CSCI 4061: Virtual Memory

Chris Kauffman

Last Updated:
Mon Mar 1 03:54:46 PM CST 2021

Logistics

Reading

- ▶ Stevens/Rago Ch 3, 4, 5, 6
    - ▶ Finish up file systems
- ▶ Stevens/Rago: Ch 14.8 on `mmap()`
- ▶ Virtual Memory Reference: Bryant/O'Hallaron, Computer Systems. Ch 9 (CSCI 2021)
    - ▶ Virtual Memory System

Goals

- ▶ Finish up File Ops
- ▶ Virtual Memory System
- ▶ Memory Mapped Files

Reminders

HW05 / Feedback Survey Due Tonight

Assignments

- ▶ Lab06: `stat()` / `lstat()`
- ▶ HW06: Directory traversals

P2 is Coming

The View of Memory Addresses so Far

▶ Consider 2 running programs about to execute the following instructions

```
## Address 1024 hold the `long x=75;`
PROGRAM 1                    PROGRAM 2
## load global from 1024     ## add to global at #1024
movq 1024, %rax              addl %esi, 1024
...                          ... # ^ esi = 5
```
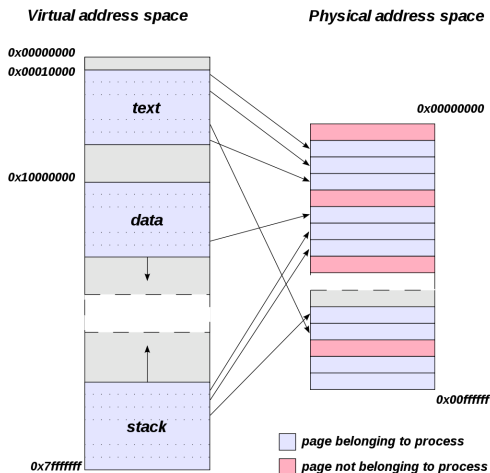
▶ Both Programs accessing physical address 1024 leads to a concurrency problem
  ▶ PROGRAM 1 first: loads 75
  ▶ PROGRAM 2 first: adds on 5, Program 1 loads 85

▶ OS is usually tasked with preventing such problems

▶ Various Hardware/Software techniques have been used over the history of computing but the most popular and important at the moment is
  Virtual Memory: Neither of Program 1/2 directly access Physical Address #1024

Addresses are a Lie

- Operating system uses tables and hardware to translate every memory address reference **on the fly**
- Processes know **virtual addresses** which are translated via the memory subsystem to physical addresses in RAM and on disk
- Hunks of addresses are translated together as **pages**

Virtual address space

0x00000000
0x00010000

text

0x10000000

data

↓

↑

stack

0x7fffffff

Physical address space

0x00000000

0x00ffffff

☐ page belonging to process

☐ page not belonging to process

Source: Wikipdia "Page Table"

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 one with something like $48 - 12 = 36$ bits for page #s
- 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 for Process #1234:
|-----------------+------+----------------------|
| Virtual Page    | Size | Physical Page        |
|-----------------+------+----------------------|
| 00007ffa0997a000 | 4K  | RAM: 0000564955aa1000 |
| 00007ffa0997b000 | 4K  | RAM: 0000321e46937000 |
| ...             |      | ...                  |
|-----------------+------+----------------------|
```

```
Address Space From Page Table for Process #1234:
|-----------------+-------------+------------------|
| 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
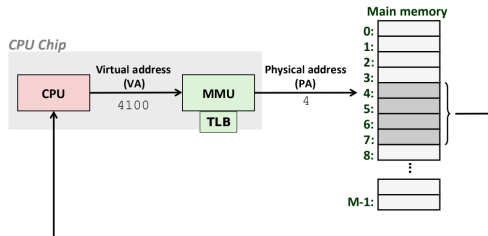
- ▶ OS Kernel interacts with MMU
- ▶ Provides location of the **Page Table**, data structure relating Virtual/Physical Addresses
- ▶ **Page Fault** : MMU couldn't map Virtual to Physical page, runs a Kernel routine to handle the fault

