EECS-343 Operating Systems
Lecture 6:
Memory management optimizations

Steve Tarzia

Spring 2019

Northwestern

Announcements

- HW 1 due Monday
- Project 2 due the following Monday
  - It's much more difficult than Project 1!
- Midterm exam in two weeks (Thurs. May 2^{nd}).

- Physical memory management for Project 2.2:
  - Free physical memory pages in xv6 is managed with a circular linked list.
  - Each free physical page stores a pointer to another free physical page.
  - `kalloc()` removes one physical page from the free list.
  - `kfree()` returns a physical page to the free list.
  - We'll talk more about free lists next week.

Last Lecture: ***Virtual Memory***

- Memory is divided into equal-sized ***pages***.
- ***Page tables*** translate virtual page numbers to physical page numbers.
- Showed the details of 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.
- VM is handled by both the OS and CPU:
  - **OS** sets up the page tables and handles exceptions (page faults).
  - **CPU** automatically translates every memory access in the program from virtual addresses to physical addresses by checking (*walking*) the page table.

Paging costs

1. ***Latency***
   - Every memory access now requires an ***additional read*** to get the physical page number from the page table
   - RAM access is slow (~50ns), so this is very bad!
2. ***Space***
   - Each process must have a page table mapping the entire address range
   - On a 32-bit system linear page tables would consume 4MB of mem *per process*
     - Assuming 4kb pages, and 32-bit addresses (4GB of virtual memory) we require one million PTEs. Each PTE is 4 bytes.
     - On a 64-bit system this would be much, much worse

Translation lookaside buffer (TLB)

- TLB is the solution to our paging latency problems
- TLB ***caches*** recently used PTEs
  - In other words, it's a small fraction of the current page table that is stored on-chip, in fast memory.
  - Usually "fully associative"
- Caches are common in computer systems
  - A cache is a record of recent transactions that allows you to skip repeated requests
  - Eg., a web browser caches all your HTTP GET requests so that you don't have to reload repeated images, like logos, menus, etc.

Why does a TLB help?

- Because programs don't access random addresses
- We're likely to need the same translations in the future
- ***Temporal locality*** – programs reuse the *exact same* memory addresses
- ***Spatial locality*** – programs typically will access memory *near* recently-used memory. For example:
  - Looping through an array (each access is adjacent to the last one)
  - A function's local variables and parameters are on the same stack frame.
  - Code has to be read from memory, and these are contiguous until a branch/jump happens

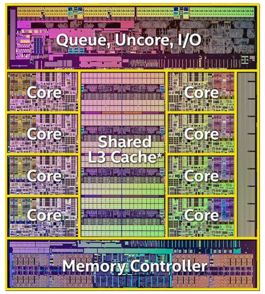
Cache dynamics

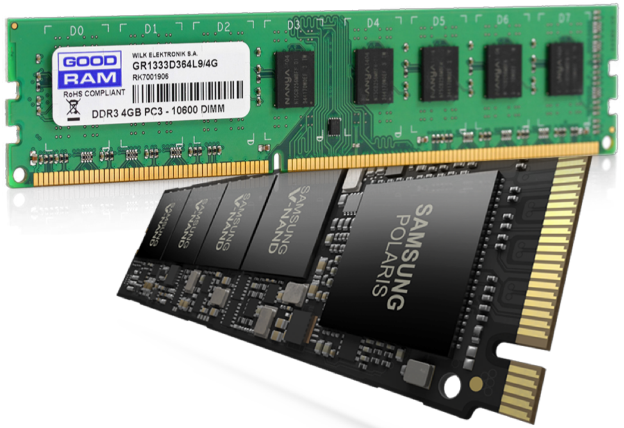
- A cache ***hit*** is when data is found in the cache
  - This is the fast case, and hopefully the most common
- A cache ***miss*** is when data is *not found* in the cache
  - Our attempt to take a shortcut failed
  - Must carry out the request normally (access page table in RAM)
  - When done, store the data in the cache for next time
  - To make space in the cache, we must chose an existing entry to *evict* (remove)
- CPU caches (like the TLB) make performance unpredictable because
  - It's usually invisible to the OS (*except for software-managed TLBs)
  - Cache status depends on prior activity, perhaps by other processes.

Computers have a hierarchy of storage

Larger, but slower

delay		*capacity*
0.3ns	CPU Registers	1 kB (kilobyte)
5ns	CPU Caches (L2)	16 MB
50ns	Random Access Memory (RAM)	16 GB
100µs	Flash Storage (SSD)	1 TB
5ms	Magnetic Disk	8 TB





- Disk is about *ten billion* times larger than registers, but has about *ten million* times larger delay (latency).
- Goal is to work as much as possible in the top levels.
- Large, rarely-needed data is stored at the bottom level
- "Memory" is not just RAM, but everything below the registers

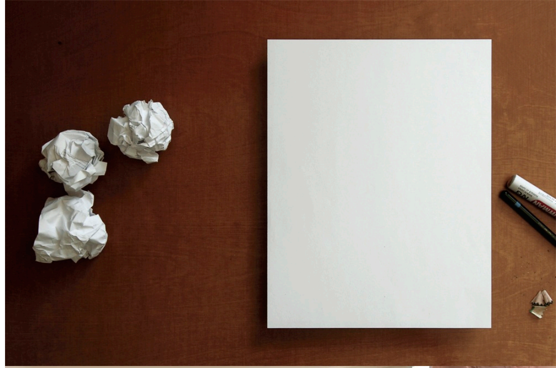
Paper-in-office analogy

- Imagine doing a really complex pen-and-paper data analysis.
- You would move papers between these storage levels, as needed:

Larger, but further ↓

delay		*capacity*
< 1s	Papers on desk	4 sheets
5s	Stack of paper on desk	20 sheets
10s	File folders in one desk drawer	1,000 sheets
5 min.	Filing cabinets in storage room	100k sheets
1 day	Off-site archival warehouse	10M sheets

- You would move papers in chunks (folders or boxes).
- Organize papers to keep related sheets together, to reduce the number of data-fetching trips.




Software-controlled paging

- Intel x86 CPUs (and xv6) use **hardware-managed** TLB
  - CPU automatically walks the page table & controls the TLB
- Some RISC CPUs use a ***software-managed TLB***:
  - These CPUs know nothing about the page tables. Just uses the TLB.
  - If a translation is not present in the TLB, CPU causes an exception
  - OS interrupt handler consults its page tables to find the address translation.
  - OS evicts an entry from the TLB and adds the new translation to the TLB, using special instructions.
  - Interrupt return instruction resumes by ***repeating*** the instruction that failed.
  - Flush the TLB before a context switch.
- This can simplify the CPU hardware and gives more control to the OS.

