**EECS 343**

**Tuesday April 9, 2019**

# Lecture 7: Memory Management Optimizations

## From last time: Virtual Memory

Page tables translate virtual page numbers to physical page numbers

There are page table entries inside page tables (PTEs)

- high bits translate from virutal page number to physical page number
- low bits tell you whether readable/writable, user accessable, kernal accessable and more

Register %CR3 value gets changed during a context switch so the OS uses a different mapping after the context switch

VM is hanlded by OS and CPU together:

- OS: sets up page tables and handle excpetions (like trying to access a memory address that isn't accesible or doesn't have a valid page)
- CPU: automatically translates every memory access in the program from virtual addresses to physical addresses by checking the page tables

# Paging Costs

### 1. Latency

Every memory access now required 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 has their own 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 we require one million PTEs and each PTE is 4kb

# Translation Lookaside Buffer (TLB)

Cache for recenetly used page table entries (to make access to them faster)

TLB is the solution for paging latency problems

- Uses small fraction of current page table that is stored on-chip, in fast memeory
- "Fully associative"

Caches are common in computer systems lol thanks Steve

- cache is a record of recent transactions that allows you to skip repeated requests
- web browser caches all you HTTP GET requests so that you don't have to reload repeated images, like logos, menus and other ish

# Why does a TLB help

Programs don't access random addresses, instead they're likely to need the same translations in the future

**Temporal Locality** - programs resuse the *exact* same memory addresses

**Spatial Locality** - programs typically access memory *near* recently used memory. Example:

- looping through an array (adjacent addresses)
- functions local variables and paramenters are on the same stack frame
- code has to be read from memory, and these are contiguous until a branch/jump happens

Tend to have *spatial locality* in memory access

# Cache Dynamics

A cache **hit** is when data is found in the cache. This is fast and hopefully most common

A cache **miss** is when the data is not found in the cache

- have to go to memory to find the page table translation
- low because we have to access page table in RAM
- When we're done with this, we store the data in the cache for next time, SO we have to choose an existing entry to *evict*

CPU Caches (like the TLB) make performance unpredictable because:

- it's usually invisible to the OS (excpet for software managed TLBs)
- Cache status depends on prior activity, perhaps by other processes

# Computers have a hierarchy of Storage

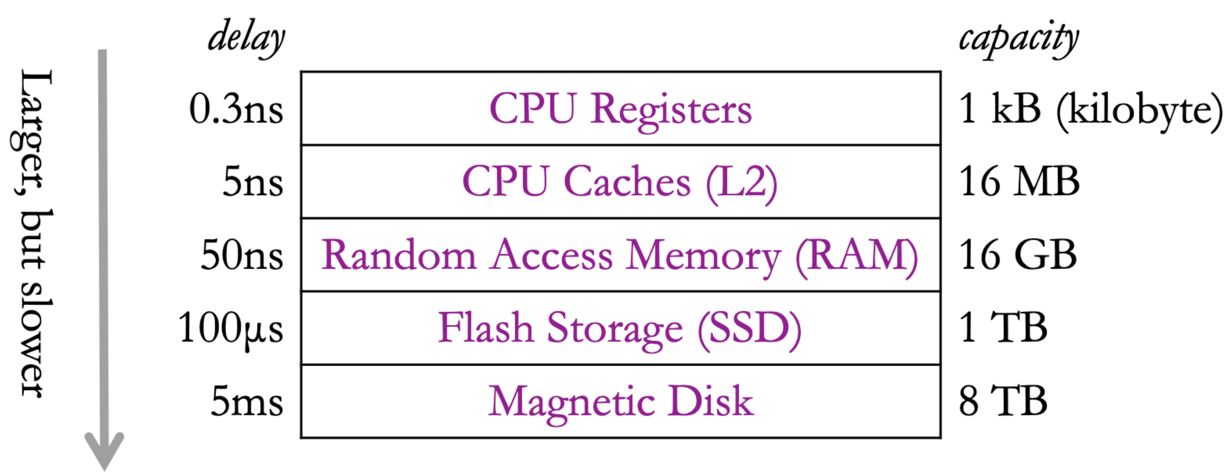
Disk is about *ten billion* times larger than registers, but has about *ten million* times larger delay (latency)

