CSCI 2021: Virtual Memory

Chris Kauffman

Last Updated:
Wed Dec 8 02:20:19 PM CST 2021

Logistics

Reading Bryant/O'Hallaron

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

P5

- ▶ 1 Required Problem
- ▶ Employ Memory Mapping to traverse binary file

Goals

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

Date	Event
12/06 Mon	Virtual Mem 1/2
12/08 Wed	Virtual Mem 2/2
	Lab 14 VirtMem
	HW 14 Linking
	P5 Release
12/10 Fri	ELF Files/Linking 1/2
12/13 Mon	Obj Code/Linking 2/2
12/15 Wed	Last Lecture, Review
	Last Lab, Review
	P5 Due
12/18 Sat	**Final Exam 1:30pm**

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
...                              ...
```

- 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       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 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
- ▶ 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 $48 - 12 = 36$ bits for Page Number
- ▶ Means a **page table** may have up to 2^{36} entries (!)

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
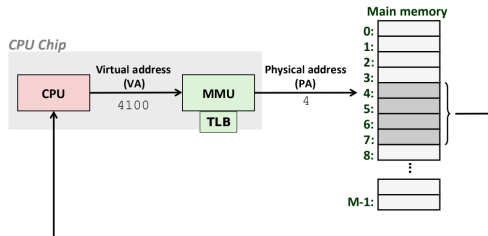

- ▶ OS Kernel interacts with MMU
- ▶ Provides location of the **Page Table**, data structure relating Virtual/Physical Addresses
- ▶ **Page Fault** : MMU couldn't map Virtual to Physical page, runs a Kernel routine to handle the fault

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 his larger?

4. What happens if PID #123 accesses its Virtual Page #2

5. What happens if PID #456 accesses its Virtual Page #2
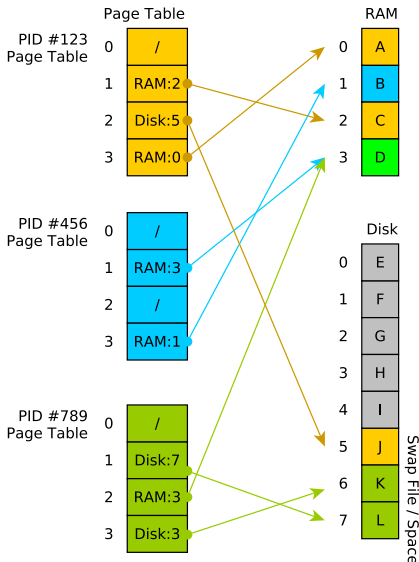
Translating Virtual Addresses 1/2

- 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
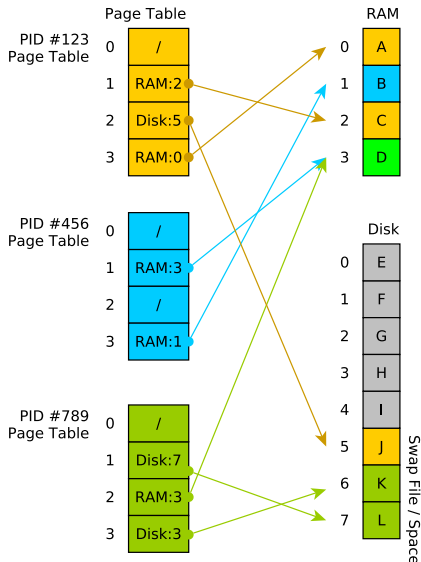
Translating Virtual Addresses 2/2

- 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"
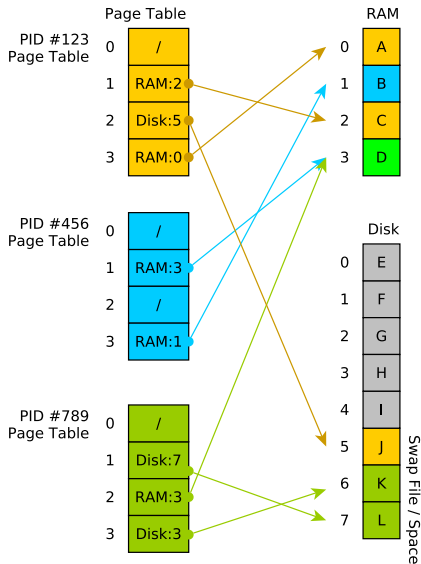
Trade-offs of Address Translation

Wins of Virtual Memory

1. Avoids 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 often outweigh Losses so Virtual Memory is used in *most* modern computing systems, 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()

Will start with last of these and the hand mmap() function

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

1. Examine `print_file.c`: reads contents of a file and prints it to the screen. Identify the key parts of this program and its **memory requirements.**
2. Examine `mmap_print_file.c`: does it contain all of these key features? Which ones are missing?

Exercise: Printing Contents of file

```c
1  // print_file.c
2  int main(int argc, char *argv[]){
3    FILE *fin = fopen(argv[1], "r");
4    char inchar;
5    while(1){
6      int result =
7        fscanf(fin, "%c", &inchar);
8      if(result == EOF){
9        break;
10     }
11     printf("%c", inchar);
12   }
13
14   fclose(fin);
15   return 0;
16 }
```