# Translating Virtual Addresses 1/2

- On using a Virtual Memory address, hardware attempts to resolve in MMU via TLB cache of pages
- If valid (hit), address is already in DRAM, translates to physical DRAM address
- Miss = **Page fault**, OS decides..
    1. Page is swapped out, move disk data to DRAM, potentially evicting another page
    2. Not mapped = Segmentation Fault
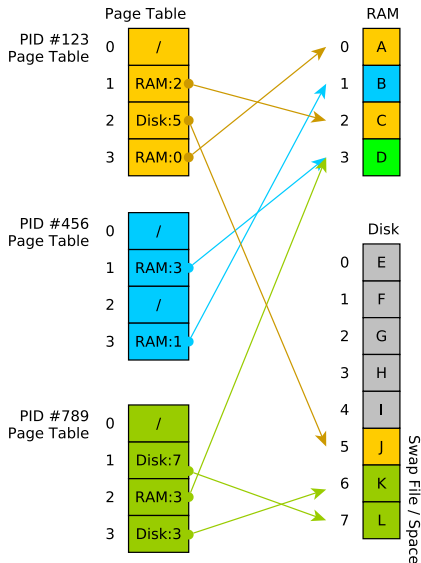
# Translating Virtual Addresses 2/2

- ▶ 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"
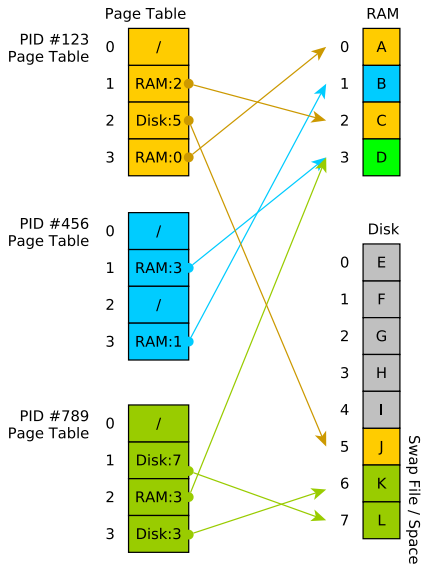
Virtual Memory Caches Physical Memory

- Virtual Memory allows illusion of 2^{48} bytes (hundreds of TBs) of memory when physical memory might only be 2^{30} to 2^{36} (few to hundreds of GBs)
- Disk space is used for space beyond main memory
- Pages that are frequently used stay in DRAM (swapped in)
- Pages that haven't been used for a while end up on disk (**swapped out**)

- DRAM (physical memory) is then thought of as a cache for Virtual Memory which can be as big as disk space allows
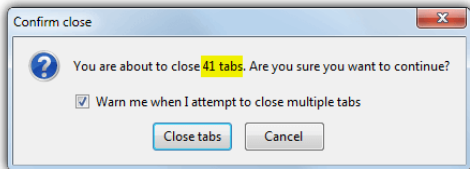
Like when I was writing my composition paper but then got distracted and opened 41 Youtube tabs and when I wanted to write again it took like 5 minutes for Word to load back up because it was swapped out.

The Many Other Advantages of Virtual Memory

▶ Caching: Seen that VirtMem can treat main memory as a cache for larger memory

▶ Security: Translation allows OS to check memory addresses for validity

▶ Debugging: Similar to above, Valgrind checks addresses for validity

▶ Sharing Data: Processes can share data with one another by requesting OS to map virtual addresses to same physical addresses

▶ Sharing Libraries: Can share same program text between programs by mapping address space to same shared library

▶ Convenient I/O: Map internal OS data structures for files to virtual addresses to make working with files free of read()/write()

But first…

Exercise: Page Table Size

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

Virtual memory (per process)

Physical memory

Another process's memory

RAM

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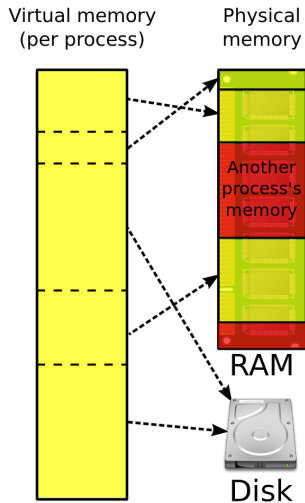
How big does the page table mapping virtual to physical pages need to be?

