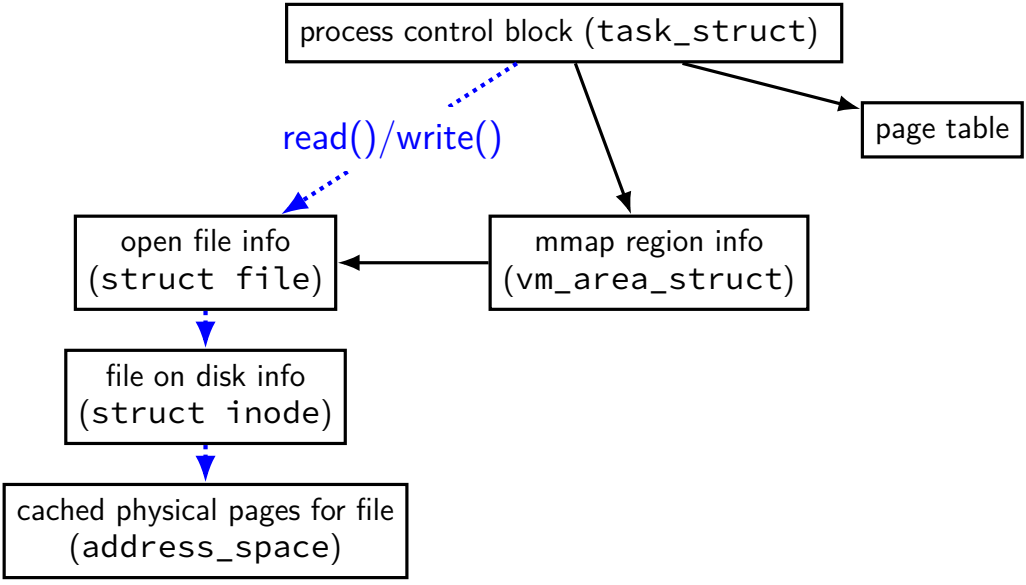
# Linux: forward mapping



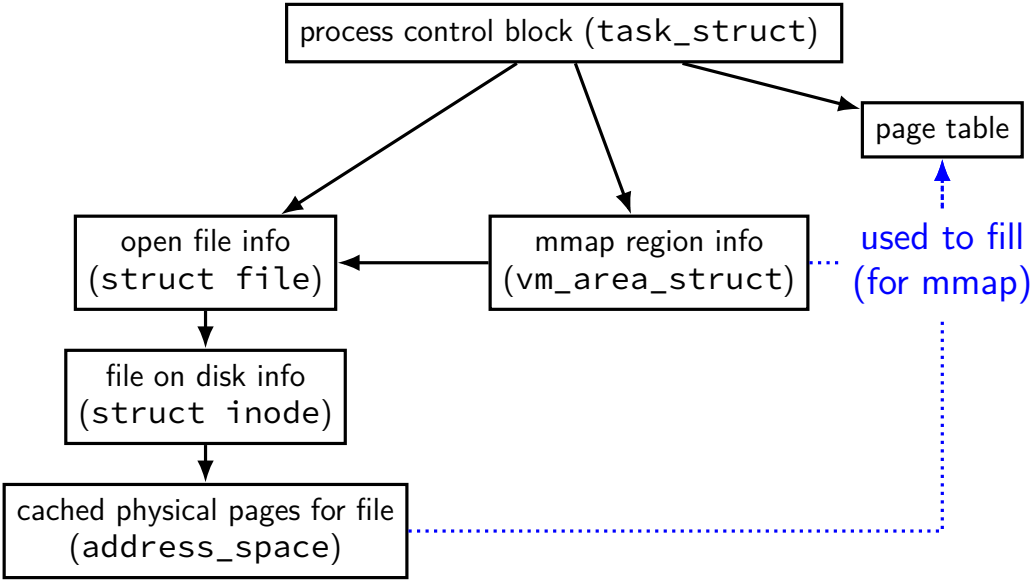
# Linux: forward mapping



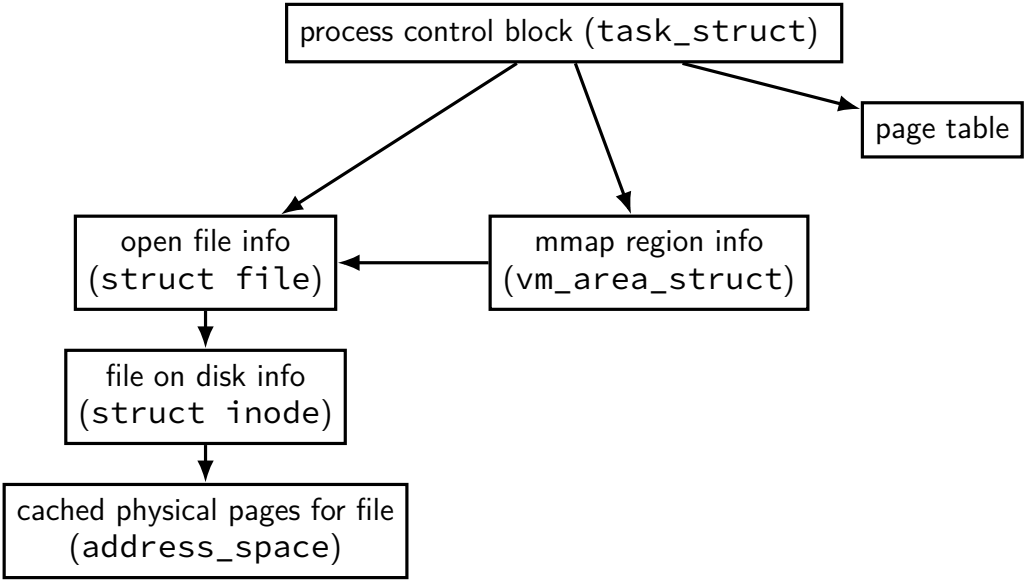
# Linux: forward mapping



# Linux: forward mapping



# Linux: forward mapping



# sketch: implementing mmap

access mapped file for first time, read from disk

(like swapping when memory was swapped out)

write "mapped" memory, write to disk eventually

need to detect whether writes happened

usually hardware support: dirty bit

extra detail: other processes should see changes

all accesses to file use same physical memory

how? OS tracks copies of files in memory

# xv6: setting process page tables (`exec()`)

exec step 1: create new page table with kernel mappings

- done in `setupkvm()`, which calls `mappages()`

exec step 2a: allocate memory for executable pages

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

exec step 2b: load from executable file

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

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

# xv6: setting process page tables (`exec()`)

exec step 1: create new page table with kernel mappings

- done in `setupkvm()`, which calls `mappages()`

exec step 2a: allocate memory for executable pages

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

exec step 2b: load from executable file

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

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

# minor and major faults

minor page fault

