CSCI 2021: Virtual Memory

Chris Kauffman

Last Updated:
Fri Apr 26 12:00:04 CDT 2019

Logistics

Reading Bryant/O'Hallaron

- Ch 9: Virtual Memory
- Ch 7: Linking (next)

Goals

- Address Spaces, Translation, Paged Memory
- `mmap()`, Sharing Pages

Assignment 5

- Memory allocator + Print ELF Symbol Table
- Due last day of classes

Date	Event
MWF 4/22	Virtual Memory
MWF 4/29	ELF and Linking
Mon 5/6	Last day of class
	A5 Due

Final Exams

- Sec 001 (12:20 MWF) Sat 5/11 1:30pm
- Sec 010 (3:35 MWF) Mon 5/13 10:30am

Exercise: The View of Memory Addresses so Far

- Every **process** (running program) has some memory, divided into roughly **4 areas (which are...?)**
- Reference different data/variables through their addresses
- If only a single program could run at time, no trouble: load program into memory and go
- Running multiple programs gets interesting particularly if they both reference the *same memory location*, e.g. address 1024

```
PROGRAM 1                      PROGRAM 2
## load global from 1024        # add to global at #1024
movq 1024, %rax                addl %esi, 1024
...                            ...
```

- Is this a **problem**? Is there a conflict between these programs?
- What are possible solutions?

Answers: The View of Memory Addresses so Far

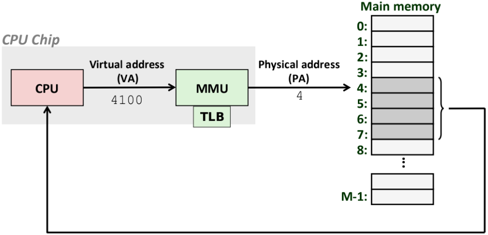
- 4 areas of memory are roughly: (1) Stack (2) Heap (3) Globals (4) Text/Instructions
- Both programs **cannot** use physical address #1024

```
PROGRAM 1                       PROGRAM 2
## load global from 1024         # add to global at #1024
movq 1024, %rax                 addl %esi, 1024
...                             ...
```

- **Solution 1:** Never let Programs 1 and 2 run together (bleck!)
- **Solution 2:** Translate every memory address in every program on loading it, run with physical addresses
  - Tough/impossible as not all addresses are known at compile/load time...
- **Solution 3:** Translate every memory address/access in every program while it runs (!!!)

Addresses are a Lie

- Operating System + Hardware translates every memory address reference **on the fly** "pretend" to "actual"
- Processes know **virtual addresses** which are translated via the memory subsystem to physical addresses in RAM and on disk
- Translation must be **FAST** so utilizes special hardware



- **MMU (Memory Manager Unit)** is a hardware element specifically designed for address translation
- Usually contains a special cache, **TLB (Translation Lookaside Buffer)**, which keeps some translated addresses

Trade-offs of Address Translation

Wins of Virtual Memory

1. Avoids conflicts between processes each referencing the same address
2. Allows each Process (running program) to believe it has entire memory to itself
3. Gives OS tons of flexibility and control over memory layout
   - Present a continuous Virtual chunk which is spread out in Physical memory
   - Use Disk Space as memory
   - Check for out of bounds memory references

Losses of Virtual Memory

1. Address translation is not constant O(1), has an impact on performance of real algorithms*
2. Requires special hardware to make translation fast enough: MMU/TLB
3. Not needed if only 1 "good" program is running on a machine

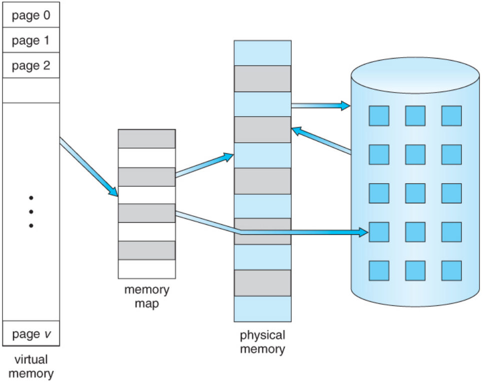
Virtual Memory is used in most modern computing systems, a "great idea" in CS

*See On a Model of Virtual Address Translation (2015)

Paged Memory Overview

- Memory is physically divided into "hunks" called **pages**
  - Page as in a "page in a notebook"
- OS maintains **page tables** to translate virtual pages to physical pages
- Two programs using the same Virtual address usually map to different physical addresses

