CSCI 2021: Virtual Memory

Chris Kauffman

Last Updated:
Wed Dec 7 03:25:04 PM CST 2022

Logistics

Reading Bryant/O'Hallaron

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

P5: 1 Problem

- Parse an ELF binary file to extract information
- Post later today with video, due last day of class
- Useful techniques introduced in Lab 14

Goals

- Memory Address Translation
- `mmap()`'d files

Date	Event
Mon 05-Dec	Virtual Mem 1/2
Wed 07-Dec	Virtual Mem 2/2 Lab 14 mmap() HW 14 Linking
Fri 09-Dec	ELF Files/Linking 1/2
Mon 12-Dec	Obj Code/Linking 2/2
Wed 14-Dec	Last Lecture, Review Review Lab P5 Due

Will Provide Walk-through Video for P5 as I am late in releasing it

Exercise: The View of Memory Addresses so Far

- Every **process** (running program) uses memory divided into roughly 4 Logical Memory Areas
- Computing systems have various Physical Memory Devices which are shared among all running programs
- 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
...                               ...
```

- What conflict exists between these programs?
- What are possible solutions to this conflict?
- Review: what are the 4 Logical Memory Areas used by programs and some examples of Physical Memory Devices?

Answers: The View of Memory Addresses so Far

- **Review**: (1) Stack (2) Heap (3) Globals (4) Text/Instructions spread across devices like Registers, Cache, DRAM, SSDs / HDDs, tape drives
- Both programs use address #1024, behavior depends on order that instructions are interleaved between them

```
ORDER A: Program 1 loads first          ORDER B: Program 2 adds first
---------------------------------       -----------------------------------
PROGRAM 1        PROGRAM 2              PROGRAM 1        PROGRAM 2
movq 1024, %rax  ...                    ...              addl %esi, 1024
...              addl %esi, 1024        movq 1024, %rax  ...
```

- **Solution 1:** Never let Programs 1 and 2 run together (bleck!)
- **Solution 2:** Translate every memory address/access in every program while it runs

As wild as it sounds, most modern systems use memory address translation schemes called **Virtual Memory** (Solution 2) due to its many powerful features

Paged Memory

- Physical devices divide memory into chunks called **pages**
- Common page size supported by many OS's (Linux) and hardware is 4KB = 4096 bytes, can be larger with OS config
- CPU models use some # of bits for **Virtual Addresses**

```
> 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
                                  ^^^^^^^
```

- Example of address with page number and offset labelled

```
  xxxxPagenumbrOff    : 48 bits used
0x00007ffa0997a428    : 64 bit address
  |   |          |
  |   |          +-> Offset 0x428 within page, 12 bits
  |   +-> Page number 0x7ffa0997a, 36 bits
  +-> Constant bits, not used by processor
```

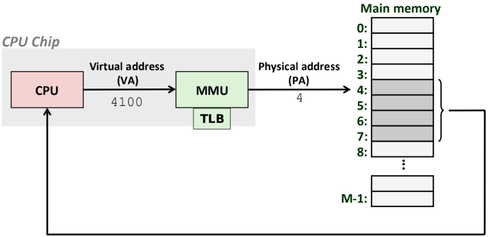
Translation happens at the Page Level

- Within a page, addresses are sequential
- Between pages, may be non-sequential

```
Page Table:
|------------------+------+----------------------|
| Virtual Page Num | Size | Physical Page Num    |
|------------------+------+----------------------|
| 00007ffa0997a000 | 4K   | RAM: 0000564955aa1000 |
| 00007ffa0997b000 | 4K   | RAM: 0000321e46937000 |
| ...              |      | ...                  |
|------------------+------+----------------------|

