

# CSCI 2021: Virtual Memory

Chris Kauffman

*Last Updated:*

*Fri Apr 22 01:11:47 PM CDT 2022*

# Logistics

## Reading Bryant/O'Hallaron

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

## P5

- ▶ 1 Problem
- ▶ Implement a small version of `malloc()` / `free()`
- ▶ Post over the weekend, due last day of class

## Goals

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

Date	Event
Fri 4/22	Virtual Mem 1/2
Mon 4/25	Virtual Mem 2/2
Wed 4/27	ELF Files/Linking 1/2 Lab 14 <code>mmap()</code> HW 14 Linking
Fri 4/29	Obj Code/Linking 2/2
Mon 5/2	Last Lecture, Review P5 Due
	<b>Final Exam 1:30pm</b>

## 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

...

## load global from #1024

movq 1024, %rax

...

PROGRAM 2

...

## add to global at #1024

addl %esi, 1024

...

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

## 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 use physical address #1024, behavior depends on order that instructions are interleaved between them

ORDER A: Program 1 loads first

PROGRAM 1	PROGRAM 2
movq 1024, %rax	...
...	addl %esi, 1024

ORDER B: Program 2 adds first

PROGRAM 1	PROGRAM 2
...	addl %esi, 1024
movq 1024, %rax	...

- ▶ **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 (!!!)

## Paged Memory

- ▶ Physical memory is divided into hunks 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:

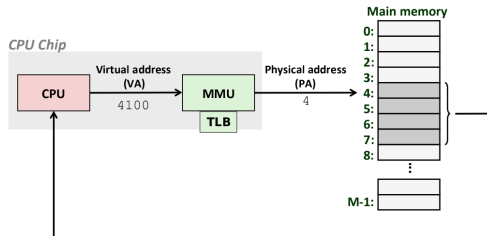
Virtual Page	Size	Physical Page
00007ffa0997a000	4K	RAM: 0000564955aa1000
00007ffa0997b000	4K	RAM: 0000321e46937000
...		...

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

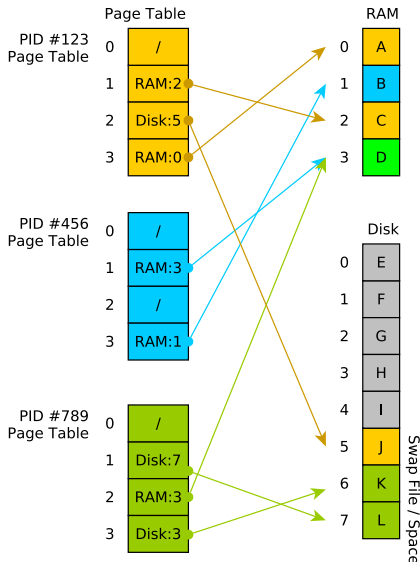


- ▶ OS Kernel interacts with MMU
- ▶ Provides location of the **Page Table**, data structure relating Virtual/Physical Addresses
- ▶ **Page Fault** : MMU couldn't map Virtual to Physical page, runs a Kernel routine to handle the fault

# Exercise: Translating Virtual Addresses

Nearby diagram illustrates relation of Virtual Pages to Physical Pages

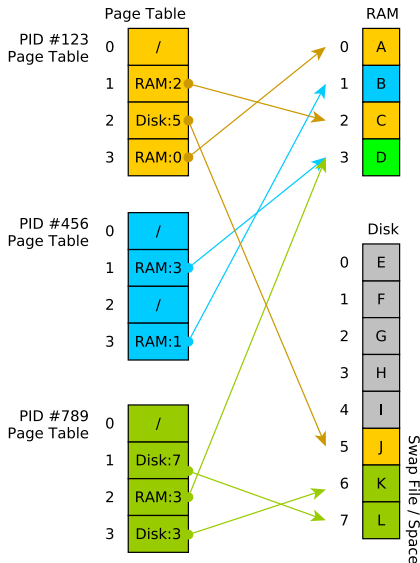
1. **How many** page tables are there?
2. **Where** can a page table entry refer to?
3. **Count** the number of Virtual pages, compare to the number of physical pages - which is larger?
4. **What happens** if PID #123 accesses its Virtual Page #2
5. **What happens** if PID #456 accesses its Virtual Page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