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

major page fault

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

# Linux: reporting minor/major faults

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

# predicting the future?

can't really...

look for common patterns

# working set intuition

say we're executing a loop

what memory does this require?

code for the loop

code for functions called in the loop

and functions they call

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

only uses a subset of the program's memory

# the working set model

one common pattern: working sets

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

...called its *working set*

rest of memory is inactive

...until program switches to different working set

# working sets and running many programs

give each program its working set

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

inactive — won't be used

# working sets and running many programs

give each program its working set

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

inactive — won't be used

replacement policy: identify working sets $\approx$ recently used data

replace anything that's not in in it

# cache size versus miss rate

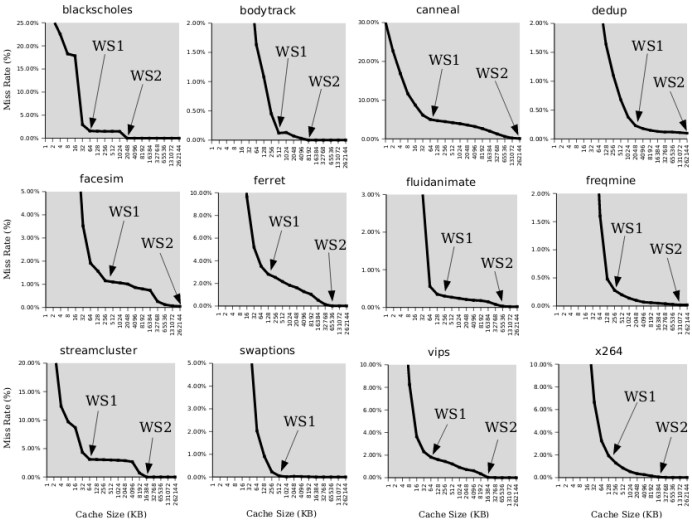



**Figure 3: Miss rates versus cache size. Data assumes a shared 4-way associative cache with 64 byte lines. WS1 and WS2 refer to important working sets which we analyze in more detail in Table 2. Cache requirements of PARSEC benchmark programs can reach hundreds of megabytes.**