Address Space From Page Table:
|------------------+-------------+------------------|
| Virtual Address  | Page Offset | Physical Address |
|------------------+-------------+------------------|
| 00007ffa0997a000 |           0 | 0000564955aa1000 |
| 00007ffa0997a001 |           1 | 0000564955aa1001 |
| 00007ffa0997a002 |           2 | 0000564955aa1002 |
| ...              |             | ...              |
| 00007ffa0997afff |        4095 | 0000564955aa1fff |
|------------------+-------------+------------------|
| 00007ffa0997b000 |           0 | 0000321e46937000 |
| 00007ffa0997b001 |           1 | 0000321e46937001 |
| ...              |             | ...              |
|------------------+-------------+------------------|
```

Addresses Translation Hardware

- Translation must be **FAST** so usually involves hardware
- **MMU (Memory Manager Unit)** is a hardware element specifically designed for address translation
- Usually contains a special cache, **TLB (Translation Lookaside Buffer)**, which stores recently translated addresses

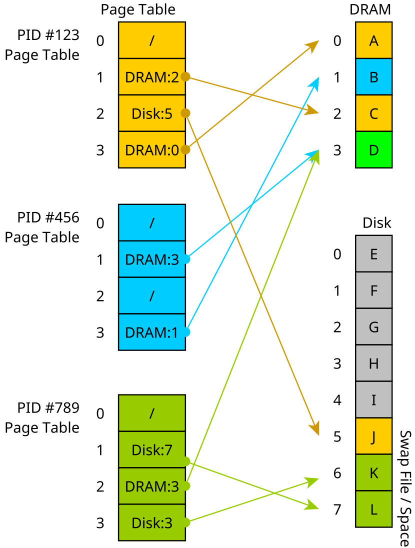


- OS Kernel interacts with MMU
- Provides location of the **Page Table**, data structure relating Virtual/Physical Addresses
- **Page Fault** : MMU couldn't map Virtual to Physical page, runs a Kernel routine to handle the fault

Exercise: Translating Virtual Addresses

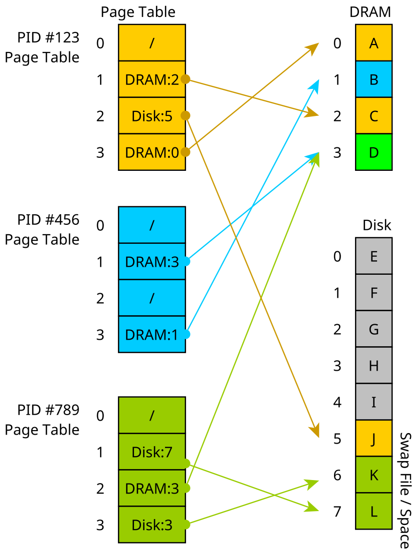
Nearby diagram illustrates relation of Virtual Pages to Physical Pages

1. How many page tables are there?
2. Where can a page table entry refer to?
3. Count the number of Virtual pages, compare to the number of physical pages - which his larger?
4. What happens if PID #123 accesses its Virtual Page #2
5. What happens if PID #456 accesses its Virtual Page #2



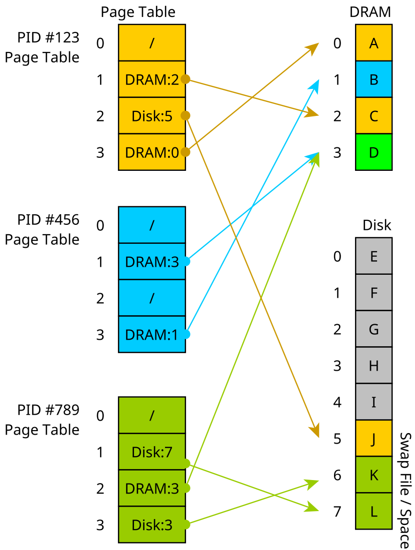
Translating Virtual Addresses 1/2

- On using a Virtual Memory address, MMU will search TLB for physical DRAM address,
- If found in TLB, Hit, use physical DRAM address
- If not found, MMU will search Page Table, if found and in DRAM, cache in TLB
- Else Miss = **Page fault**, OS decides..
  1. Page is swapped to Disk, move to DRAM, potentially evicting another page
  2. Page not in page table = Segmentation Fault



Translating Virtual Addresses 2/2

- Each process has its own page table, OS maintains mapping of Virtual to Physical addresses
- Processes “compete” for RAM
- OS gives each process impression it owns all of RAM
- OS may not have enough memory to back up all or even 1 process
- Disk used to supplement ram as **Swap Space**
- **Thrashing** may occur when too many processes want too much RAM, “constantly swapping”



Trade-offs of Address Translation

Wins of Virtual Memory

1. Avoids memory Conflicts where separate programs each use the same memory address
2. Programs can be compiled to assume they will have all memory to themselves
3. OS can make decisions about DRAM use and set policies for security and efficiency (next slide)

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 a single program is running on a machine

Wins outweigh Losses in most systems so Virtual Memory is used widely, a *great idea* in CS

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

The Many Other Advantages of Virtual Memory

1. Swap Space: System can project larger total memory than available DRAM by using Disk Space, DRAM is a "cache" for larger disk space, Swap program memory between DRAM+Disk as it is used
2. Security: Translation allows OS to check memory addresses for validity, segfault on out-of bounds access
3. Debugging: Valgrind checks addresses for validity
4. Sharing Data: Processes can share data with one another; request OS to map virtual addresses to same physical addresses
5. **Sharing Libraries**: Can share same program text between programs by mapping address space to same shared library
6. **Convenient I/O**: Map internal OS data structures for files to virtual addresses to make working with files free of read()/write()

Virtual Memory and `mmap()`

- Normally programs interact indirectly with Virtual Memory system
  - Stack/Heap/Globals/Text are mapped automatically to regions in Virtual Memory System
  - Maps are adjusted as Stack/Heap Grow/Shrink
- `mmap()` / `munmap()` directly manipulate page tables
  - `mmap()` creates new entries in page table
  - `munmap()` deletes entries in the page table
  - Can map arbitrary or specific addresses into memory
- `mmap()` is used to initially set up Stack / Heap / Globals / Text when a program is loaded by the program loader
- While a program is running can also use `mmap()` to interact with virtual memory
- A convenient way to do File I/O via **Memory Mapped Files**

Exercise: Printing Contents of file

Examine the two programs below which print the contents of a file

- Identify differences between them
- Which has a higher memory requirement?