Answers: Page Table Size

Virtual Address

Page Number	Offset
36 bits	12 bits

Physical Page

Physical Address

Page Table 2^{36} entries
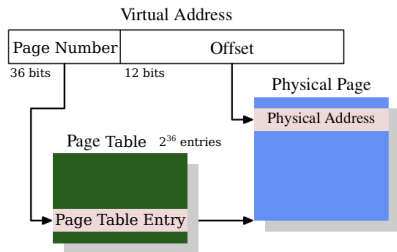
Page Table Entry

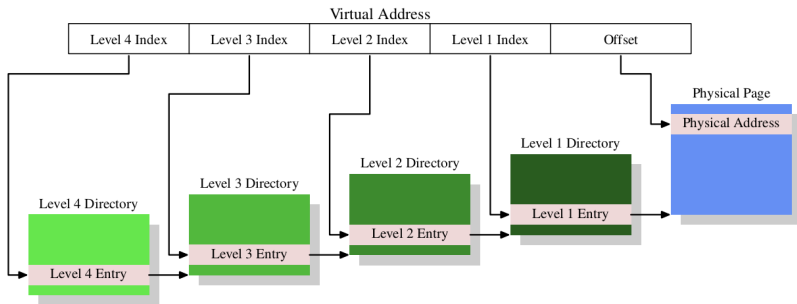"What Every Programmer Should Know About Memory" by Ulrich Drepper, Red Hat, Inc.

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

So, 2^{36} virtual pages…

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

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

Page Tables Usually Have Multiple Levels

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

▶ Fix this absurdity with **multi-level page tables**: a sparse tree
▶ Virtual address divided into sections which indicate which PTE to access at different table levels
▶ 3-4 level page table is common in modern architectures
▶ Many entries in different levels are `NULL` (not mapped) most of the 2^{36} virtual pages are not mapped to a physical page (see next diagram)

Direct Page Table vs Sparse Tree Page Table

Direct Page Table: Array-Like, 5-bit addresses

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

Direct Table
3 pages mapped
32 entries required

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

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.

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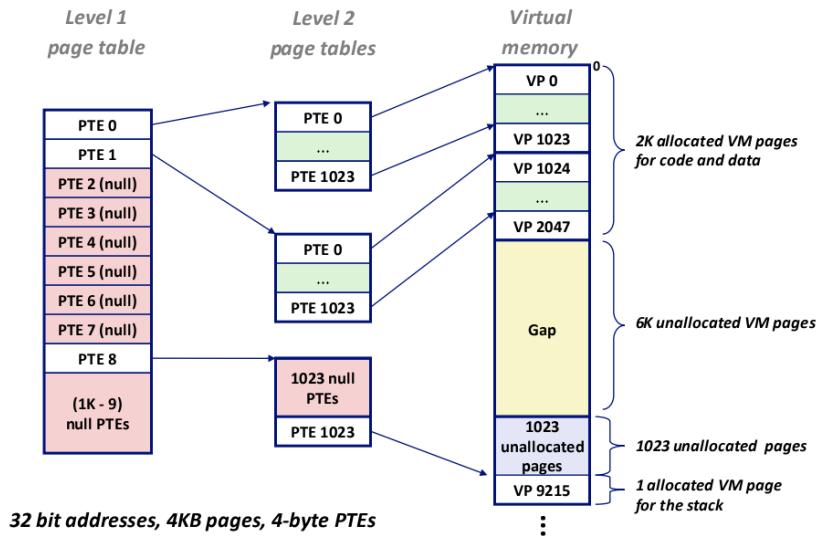
Source: Bryant/O'Hallaron, CSAPP 3rd Ed

Exercise: Printing Contents of file

1. Write a simple program to print all characters in a file **using read() / write()** system calls. What are key features of this program?