Goal is to work as much as possible in the top level

Large, rarely-needed data is stored at the bottom level

"memory" is not just RAM, but everything below the registers

The reason they get slower is because they are bigger, and because they are bigger, things get farther away so you have to go a further distance to access your data. The same tech more or less is used in all the different forms of memory, it's just that the distance is larger




# Software-controlled Paging

Intel x86 CPUs use **hardware-managed TLB**

- CPU automatically wlaks the page table and controls the TLB

RISC CPUs (a lot of mobile) use a **software-managed TLB**

- These CPUs know nothing about page tables, just use the TLB
- If a translation is not present in the TLB, CPU causes an expcetion
- OS interrput handler consults its page tables to find the address translation
- OS evicts an entry from the TLB and addes the new translation to the TLB using special instructions
- Interrupt return instruction resumes by **repeating** the intstruction that failed since the TLB has been changed
- 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 and 4kb pages

We can reduce the page table size by make pages larger

- 4MB "superpages" on the x86 lead to just 1000 PTEs (4kb overhead) per process
- Also leads to more TLB hits, because each page translations serves more data
- However, superpages are *not* a full solution
- Allocating huge pages for everything will lead to wasted space

We want to keep fine-grained page allocation, but lose some of the overhead

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

Basically they waste space as stated before

**Two-level** page table can start small and **adapt** its size as needed

# Linear Page Table Addressing Clarification

How are 18 bits from PDE + 22bit offest (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 offest finds the location with that 4mb page

# Linear Page table has fixed space overhead

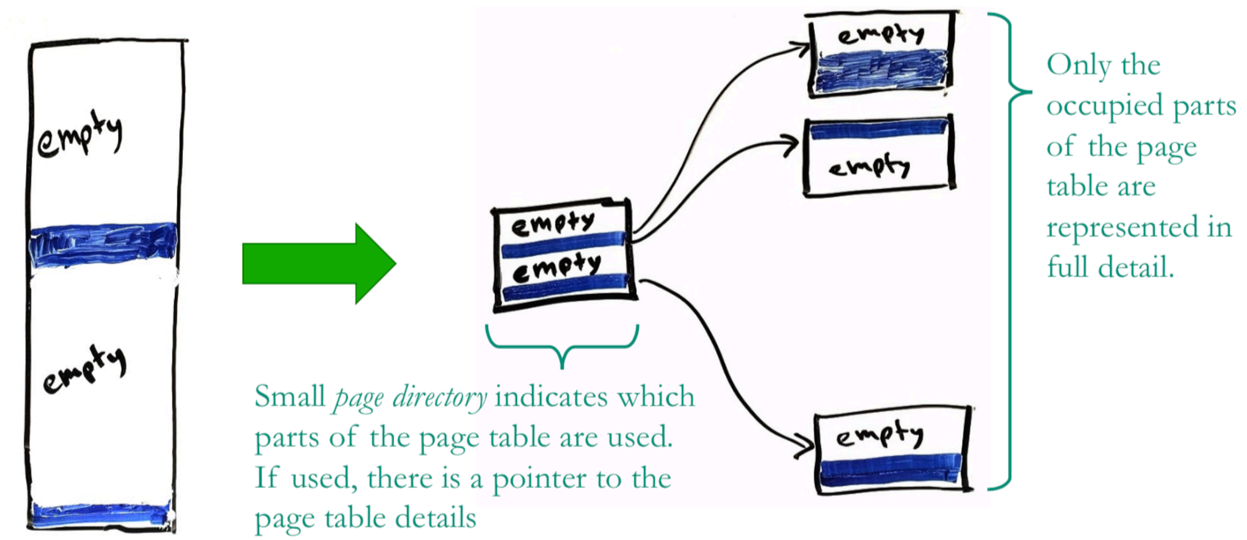
THe page table space overhead is actually OK for large process

- 4MB page table is just 0.1% of a process using the full 4GB of memory

However, the 4MB over head 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




## Multi-level page table mechanics

- Virtual address is broken into 3 or more parts
- Highest bits index into the highest-level page table
- A pagefault 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

## 2-level page table addressing clarification

## Multi-level paging example

- 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.

