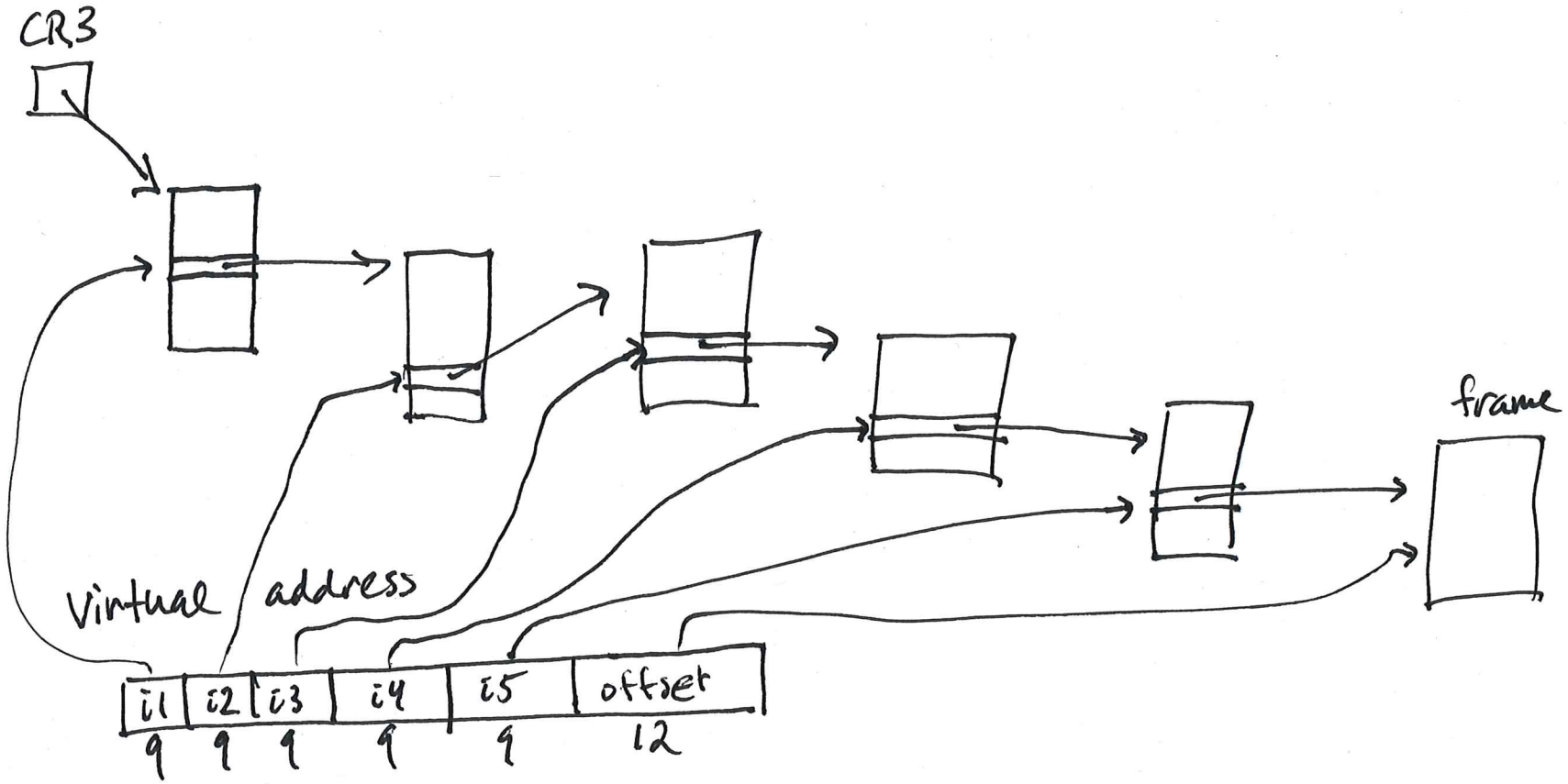
Lec 14

TLBs + Superpages

Hardware address translation



- How many memory references required for translation?
- How can processes share memory?
  - ↳ how does OS track sharing?