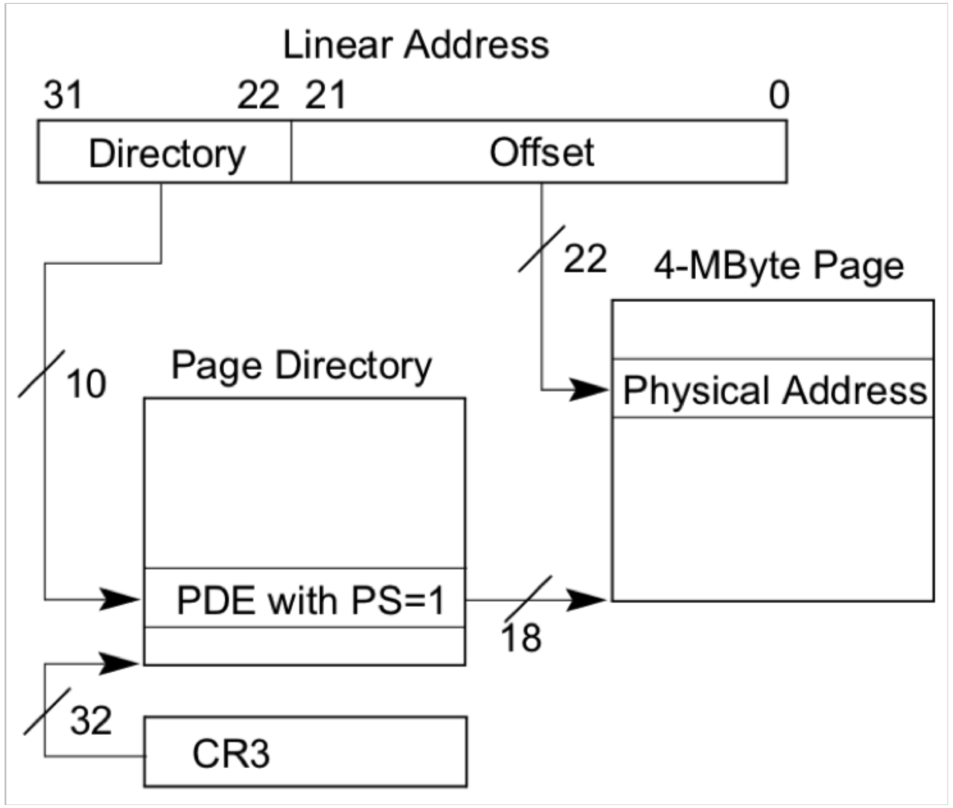
Reducing space overhead of paging

- Recall that we need 10^6 PTEs for 32-bit address space & 4kb pages
- We can reduce the page table size by making pages larger:
  - 4MB "superpages" on x86 lead to just 1000 PTEs (4kb overhead) per process
  - Also leads to more TLB hits, because each page translation serves more data
  - However, superpages are ***not*** a full solution
  - Allocating huge pages for everything will lead to wasted space
- We would like to keep fine-grained page allocation, but lose some of the overhead.

Linear (one-level) page table with **4mb (big) pages**

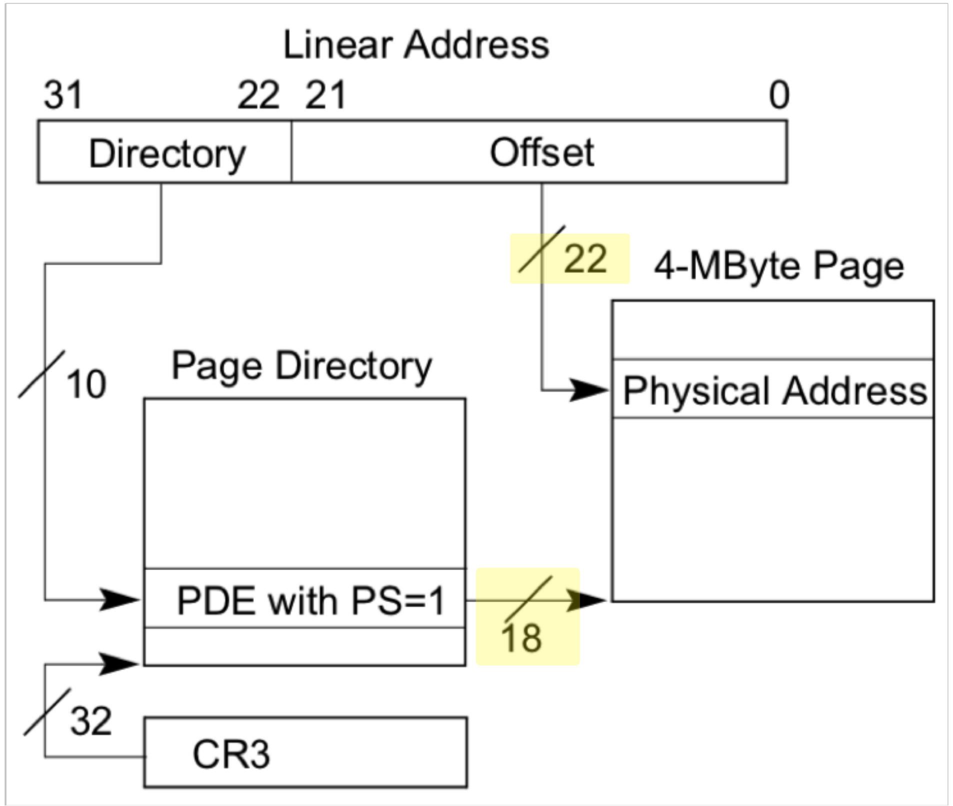
- This is a real option on Intel x86.
- Page table size = 4gb/4**m**b = 1,000 * 32bit

- We actually want small (4**k**b) pages, to waste less space when allocating.
- Theoretically, a linear page table could also be used for regular-sized (4kb) pages…
- But it would be huge even for a 32-bit system with just 4gb of RAM:
  - 4gb/4kb = 1 million entries
  - Each process would need a 4mb page table!
  - That's OK for a large process, but unacceptably wasteful for small processes.
- A **two-level** page table can start small and it **adapts** its size as needed.



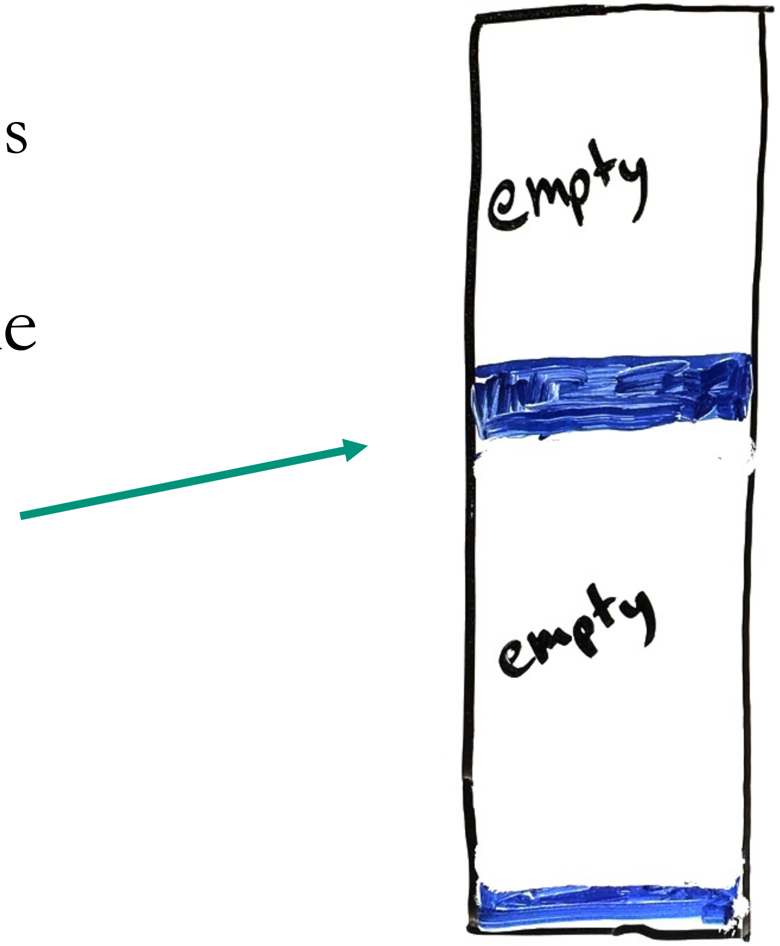
Linear page table addressing clarification

- How are 18 bits from PDE + 22 bit offset (40 bits) used to find a 32-bit address?
- Add 14 zeros to end of 18-bit PDE value to find the 32-bit starting address of the 4mb page (page must be aligned to a 16kb frame).
- 22 bit offset finds the location within that 4mb page.