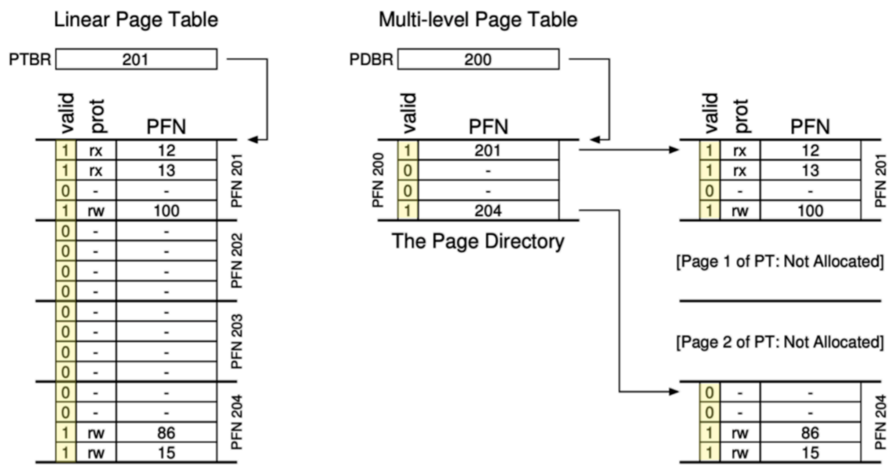



Figure 20.2: **Linear (Left) And Multi-Level (Right) Page Tables**

# Improper Virtual Memory Access causes an Exception

Project 2.2 requires a new interrupt handler in trap.c

```
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:
      //handler code
    case SOME_THING :
      // bunch of cases for different handlers
      // more cases
      // more cases
    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;
    }
  }
```

If there was no handler for the interrupt, the OS will either panic or it will kill the process. Default case

Allow both processes to write to that memory, but at first they can't. Need new interrupt handler for page faults essentially

# 64-bit address space requires > 3 levels

64bit address space allows $1.8 \times 10^{19} = 18 \text{ billion gigabytes}$ of memory

