CMSC216: Virtual Memory

Chris Kauffman

Last Updated:
Tue Dec 2 09:24:04 AM EST 2025

Logistics

Assignments

Last Batch...

- Lab12: Mapped Array / Threads
- HW12: Threads / Worms
- P5 up by tomorrow

Goals

- Virtual Memory System
- Isolation of process memory
- Detection of Segmentation Faults
- OS Allocation of Memory
- Memory Mapped Files

Reading Bryant/O'Hallaron

Ch	Read?	Topic
Ch 6		**The Memory Hierarchy**
Ch 6.1	skim	Storage Technologies
Ch 6.2	READ	Locality
Ch 6.3	READ	The Memory Hierarchy
Ch 6.4	opt	Cache Memories
Ch 6.5	READ	Writing Cache Friendly Code
Ch 6.6	skim	Impacts of Cache on Performance
Ch 9		**Virtual Memory**
Ch 9.1-6	skim	VM Overview, Address Translation
Ch 9.7	opt	Case Study
Ch 9.8	READ	Memory mapping and `mmap()`
Ch 9.9	READ	Dynamic Memory Allocation
Ch 9.10	opt	Garbage Collection
Ch 9.11	skim	Memory Bugs in C Programs

Next: Ch 12 Concurrent Programming (Threads)

Announcements

Exercise: Potential Conflicts in Memory

- Running multiple programs gets interesting particularly if they both reference the *same memory location*, e.g. address 8192

```
PROGRAM 1                  PROGRAM 2
...                        ...
## load global from #8192  ## add to global at #8192
movq 8192, %rax            addl %esi, 8192
...                        ...
```

- What conflict exists between these programs?
- What are possible solutions to this conflict?

Answers: Potential Conflicts in Memory

- Both programs use address #8192, 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 8192, %rax   ...                   ...               addl %esi, 8192
...               addl %esi, 8192       movq 8192, %rax   ...
```

- **Solution 1:** Never let Programs 1 and 2 run together (bleck!)
- **Solution 2:** Neither program actually uses address #8192: translate memory addresses to other places

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
```

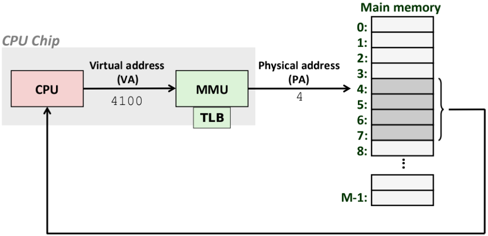
Translation happens at the Page Level

- 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

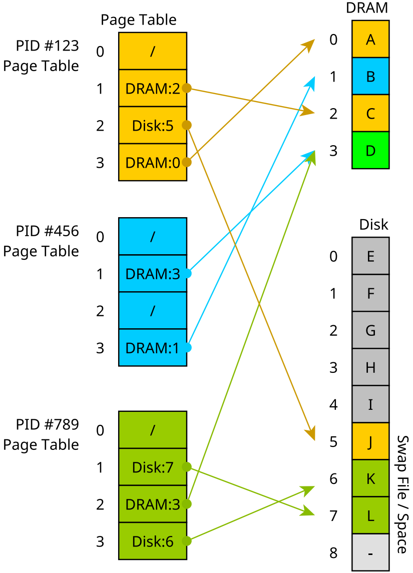


- OS Kernel interacts with MMU
- Provides location of the **Page Table**, data structure relating Virtual/Physical Addresses
- **Page Fault** : MMU couldn't map Virtual to Physical page, runs a Kernel routine to handle the fault