```c
// print_file.c
int main(int argc, char *argv[]){
  FILE *fin = fopen(argv[1], "r");
  char inbuf[256];
  while(1){
    int nread =
      fread(inbuf, sizeof(char),
            256, fin);
    if(nread == 0){
      break;
    }
    for(int i=0; i<nread; i++){
      printf("%c",inbuf[i]);
    }
  }

  fclose(fin);
  return 0;
}
```

```c
// mmap_print_file.c
int main(int argc, char *argv[]){
  int fd = open(argv[1], O_RDONLY);

  struct stat stat_buf;
  fstat(fd, &stat_buf);
  int size = stat_buf.st_size;

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

  for(int i=0; i<size; i++){
    printf("%c",file_chars[i]);
  }
  printf("\n");

  munmap(file_chars, size);
  close(fd);
  return 0;
}
```

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 up to 256 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 Amazing

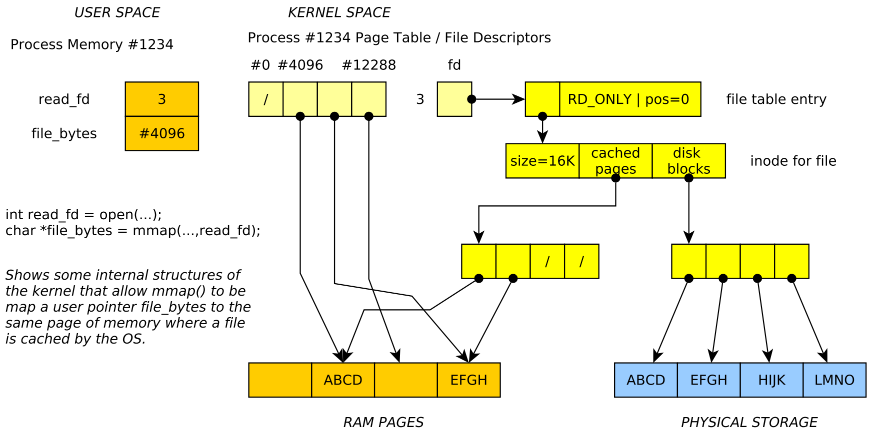
- `ptr = mmap(NULL, size,...,fd,0)` arranges backing entity of `fd` to be mapped to be mapped to `ptr`
- `fd` often a file opened with `open()` system call

```
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
```

OS usually Caches Files in RAM

- For efficiency, part of files are stored in RAM by the OS
- OS manages internal data structures to track which parts of a file are in RAM, whether they need to be written to disk
- `mmap()` alters a process Page Table to translate addresses to the cached file page
- OS tracks whether page is changed, either by file write or `mmap()` manipulation
- Automatically writes back to disk when needed
- Changes by one process to cached file page will be seen by other processes
- **See diagram on next slide**

Diagram of Kernel Structures for `mmap()`