SO, 64-bit addresss spaces are very, very sparse

Requires 3 or 4 paging levels to keep page tables small

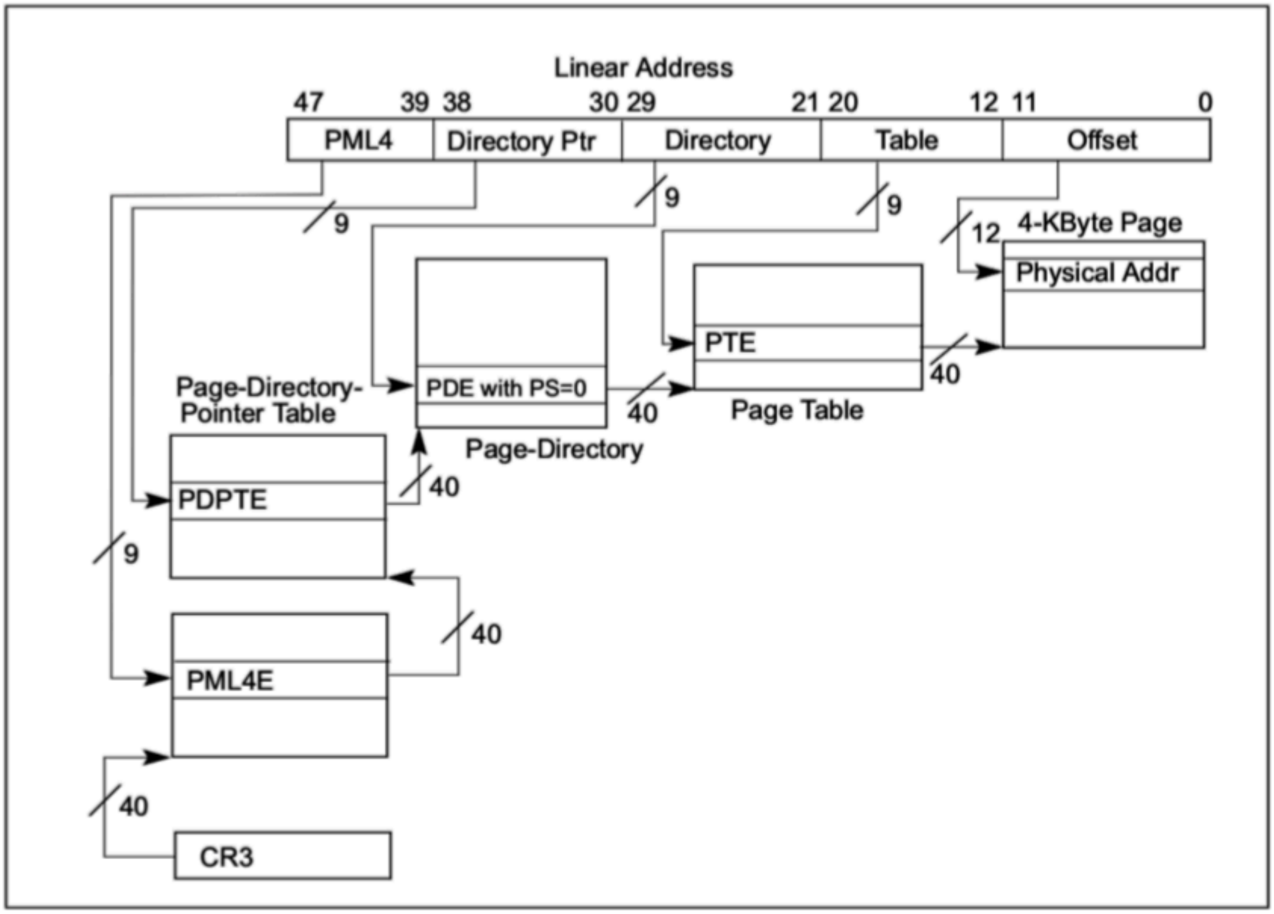



**Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging**

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

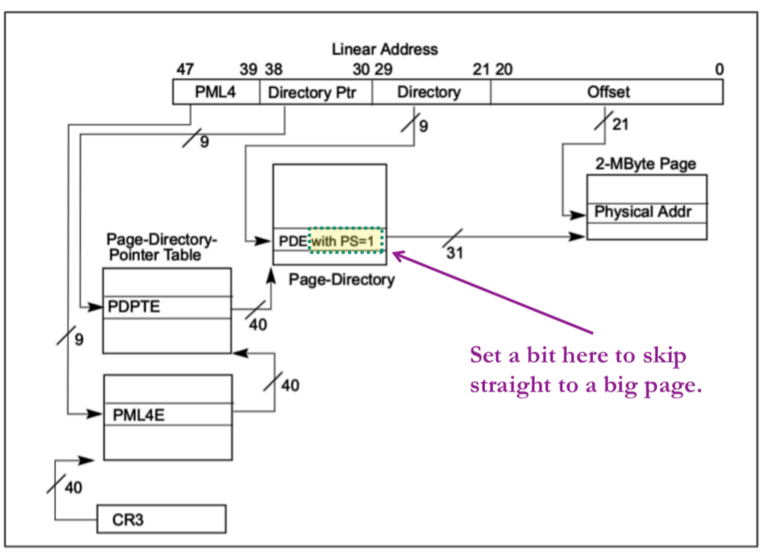



**Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging**

## … or even a 1GB huge page

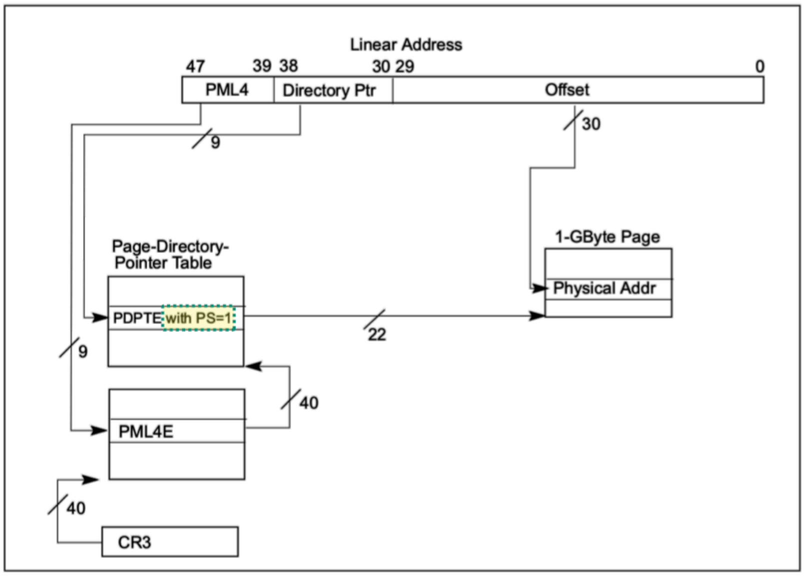


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

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)

# `top`

- gives machine level statistics
- RES column is "resident memory" - amount of physical memory being consumed
- "q" to quit
- For each process it shows how much virtual memory is being "used" by that process
- You can see that clever implementation of the OS allows for users to think they have like 10x more memory than is actually being used
- SHR is the shared memory that is being used for the process
- Virtual means program has made system calls to trigger memory
- This is on murphy or some machine like it that's why there's so many processes going on

```
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

- `Fork` + `exec` is the only way to create a child process in unix
- This is what we're doing on part 2 of project 2 OUR PROJECT HAS NEVER BEEN DONE BEFORE YAY
- Fork clones the entire process, including all of virutal memory
    - this can be 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 paren;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 excpetion will occur
    - OS handles the page fault by:

        - Copying parent's page to the child and marking both copies as writeable
        - when the faulting process is resumed, it retries the memory write

Essentially, don't copy until you absolutely KNOW that you need it. If it's just going to be overwritten anyway, why not just copy if you need it

# Demand Zeroing

another lazy optimization

- If a process asks for more memory with `sbrk` or `mmap` the OS can allocate is **lazily**
    - in other words, don't allocate the full block immediately
    - lazy allocation minimizes latency of fulfilling the reuest
    - and it prevents OS from allocatin memory that will not be used
- OS must also write zeros to newly assigned physical frames
    - So that they can't access memory in that same location that was used by the previous program
    - program does not necessarily expect the new memory to contain zeros
    - just for security so other process' data is not leaked
- OS can keep one read-only physical page filled with zeros and just five a reference to this at first
    - After the first page fault (due to writing a read-only page), then allocate a real page

# Virtual Memory in Practice

On linux `pmap` command shows a process' VM mapping

We see:

- OS tracks while 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

**Steve freaking likes emacs**

```
[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
    
    $\Big\}$ *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

# Recap

**Latency Cost**:

**Space Cost**: we have to store page tables, linear page tables are the biggest time, we can get smaller page tables by making pages bigger so we have multiople levels and only fill in lower levels wheen they are needed. We save space with fine-grained something or other. Also making page tables thmeselves is shorter

- **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.