OS paging data structures

- separate from hardware data structures
  - portability
  - additional features: copy on write, stack growth, etc.
- code to keep hardware in sync

```
// 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.
pte_t * walkpml4(pml4e_t *pml4, const void *va, int alloc)
{
  pml4e_t *pml4e;              pdpte_t *pdpt, *pdpte;
  pde_t *pgdir, *pde;          pte_t *pgtab;

  pml4e = &pml4[PML4_INDEX(va)];
  if (*pml4e & PTE_P) {
    pdpt = (pdpte_t*)P2V(PDPT_ADDR(*pml4e));
  } else {
    if(!alloc || (pdpt = (pdpte_t*)kalloc()) == 0)
      return 0;
    memset(pdpt, 0, PGSIZE);
    *pml4e = V2P(pdpt) | PTE_P | PTE_W | PTE_U;
  }

  pdpte = &pdpt[PDPT_INDEX(va)];
  if (*pdpte & PTE_P) {
    pgdir = (pde_t*)P2V(PDE_ADDR(*pdpte));
  } else {
    if(!alloc || (pgdir = (pde_t*)kalloc()) == 0)
      return 0;
    memset(pgdir, 0, PGSIZE);
    *pdpte = V2P(pgdir) | PTE_P | PTE_W | PTE_U;
  }

  pde = &pgdir[PD_INDEX(va)];
  if (*pde & PTE_P) {
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    memset(pgtab, 0, PGSIZE);
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }

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

14.3

```
void scheduler(void) {
  struct proc *p;

  for (;;) {
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      mycpu()->proc = p;
      vspaceinstall(p);
      p->state = RUNNING;
      swtch(&mycpu()->scheduler, p->context);
      vspaceinstallkern();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      mycpu()->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

14.4

```
void vspaceinstall(struct proc *p)
{
  if (!p)
    panic("mrinstall: null proc");
  if (!p->kstack)
    panic("mrinstall: null kstack");
  if (!p->vspace.pgtbl)
    panic("mrinstall: page table not initialized");

  pushcli();  // turn off interrupts
  mycpu()->ts.rsp0 = (uint64_t)p->kstack + KSTACKSIZE;
  lcr3(V2P(p->vspace.pgtbl));
  popcli();  // turns on interrupts
}

void vspaceinstallkern(void)
{
  lcr3(V2P(kpml4));
}
```

```
struct vspace {
  struct vregion regions[NREGIONS];
  pml4e_t* pgtbl;
};

struct vregion {
  enum vr_direction dir;   // Direction that the vregion grows
  uint64_t va_base;
  uint64_t size;           // Number of bytes in the vregion.

  struct vpi_page *pages;
};

struct vpi_page {
  struct vpage_info infos[VPIPPAGE];  // info struct for the given page
  struct vpi_page *next;              // the next page
};

struct vpage_info {
  short used;     // 1 if the page is in use.
  uint64_t ppn;   // physical page number
  short present;  // whether the page is in physical memory
  short writable; // does the page have write permissions
};
```