```
int read_fd = open(...);
char *file_bytes = mmap(...,read_fd);
```

Shows some internal structures of the kernel that allow mmap() to be map a user pointer file_bytes to the same page of memory where a file is cached by the OS.

Changing Files

- `mmap()` exposes several capabilities from the OS

```
char *file_chars =
  mmap(NULL, size,
       PROT_READ | PROT_WRITE, // map allowing read + write
       MAP_SHARED,             // share changes with original file
       fd, 0);                 // file to map + offset from start
```

- 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: demonstrate working with mmap()'d binary data

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()`

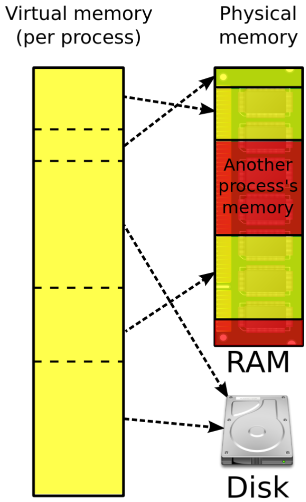
- Avoid following cycle
  - `fread()/fscanf()` file contents into memory
  - Analyze/Change data
  - `fwrite()/fscanf()` write memory back into file
- Saves memory and time
- Many Linux mechanisms backed by `mmap()` like processes sharing memory

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

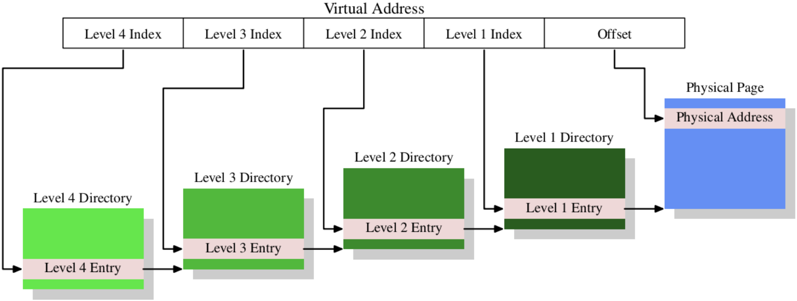
Page Table Size

- Page tables map a virtual page to physical location
- Page tables maintained by operating system in Kernel 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} bytes
- So, 2^{36} virtual pages mapped in the page table...
  - 68,719,476,736 pages
  - At 8 bytes per page entry...
  - 1 Terabyte for a page table



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

Page "Tables" are Multi-Level Sparse Trees



"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, 5-bit addresses

Direct Page Table

VP#	Valid	PP#
00000	0	/
00001	0	/
00010	1	01001
00011	0	/
00100	0	/
00101	0	/
00110	0	/
00111	0	/
01000	0	/
01001	0	/
...	0	/
11011	0	/
11100	1	00001
11101	0	/
11110	1	11100
11111	0	/

Physical Memory

PP#	Contents
00000	-
00001	654
00010	-
00011	-
00100	-
00101	-
00110	-
00111	-
01000	-
01001	987
...	-
11011	-
11100	321
11101	-
11110	-
11111	-

Two-level Page Table: Sparse Tree, 5-bit addresses

Two-level Page Table

VP High Bits	Valid	Node
000	1	
001	0	/
010	0	/
011	0	/
100	0	/
101	0	/
110	0	/
111	1	

VP Low Bits	Valid	PP#
00	0	/
01	0	/
10	1	01001
11	0	/

VP Low Bits	Valid	PP#
00	1	00001
01	0	/
10	1	11100
11	0	/

Physical Memory

PP#	Contents
00000	-
00001	654
00010	-
00011	-
00100	-
00101	-
00110	-
00111	-
01000	-
01001	987
...	-
11011	-
11100	321
11101	-
11110	-
11111	-

VP#	PP#
00010	01001
11100	00001
11110	11100

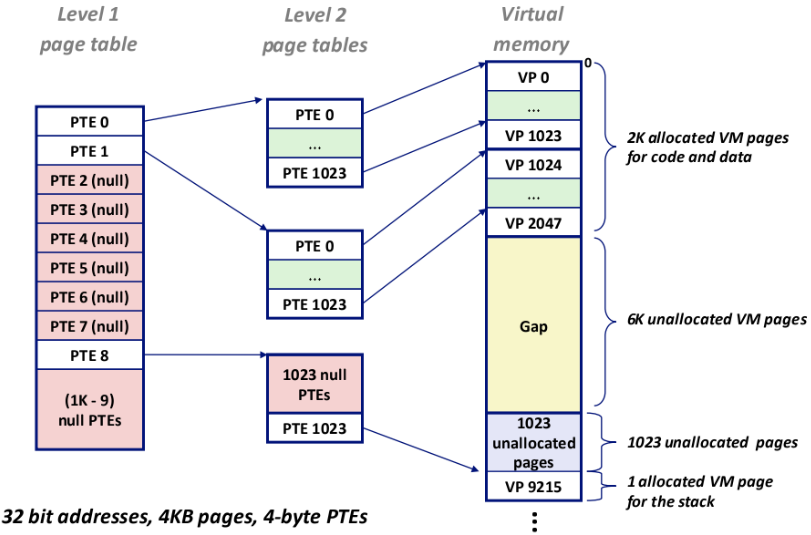
Both data structures map 3 virtual pages to 3 physical page as indicated in the map to the left but use different amounts of space to do so.

Direct Table
3 pages mapped
32 entries required

Mult-level Table
3 pages mapped
16 entries required
50% space saved

Textbook Example: Two-level Page Table

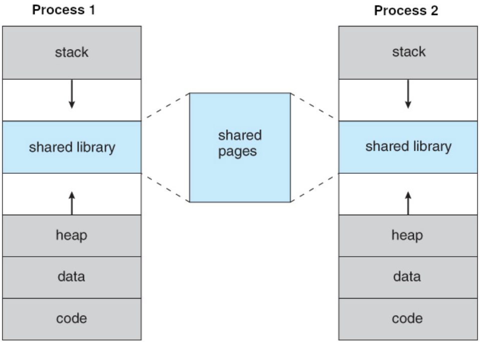
Space savings gained via `NULL` portions of the page table/tree



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

Virtual Memory Enables Shared Libraries: `*.so` Files

- Many programs need to use `malloc()`, `printf()`, `fopen()`, etc.
- Rather than each program having its own copy, modern systems use **Shared Objects** and **Shared Libraries**



Source: John T. Bell Operating Systems Course Notes

- Example: `libc.so` is the C Library which contains Code/Text for `malloc()`, `printf()`, `fopen()`, `etc.`, 1-2MB of code
- One copy of `libc.so` exists in DRAM
- Many programs "share it" via Page Table mappings in Virtual Memory, reduces overall memory required

pmap: show virtual address space of running process