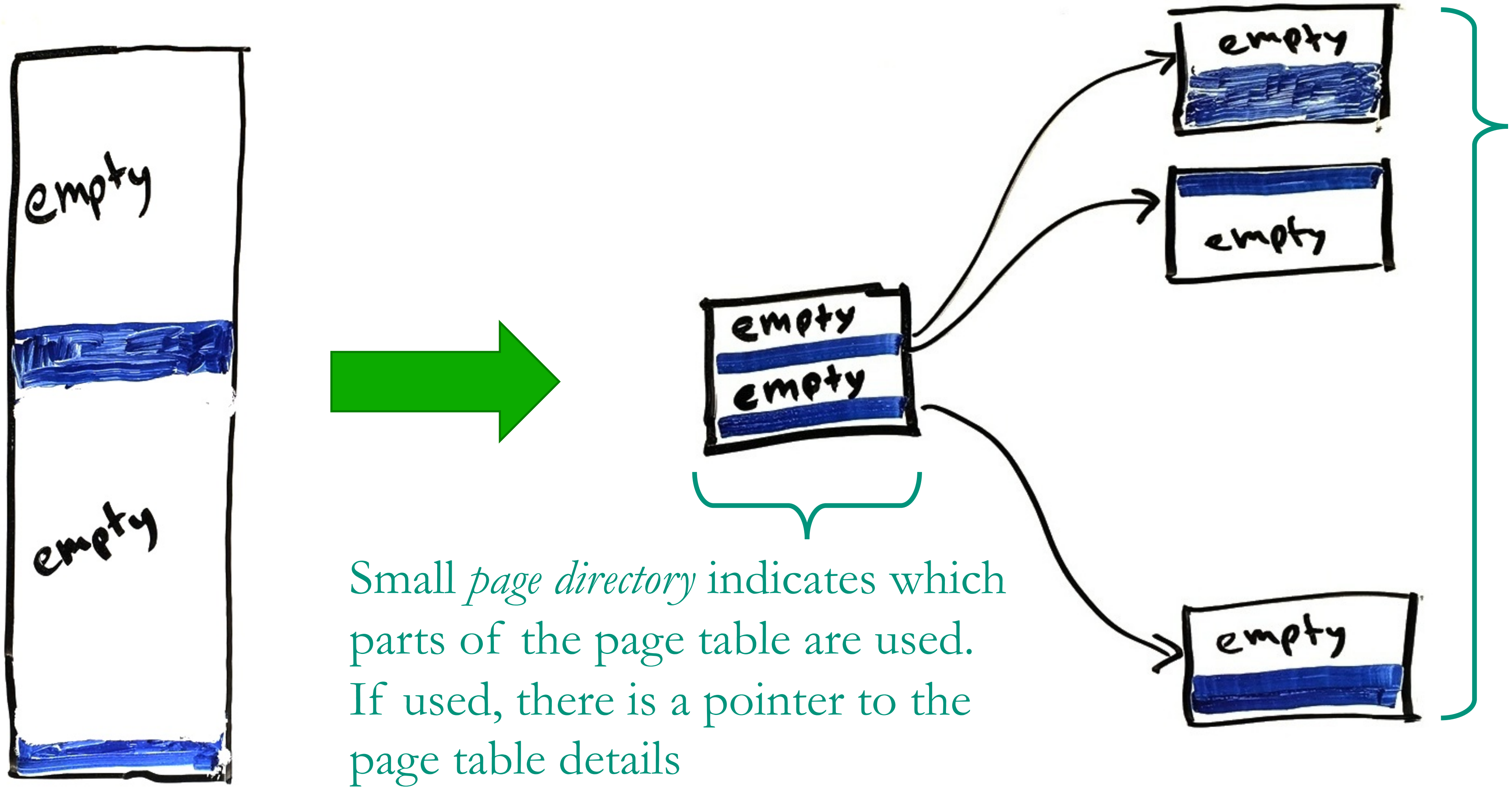


Linear page table has fixed space overhead

- The page table space overhead is actually OK for large processes.
  - 4MB page table is just 0.1% of a process using the full 4GB of memory
- However, the 4MB overhead is terrible for small processes
  - Most of the page table will be empty:
  - (PTEs will have "present" bit = 0)



Multi-level page tables eliminate wasted space



Small *page directory* indicates which parts of the page table are used. If used, there is a pointer to the page table details

Only the occupied parts of the page table are represented in full detail.

Multi-level page table mechanics

- Virtual address is broken into 3 or more parts
- Highest bits index into the highest-level page table
- A page fault can occur if an entry is missing at any level
- OS can initialize a process with just a highest-level table and just a few lower-level tables.
- More tables are added as a process demands more memory

