EECS-343 Operating Systems
Lecture 5:
Virtual Memory & Paging

Steve Tarzia

Spring 2019

Northwestern

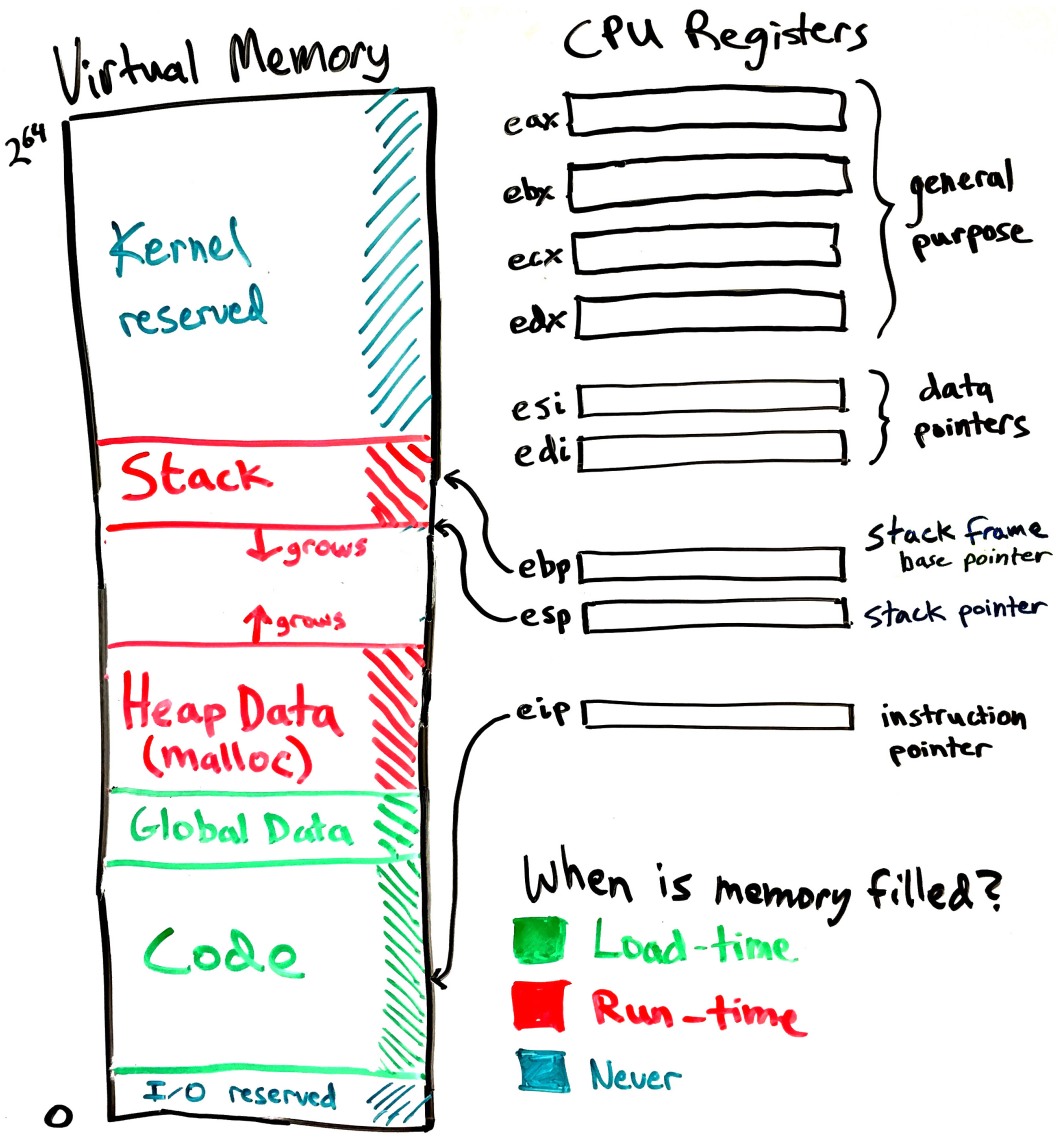
Announcements

- Project 1 due yesterday
- Late policy: up to two days at 10% off per day
- HW 1 due Monday
- Project 2 due the following Monday
  - It's much more difficult than Project 1!
- Additional weekly office hours: Tuesdays 2-4pm in Wilkinson

Last Lecture

- Defined two conflicting metrics: ***turnaround time*** and ***response time***
  - Cannot optimize both – must tradeoff, or balance, the two
- Optimized by ***shortest job first*** and ***round robin***, respectively
- Context switching overhead is due to the CPU caches
  - CPU keeps most recently used data in nearby caches, so it's more efficient to let an ongoing process continue.
- ***I/O-blocked*** processes make progress without using the CPU
  - We should prioritize I/O-bound processes
- ***Multi-Level Feedback Queues*** are often used in real OS schedulers
  - Prioritizes "polite" processes that use little CPU time when scheduled
  - CPU-bound processes squander their time quotas and lose priority

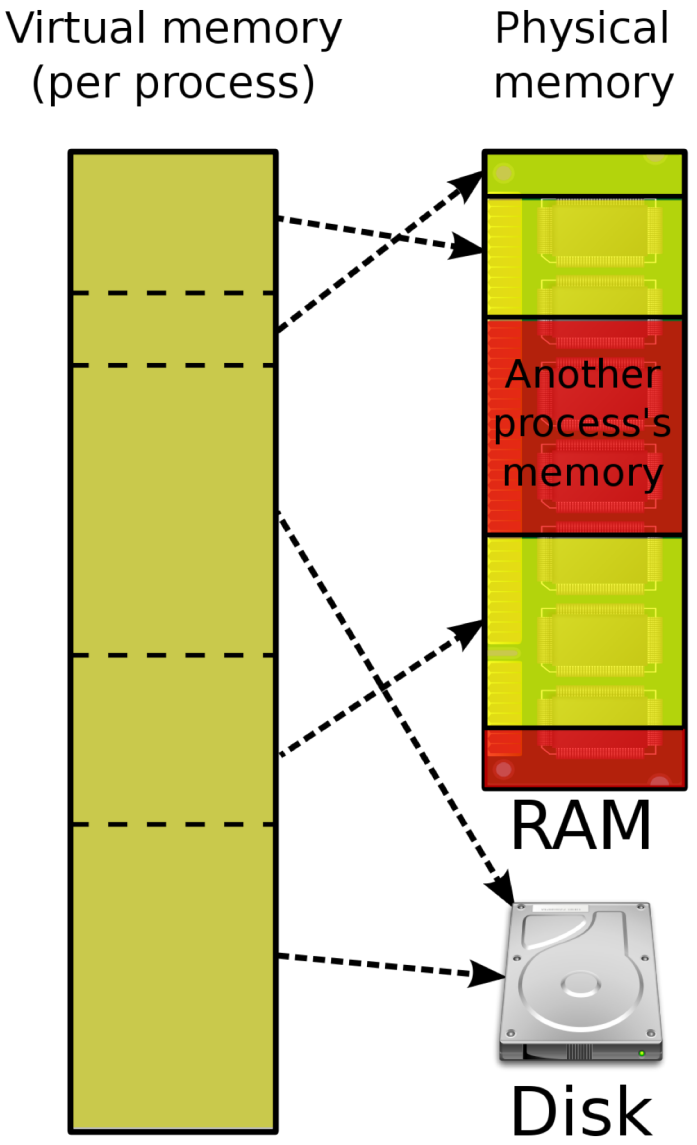
Program's view of the Machine



- Programs are compiled to run **alone** on a machine with **lots of memory**.
  - "64-bit" machine = 2^{64} bytes of memory
  - "32-bit" machine = 2^{32} bytes = 4GB of memory (xv6 is a 32-bit OS).
- In reality:
  - Programs share memory with others
  - Machine has less than the maximum amount of memory.