- If many programs are using lots of memory, OS may use **swap space** on disk for some pages of memory



Source: John T. Bell Operating Systems Course Notes

Paged Memory

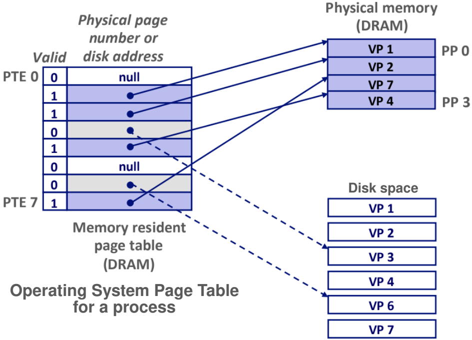
- Physical memory is divided into hunks called **pages**
- Common page size supported by many OS's (Linux) and hardware MMU's is 4KB $= 4096$ bytes
- Memory is usually byte addressable so need offset into page
- 12 bits for offset into page
- $A - 12$ bits for **page number** where A is the address size in bits
- Usually A is NOT 64-bits

```
> cat /proc/cpuinfo
vendor_id      : GenuineIntel
cpu family     : 6
model          : 79
model name     : Intel(R) Xeon(R) CPU E5-1620 v4 @ 3.50GHz
...
address sizes : 46 bits physical, 48 bits virtual
```

- Leaves one with something like $48 - 12 = 36$ bits for page #s
- Means a **page table** may have up to 2^{36} entries (!)

Page Tables and Page Table Entries (PTE)

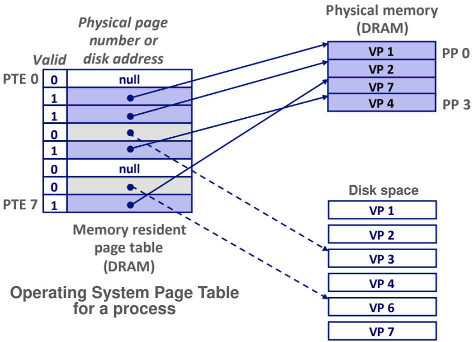
- OS **Page Table** allows translation from virtual address to physical addresses
- Each **Page Table Entry (PTE)** contains a physical page number in DRAM or Disk
- Page table contains all possible addresses for a process and where that data currently exists
- Some virtual addresses like `0x00` are **unmapped** (`null` page table entry)



- Accessing unmapped addresses leads to a **segmentation fault**

Translating Addresses

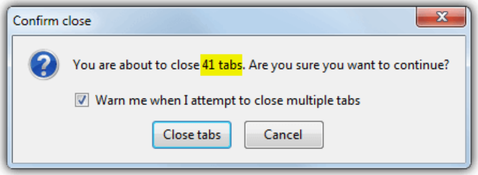
- On using a Virtual Memory address, hardware looks it up in the Page Table
- If valid (hit), address is already in DRAM, translates to physical DRAM address
- If not valid (miss), address is on disk, move to DRAM potentially evicting a current resident
- Miss = **Page fault**, notifies OS to move disk data to DRAM



- Lookup.. Hits.. misses.. this should sound **familiar to**..

Virtual Memory with DRAM as a Cache

- Virtual Memory allows illusion of 2^{48} bytes (hundreds of TBs) of memory when DRAM might only be 2^{30} to 2^{36} (tens to hundreds of GBs)
- OS can use both DRAM and Disk Space for Virtual Memory Pages
  - Pages that are frequently used stay in DRAM (swapped in)
  - Pages that haven't been used for a while end up on disk (**swapped out**)
- DRAM becomes of kind of **cache** for recently accessed Virtual Pages



Like when I was writing my composition paper but then got distracted and opened 41 Youtube tabs and when I wanted to write again it took like 5 minutes for Word to load back up because it was swapped out.

The Many Other Advantages of Virtual Memory

- Caching: Seen that VirtMem can treat main memory as a cache for larger memory
- Security: Translation allows OS to check memory addresses for validity
- Debugging: Similar to above, Valgrind checks addresses for validity
- Sharing Data: Processes can share data with one another by requesting OS to map virtual addresses to same physical addresses
- Sharing Libraries: Can share same program text between programs by mapping address space to same shared library
- Convenient I/O: Map internal OS data structures for files to virtual addresses to make working with files free of fread()/fwrite()

But first...