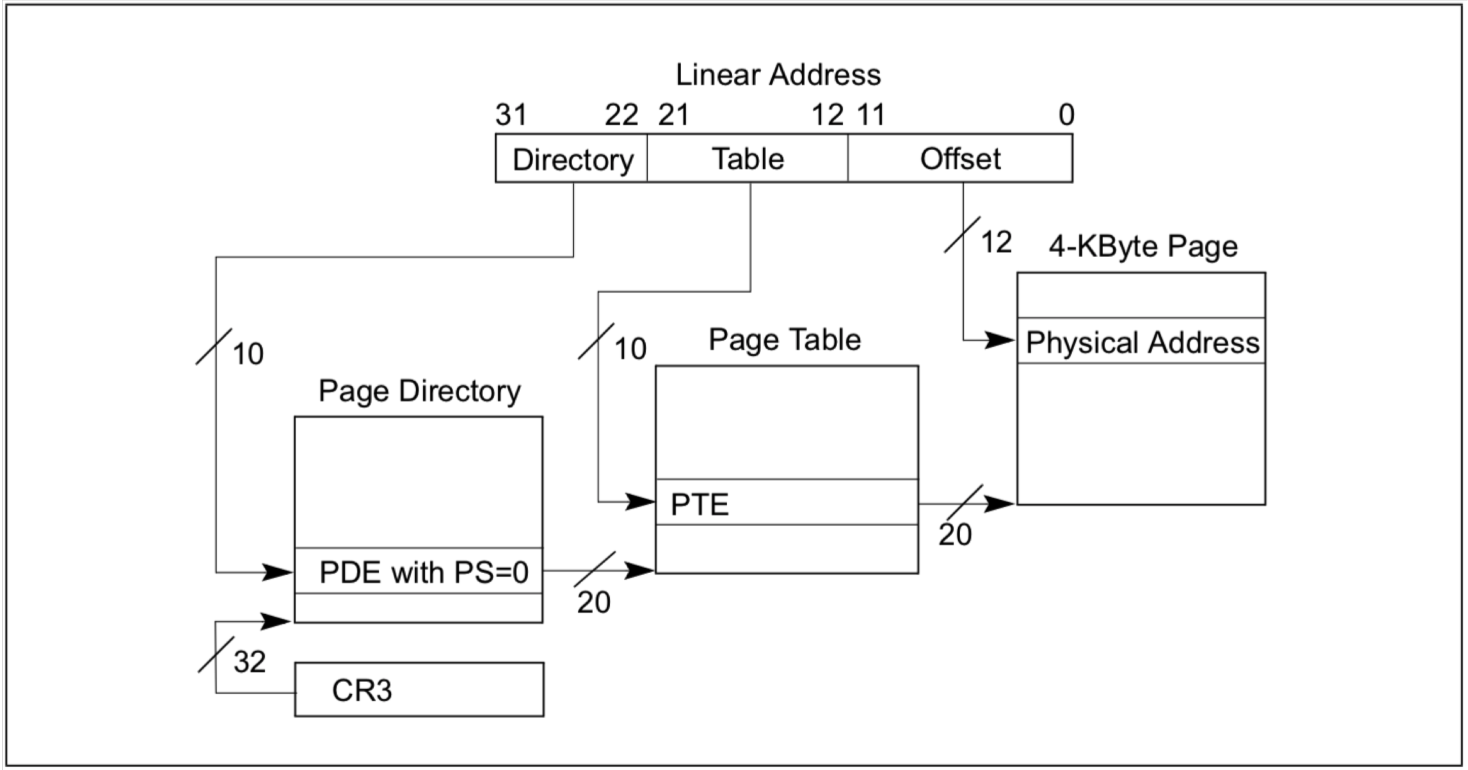


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

2-level page table addressing clarification

- Page table pages must be aligned to a 4096 byte page
  - In other words, the bottom 12 bits of the address must be zero
- In two level paging:
  - PTE address is constructed with:
    - 20 bits from PDE
    - 10 bits from middle of linear address
    - 2 remaining bits are zero because PTEs are 4 bytes long
    - = total of **32 bits**

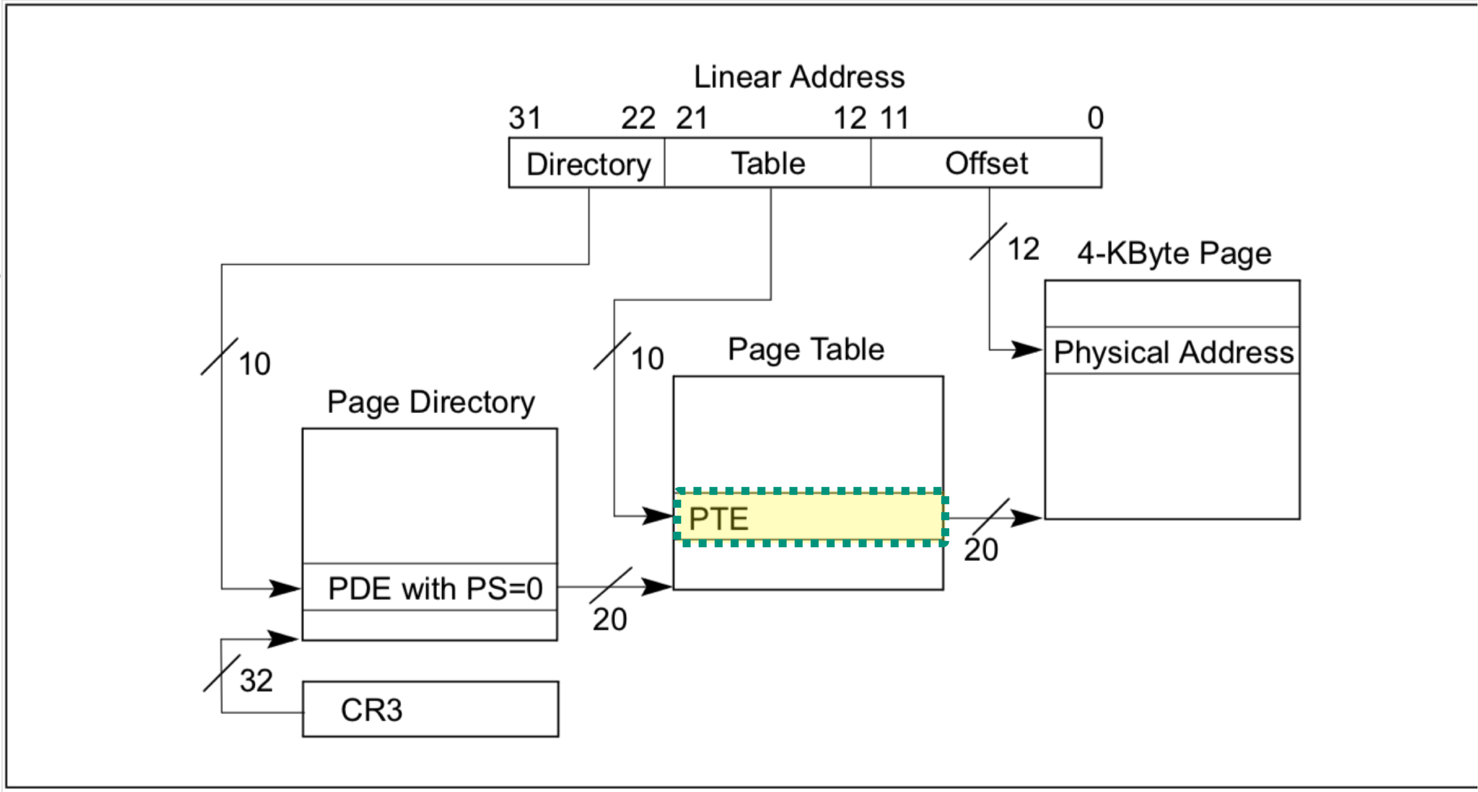


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Multi-level paging example (from 3EP book)

- Notice the valid bits.
- CPU will cause a page fault exception if it encounters a valid=0 PTE when walking the table.
    - Will also cause an exception if writing to an address whose PTE is marked not writable, etc.