2. Examine mmap_print_file.c: does it contain all of these key features? Which ones are missing?

Answers: Printing Contents of file

1. Write a simple program to print all characters in a file. What are key features of this program? **From my_cat.c:**
   - ▶ Open file
   - ▶ Read characters **into memory** buf using read()
   - ▶ Write characters to screen with write()
   - ▶ Repeat this until 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(): Mapping Addresses is Ammazing

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

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

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

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

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

- ▶ Memory mapped files are not just for reading
- ▶ With appropriate options, writing is also possible
  ```
  char *file_chars =
    mmap(NULL, size, PROT_READ | PROT_WRITE,
         MAP_SHARED, fd, 0);
  ```
- ▶ Assign new value to memory, OS writes changes into the file
- ▶ Example: mmap_tr.c to transform one character to another

Mapping things that aren't characters

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

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

```c
// mmap_increment.c

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

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

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

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

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

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

read_fd   3

file_bytes   #4096

```
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.

KERNEL SPACE

Process #1234 Page Table / File Descriptors

#0 #4096   #12288   fd

/

3

RD_ONLY | pos=0   file table entry

size=16K   cached pages   disk blocks   inode for file

/   /

ABCD   EFGH

RAM PAGES

ABCD   EFGH   HIJK   LMNO

PHYSICAL STORAGE

Exercise: `mmap_tr.c`

Speculate on how to use `mmap()` to write the following program.

```
> gcc -o mmap_tr mmap_tr.c
> mmap_tr gettysburg.txt f p
Transforming 'f' to 'p' in gettysburg.txt
Transformation complete

> mmap_tr gettysburg.txt F P
Transforming 'F' to 'P' in gettysburg.txt
Transformation complete

> head gettysburg.txt
Pour score and seven years ago our pathers brought porth on this continent, a
new nation, conceived in Liberty, and dedicated to the proposition that all men
are created equal.

Now we are engaged in a great civil war, testing whether that nation, or any
nation so conceived and so dedicated, can long endure. We are met on a great
battle-pield op that war. We have come to dedicate a portion op that pield, as
pinal resting place por those who here gave their lives that that nation might
live. It is altogether pitting and proper that we should do this.

>
```

Answers in `mmap_tr.c`

mmap() Compared to Traditional read()/write() I/O

Advantages of mmap()

- ▶ Avoid following cycle
    - ▶ read()/fread()/fscanf() file contents into memory
    - ▶ Analyze/Change data
    - ▶ write()/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
- ▶ Memory Map is fixed size while write() and ftruncate() can change the size of a file (though can mremap() once a file changes in size)
- ▶ No bounds checking, just like everything else in C

Malloc Uses `mmap()`

- ▶ `mmap()` system call manipulates the process Page Table
- ▶ `malloc()` is a request for memory from the "Heap" but for large requests, may invoke `mmap()` to map in space directly
- ▶ **Demo**: `strace` to trace system calls on the `max_memory.c`

```
...
# "small" malloc() - no system calls
write(1, "        65536 bytes: Success\n", 28) = 28

# "small" malloc() - no system calls
write(1, "       131072 bytes: Success\n", 28) = 28

# "big" malloc() - use mmap()
mmap(NULL, 266240, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
    = 0x7f52f2ed5000
write(1, "       262144 bytes: Success\n", 28) = 28
munmap(0x7f52f2ed5000, 266240)        = 0

# "big" malloc() - use mmap()
mmap(NULL, 528384, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
    = 0x7f52f2e95000
write(1, "       524288 bytes: Success\n", 28) = 28
munmap(0x7f52f2e95000, 528384)        = 0
...
```

One Page Table Per Process

- ▶ OS maintains a page table for each running 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 O   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 O 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

*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.*

Shared Memory Calls: Modern Posix

- ▶ Using OS system calls, can usually create shared memory
- ▶ Modern POSIX systems favor mmap()/fork() for this
- ▶ Will cover more **interprocess communication (IPC)** later

```
char *shared_name = "/something_shared";
int SHM_SIZE = 1024
// global params for shared memory segment

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

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

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

int child_pid = fork();
// child/parent share array shared_bytes
```

Shared Memory Calls: Historical System V

- Using OS system calls, can usually create shared memory
- Unix System V (five) IPC includes the following

```
key_t key  = ftok("crap", 'R');
// make the SysV IPC key

int shmid  = shmget(key, 1024, 0644 | IPC_CREAT);
// connect to (and possibly create) the segment:

char *data = shmat(shmid, (void *)0, 0);
// attach to the segment to get a pointer to it:
```

- Multiple processes can all "see" the same unit of memory
- This is an old style but still useful
- Will cover more **interprocess communication (IPC)** later
- Modern incarnations favor mmap() followed by fork()

Exercise: What can be shared safely?