Virtual Memory

- Splits memory between various processes,
- But gives each process the illusion that it has the full address space.
  - Code uses ***virtual memory addresses***
    - Called "logical address" by Intel
  - CPU somehow translates to ***physical addresses*** assigned to that process
- Virtual memory also gives the illusion that memory is huge.
  - OS ***swaps*** memory to disk if there is no space in physical memory. (More on this in later lectures.)
- So, no memory is moved during context switch.
- We just need to configure the CPU to use a different virtual-to-physical address translation

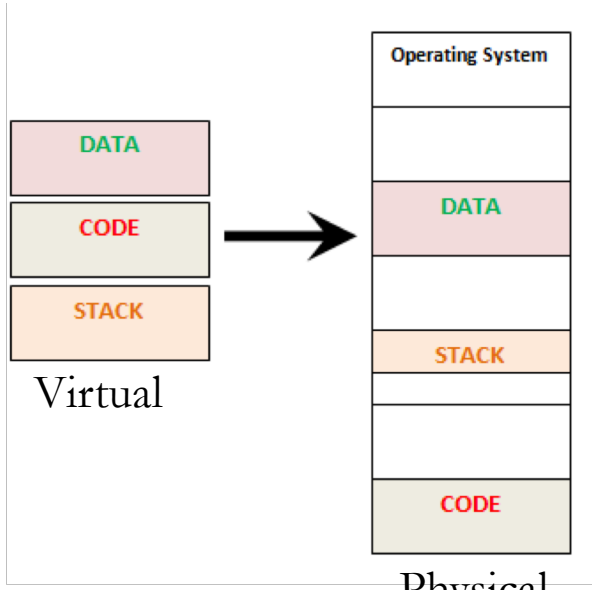


Virtual Memory

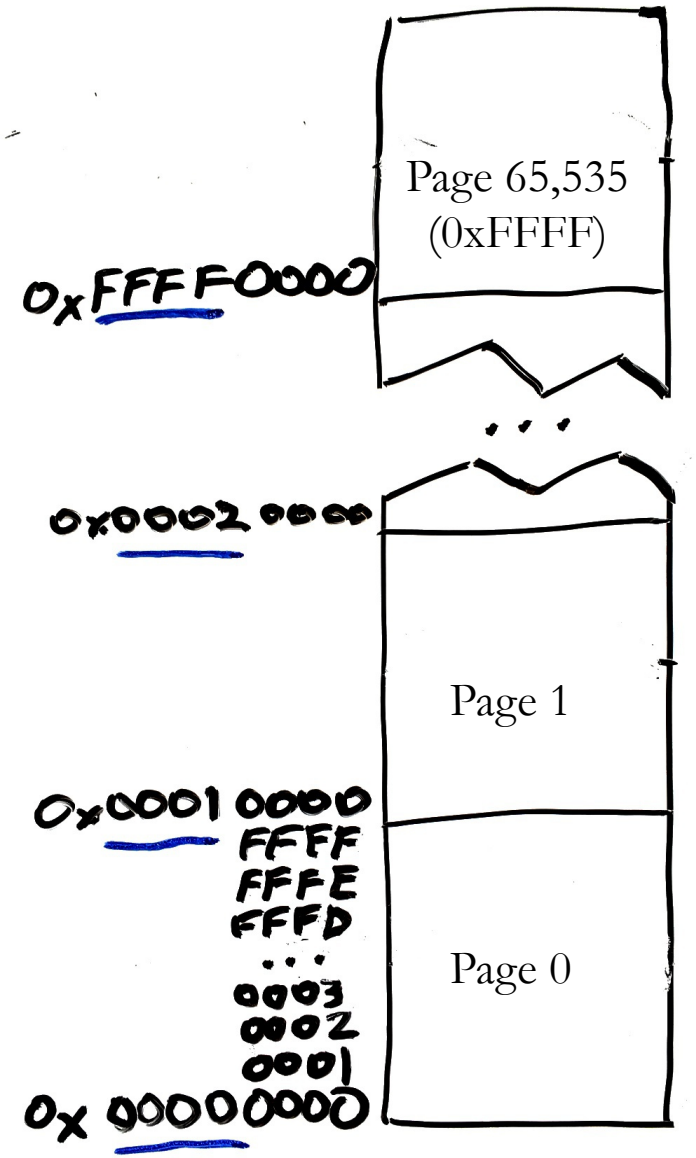
- Virtual Memory allows the OS and hardware to control how CPU instructions' memory addresses are translated to physical memory addresses.
  - adds a layer of ***indirection*** between programs and the physical memory.
- Virtual Memory gives each programs the illusion that it is not competing with other processes for use of the machine's memory
  - This does not exactly mean that a program can access the full address range
  - OS may still place restrictions on what memory can be accessed
  - ***sbrk*** or ***mmap*** syscalls may be required for a process to request access to areas of its address space.
- VM is disabled at boot time, but soon used by both kernel & user code

Segmentation

- Modern systems use ***paging*** to translate virtual memory addresses
- Earlier systems used ***segmentation***, which is simpler:
  - Each process has a memory segment for its code, stack, and heap.
  - Segments can be any size, defined by a ***base*** and ***bound*** (size)
  - Memory accesses are offset by the appropriate base and checked that they don't fall outside the bounds (otherwise you get a segmentation fault).
- Segment registers (or table) would be altered by the OS on context switch
- Flaw is that a program's memory must reside in a large chunk of physical memory. **Paging** can distribute a program's memory.