```
> ./memory_parts
0x5575555a71e9 : main()
0x5575555aa0c0 : global_arr
0x557555b482a0 : heap_arr
0x600000000000 : mmap'd block1
0x600000001000 : mmap'd block2
0x7f2244dc4000 : mmap'd file
0x7ffff0133b70 : stack_arr
my pid is 496605
press any key to continue
```

- Determine **process id** of running program
- pmap reports its virtual address space
- Reports features of each mapped page range such as size, permissions, possibly logical area

```
> pmap 496605
496605:   ./memory_parts
00005575555a6000     4K r---- memory_parts
00005575555a7000     4K r-x-- memory_parts TEXT
00005575555a8000     4K r---- memory_parts
00005575555a9000     4K r---- memory_parts
00005575555aa000     4K rw--- memory_parts GLOBALS
00005575555ab000     4K rw---   [ anon ]
0000557555b48000   132K rw---   [ anon ]   HEAP
0000600000000000     8K rw---   [ anon ]
00007f2244bca000     8K rw---   [ anon ]
00007f2244bcc000   152K r---- libc-2.32.so
00007f2244bf2000  1332K r-x-- libc-2.32.so
00007f2244d3f000   304K r---- libc-2.32.so
00007f2244d8e000    12K rw--- libc-2.32.so
00007f2244d91000    24K rw---   [ anon ]
00007f2244dc4000     4K r---- gettysburg.txt
00007f2244dc5000     8K r---- ld-2.32.so
00007f2244dc7000   132K r-x-- ld-2.32.so
00007f2244de8000    36K r---- ld-2.32.so
00007f2244df2000     8K rw--- ld-2.32.so
00007ffff0114000   132K rw---   [ stack ]  STACK
00007ffff014d000    12K r----   [ anon ]
 total            2352K
```

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

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

Additional Review Questions

- What OS data structure facilitates the Virtual Memory system? What kind of data structure is it?
- What does `pmap` do?
- What does the `mmap()` system call do that enables easier I/O? How does this look in a C program?
- Describe at least 3 benefits a Virtual Memory system provides to a computing system