Exercise: Quick Review

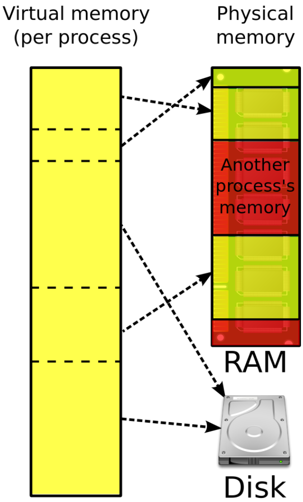
1. While running a program, memory address #1024 always refers to a physical location in DRAM (True/False: why?)
2. Two programs which both use the address #1024 cannot be simultaneously run (True/False: why?)
3. What do MMU and TLB stand for and what do they do?
4. What is a memory page? How big is it usually?
5. What is a Page Table and what is it good for?

Answers: Quick Review

1. While running a program, memory address #1024 always refers to a physical location in DRAM (True/False: why?)
   - False: #1024 is usually a **virtual address** which is translated by the OS/Hardware to a physical location which *may* be in DRAM but may instead be paged out to disk
2. Two programs which both use the address #1024 cannot be simultaneously run (True/False: why?)
   - False: The OS/Hardware will likely translate these identical virtual addresses to **different physical locations** so that the programs doe not clobber each other's data
3. What do MMU and TLB stand for and what do they do?
   - Memory Management Unit: a piece of hardware involved in translating Virtual Addresses to Physical Addresses/Locations
   - Translation Lookaside Buffer: a special cache used by the MMU to make address translation **fast**
4. What is a memory page? How big is it usually?
   - A discrete hunk of memory usually 4Kb (4096 bytes) big
5. What is a Page Table and what is it good for?
   - A table maintained by the operating system that is used to map Virtual Addresses to Physical addresses for each page

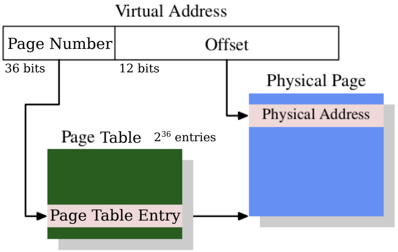
Exercise: Page Table Size

- Page tables map a virtual page to physical location
- Maintained by operating system in memory
- A **direct page** table has one entry per virtual page
- Each page is $4K = 2^{12}$ bytes, so 12 bits for offset of address into a page
- Virtual Address Space is 2^{48}
- **How many** pages of virtual memory are there?
  - How many bits specify a virtual page number?
  - How big is the page table? Is this a problem?



How big does the page table mapping virtual to physical pages need to be?

Answers: Page Table Size



"What Every Programmer Should Know About Memory" by Ulrich Drepper, Red Hat, Inc.

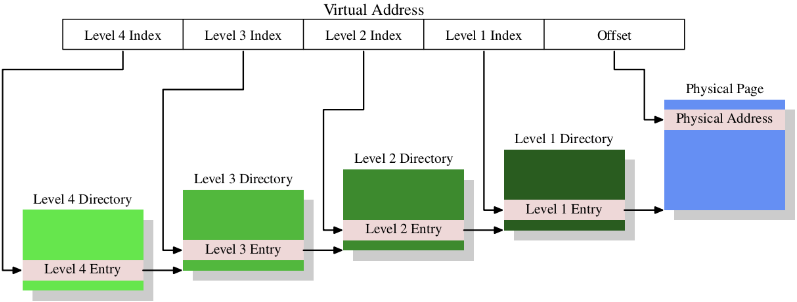
```
  48 bits for virtual address
- 12 bits for offset
--------------------
  36 bits for virtual page number
```

So, 2^{36} virtual pages…

- Every page table entry needs at least 8 bytes for a physical address
- Plus maybe 8 bytes for other stuff (on disk, permissions)
- 16 bytes per PTE $= 2^4$ bytes $\times$ 2^{36} PTEs $=$
- 2^{40} $=$ 1 Terabyte of space for the Page Table (!!!)

You've been lying again, haven't you professor…

Page Tables Usually Have Multiple Levels



"What Every Programmer Should Know About Memory" by Ulrich Drepper, Red Hat, Inc.