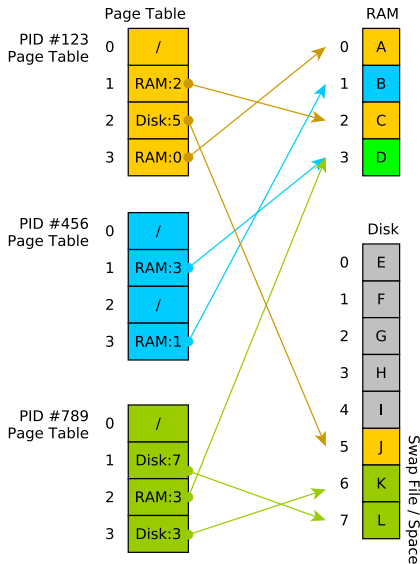
# Translating Virtual Addresses 1/2

- ▶ 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 processes each referencing the same address, conflicting
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 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. Caching: Seen that VirtMem can treat main memory as a cache for larger memory
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()` function allows direct manipulation of the page table
  - ▶ `mmap()` creates new entries in page table, `munmap()` deletes entries
  - ▶ Can map arbitrary or specific addresses into memory
- ▶ `mmap()` is used to initially set up the Stack/Heap/Globals/Text when a program is loaded by the program loader
- ▶ Can also use `mmap()` as a convenient way to interact with files 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?

```
1 // print_file.c
2 int main(int argc, char *argv[]){
3     FILE *fin = fopen(argv[1], "r");
4     char inbuf[256];
5     while(1){
6         int nread =
7             fread(inbuf, sizeof(char),
8                 256, fin);
9         if(nread == 0){
10             break;
11         }
12         fwrite(inbuf, sizeof(char),
13             nread, stdout);
14     }
15     fclose(fin);
16     return 0;
17 }
18 }
```

```
1 // mmap_print_file.c
2 int main(int argc, char *argv[]){
3     int fd = open(argv[1], O_RDONLY);
4
5     struct stat stat_buf;
6     fstat(fd, &stat_buf);
7     int size = stat_buf.st_size;
8
9     char *file_chars =
10         mmap(NULL, size,
11             PROT_READ, MAP_SHARED,
12             fd, 0);
13
14     for(int i=0; i<size; i++){
15         printf("%c", file_chars[i]);
16     }
17     printf("\n");
18
19     munmap(file_chars, size);
20     close(fd);
21     return 0;
22 }
```