# estimating working sets

working set $\approx$ what's been used recently

except when program switching working sets

so, what a program recently used $\approx$ working set

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

# estimating working sets

working set $\approx$ what's been used recently

except when program switching working sets

so, what a program recently used $\approx$ working set

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

# CLOCK-Pro: special casing for one-use pages

by default, Linux tries to handle scanning of files

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

basic idea: delay considering pages active until second access

- second access = second scan of accessed bits/etc.

single scans of file won't "pollute" cache

without this change: reading large files slows down other programs

- recently read part of large file steals space from active programs

# readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

when to start reads?

how much to readahead?

# readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern
- not enough to look at misses!
- want to check when readahead pages are used — keep up with program

when to start reads?

how much to readahead?

# readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern
- not enough to look at misses!
- want to check when readahead pages are used — keep up with program

when to start reads?

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

how much to readahead?

# readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern
- not enough to look at misses!
- want to check when readahead pages are used — keep up with program

when to start reads?

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

how much to readahead?

- if too much: evict other stuff programs need
- if too little: won't keep up with program
- if too little: won't make efficient use of HDD/SSD/etc.

# recording accesses

goal: "check is this physical page still being used?"

software support: temporarily mark page table invalid

- use resulting page fault to detect "yes"

hardware support: accessed bits in page tables

- hardware sets to 1 when accessed

# temporarily invalid PTE (software support)

program 1

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

the kernel

…
(OS exception's handler)
…

page table for program 1

| VPN | present? | writable? | … | PPN |
|---|---|---|---|---|
| 0x00000 | 0 | --- | … | --- |
| 0x00001 | 0 | --- | … | --- |
| … | … | … | … | … |
| 0x00123 | 0 | 0 | … | 0x4442 |
| … | … | … | … | … |

OS page info

| PPN | last known access? | … |
|---|---|---|
| … | … | … |
| 0x04442 | (never) | … |
| … | … | … |

# temporarily invalid PTE (software support)

program 1

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

the kernel

```
…
(OS exception's handler)
…
```

oops! page fault

processor does lookup

page table for program 1

| VPN | present? | writable? | … | PPN |
|---|---|---|---|---|
| 0x00000 | 0 | --- | … | --- |
| 0x00001 | 0 | --- | … | --- |
| … | … | … | … | … |
| 0x00123 | 0 | 0 | … | 0x4442 |
| … | … | … | … | … |

OS page info

| PPN | last known access? | … |
|---|---|---|
| … | … | … |
| 0x04442 | (never) | … |
| … | … | … |

# temporarily invalid PTE (software support)

program 1

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

the kernel

…
(OS exception's handler)
…

update page info +
mark present

page table for program 1

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

OS page info

| PPN | last known access? | … |
|---|---|---|
| … | … | … |
| 0x04442 | at time X | |
| … | … | … |

# temporarily invalid PTE (software support)

program 1

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

the kernel

```
…
(OS exception's handler)
…
```

processor does lookup

no page fault, not recorded in OS info

page table for program 1

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

OS page info

| PPN | last known access? | … |
|---|---|---|
| … | … | … |
| 0x04442 | at time X | … |
| … | … | … |

# temporarily invalid PTE (software support)

program 1

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

the kernel

```
…
(OS exception's handler)
…
```

processor does lookup

no page fault, not recorded in OS info

page table for program 1

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

OS page info

| PPN | last known access? | … |
|---|---|---|
| … | … | … |
| 0x04442 | at time X | … |
| … | … | … |

# temporarily invalid PTE (software support)

program 1

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

the kernel

```
…
(OS exception's handler)
…
```

OS clears present bit
to check for next access

page table for program 1

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

OS page info

| PPN | last known access? | … |
|---|---|---|
| … | … | … |
| 0x04442 | at time X | … |
| … | … | … |

# temporarily invalid PTE (software support)

program 1

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

the kernel

```
…
(OS exception's handler)
…
```

OS clears present bit
to check for next access

page table for program 1

| VPN | present? | writable? | … | PPN |
|---|---|---|---|---|
| 0x00000 | 0 | --- | … | --- |
| 0x00001 | 0 | --- | … | --- |
| … | … | … | … | … |
| 0x00123 | 0 | 0 | … | 0x4442 |
| … | … | … | … | … |