Traditional Unix Processes didn't share anything in **user memory** with each other to preserve security: processes are isolated.

1. Why is this wasteful of Memory?
2. What items in user memory could be shared safely?
3. What is shared between processes **outside of user space?**

Answers: What can be shared safely?

Traditional Unix Processes didn't share anything in **user memory** with each other to preserve security: processes are isolated.

1. Why is this wasteful of Memory?
   Every program has its own identical copies of printf(), malloc(), etc. Lots of copies of the same stuff sitting in MANY processes.

2. What items in user memory could be shared safely?
   The Text (Code) and other Read-only data for libraries can be shared: every program executes them, no program changes them.
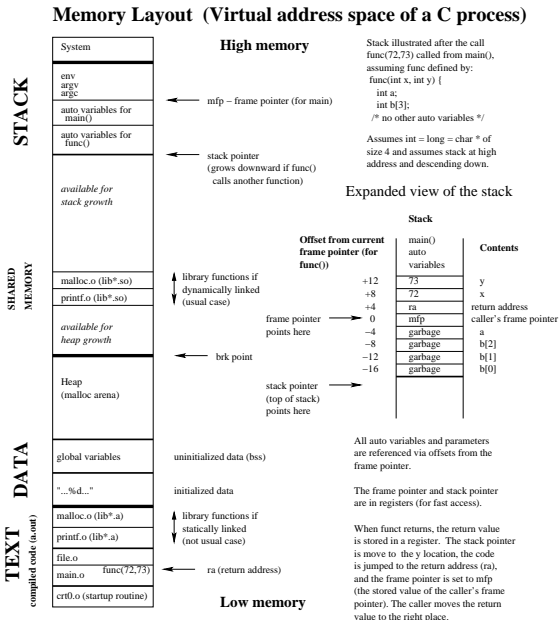
3. What is shared between processes **outside of user space?**
   Kernel data is shared by all processes; some parent/child processes directly share file table entries, all access same kernel structures

Process Memory Image and Libraries

Most programs on the system need to use malloc() and printf(): **shared library** versions of these reduce memory footprint w/o being dangerous

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

Memory Layout (Virtual address space of a C process)

STACK	System	**High memory**	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 */
	env argv argc		Assumes int = long = char * of size 4 and assumes stack at high address and descending down.
	auto variables for main()	← mfp – frame pointer (for main)	
	auto variables for func()		
		← stack pointer (grows downward if func() calls another function)	Expanded view of the stack
	available for stack growth		

Stack

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

SHARED MEMORY	malloc.o (lib*.so)	↕ library functions if dynamically linked (usual case)	
	printf.o (lib*.so)		
	available for heap growth		stack pointer (top of stack) points here
		← brk point	
	Heap (malloc arena)		All auto variables and parameters are referenced via offsets from the frame pointer.
DATA	global variables	uninitialized data (bss)	The frame pointer and stack pointer are in registers (for fast access).
	"...%d..."	initialized data	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.
TEXT compiled code (a.out)	malloc.o (lib*.a) printf.o (lib*.a)	↕ library functions if statically linked (not usual case)	
	file.o		
	main.o func(72,73)		
	crt0.o (startup routine)	← ra (return address)	
		Low memory	