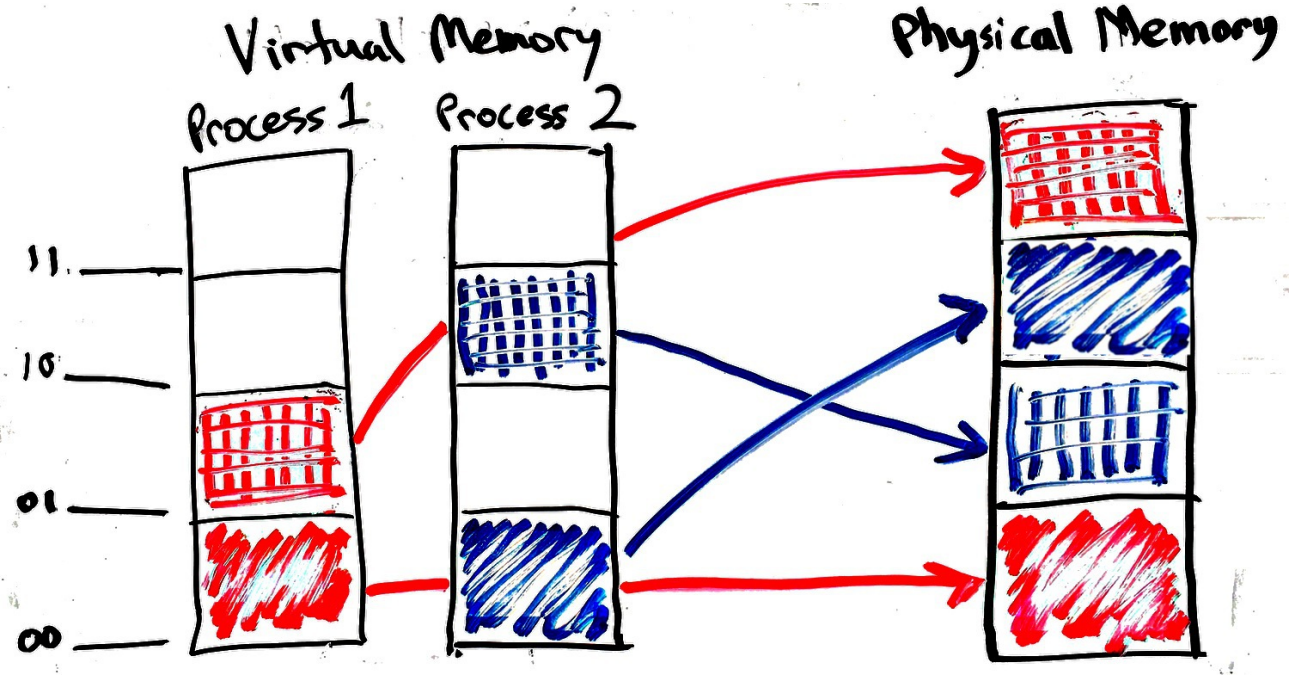


Pages are contiguous blocks of memory



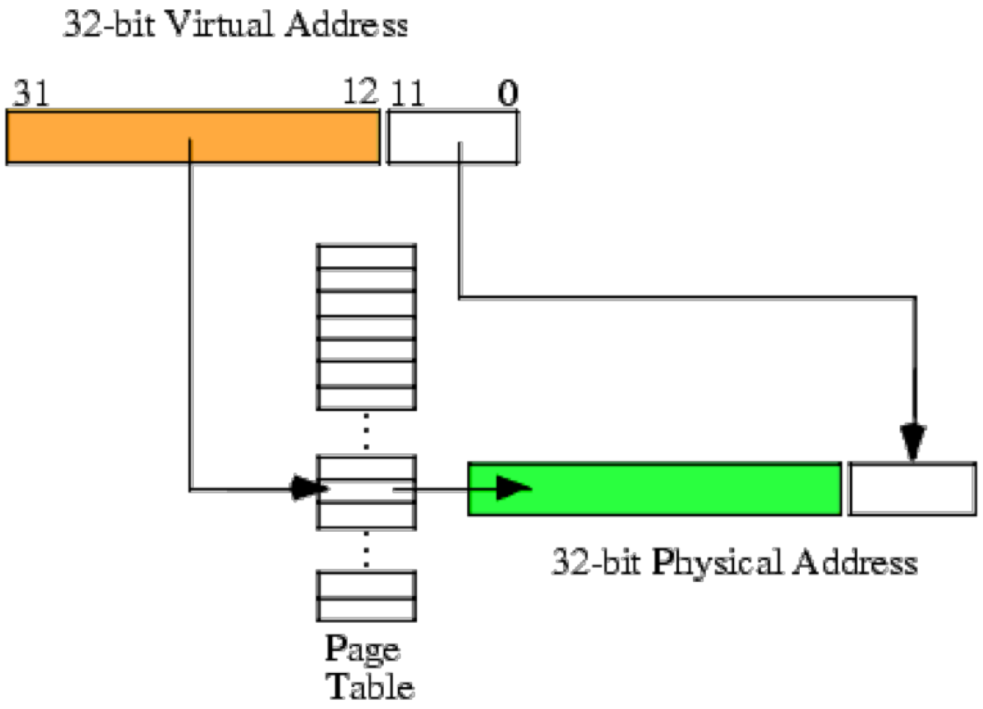
- Pages are *usually* all the same size
  - Configurable by the OS (at boot time) from 4 kb to 4 Mb
- ***High bits*** of virtual address identify the page
- ***Low bits*** identify the offset within the page
- **Larger** pages lead to:
  - More low bits identifying offset
  - Fewer high bits identifying page#

Physical memory is split among multiple processes



- OS needs a ***policy*** for splitting physical memory among competing processes
  - (We'll discuss memory management policies in a later lecture.)
- CPU needs a ***mechanism*** to implement that policy
  - At runtime the CPU translates every virtual address mentioned by user code into the appropriate physical address.

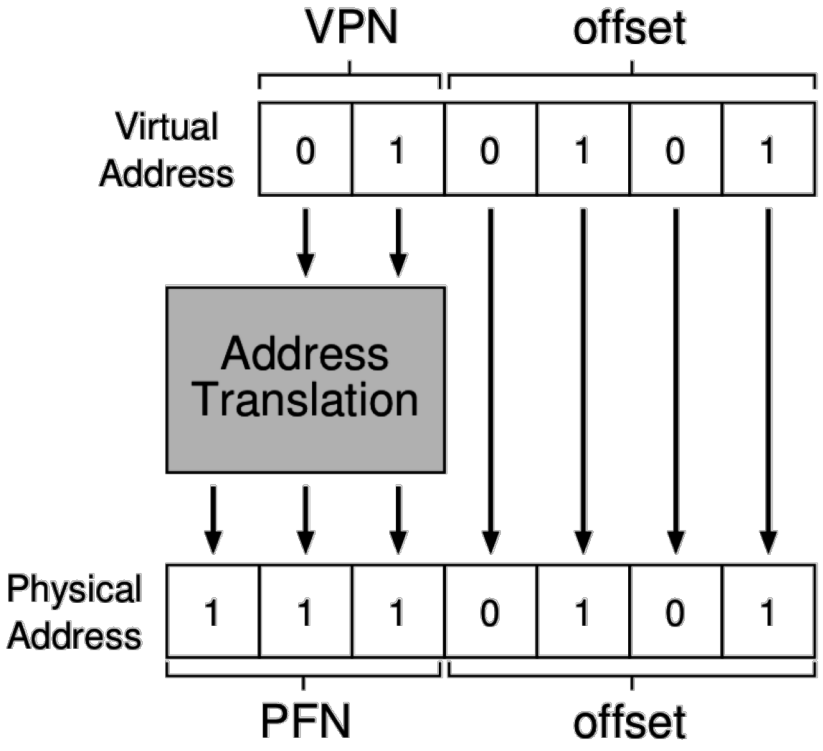
Page table translates virtual to physical addresses