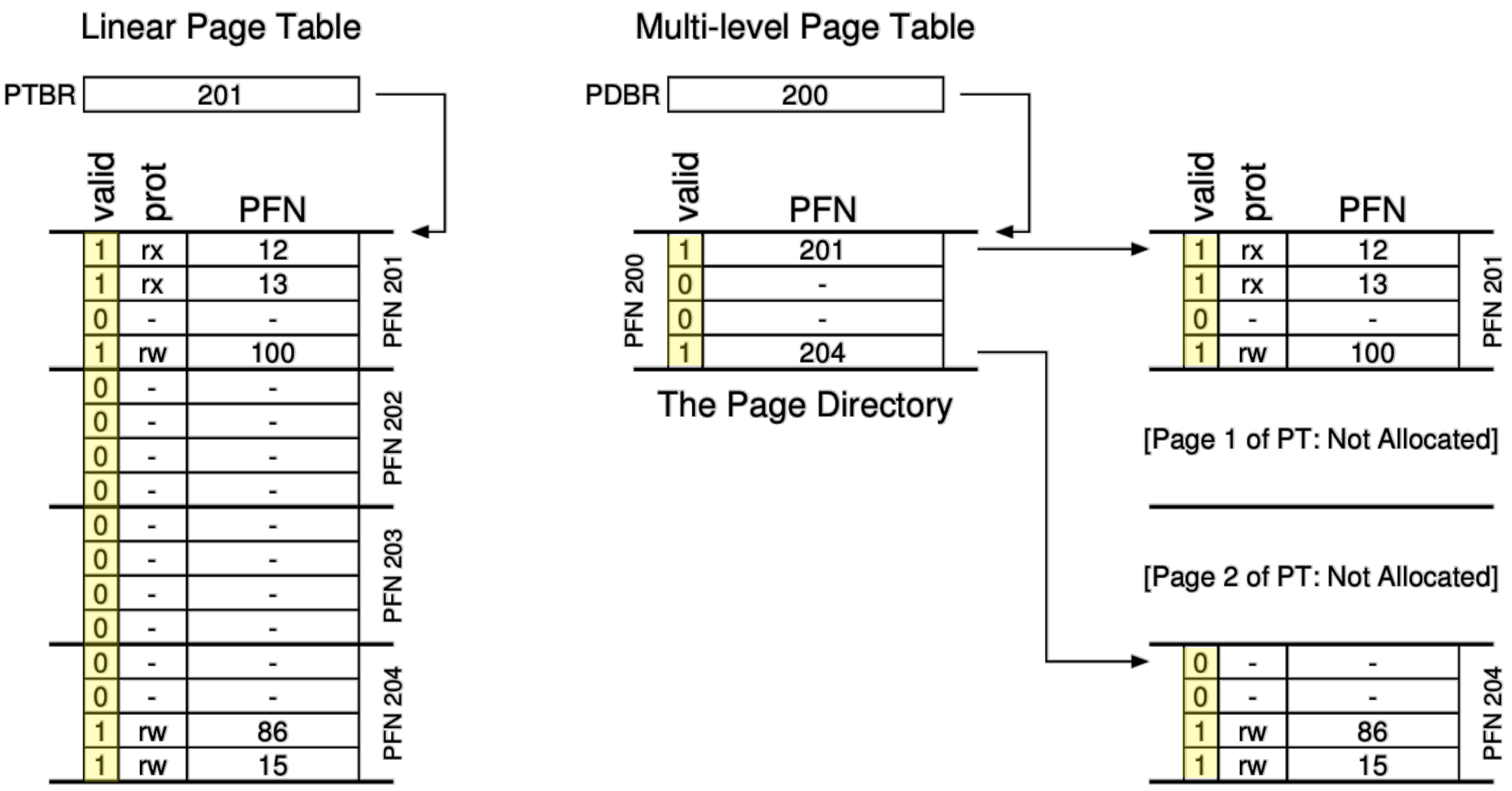


Figure 20.2: **Linear (Left) And Multi-Level (Right) Page Tables**

Improper virtual memory access causes an exception

- Project 2.2 requires a new interrupt handler in trap.c:

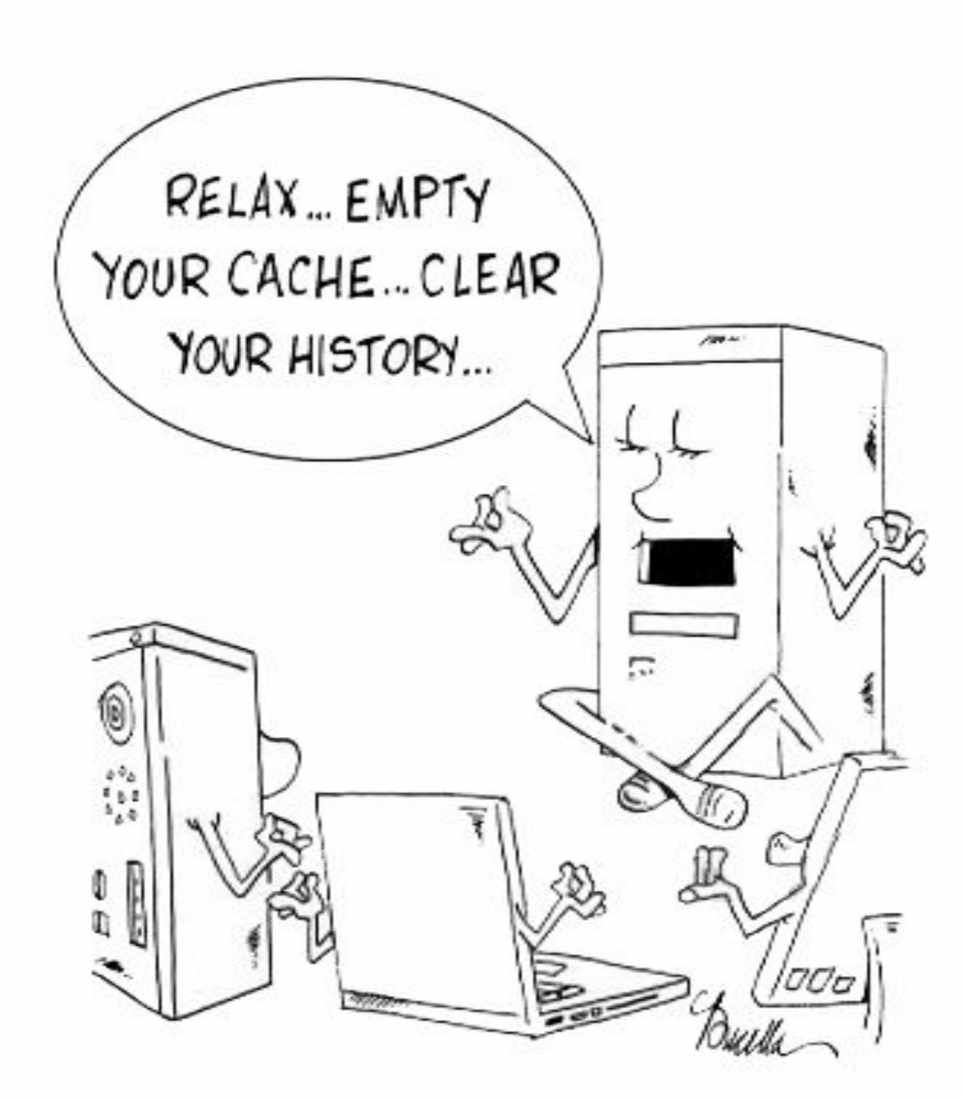
```c
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

…