14.6

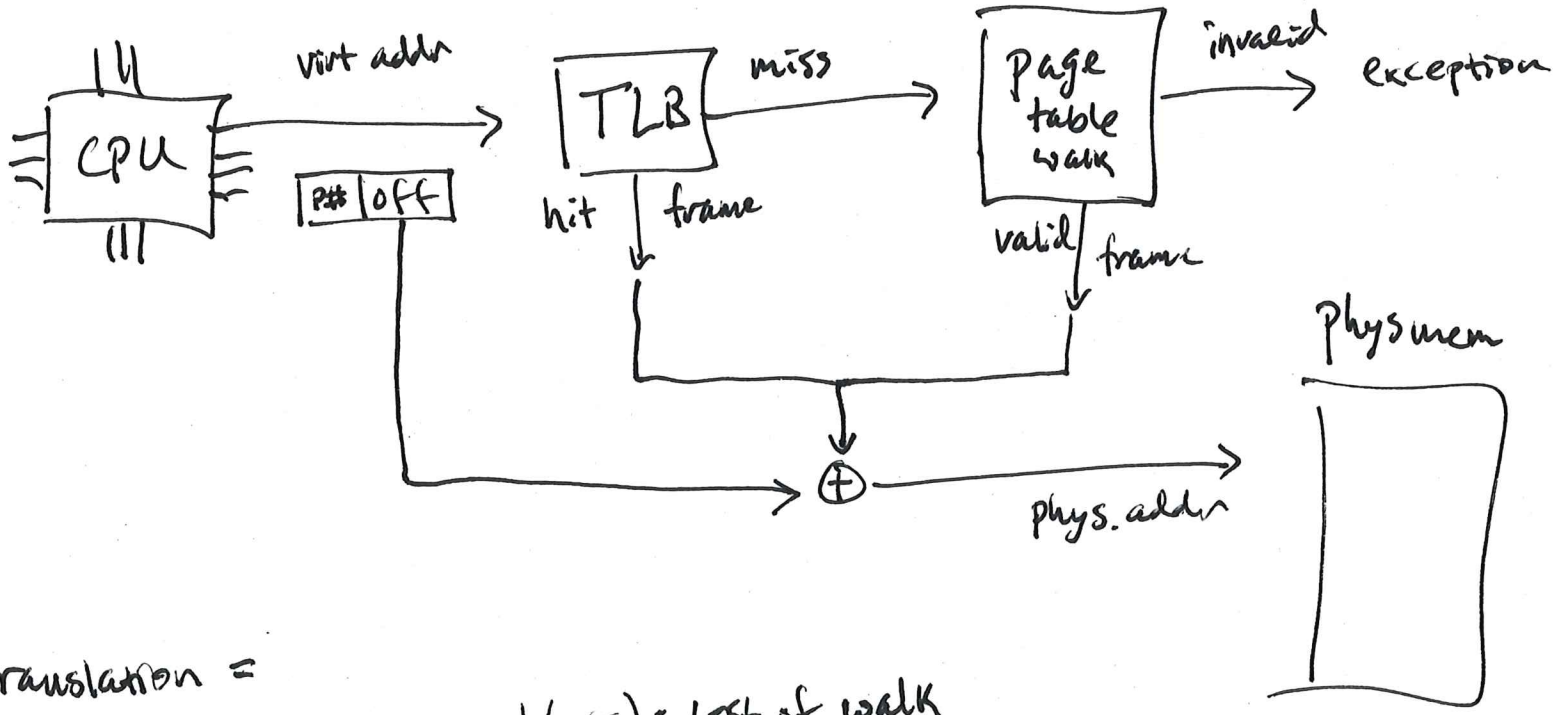
Using virtual memory to do stuff

- basic idea: set up page table so that hardware will interrupt
  - then implement feature in handler

- examples:
  - copy-on-write
  - stack growth (transparent)
  - demand paging

Translation Lookaside Buffers (TLBs)