```c
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 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
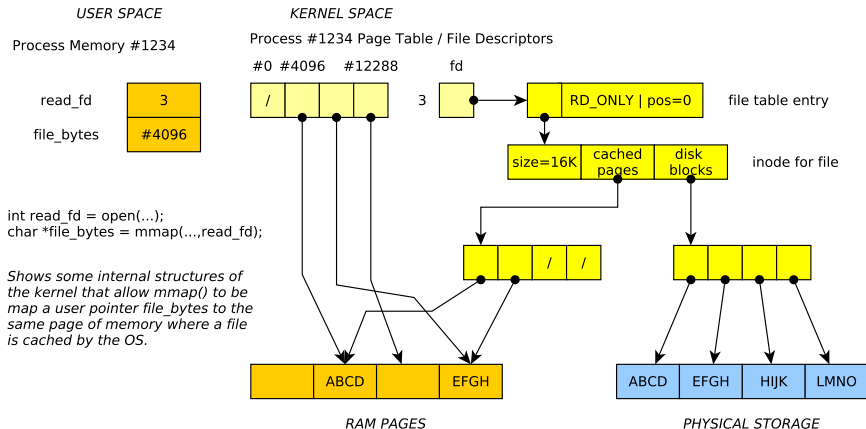
mmap(): Mapping Addresses is Amazing

- ▶ `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()`

USER SPACE

Process Memory #1234

KERNEL SPACE

Process #1234 Page Table / File Descriptors

#0 #4096   #12288   fd

read_fd   3

file_bytes   #4096

/

3

RD_ONLY | pos=0   file table entry

size=16K   cached pages   disk blocks   inode for file

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.

/   /

ABCD   EFGH

ABCD   EFGH   HIJK   LMNO

RAM PAGES

PHYSICAL STORAGE

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

```c
// 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

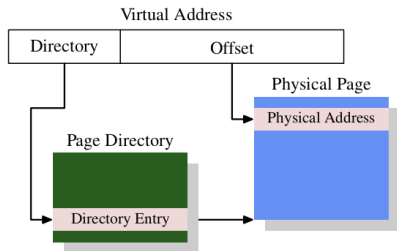# Page Table Size

- 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?

Virtual memory (per process)   Physical memory

Another process's memory

RAM

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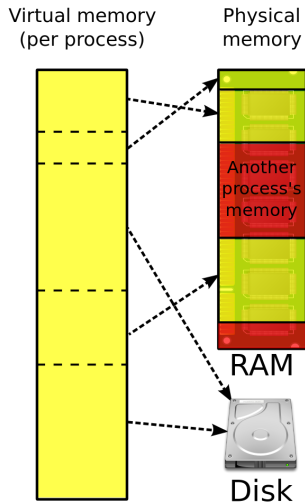
How big does the page table mapping virtual to physical pages need to be?

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
```
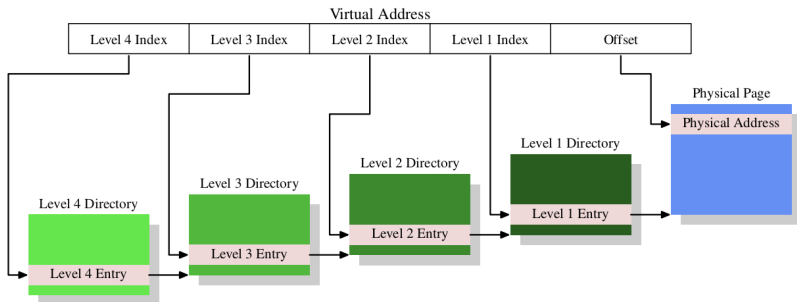
So, 2^{36} virtual pages…

- ▶ 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

Textbook Example: Two-level Page Table

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

32 bit addresses, 4KB pages, 4-byte PTEs
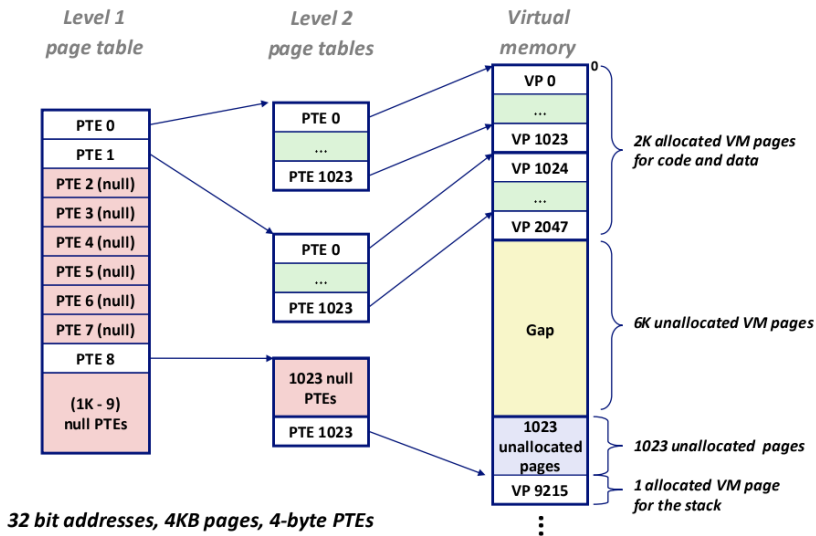
Source: Bryant/O'Hallaron, CSAPP 3rd Ed