```c
  default:
    if(myproc() == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpuid(), tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.
    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;
  }
```

Intermission



64-bit address space requires > 3 levels

- 64-bit address space allows 1.8×10^{19} = 18 *billion gigabytes* of memory
- So, 64-bit address spaces are very, very sparse
- Requires 3 or 4 paging levels to keep page tables small:
- x86-64 CPUs actually use 48-bit memory addresses, not 64.
  - But it still requires 3 or 4 levels

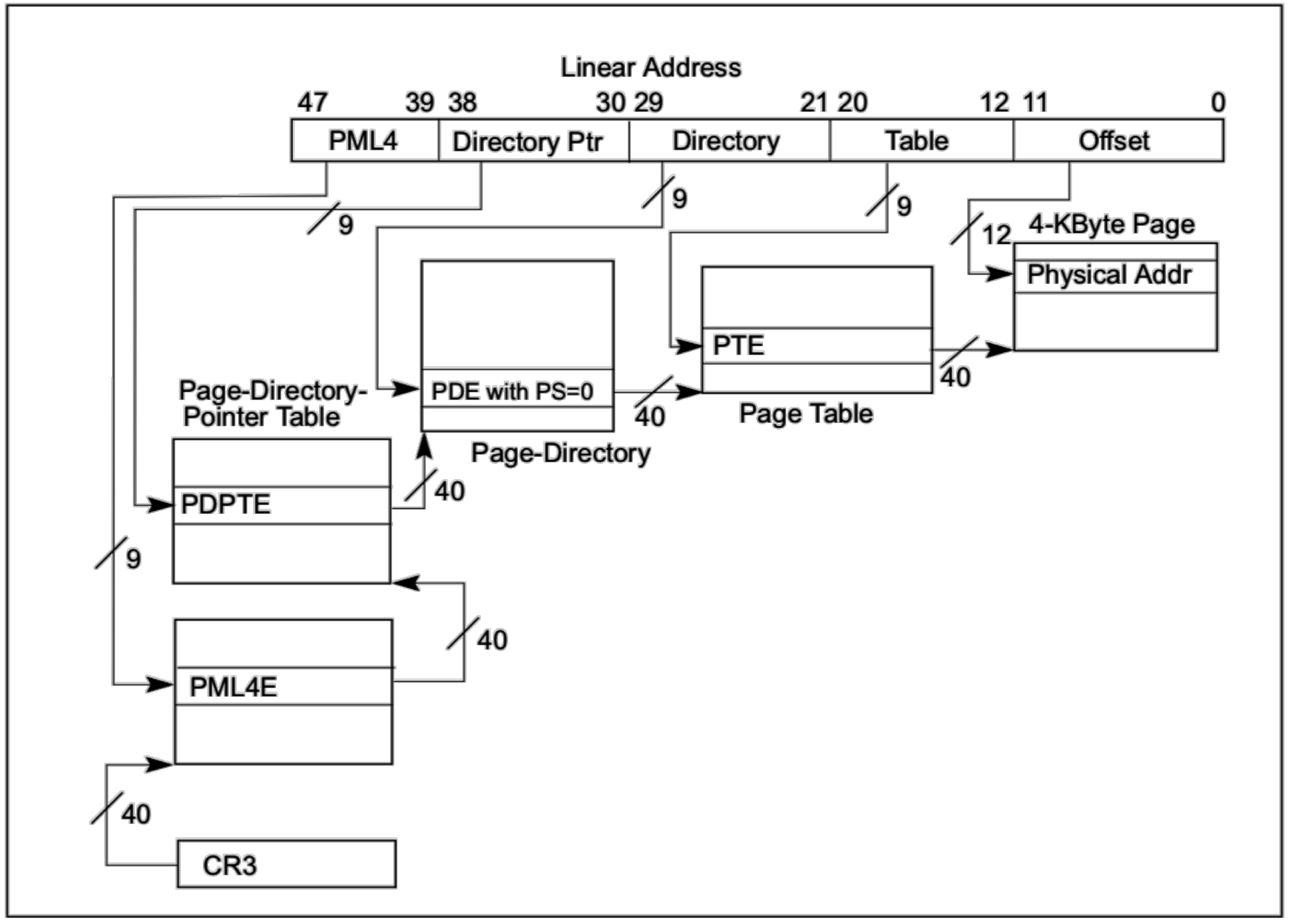


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

x86 lets you mix page sizes – *throw in a 4mb page!*

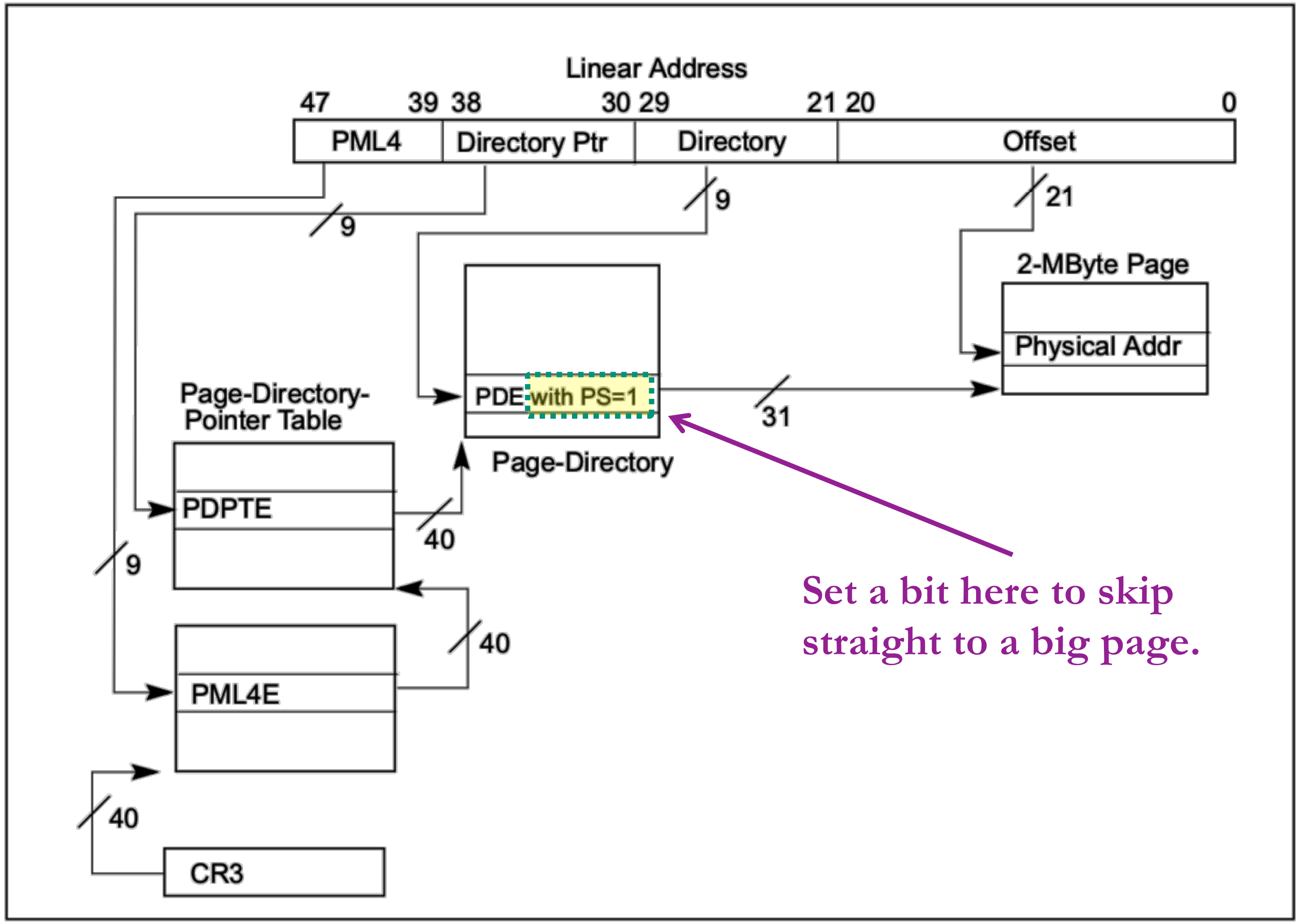


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

… or even a 1GB huge page

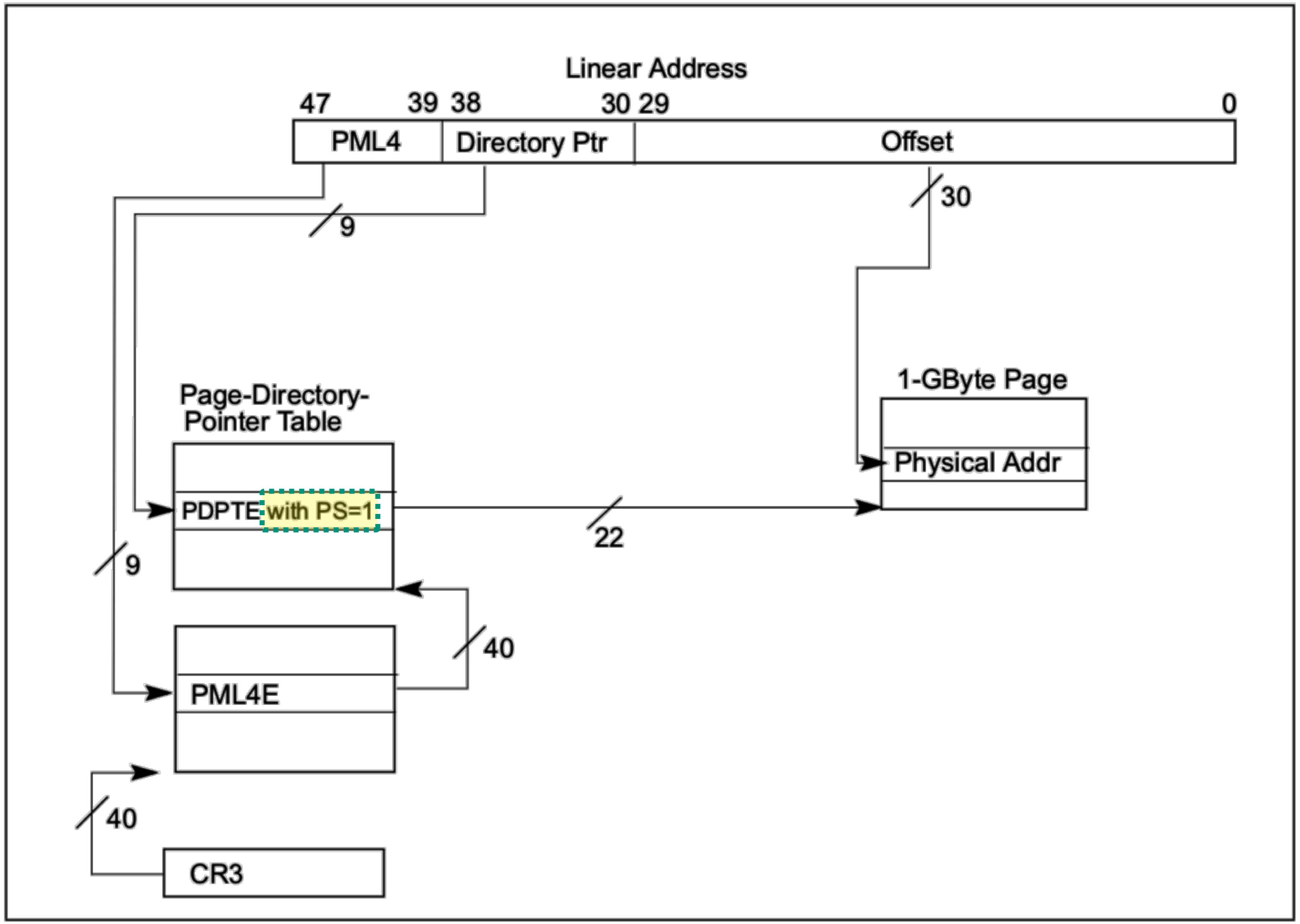


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

Why use a huge page?

- If you're using a huge chunk of data…

  (it makes the page table smaller, but that's not too important)

- **Just one TLB entry** can be used for 1GB of data.
  - Conserves precious TLB space.
- Thus, reduces TLB miss rate!

To see VM info on Linux

- `cat /proc/meminfo`
- `vmstat`
- `top`
  - (resident)

