Logistics

Reading Bryant/O’Hallaron

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

Goals

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

P5

▶ 1 Required Problem
▶ Post Monday

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>4/23 Fri</td>
<td>HW 13 Virt Mem</td>
</tr>
<tr>
<td></td>
<td>Virtual Mem 1/2</td>
</tr>
<tr>
<td>4/23 Mon</td>
<td>Virtual Mem 2/2</td>
</tr>
<tr>
<td></td>
<td>P5 Release</td>
</tr>
<tr>
<td>4/26 Wed</td>
<td>ELF Files/Linking 1/2</td>
</tr>
<tr>
<td></td>
<td>Lab 14 VirtMem</td>
</tr>
<tr>
<td></td>
<td>HW 14 Linking</td>
</tr>
<tr>
<td>4/28 Fri</td>
<td>Obj Code/Linking 2/2</td>
</tr>
<tr>
<td>5/03 Mon</td>
<td>Last Lecture, Review</td>
</tr>
<tr>
<td></td>
<td>P5 Due</td>
</tr>
<tr>
<td>5/08 Sat</td>
<td>Final Exam Opens 1:30pm</td>
</tr>
<tr>
<td>5/12 Wed</td>
<td>Final Exam Closes 3:30pm</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>ORDER A: Program 1 loads first</th>
<th>ORDER B: Program 2 adds first</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM 1</td>
<td>PROGRAM 2</td>
</tr>
<tr>
<td>movq 1024, %rax</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>addl %esi, 1024</td>
</tr>
<tr>
<td>addl %esi, 1024</td>
<td>movq 1024, %rax</td>
</tr>
</tbody>
</table>

- **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 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:

<table>
<thead>
<tr>
<th>Virtual Page</th>
<th>Size</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>00007ffa0997a000</td>
<td>4K</td>
<td>RAM: 0000564955aa1000</td>
</tr>
<tr>
<td>00007ffa0997b000</td>
<td>4K</td>
<td>RAM: 0000321e46937000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Address Space From Page Table:

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Page Offset</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00007ffa0997a000</td>
<td>0</td>
<td>0000564955aa1000</td>
</tr>
<tr>
<td>00007ffa0997a001</td>
<td>1</td>
<td>0000564955aa1001</td>
</tr>
<tr>
<td>00007ffa0997a002</td>
<td>2</td>
<td>0000564955aa1002</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>00007ffa0997afff</td>
<td>4095</td>
<td>0000564955aa1fff</td>
</tr>
<tr>
<td>00007ffa0997b000</td>
<td>0</td>
<td>0000321e46937000</td>
</tr>
<tr>
<td>00007ffa0997b001</td>
<td>1</td>
<td>0000321e46937001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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

*Diagram:*

- 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, MMU will search TLB for physical DRAM address,
- If found in TLB, Hit, use physical DRAM address
- If not found, MMU will searches 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”
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.
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

▶ 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()
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

- 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

"What Every Programmer Should Know About Memory" by Ulrich Drepper, Red Hat, Inc.
Direct Page Table vs Sparse Tree Page Table

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

<table>
<thead>
<tr>
<th>VP#</th>
<th>Valid</th>
<th>PP#</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>00000</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>00001</td>
<td>654</td>
</tr>
<tr>
<td>0010</td>
<td>1 01001</td>
<td>00010</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>00011</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td>00100</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
<td>00101</td>
<td>-</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>00110</td>
<td>-</td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td>00111</td>
<td>-</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>01000</td>
<td>-</td>
</tr>
<tr>
<td>1001</td>
<td>0</td>
<td>01001</td>
<td>987</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>11011</td>
<td>-</td>
</tr>
<tr>
<td>1110</td>
<td>1 00001</td>
<td>11100</td>
<td>321</td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
<td>11101</td>
<td>-</td>
</tr>
<tr>
<td>11110</td>
<td>1 11100</td>
<td>11110</td>
<td>-</td>
</tr>
<tr>
<td>11111</td>
<td>0</td>
<td>11111</td>
<td>-</td>
</tr>
</tbody>
</table>