OS page info

| PPN | last known access? | … |
|---|---|---|
| … | … | … |
| 0x04442 | at time X | … |
| … | … | … |

# temporarily invalid PTE (software support)

program 1

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

the kernel

…
(OS exception's handler)
…

oops! page fault

processor does lookup

page table for program 1

| VPN | present? | writable? | … | PPN |
|---|---|---|---|---|
| 0x00000 | 0 | --- | … | --- |
| 0x00001 | 0 | --- | … | --- |
| … | … | … | … | … |
| 0x00123 | 0 | 0 | … | 0x4442 |
| … | … | … | … | … |

OS page info

| PPN | last known access? | … |
|---|---|---|
| … | … | … |
| 0x04442 | at time X | … |
| … | … | … |

# temporarily invalid PTE (software support)

program 1

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

the kernel

…
(OS exception's handler)
…

update page info + mark present

page table for program 1

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

OS page info

| PPN | last known access? | … |
|---|---|---|
| … | … | … |
| 0x04442 | at time Y | |
| … | … | … |

# accessed bit usage (hardware support)

program 1

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

the kernel

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

page table for program 1

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

# accessed bit usage (hardware support)

program 1

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

the kernel

…
(OS exception's handler)
…

processor does lookup
sets accessed bit to 1

page table for program 1

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

# accessed bit usage (hardware support)

program 1

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

the kernel

…
(OS exception's handler)
…

processor does lookup
sets accessed bit to 1

page table for program 1

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

# accessed bit usage (hardware support)

program 1

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

the kernel

…
(OS exception's handler)
…

processor does lookup
keeps access bit set to 1

page table for program 1

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

# accessed bit usage (hardware support)

program 1

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

the kernel

…
(OS exception's handler)
…

processor does lookup
keeps access bit set to 1

page table for program 1

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

# accessed bit usage (hardware support)

program 1

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

the kernel

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

OS reads + records + clears access bit

page table for program 1

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

# accessed bit usage (hardware support)

program 1

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

the kernel

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

OS reads + records + clears access bit

page table for program 1

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

# accessed bit usage (hardware support)

program 1

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

the kernel

…
(OS exception's handler)
…

processor does lookup
sets accessed bit to 1 (again)

page table for program 1

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

# accessed bit usage (hardware support)

program 1

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

the kernel

…
(OS exception's handler)
…

processor does lookup
sets accessed bit to 1 (again)

page table for program 1

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

# accessed bits: multiple processes

page table for program 1

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

page table for program 2

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

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

# dirty bits

"was this part of the mmap'd file changed?"

"is the old swapped copy still up to date?"

software support: temporarily mark read-only

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

same idea as accessed bit, but only changed on writes

# x86-32 accessed and dirty bit

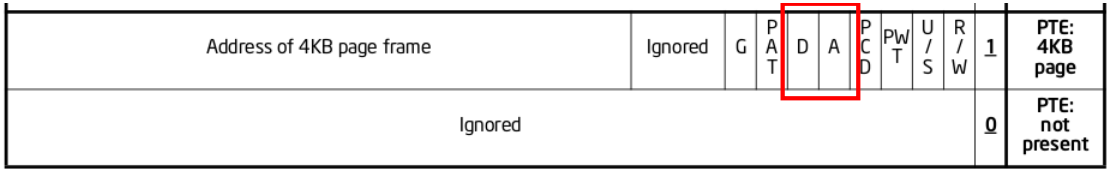



Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

A: acccessed — processor sets to 1 when PTE used

used = for read or write or execute

likely implementation: part of loading PTE into TLB

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

# lazy replacement?

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

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

# lazy replacement?

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

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

but real OSes are more proactive

# non-lazy writeback

what happens when a computer loses power

how much data can you lose?

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

- no changed data written back

solution: track or scan for dirty pages and writeback

example goals:

- lose no more than 90 seconds of data
- force writeback at file close
- ...

# non-lazy eviction

so far — allocating memory involves evicting pages

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

# non-lazy eviction

so far — allocating memory involves evicting pages

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

alternative: evict earlier "in the background"

- "free": probably have some idle processor time anyways

allocation = remove already evicted page from linked list

- (instead of changing page tables, file cache info, etc.)

# backup slides