```
[spt175@murphy ~]$ cat /proc/meminfo
MemTotal:        132144848 kB
MemFree:         130263996 kB
Buffers:             63880 kB
Cached:             539824 kB
SwapCached:              0 kB
Active:             665300 kB
Inactive:           323932 kB
Active(anon):       385768 kB
Inactive(anon):       2460 kB
Active(file):       279532 kB
Inactive(file):     321472 kB
Unevictable:             0 kB
Mlocked:                 0 kB
SwapTotal:        16383996 kB
SwapFree:         16383996 kB
Dirty:                  96 kB
Writeback:               0 kB
AnonPages:          387972 kB
Mapped:              61012 kB
Shmem:                2688 kB
Slab:                88844 kB
SReclaimable:        28140 kB
SUnreclaim:          60704 kB
KernelStack:         12672 kB
PageTables:          15000 kB
NFS_Unstable:            0 kB
Bounce:                  0 kB
WritebackTmp:            0 kB
CommitLimit:      82456420 kB
Committed_AS:      1659096 kB
VmallocTotal:   34359738367 kB
VmallocUsed:        486616 kB
VmallocChunk:   34291646280 kB
HardwareCorrupted:       0 kB
AnonHugePages:      276480 kB
HugePages_Total:         0
HugePages_Free:          0
HugePages_Rsvd:          0
HugePages_Surp:          0
Hugepagesize:         2048 kB
DirectMap4k:          5604 kB
DirectMap2M:       2078720 kB
DirectMap1G:     132120576 kB
```

top

RES column is "resident memory"

"q" to quit

```
top - 10:25:45 up 7 days, 48 min,  3 users,  load average: 0.04, 0.06, 0.09
Tasks: 650 total,   1 running, 649 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni, 99.9%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  132144848k total, 129331984k used,  2812864k free, 37895660k buffers
Swap: 16383996k total,      436k used, 16383560k free, 45074412k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 9213 mysql     20   0 1263m 156m  14m S  0.0  0.1   3:57.24 mysqld
10001 root      20   0 5748m 219m  14m S  0.3  0.2  15:02.22 dsm_om_connsvcd
 9382 root      20   0  337m  18m  11m S  0.0  0.0   0:10.67 httpd
 8304 apache    20   0  352m  19m  10m S  0.0  0.0   0:00.29 httpd
 8302 apache    20   0  339m  14m 7144 S  0.0  0.0   0:00.16 httpd
 8298 apache    20   0  339m  14m 7140 S  0.0  0.0   0:00.12 httpd
 8299 apache    20   0  339m  14m 7136 S  0.0  0.0   0:00.17 httpd
 8303 apache    20   0  339m  14m 7136 S  0.0  0.0   0:00.17 httpd
 8300 apache    20   0  339m  14m 7120 S  0.0  0.0   0:00.13 httpd
 8301 apache    20   0  339m  14m 7120 S  0.0  0.0   0:00.16 httpd
 8305 apache    20   0  339m  14m 7112 S  0.0  0.0   0:00.13 httpd
 1386 apache    20   0  339m  14m 7096 S  0.0  0.0   0:00.06 httpd
 1387 apache    20   0  339m  14m 7084 S  0.0  0.0   0:00.07 httpd
 1122 spt175    20   0  251m  14m 6484 S  0.0  0.0   0:00.26 emacs
 2615 root      20   0 92996 6200 4816 S  0.0  0.0   0:00.93 NetworkManager
 9865 root      20   0 1043m  23m 4680 S  0.3  0.0   9:44.98 dsm_sa_datamgrd
 8737 postgres  20   0  219m 5380 4588 S  0.0  0.0   0:01.00 postmaster
 2786 haldaemo  20   0 45448 5528 4320 S  0.0  0.0   0:03.99 hald
 9956 root      20   0  491m 7268 3280 S  0.0  0.0   3:16.30 dsm_sa_snmpd
  990 root      20   0  103m 4188 3172 S  0.0  0.0   0:00.01 sshd
 1014 root      20   0  103m 4196 3172 S  0.0  0.0   0:00.02 sshd
19701 root      20   0  103m 4244 3172 S  0.0  0.0   0:00.01 sshd
```

Copy-on-write with Fork

- Recall that ***fork + exec*** is the only way to create a child process in unix
- Fork clones the entire process, including all virtual memory
  - This can be very slow and inefficient, especially if the memory will just be overwritten by a call to **exec**.
- ***Copy on write*** is a performance optimization:
  - Don't copy the parent's pages, ***share*** them
    - Make the child process' page table point to the parent's physical pages
    - Mark all the pages as "read only" in the PTEs (temporarily)
  - If parent or child writes to a shared page, a page fault exception will occur
  - OS handles the page fault by:
    - Copying parent's page to the child & marking both copies as writeable
    - When the faulting process is resumed, it retries the memory write.

Demand zeroing

- If a process asks for more memory with ***sbrk*** or ***mmap*** the OS can allocate it ***lazily***.
  - In other words, don't allocation the full block immediately.
  - Lazy allocation minimizes latency of fulfilling the request
  - and it prevents OS from allocating memory that will not be used.
- OS must also write zeros to newly assigned physical frames
  - Program does not necessarily expect the new memory to contain zeros,
  - But we clear the memory for security, so that other process' data is not leaked.
- OS can keep one read-only physical page filled with zeros and just give a reference to this at first.
  - After the first page fault (due to writing a read-only page), allocate a real page.

Virtual memory in practice

- On Linux, the `pmap` command shows a process' VM mapping.
- We see:
  - OS tracks which file code is loaded from, so it can be lazily loaded
  - The main process binary and libraries are ***lazy loaded***, not fully in memory
  - Libraries have read-only sections that can be shared with other processes
- `cat /proc/<pid>/smaps` shows even more detail

References:

- https://unix.stackexchange.com/a/116332
- https://www.akkadia.org/drepper/dsohowto.pdf