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



Page Table

PID #123 Page Table

0	/
1	DRAM:2
2	Disk:5
3	DRAM:0

PID #456 Page Table

0	/
1	DRAM:3
2	/
3	DRAM:1

PID #789 Page Table

0	/
1	Disk:7
2	DRAM:3
3	Disk:6

DRAM

0	A
1	B
2	C
3	D

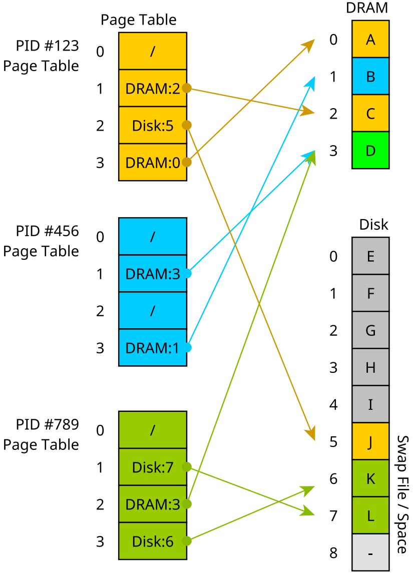
Disk

Swap File / Space

0	E
1	F
2	G
3	H
4	I
5	J
6	K
7	L
8	-

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



Page Table

PID #123 Page Table

0	/
1	DRAM:2
2	Disk:5
3	DRAM:0

PID #456 Page Table

0	/
1	DRAM:3
2	/
3	DRAM:1

PID #789 Page Table

0	/
1	Disk:7
2	DRAM:3
3	Disk:6

DRAM

0	A
1	B
2	C
3	D

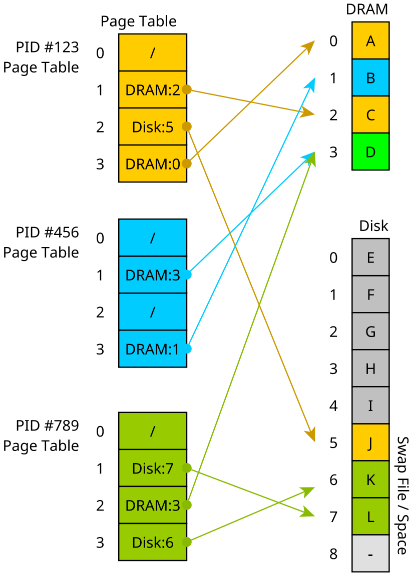
Disk

0	E
1	F
2	G
3	H
4	I
5	J
6	K
7	L
8	-

Swap File / Space

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
- We will use `mmap()` for 2 specific purposes
  1. Implement our own `malloc()` / `free()` system (Project 5)
  2. A convenient way to interact with files via **Memory Mapped Files** (in lecture/lab)

Basic Use of mmap() System Call

```
// memory_parts.c: demo mmap() and allow inspection of memory
{
  // create 2 blocks of mmap()'d space starting at a fixed address
  // which are contiguous
  char *address = (char *) 0x0000600000000000; // requested starting address for block
  size_t bsize = 0x1000;                       // 1*16^3 = 4096

  char *block1 =
    mmap(address, bsize,                       // request start address and size
         PROT_READ | PROT_WRITE,               // can read and write this block
         MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,  // no sharing/file, exact address or fail
         -1, 0);                               // default options for anonymous block
  char *block2 =
    mmap(address+bsize, bsize,                 // start at end of previous block
         PROT_READ | PROT_WRITE,               // similar options to previous block
         MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,
         -1, 0);
  // create 3rd block that is not contiguous
  char *block3 =
    mmap(NULL, 3*bsize,                        // NULL: allow OS to choose address
         PROT_READ | PROT_WRITE,               // similar options to previous block
         MAP_PRIVATE | MAP_ANONYMOUS,
         -1, 0);
}
```