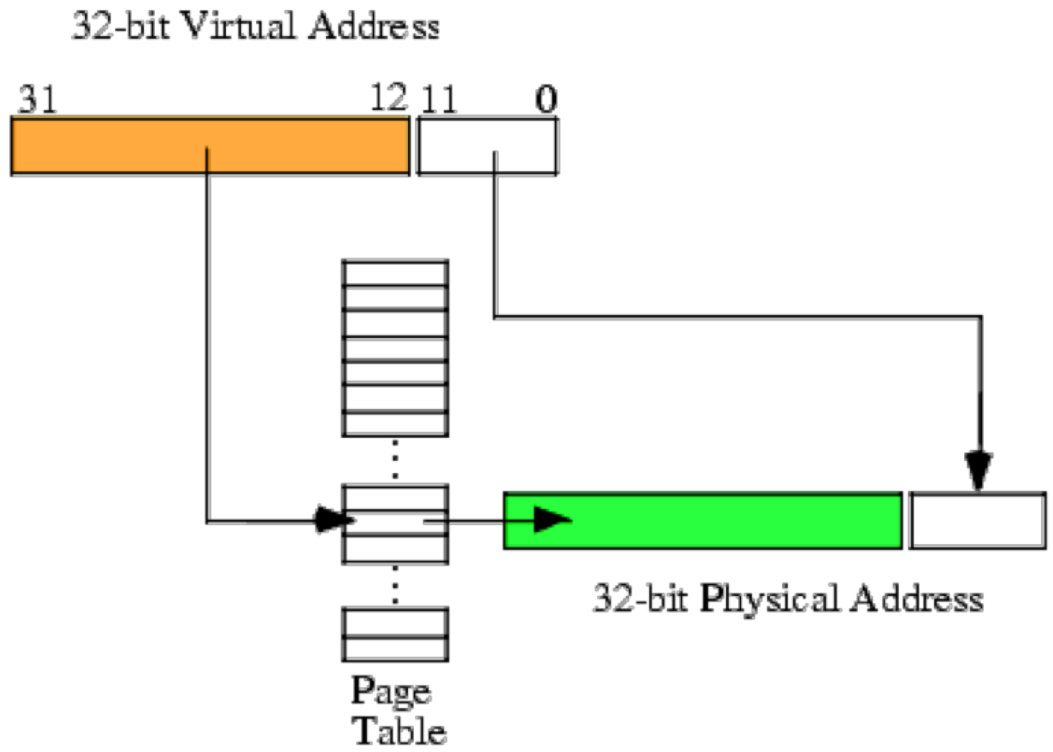


- Just need to translate the page numbers
  - The high bits of the virtual address
- The simplest type of page table is an array of physical addresses, one for each virtual page. (Linear page table)
  - Special values can be used to indicate that page has never been used or is on disk.
- To switch process contexts, just switch to a different page table
  - Different page table will cause the program to access a totally different set of physical memory locations.

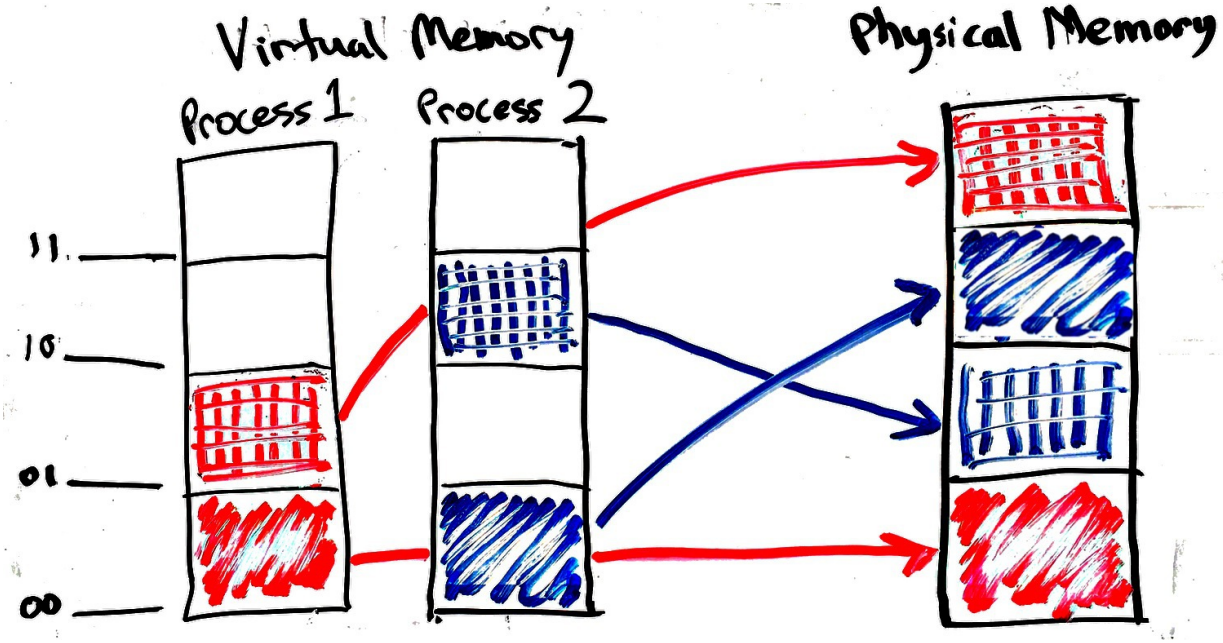
Address translation

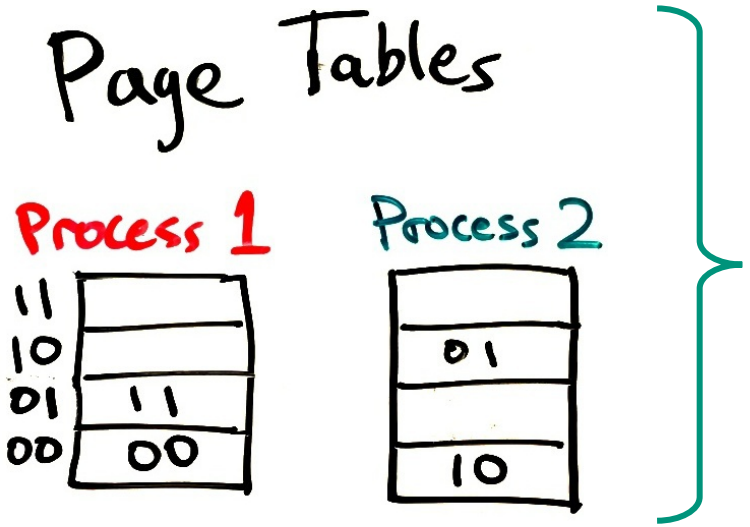
- Recall that the high bits of a virtual address refer to the page number, and low bits refer to the offset within that page
- System must translate *virtual page numbers* to *physical frame numbers*





Page table example





Stored in kernel memory

Example address translations

Process 1:

00011101 → 00011101

01001000 → 11001000

Process 2:

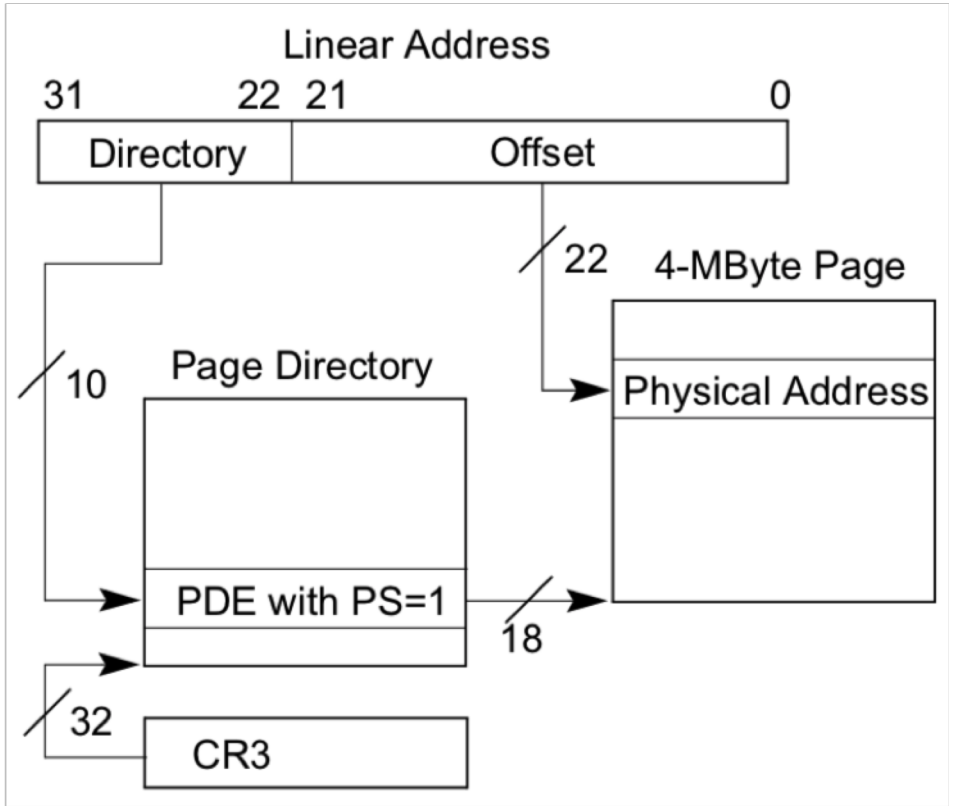
00011101 → 10011101

10000001 → 01000001

01001000 → page fault!

Changing page tables

- Intel CPUs' `%CR3` register stores the page table's address.
  - Cannot be changed in user mode.
- OS kernel changes the `%CR3` value when switching processes.
- CPU will use the page tables to translate address of every single memory access
  - Except the OS's boot code uses physical addresses directly to set things up.



Inactive process state in xv6's proc.h

- `pde_t* pgdir` points to this process' page table

```
62  // Per-process state
63  struct proc {
64    uint sz;                     // Size of process memory (bytes)
65    pde_t* pgdir;                // Page table
66    char *kstack;                // Bottom of kernel stack for this process
67    enum procstate state;        // Process state
68    volatile int pid;            // Process ID
69    struct proc *parent;         // Parent process
70    struct trapframe *tf;        // Trap frame for current syscall
71    struct context *context;     // swtch() here to run process
72    void *chan;                  // If non-zero, sleeping on chan
73    int killed;                  // If non-zero, have been killed
74    struct file *ofile[NOFILE];  // Open files
75    struct inode *cwd;           // Current directory
76    char name[16];               // Process name (debugging)
77  };
```

Context switch in xv6

In vm.c and proc.c

```
171  // Switch TSS and h/w page table to correspond to process p.
172  void
173  switchuvm(struct proc *p)
174  {
175    pushcli();
176    cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
177    cpu->gdt[SEG_TSS].s = 0;
178    cpu->ts.ss0 = SEG_KDATA << 3;
179    cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
180    ltr(SEG_TSS << 3);
181    if(p->pgdir == 0)
182      panic("switchuvm: no pgdir");
183    lcr3(PADDR(p->pgdir));  // switch to new address space
184    popcli();
185  }
```

```
248  // Per-CPU process scheduler.
249  // Each CPU calls scheduler() after setting itself up.
250  // Scheduler never returns.  It loops, doing:
251  //  - choose a process to run
252  //  - swtch to start running that process
253  //  - eventually that process transfers control
254  //      via swtch back to the scheduler.
255  void
256  scheduler(void)
257  {
258    struct proc *p;
259
260    for(;;){
261      // Enable interrupts on this processor.
262      sti();
263
264      // Loop over process table looking for process to run.
265      acquire(&ptable.lock);
266      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
267        if(p->state != RUNNABLE)
268          continue;
269
270        // Switch to chosen process.  It is the process's job
271        // to release ptable.lock and then reacquire it
272        // before jumping back to us.
273        proc = p;
274        switchuvm(p);
275        p->state = RUNNING;
276        swtch(&cpu->scheduler, proc->context);
277        switchkvm();
278
279        // Process is done running for now.
280        // It should have changed its p->state before coming back.
281        proc = 0;
282      }
283      release(&ptable.lock);
284
285    }
286  }
```