```
[spt175@murphy ~]$ pmap -x 1122
1122:   emacs kernel/proc.c
Address           Kbytes     RSS   Dirty Mode   Mapping
0000000000400000    2032    1344       0 r-x--  emacs-23.1
00000000007fb000    8856    8192    6140 rw---  emacs-23.1
0000000001dd5000    1204    1204    1204 rw---    [ anon ]
00000035cc600000      16      12       0 r-x--  libuuid.so.1.3.0
00000035cc604000    2044       0       0 -----  libuuid.so.1.3.0
00000035cc803000       4       4       4 rw---  libuuid.so.1.3.0
00000035cca00000      28      12       0 r-x--  libSM.so.6.0.1
00000035cca07000    2048       0       0 -----  libSM.so.6.0.1
00000035ccc07000       4       4       4 rw---  libSM.so.6.0.1
00000035d0e00000      32      12       0 r-x--  libgif.so.4.1.6
00000035d0e08000    2048       0       0 -----  libgif.so.4.1.6
00000035d1008000       4       4       4 rw---  libgif.so.4.1.6
0000003f65a00000     128     116       0 r-x--  ld-2.12.so
0000003f65c20000       4       4       4 r----  ld-2.12.so
0000003f65c21000       4       4       4 rw---  ld-2.12.so
0000003f65c22000       4       4       4 rw---    [ anon ]
0000003f65e00000    1576     536       0 r-x--  libc-2.12.so
0000003f65f8a000    2048       0       0 -----  libc-2.12.so
0000003f6618a000      16      16       8 r----  libc-2.12.so
0000003f6618e000       8       8       8 rw---  libc-2.12.so

      …                        …                   …

00007fca3aa85000      52      20       0 r-x--  libnss_files-2.12.so
00007fca3aa92000    2044       0       0 -----  libnss_files-2.12.so
00007fca3ac91000       4       4       4 r----  libnss_files-2.12.so
00007fca3ac92000       4       4       4 rw---  libnss_files-2.12.so
00007fca3ac93000   96848      44       0 r----  locale-archive
00007fca40b27000     104     104     104 rw---    [ anon ]
00007fca40b54000      80      80      80 rw---    [ anon ]
00007ffccb300000     164     128     128 rw---    [ stack ]
00007ffccb341000       4       4       0 r-x--    [ anon ]
ffffffffff600000       4       0       0 r-x--    [ anon ]
----------------  ------  ------  ------
total kB          257068   14604    8128
```

emacs

- "Mapping" shows source of the section, more code can be loaded from here later.
  - "**anon**" are regular program data, requested by *sbrk* or *mmap*. (In other words, heap data.)
- Each library has several sections:
  - "r-x--" for code
  - "r----" for constants

    } *can be shared*
  - "rw---" for global data
  - "-----" for guard pages:

    (not mapped to anything, just reserved to generate page faults)
- RSS means resident in physical mem.
- Dirty pages have been written and therefore cannot be shared with others

`top` has a column showing shared memory

```
top - 10:25:45 up 7 days, 48 min,  3 users,  load average: 0.04, 0.06, 0.09
Tasks: 650 total,   1 running, 649 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni, 99.9%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  132144848k total, 129331984k used,  2812864k free, 37895660k buffers
Swap: 16383996k total,      436k used, 16383560k free, 45074412k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 9213 mysql     20   0 1263m 156m  14m S  0.0  0.1   3:57.24 mysqld
10001 root      20   0 5748m 219m  14m S  0.3  0.2  15:02.22 dsm_om_connsvcd
 9382 root      20   0  337m  18m  11m S  0.0  0.0   0:10.67 httpd
 8304 apache    20   0  352m  19m  10m S  0.0  0.0   0:00.29 httpd
 8302 apache    20   0  339m  14m 7144 S  0.0  0.0   0:00.16 httpd
 8298 apache    20   0  339m  14m 7140 S  0.0  0.0   0:00.12 httpd
 8299 apache    20   0  339m  14m 7136 S  0.0  0.0   0:00.17 httpd
 8303 apache    20   0  339m  14m 7136 S  0.0  0.0   0:00.17 httpd
 8300 apache    20   0  339m  14m 7120 S  0.0  0.0   0:00.13 httpd
 8301 apache    20   0  339m  14m 7120 S  0.0  0.0   0:00.16 httpd
 8305 apache    20   0  339m  14m 7112 S  0.0  0.0   0:00.13 httpd
 1386 apache    20   0  339m  14m 7096 S  0.0  0.0   0:00.06 httpd
 1387 apache    20   0  339m  14m 7084 S  0.0  0.0   0:00.07 httpd
 1122 spt175    20   0  251m  14m 6484 S  0.0  0.0   0:00.26 emacs
 2615 root      20   0 92996 6200 4816 S  0.0  0.0   0:00.93 NetworkManager
 9865 root      20   0 1043m  23m 4680 S  0.3  0.0   9:44.98 dsm_sa_datamgrd
 8737 postgres  20   0  219m 5380 4588 S  0.0  0.0   0:01.00 postmaster
 2786 haldaemo  20   0 45448 5528 4320 S  0.0  0.0   0:03.99 hald
 9956 root      20   0  491m 7268 3280 S  0.0  0.0   3:16.30 dsm_sa_snmpd
  990 root      20   0  103m 4188 3172 S  0.0  0.0   0:00.01 sshd
 1014 root      20   0  103m 4196 3172 S  0.0  0.0   0:00.02 sshd
19701 root      20   0  103m 4244 3172 S  0.0  0.0   0:00.01 sshd
```

- The duplicate processes are using a lot of shared memory:
  - ~50% of resident memory for `httpd` is shared (RES/2 == SHR)
  - ~75% of resident memory for `sshd` is shared
- Even if there is just one instance of `emacs` running, it may share many libraries with other running programs.
- Total virtual memory is ~10x larger than resident memory
  - Processes only use a small fraction of their VM!
  - Due to sharing and lazy loading.

Recap: the costs of virtual memory and paging

- **Latency cost**, because each memory access must be translated.
  - **Translation lookaside buffer (TLB)** caches recent virtual to physical page number translations.
  - Software-controlled paging removes page tables from the CPU spec and lets OS handle translations in software, in response to TLB miss exceptions.
- **Space cost**, due to storing a page table for each process.
  - Linear (one-level) page tables are large.
  - Smaller pages lead to less wasted space during allocation, but more space is consumed by page tables.
  - **Multi-level page tables** are the only way to truly conserve space.
  - Mixed-size pages reduce TLB misses.
- Copy-on-write fork, demand zeroing, lazy loading, and library sharing all reduce physical memory demands.