pmap: show virtual address space of running process

```
> ./memory_parts
0x5c9d813151e9 : main()
0x5c9d813180a0 : global_arr
0x5c9d826b92a0 : heap_arr
0x600000000000 : mmap'd block1
0x600000001000 : mmap'd block2
0x7b4a8f83c000 : mmap'd block3
0x7b4a8f83b000 : mmap'd file
0x7ffdc5499050 : 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
00005c9d81314000      4K r---- memory_parts
00005c9d81315000      4K r-x-- memory_parts    TEXT
00005c9d81316000      4K r---- memory_parts
00005c9d81317000      4K r---- memory_parts
00005c9d81318000      4K rw--- memory_parts    GLOBALS
00005c9d81319000      4K rw---   [ anon ]
00005c9d826b9000    132K rw---   [ anon ]      HEAP
0000600000000000      8K rw---   [ anon ]      Block 1+2
00007b4a8f613000     12K rw---   [ anon ]
00007b4a8f616000    144K r---- libc.so.6
00007b4a8f63a000   1388K r-x-- libc.so.6       C LIBRARY
00007b4a8f795000    340K r---- libc.so.6       (SHARED)
00007b4a8f7ea000     16K r---- libc.so.6
00007b4a8f7ee000      8K rw--- libc.so.6
00007b4a8f7f0000     40K rw---   [ anon ]
00007b4a8f83b000      4K r---- gettysburg.txt  mmap()'d FILE
00007b4a8f83c000     12K rw---   [ anon ]      BLOCK 3
00007b4a8f83f000      4K r---- ld-linux-x86-64.so.2
00007b4a8f840000    156K r-x-- ld-linux-x86-64.so.2
00007b4a8f867000     44K r---- ld-linux-x86-64.so.2
00007b4a8f872000      8K r---- ld-linux-x86-64.so.2
00007b4a8f874000      8K rw--- ld-linux-x86-64.so.2
00007ffdc547a000    132K rw---   [ stack ]     STACK
00007ffdc5589000     16K r----   [ anon ]
00007ffdc558d000      8K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total             2508K
```

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: 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[]){
  int fin = open(argv[1], O_RDONLY);
  char inbuf[256];
  while(1){
    int nread =
      read(fin, inbuf, 256);
    if(nread == 0){
      break;
    }
    for(int i=0; i<nread; i++){
      printf("%c",inbuf[i]);
    }
  }

  close(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 `read()/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 `read()/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() for File Operations

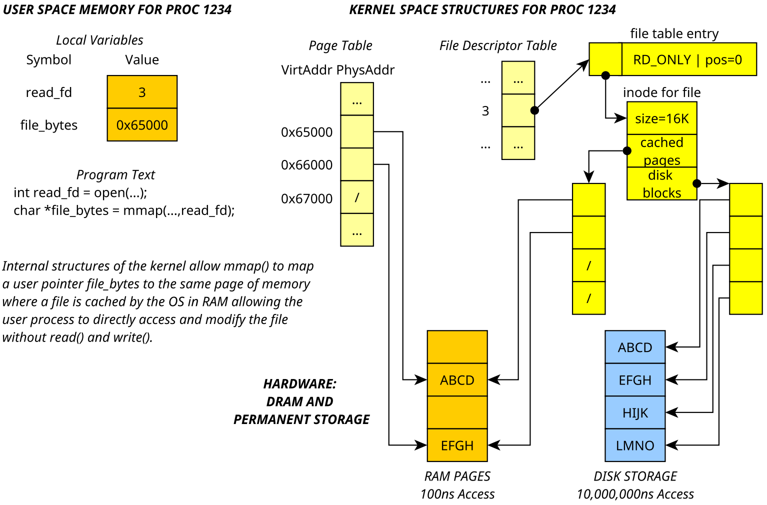
- ptr = mmap(NULL, size,...,fd,0) arranges backing data of fd to be mapped to be mapped to ptr
- fd often a file opened with open() system call

```
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**

Kernel Structures for `mmap()`'d Files



USER SPACE MEMORY FOR PROC 1234

Local Variables

Symbol	Value
read_fd	3
file_bytes	0x65000

Program Text

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

Internal structures of the kernel allow mmap() to map a user pointer file_bytes to the same page of memory where a file is cached by the OS in RAM allowing the user process to directly access and modify the file without read() and write().

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 read()/write() I/O

Advantages of mmap()

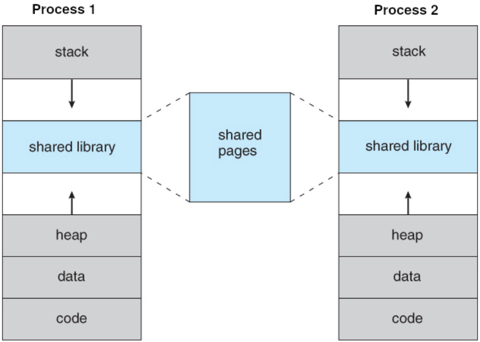
- Avoid following cycle
  - read()/fscanf() file contents into memory
  - Analyze/Change data
  - write()/fprintf() write memory back into file
- Saves memory and time
- Many Linux mechanisms backed by mmap() like processes sharing memory

Drawbacks of mmap()

- Always maps **pages** of memory: multiple of 4096b (4K)
- For small maps, lots of wasted space
- write() automatically grows files but mmap() doesn't; must use other means to change file size
- No bounds checking, just like everything else in C

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

(Optional) Physical Locations of Pages

- UMN Kernel Object Student group members put together a `vpmap` program to print virtual to physical page locations on Linux
- Requires Administrator rights to use as physical locations are OS business
- https://github.com/UMN-Kernel-Object/virtmem

vpmap Sample Output