Actually, Intel x86 supports more complex VM schemes

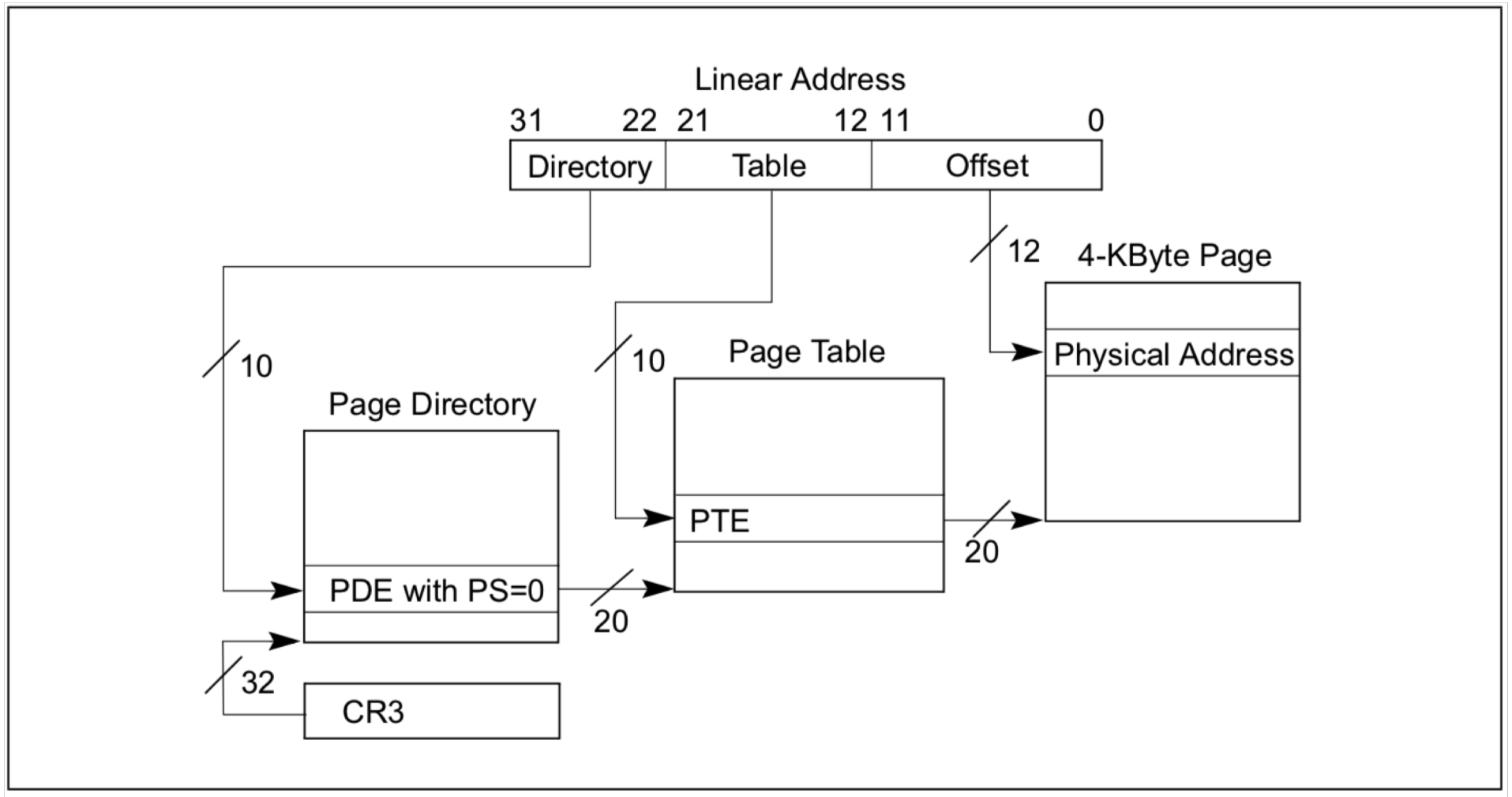


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

…and even more complex schemes

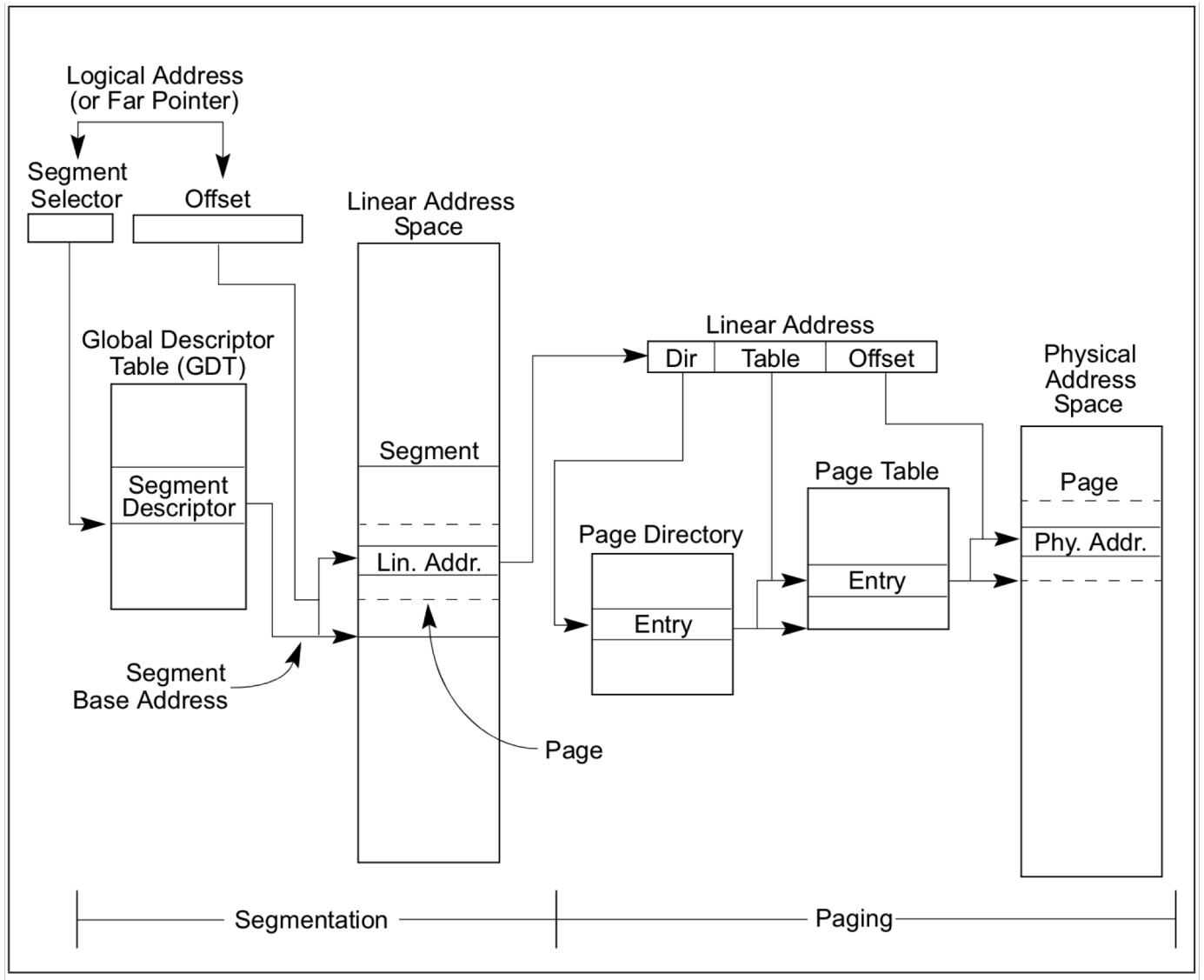


Figure 3-1. Segmentation and Paging

CPU behaviors configured by the OS

Most importantly:

- Interrupt tables - %IDTR
- Page tables - %CR3

- Store tables in kernel memory
- Store pointers to tables in control registers

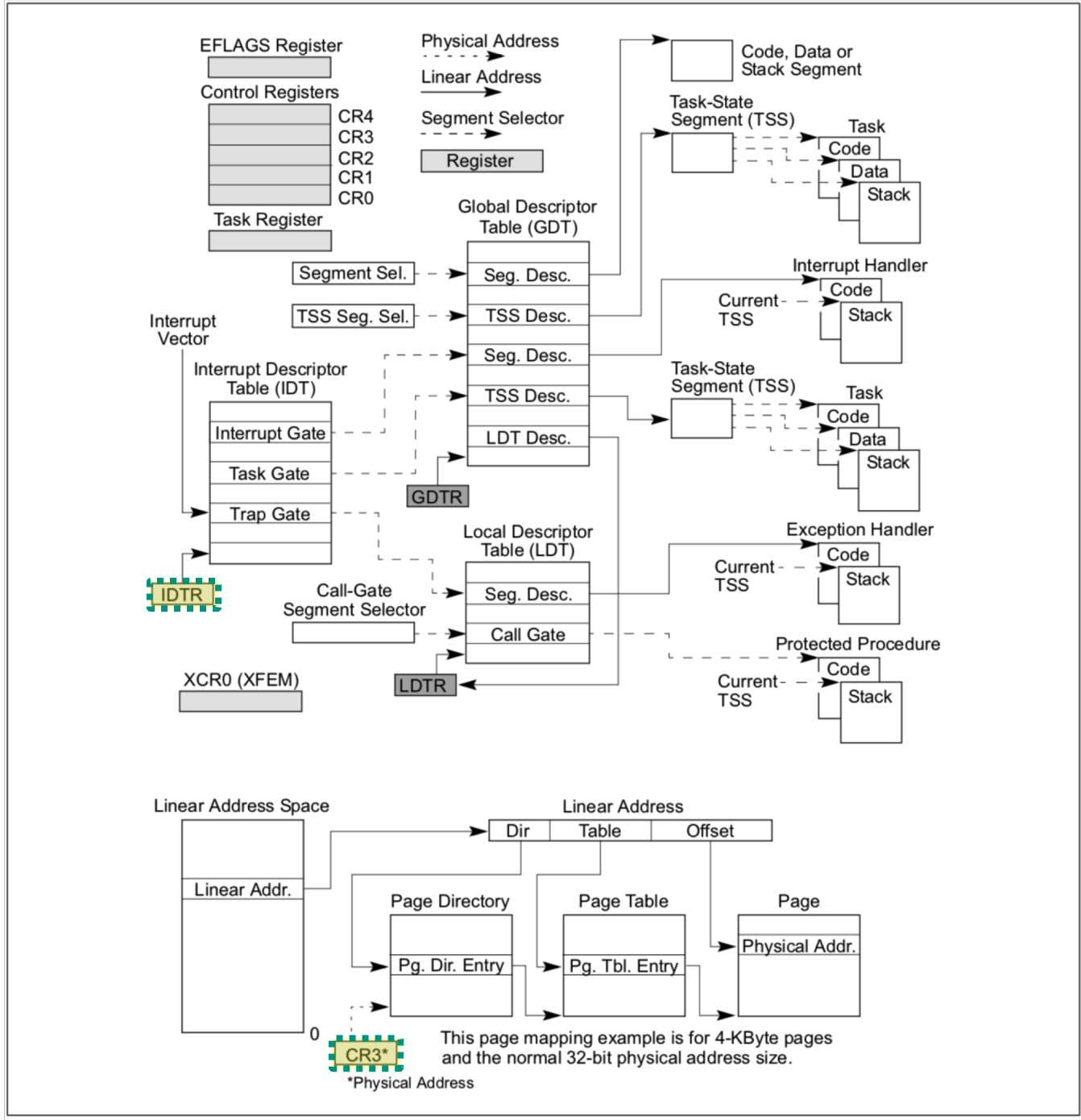


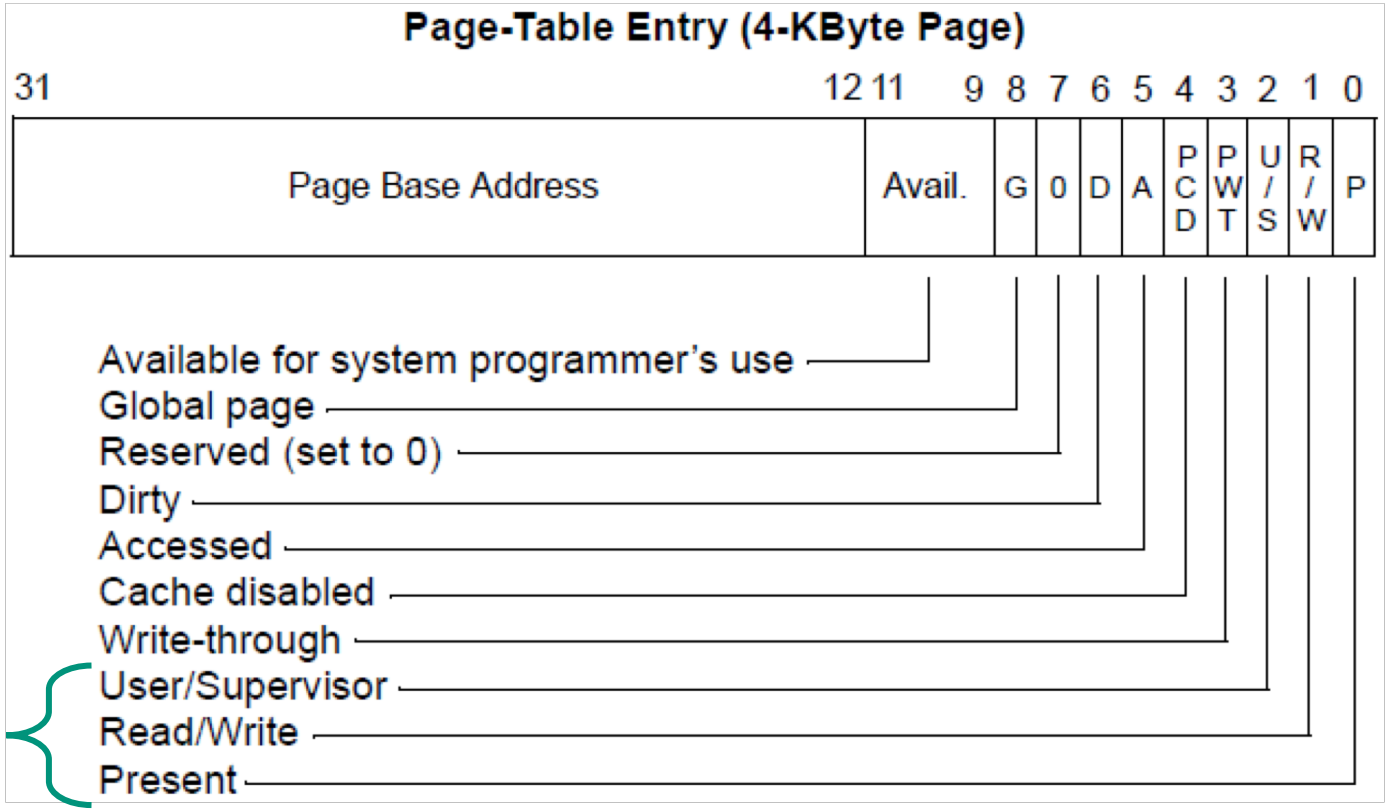
Figure 2-1. IA-32 System-Level Registers and Data Structures

The many benefits of Virtual Memory & Paging

- Program code need not worry about:
  - where in memory it will run
  - the size of physical memory
  - Compiler and Loader's job is now much easier
- Process memory is isolated & secure
  (assuming pages table entries point to unique locations)
  Changing page tables is a privileged instruction
- Can efficiently share memory between processes when desired
  (just make page table entries for two processes point to same physical page)
- Can run programs needing more memory than we physically have
  (by swapping to disk)
- Can build a machine with more memory than is addressable by programs
  (eg., "physical address extension" on 32-bit Intel processors)

Page table entries (PTEs) in detail (32-bit Intel x86)