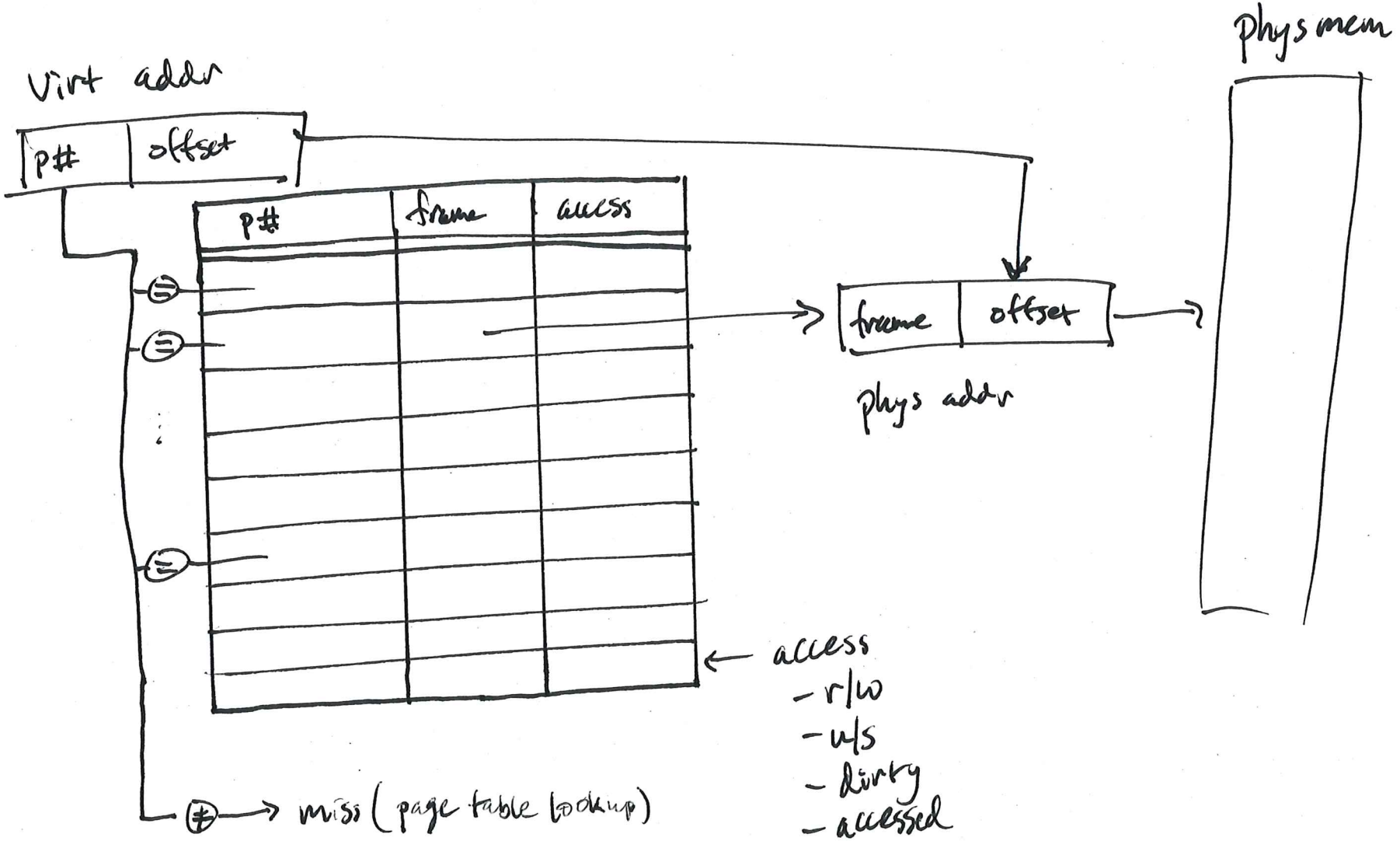
- cache virtual → physical address translation
  - avoids multilevel page walk in common case



Cost of translation =

cost of TLB lookup + prob(miss) · cost of walk

TLB Lookup


Phys mem
Virt addr
p#
offset
p#
frame
access
frame
offset
phys addr
access
- r/w
- u/s
- dirty
- accessed
miss (page table lookup)

TLB consistency (coherence)

- context switch changes which page table is in use
  - hardware must know not to reuse TLB entries from other process
    - ↳ tag each entry w/ a process identifier or invalidate all entries on context switch (xK)
- what if OS changes page table entry? (eg grow stack)
  - adding permissions is no problem; out of date entry causes fault (and OS can handle)
  - removing permissions requires invalidating old entries (invlpg on x86)
- on multicore, ("TLB shootdown") ask each CPU to invalidate its TLB

14.10

Superpages

- architecture support for mapping large contiguous regions of memory

- on x86, can replace a page table of contiguous mapping

  $2^9 = 512$ entries

  each mapping 2^{12} bytes

  ↳ replace with one mapping of $2^{21}_{bytes} = 2MB$ $\quad 9+12=21$

- or replace page directory for another 2^9:

  2^{30} bytes $= 1GB$

- super pages make TLB more efficient

14.11