- Fix this absurdity with **multi-level page tables**: a sparse tree
- Virtual address divided into sections which indicate which PTE to access at different table levels
- 3-4 level page table is common in modern architectures
- Programs typically use only small amounts of virtual memory: most entries in different levels are `NULL` (not mapped) leading to much smaller page tables than a direct (array) map

Direct Page Table vs Sparse Tree Page Table

Direct Page Table: Array-Like

Direct Page Table

VP#	Valid	PP#
0000	0	/
0001	0	/
0010	1	1010
0011	0	/
0100	0	/
0101	0	/
0110	0	/
0111	0	/
1000	0	/
1001	0	/
1010	0	/
1011	0	/
1100	1	0001
1101	0	/
1110	1	1100
1111	0	/

Physical Memory

PP#	Contents
0000	-
0001	654
0010	-
0011	-
0100	-
0101	-
0110	-
0111	-
1000	-
1001	-
1010	987
1011	-
1100	321
1101	-
1110	-
1111	-

Two-level Page Table: Sparse Tree

Two-level Page Table

VP High Bits	Valid	Node
00	1	
01	0	/
10	0	/
11	1	

VP Low Bits	Valid	PP#
00	0	/
01	0	/
10	1	1010
11	0	/

VP Low Bits	Valid	PP#
00	1	0001
01	0	/
10	1	1100
11	0	/

VP#	PP#
0010	1010
1100	0001
1110	1100

Both data structures map 3 virtual pages to 3 physical page as above but use different amounts of space to do so.

Direct Table
3 pages mapped
16 entries required

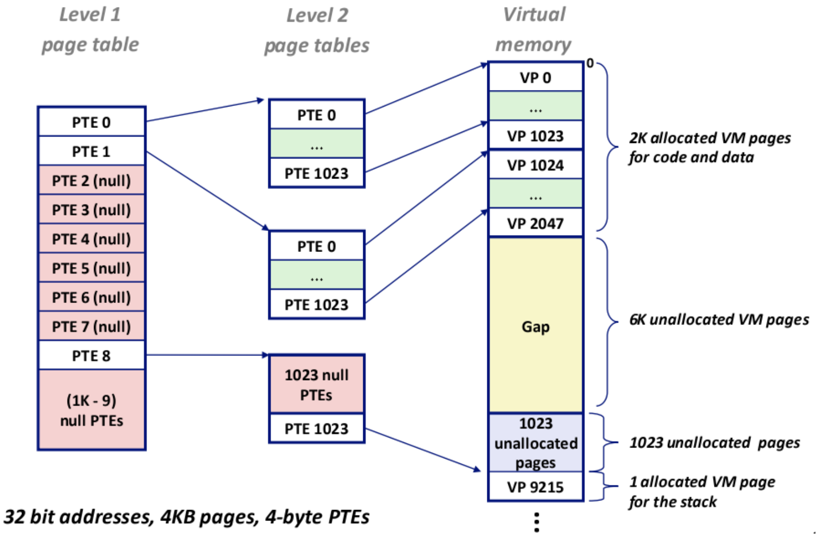
Mult-level Table
3 pages mapped
12 entries required
25% space saved

Physical Memory

PP#	Contents
0000	-
0001	654
0010	-
0011	-
0100	-
0101	-
0110	-
0111	-
1000	-
1001	-
1010	987
1011	-
1100	321
1101	-
1110	-
1111	-

Textbook Example: Two-level Page Table

Space savings gained via `NULL` portions of the page table/tree



Source: Bryant/O'Hallaron, CSAPP 3rd Ed

Exercise: Printing Contents of file

1. Write a simple program to print all characters in a file. What are key features of this program?
2. Examine `mmap_print_file.c`: does it contain all of these key features? Which ones are missing?

Answers: Printing Contents of file

1. Write a simple program to print all characters in a file. What are key features of this program?
   - Open file
   - Read 1 or more characters into memory using `fread()/fscanf()`
   - Print those characters with `printf()`
   - Read more characters and print
   - Stop when end of file is reached
   - Close file
2. Examine `mmap_print_file.c`: does it contain all of these key features? Which ones are missing?
   - Missing the `fread()/fscanf()` portion
   - Uses `mmap()` to get **direct access** to the bytes of the file
   - Treat bytes as an array of characters and print them directly

mmap(): Mapping Addresses is Ammazing

- `ptr = mmap(NULL, size,...,fd,0)` arranges backing entity of `fd` to be mapped to be mapped to `ptr`
- `fd` often a file opened with `open()` system call

```
int fd = open("gettysburg.txt", O_RDONLY);
// open file to get file descriptor

char *file_chars = mmap(NULL, size, PROT_READ, MAP_SHARED,
                        fd, 0);
// call mmap to get a direct pointer to the bytes in file associated
// with fd; NULL indicates don't care what address is returned;
// specify file size, read only, allow sharing, offset 0

printf("%c",file_chars[0]);                // print 0th file char
printf("%c",file_chars[5]);                // print 5th file char
```

mmap() allows file reads/writes without read()/write()