32

Shared Libraries: *.so Files

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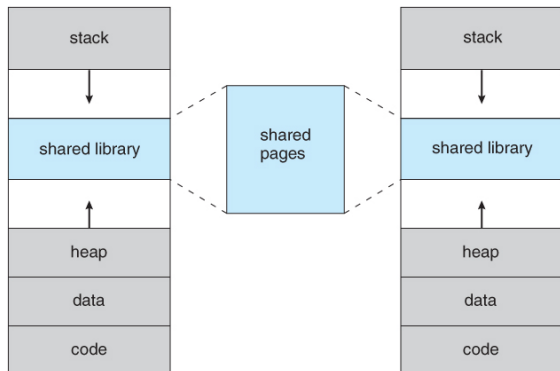
Source: John T. Bell Operating Systems Course Notes

pmap: show virtual address space of running process

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

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

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

Memory Protection

- ▶ Output of `pmap` indicates another feature of virtual memory: protection
- ▶ OS marks pages of memory with Read/Write/Execute/Share permissions
- ▶ 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

Fork and Shared Pages

- `fork()`'ing a process creates a nearly identical copy of a process
- Might need to copy all memory form parent to child pages
- Can save a lot of time if memory pages of child process are **shared with parent** - no copying needed (initially)
- What's the major danger here?
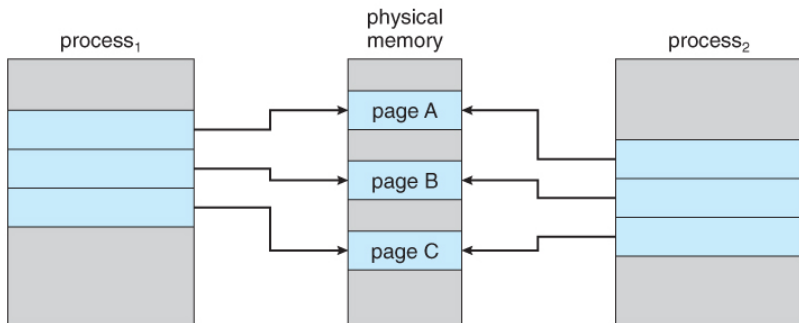
Source: John T. Bell Operating Systems Course Notes

Fork, Shared Pages, Copy on Write (COW Pages)

- ▶ If neither process writes to the page, sharing doesn't matter
- ▶ If either process writes, OS will make a copy and remap addresses to copy so it is exclusive
- ▶ Fast if hardware Memory Management Unit and OS know what they are doing (Linux + Parallel Python/R + Big Data)
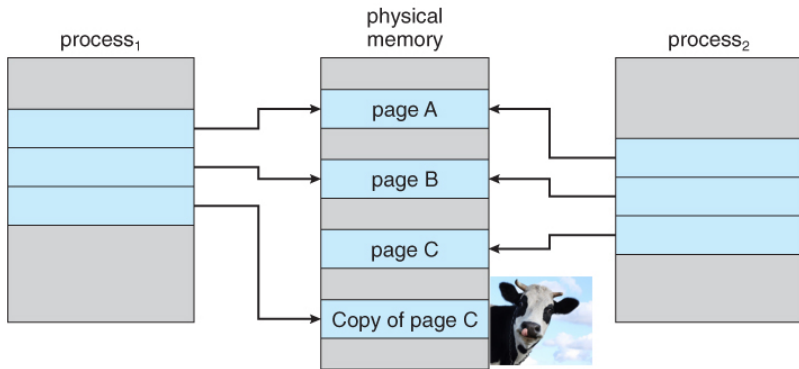
Source: John T. Bell Operating Systems Course Notes

Summary

A computer using a Virtual Memory system sees the OS and hardware cooperate to translate every program address from a virtual address to a physical RAM address.

- **Consequence 1**: All programs see a linear address space but each has its own #1024 which does not conflict

- **Consequence 2**: The OS must maintain Page Table data structures that map each process's virtual addresses to physical locations

- **Consequence 3**: Computers with small amounts of RAM can "fake" larger amounts by using disk space; RAM serves as a cache for this larger virtual memory

- **Consequence 4**: Every address deference is translated so the OS can catch out of bounds references or direct the dereference to special RAM areas like memory mapped files

- **Consequence 5**: Memory can shared and manipulated for efficiency such as fork() using Copy on Write

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 on onto 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