## 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
```

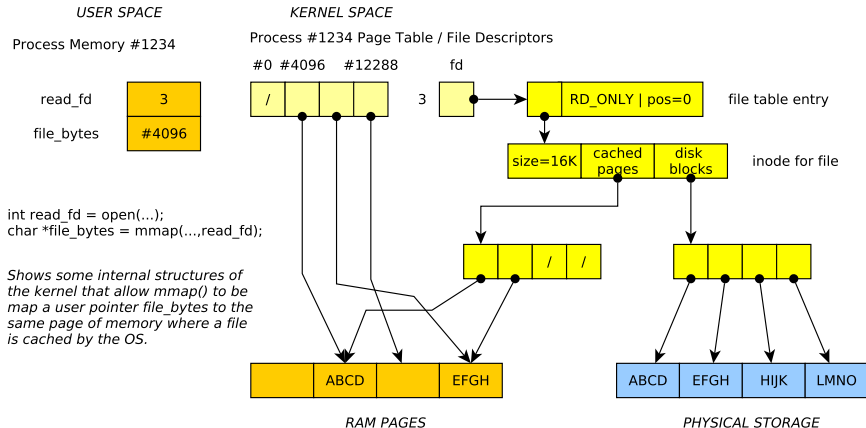
```
printf("%c", file_chars[0]);           // print 0th file char  
printf("%c", file_chars[5]);          // print 5th file char
```



## 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()



# 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()`

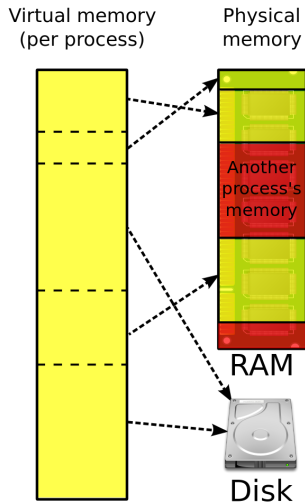
- ▶ Avoid following cycle
  - ▶ `fread()/fscanf()` file contents into memory
  - ▶ Analyze/Change data
  - ▶ `fwrite()/fscanf()` write memory back into file
- ▶ Saves memory and time
- ▶ Many Linux mechanisms backed by `mmap()` like processes sharing memory

## 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

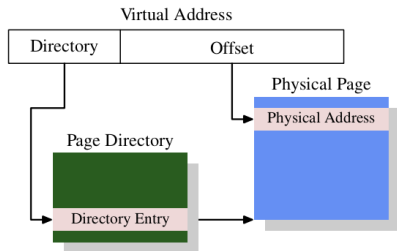
# Page Table Size

- ▶ 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
- ▶ **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.

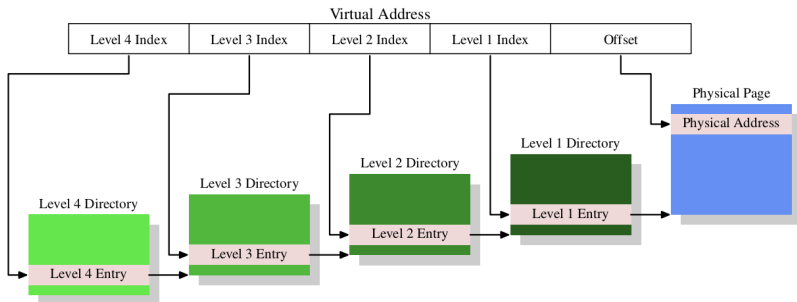
48 bits for virtual address  
- 12 bits for offset  
-----  
36 bits for virtual page number

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 = ..
- ▶  $2^{40}$  bytes  
= 1 Terabyte of space for the Page Table (!!!)

Clearly a system with 4-32 GB (Gigabytes) of main memory can't use a Direct Mapped page table

# 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, 5-bit addresses

Direct Page Table			Physical Memory	
VP#	Valid	PP#	PP#	Contents
00000	0	/	00000	-
00001	0	/	00001	654
00010	1	01001	00010	-
00011	0	/	00011	-
00100	0	/	00100	-
00101	0	/	00101	-
00110	0	/	00110	-
00111	0	/	00111	-
01000	0	/	01000	-
01001	0	/	01001	987
...	0	/	...	-
11011	0	/	11011	-
11100	1	00001	11100	321
11101	0	/	11101	-
11110	1	11100	11110	-
11111	0	/	11111	-

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

Two-level Page Table

Physical Memory

The diagram illustrates the mapping of virtual pages to physical memory using a two-level page table. It shows a VP High Bits table, a VP Low Bits table, and a Physical Memory table. Arrows indicate the mapping from virtual pages to physical pages.

**VP High Bits 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 Table:**

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

**Physical Memory Table:**

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

**Mapping:**

- Virtual page 000 (VP High Bits 000, Node 00) maps to physical page 00001 (VP Low Bits 01, PP# 00001).
- Virtual page 111 (VP High Bits 111, Node 10) maps to physical page 11100 (VP Low Bits 10, PP# 11100).
- Virtual page 010 (VP High Bits 010, Node 01) maps to physical page 00010 (VP Low Bits 01, PP# 00010).