- Memory mapped files are not just for reading
- With appropriate options, writing is also possible

```
char *file_chars =
  mmap(NULL, size, PROT_READ | PROT_WRITE,
       MAP_SHARED, fd, 0);
```

- Assign new value to memory, OS writes changes into the file
- Example: `mmap_tr.c` to transform one character to another

Mapping things that aren't characters

`mmap()` just gives a pointer: can assert type of what it points at

- Example `int *`: treat file as array of binary ints
- Notice changing array will write to file

```
// mmap_increment.c

int fd = open("binary_nums.dat", O_RDWR);
// open file descriptor, like a FILE *

int *file_ints = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
// get pointer to file bytes through mmap,
// treat as array of binary ints

int len = size / sizeof(int);
// how many ints in file

for(int i=0; i<len; i++){
  printf("%d\n",file_ints[i]);  // print all ints
}

for(int i=0; i<len; i++){
  file_ints[i] += 1;  // increment each file int, writes back to disk
}
```

`mmap()` Compared to Traditional `fread()/fwrite()` I/O

Advantages of `mmap()`

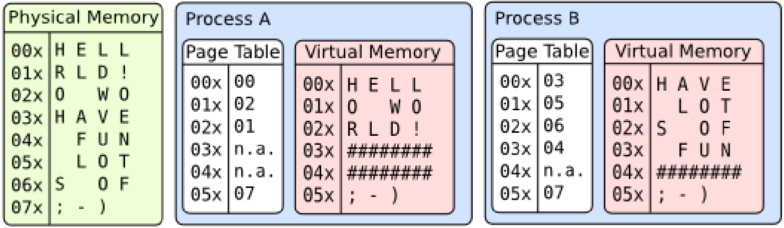
- Avoid following cycle
  - `fread()/fscanf()` file contents into memory
  - Analyze/Change data
  - `fwrite()/fscanf()` write memory back into file
- Saves memory and time
- Many Linux mechanisms backed by `mmap()` like processes sharing memory

Drawbacks of `mmap()`

- Always maps **pages** of memory: multiple of 4096b (4K)
- For small maps, lots of wasted space
- Cannot change size of files with `mmap()`: must used `fwrite()` to extend or other calls to shrink
- No bounds checking, just like everything else in C

One Page Table Per Process

- OS maintains a page table for each running program (**process**)
- Each process believes its address space ranges from `0x00` to `0xBIG` (0 to 2^{48}), its virtual address space
- Virtual addresses are mapped to physical locations in DRAM or on Disk via page tables

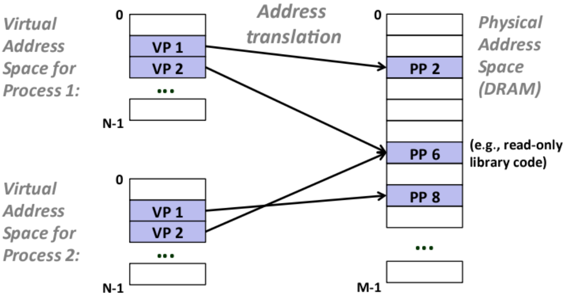


Source: OSDev.org

Two processes with their own page tables. Notice how contiguous virtual addresses are mapped to non-contiguous spots in physical memory. Notice also the ***sharing*** *of a page.*

Pages and Mapping

- Memory is segmented into hunks called **pages**, 4Kb is common (use `page-size.c` to see your system's page size)
- OS maintains tables of which pages of memory exist in RAM, which are on disk
- OS maintains tables per process that translate process virtual addresses to physical pages
- **Shared Memory** can be arranged by mapping virtual addresses for two processes to the same memory page



Shared Memory Calls

- Using OS system calls, can usually create shared memory
- Unix POSIX standard specifies following setup:

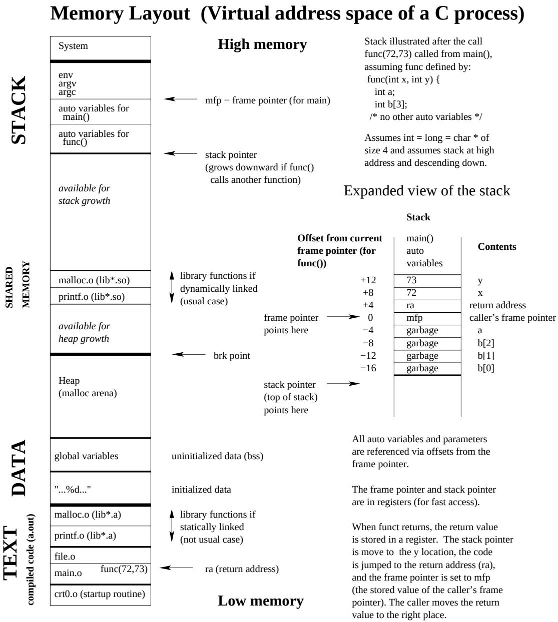
```
char *shared_name = "/something_shared";
int shared_fd =
    shm_open(shared_name, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
// retrieve a file descriptor for shared memory

ftruncate(shared_fd, SHM_SIZE);
// set the size of the shared memory area

char *shared_bytes =
    mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE,
         MAP_SHARED, shared_fd, 0);
// map into process address space
```

- Multiple processes can all "see" the same unit of memory
- Discussed in intro OS classes (CSCI 4061)
- This is an old style but still useful
- Modern incarnations use `mmap()` which we'll get momentarily

Exercise: Process Memory Image and Libraries

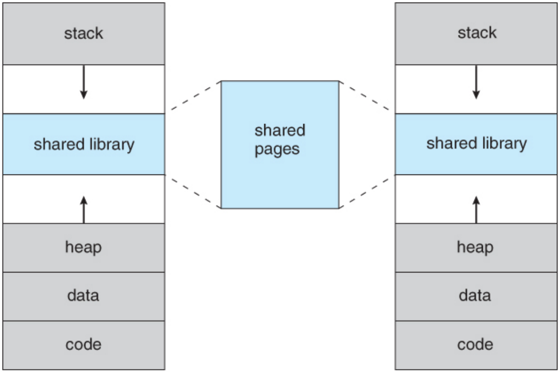
- How many programs on the system need to use `malloc()` and `printf()`?
- Where is the code for `malloc()` or `printf()` in the process memory?

Right: A detailed picture of the virtual memory image, by Wolf Holzman



Shared Libraries: `*.so` Files

- Code for libraries can be shared
- `libc.so`: shared library with `malloc()`, `printf()` etc in it
- OS puts into one page, maps all linked procs to it



Source: John T. Bell Operating Systems Course Notes

pmap: show virtual address space of running process

```
> ./memory_parts
0x5579a4cbe0c0 : global_arr
0x7fff96aff6f0 : local_arr
0x5579a53aa260 : malloc_arr
0x7f441f8bb000 : mmap'd file
my pid is 7986
press any key to continue
```

- While a program is running, determine its **process id**
- Call pmap to see how its virtual address space maps
- For full details of pmap output, refer to this article from Andreas Fester
- His diagram is awesome

```
pmap 7986
7986:   ./memory_parts
00005579a4abd000      4K r-x-- memory-parts
00005579a4cbd000      4K r---- memory-parts
00005579a4cbe000      4K rw--- memory-parts
00005579a4cbf000      4K rw---   [ anon ]
00005579a53aa000    132K rw---   [ heap ]
00007f441f2e1000   1720K r-x-- libc-2.26.so
00007f441f48f000   2044K ----- libc-2.26.so
00007f441f68e000     16K r---- libc-2.26.so
00007f441f692000      8K rw--- libc-2.26.so
00007f441f694000     16K rw---   [ anon ]
00007f441f698000    148K r-x-- ld-2.26.so
00007f441f88f000      8K rw---   [ anon ]
00007f441f8bb000      4K r--s- gettysburg.txt
00007f441f8bc000      4K r---- ld-2.26.so
00007f441f8bd000      4K rw--- ld-2.26.so
00007f441f8be000      4K rw---   [ anon ]
00007fff96ae1000    132K rw---   [ stack ]
00007fff96b48000     12K r----   [ anon ]
00007fff96b4b000      8K r-x--   [ anon ]
 total             4276K
```

Memory Protection

- Output of `pmap` indicates another feature of virtual memory: **protection**
- OS marks pages of memory with Read/Write/Execute/Share permissions like files
- Attempt to violate these and get segmentation violations (segfault)
- Ex: Executable page (instructions) usually marked as `r-x`: no write permission.
- Ensures program don't accidentally write over their instructions and change them
- Ex: By default, pages are not shared (no `s` permission) but can make it so with the right calls