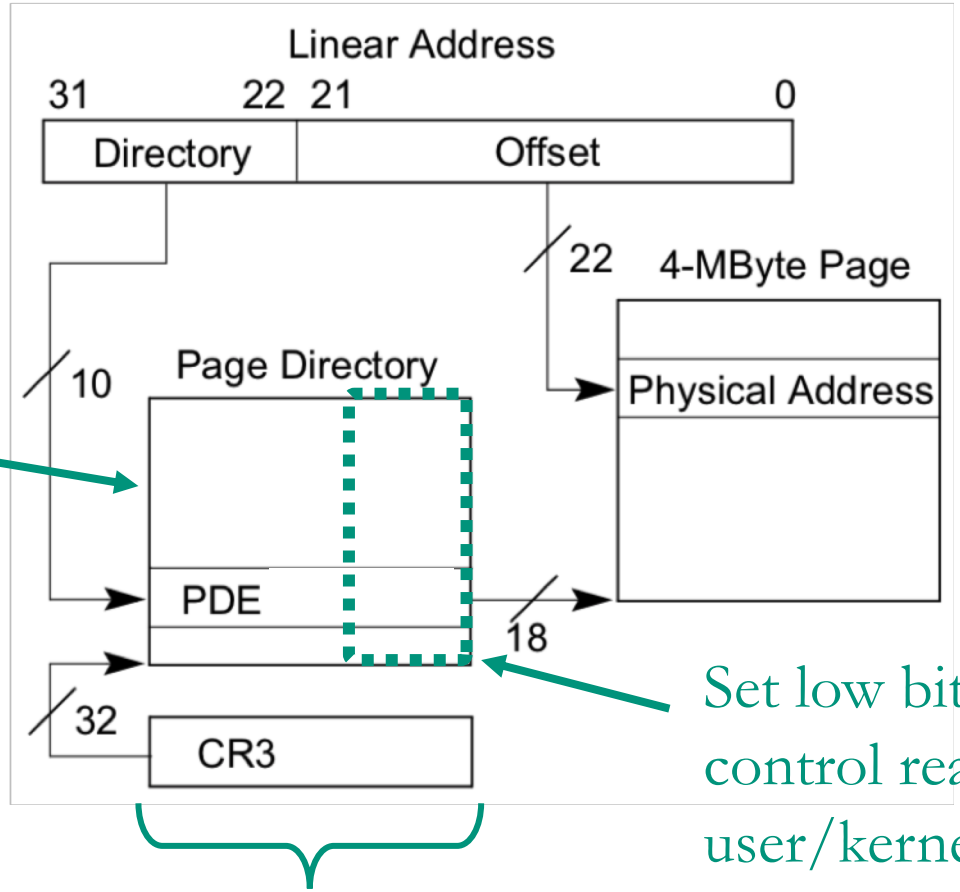
- PTEs just need the high bits for the physical frame number
- Low bits can be set by kernel to control access, etc.
- Hardware will automatically check these bits when "walking" the page table for a memory access.



Last 3 bits control user program's permission to use this page

How OS controls virtual memory (x86)

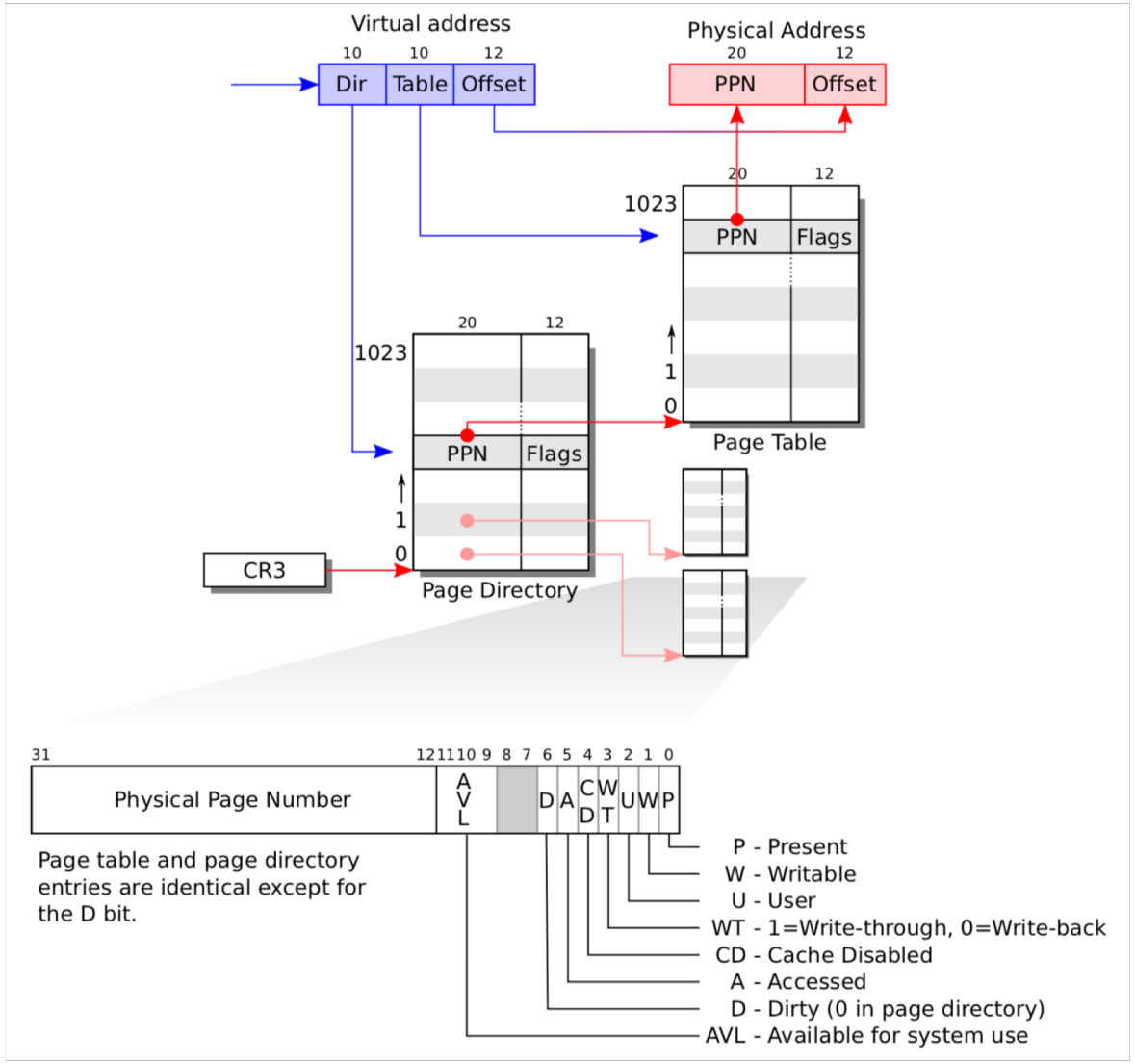
- Allocate a page table for each process.
- Set high bits of PTEs to control which physical pages are used



Set low bits of PTEs to control read/write and user/kernel access

Change the page table register (%CR3) on context switch

Complete view of paging (from xv6 book)



Recap

- Introduced about ***Virtual Memory***
- Showed the details of ***page tables*** and page table entries (PTEs)
  - High bits translate from virtual page number to physical page number.
  - Low bits in the PTE are used to indicate present/rw/kernel page.
- During a context switch, kernel changes the **%CR3** register to switch from the page table (VM mapping) of one process to another.
- Next time:
  - Page faults
  - Page table performance optimizations