```
################################################################################
## vpmap shows Virtual Page Number (vpn) followed by Page Frame Number (pfn)
$> sudo ./vpmap 64814
[sudo] password for sudo:
Process 64814
55d11d5c7000-55d11d5c8000 r--p 00000000 fe:01 5119082       /virtmem/memory_parts
| vpn: 55d11d5c7  present  pfn: 2a9314  dirty: 1  exclu: 1  wprot: 0  isfile: 1

55d11d5c8000-55d11d5c9000 r-xp 00001000 fe:01 5119082       /virtmem/memory_parts
| vpn: 55d11d5c8  present  pfn: 1fddc6  dirty: 1  exclu: 1  wprot: 0  isfile: 1
...

55d11e7f0000-55d11e811000 rw-p 00000000 00:00 0             [heap]
| vpn: 55d11e7f0  present  pfn: 440dc0  dirty: 1  exclu: 1  wprot: 0  isfile: 0
| vpn: 55d11e7f1
| vpn: 55d11e7f2
| vpn: 55d11e7f3  ## unmapped pages (promised but not delivered)
...

7fc074a41000-7fc074a63000 r--p 00000000 fe:01 19139877      /usr/lib/libc.so.6
| vpn: 7fc074a41  present  pfn: 22b275  dirty: 1  exclu: 0  wprot: 0  isfile: 1
| vpn: 7fc074a42  present  pfn: 3b677d  dirty: 1  exclu: 0  wprot: 0  isfile: 1
...

7fc074a63000-7fc074bbd000 r-xp 00022000 fe:01 19139877      /usr/lib/libc.so.6
| vpn: 7fc074a63  present  pfn: 3ac617  dirty: 1  exclu: 0  wprot: 0  isfile: 1
...
| vpn: 7fc074a6b  present  pfn: 3ac61f  dirty: 1  exclu: 0  wprot: 0  isfile: 1
| vpn: 7fc074a6c  present  pfn: 22b200  dirty: 1  exclu: 0  wprot: 0  isfile: 1
| vpn: 7fc074a6d  present  pfn: 22b201  dirty: 1  exclu: 0  wprot: 0  isfile: 1

7ffd46c53000-7ffd46c74000 rw-p 00000000 00:00 0             [stack]
...                 ## Highest addresses in stack in use but no physical pages
| vpn: 7ffd46c6f  ## yet assigned to lower pages
| vpn: 7ffd46c70
| vpn: 7ffd46c71  present  pfn: 403934  dirty: 1  exclu: 1  wprot: 0  isfile: 0
| vpn: 7ffd46c72  present  pfn: 21b607  dirty: 1  exclu: 1  wprot: 0  isfile: 0
| vpn: 7ffd46c73  present  pfn: 18ef8e  dirty: 1  exclu: 1  wprot: 0  isfile: 0
...
```

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