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

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
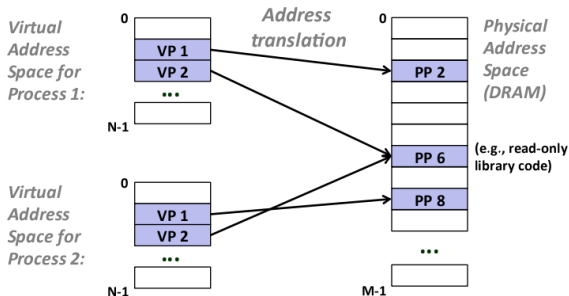
Exercise: Process Memory Image and Libraries

- ▶ 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

Memory Layout (Virtual address space of a C process)

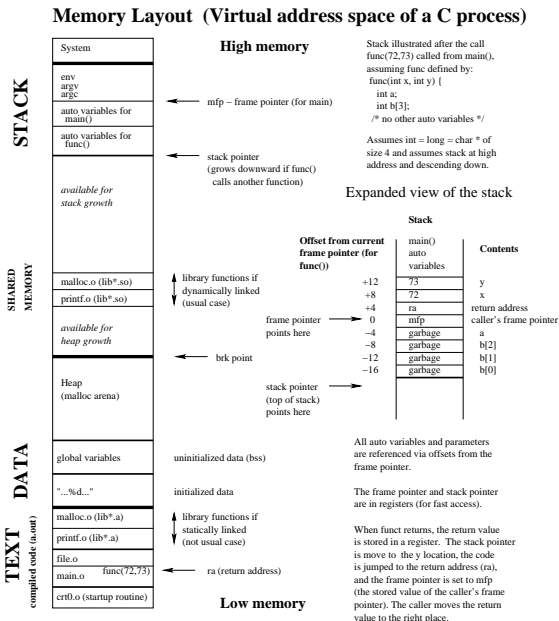|	
System	**High memory**
env argv argc	
auto variables for main()	← mfp – frame pointer (for main)
auto variables for func()	
	← stack pointer (grows downward if func() calls another function)
available for stack growth	
malloc.o (lib*.so)	library functions if dynamically linked (usual case)
printf.o (lib*.so)	
available for heap growth	
	← brk point
Heap (malloc arena)	
global variables	uninitialized data (bss)
"...%d..."	initialized data
malloc.o (lib*.a)	library functions if statically linked (not usual case)
file.o	
main.o func(72,73)	← ra (return address)
crt0.o (startup routine)	
	Low memory

STACK / SHARED MEMORY / DATA / TEXT compiled code (a.out)

Stack illustrated after the call func(72,73) called from main(), assuming func defined by:
```
func(int x, int y) {
    int a;
    int b[3];
    /* no other auto variables */
```
Assumes int = long = char * of size 4 and assumes stack at high address and descending down.

Expanded view of the stack

Stack

Offset from current frame pointer (for func())	main() auto variables	Contents
+12	73	y
+8	72	x
+4	ra	return address
0 ← frame pointer points here	mfp	caller's frame pointer
−4	garbage	a
−8	garbage	b[2]
−12	garbage	b[1]
−16 ← stack pointer (top of stack) points here	garbage	b[0]

All auto variables and parameters are referenced via offsets from the frame pointer.

The frame pointer and stack pointer are in registers (for fast access).

When funct returns, the return value is stored in a register. The stack pointer is move to the y location, the code is jumped to the return address (ra), and the frame pointer is set to mfp (the stored value of the caller's frame pointer). The caller moves the return value to the right place.

30

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
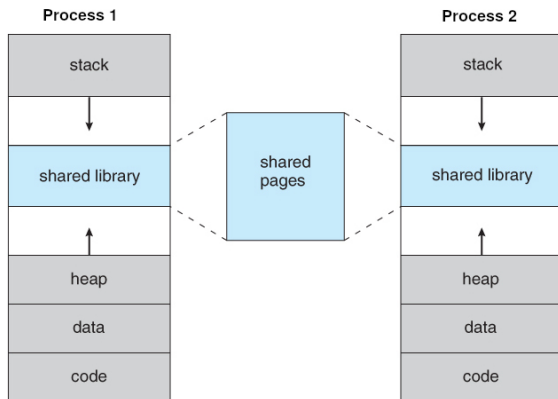
Source: John T. Bell Operating Systems Course Notes

pmap: show virtual address space of running process

```
> ./memory_parts
0x5575555a71e9 : main()
0x5575555aa0c0 : global_arr
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

- ▶ More details of pmap output in this article from Andreas Fester

- ▶ His diagram is awesome

```
> 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