**Direct Table:**

VP#	PP#
00010	01001
11100	00001
11110	11100

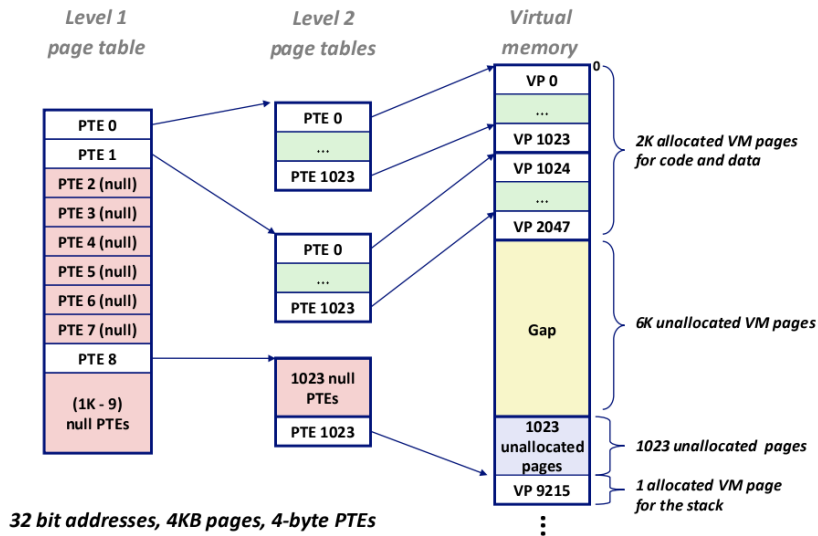
Both data structures map 3 virtual pages to 3 physical page as indicated in the map to the left but use different amounts of space to do so.

**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

# One Page Table Per Process

- ▶ OS maintains a page table for each running program (1 page table per **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

Physical Memory	
00x	H E L L
01x	R L D !
02x	0 W O
03x	H A V E
04x	F U N
05x	L O T
06x	S O F
07x	; - )

Process A			
Page Table		Virtual Memory	
00x	00	00x	H E L L
01x	02	01x	0 W O
02x	01	02x	R L D !
03x	n.a.	03x	#####
04x	n.a.	04x	#####
05x	07	05x	; - )

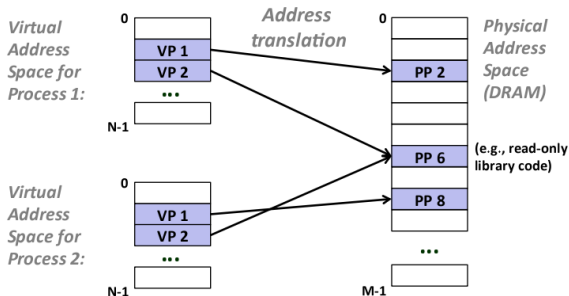
Process B			
Page Table		Virtual Memory	
00x	03	00x	H A V E
01x	05	01x	L O T
02x	06	02x	S O F
03x	04	03x	F U N
04x	n.a.	04x	#####
05x	07	05x	; - )

Source: [OSDev.org](http://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

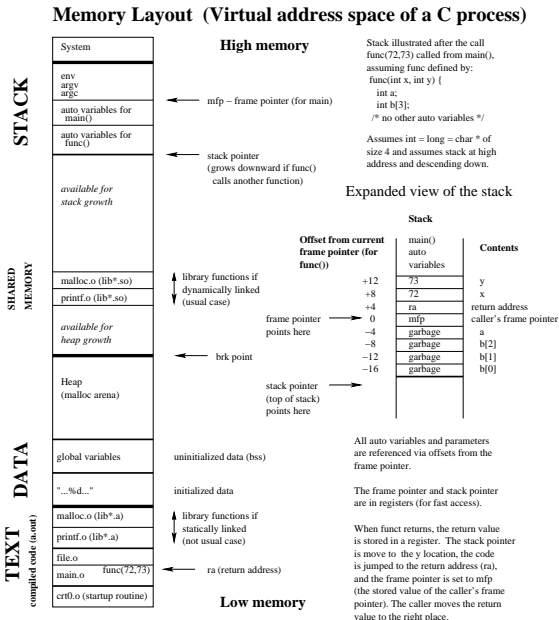
- ▶ Memory is segmented into hunks called **pages**, 4Kb is common (use `page-size.c` to see your system's page size)
- ▶ OS maintains tables of which pages of memory exist in RAM, which are on disk
- ▶ OS maintains tables per process that translate process virtual addresses to physical pages
- ▶ **Shared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



# Exercise: Process Memory Image and Libraries

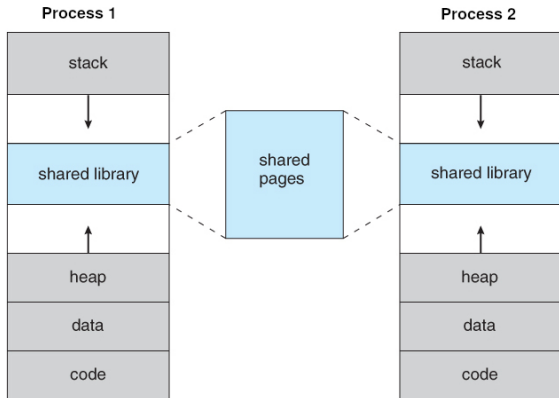
- ▶ How many programs on the system need to use `malloc()` and `printf()`?
- ▶ Where is the code for `malloc()` or `printf()` in the process memory?

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
0x5575555a71e9 : main()
0x5575555aa0c0 : global_arr
0x5575555b482a0 : heap_arr
0x6000000000000 : mmap'd block1
0x6000000001000 : mmap'd block2
0x7f2244dc4000 : mmap'd file
0x7fffff0133b70 : stack_arr
my pid is 496605
press any key to continue
```

```
> 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 ]
00005575555b48000  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
00007fffff0114000  132K rw--- [ stack ]    STACK
00007fffff014d000    12K r---- [ anon ]
total                2352K
```

- ▶ Determine **process id** of running program
- ▶ pmap reports its virtual address space
- ▶ More details of pmap output in this [article from Andreas Fester](#)
- ▶ His diagram is awesome

# 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 do 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