**Physical Memory**

<table>
<thead>
<tr>
<th>VP#</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>654</td>
</tr>
<tr>
<td>0010</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
</tr>
<tr>
<td>0110</td>
<td>-</td>
</tr>
<tr>
<td>0111</td>
<td>-</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
</tr>
<tr>
<td>1001</td>
<td>987</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1101</td>
<td>-</td>
</tr>
<tr>
<td>1110</td>
<td>321</td>
</tr>
<tr>
<td>1111</td>
<td>-</td>
</tr>
</tbody>
</table>

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

**Physical Memory**

<table>
<thead>
<tr>
<th>VP#</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>654</td>
</tr>
<tr>
<td>0010</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
</tr>
<tr>
<td>0110</td>
<td>-</td>
</tr>
<tr>
<td>0111</td>
<td>-</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
</tr>
<tr>
<td>1001</td>
<td>987</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1101</td>
<td>-</td>
</tr>
<tr>
<td>1110</td>
<td>321</td>
</tr>
<tr>
<td>1111</td>
<td>-</td>
</tr>
</tbody>
</table>

**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
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
- We will use `mmap()` for 2 specific purposes
  1. A convenient way to interact with files via **Memory Mapped Files** (in lecture/lab)
  2. Implement our own `malloc()` / `free()` system (Project 5)
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
// print_file.c
int main(int argc, char *argv[]){
    FILE *fin = fopen(argv[1], "r");
    char inchar;
    while(1){
        int result =
            fscanf(fin, "%c", &inchar);
        if(result == EOF){
            break;
        }
        printf("%c", inchar);
    }
    fclose(fin);
    return 0;
}
```

```c
// mmap_print_file.c
int main(int argc, char *argv[]){
    int fd = open(argv[1], O_RDONLY);
    struct stat stat_buf;
    fstat(fd, &stat_buf);
    int size = stat_buf.st_size;
    char *file_chars =
        mmap(NULL, size,
            PROT_READ, MAP_SHARED,
            fd, 0);
    for(int i=0; i<size; i++){
        printf("%c", file_chars[i]);
    }
    printf("

    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 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() is a system call that maps a file's contents into a process's address space. It takes a file descriptor (fd) and a size, and returns a pointer (ptr) to the mapped file. The function call looks like this:

```c
int fd = open("gettysburg.txt", O_RDONLY);
char *file_chars = mmap(NULL, size, PROT_READ, MAP_SHARED, fd, 0);
```

This code snippet demonstrates how to open a file and map it into memory using mmap(). The file is opened in read-only mode with the `open()` system call:

```c
int fd = open("gettysburg.txt", O_RDONLY);
```

Then, a pointer to the file's bytes is mapped into memory using `mmap()`, allowing the program to access the file's contents directly:

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

With the pointer to the mapped file, you can access individual characters:

```c
printf("%c",file_chars[0]); // print 0th file char
printf("%c",file_chars[5]); // print 5th file char
```
mmap() allows file reads/writes without read()/write()

- Memory mapped files are not just for reading
- With appropriate options, writing is also possible
  ```c
  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

// 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
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

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)

<table>
<thead>
<tr>
<th>Stack</th>
<th>High memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfp – frame pointer (for main)</td>
<td></td>
</tr>
<tr>
<td>stack pointer (grows downward if <code>func()</code> calls another function)</td>
<td></td>
</tr>
</tbody>
</table>

Expanded view of the stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Offset from current frame pointer (for `func())</th>
</tr>
</thead>
<tbody>
<tr>
<td>main() auto variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+12</td>
</tr>
<tr>
<td></td>
<td>+8</td>
</tr>
<tr>
<td></td>
<td>+4</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td>-8</td>
</tr>
<tr>
<td></td>
<td>-12</td>
</tr>
<tr>
<td></td>
<td>-16</td>
</tr>
<tr>
<td>Stack</td>
<td>Contents</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>return address</td>
</tr>
<tr>
<td>ra</td>
<td>caller’s frame pointer</td>
</tr>
<tr>
<td>mfp</td>
<td>a</td>
</tr>
<tr>
<td>garbage</td>
<td>b[2]</td>
</tr>
<tr>
<td>garbage</td>
<td>b[1]</td>
</tr>
<tr>
<td>garbage</td>
<td>b[0]</td>
</tr>
</tbody>
</table>

Low memory

- `malloc()` and `printf()` library functions if dynamically linked (usual case)
- `malloc()` and `printf()` library functions if statically linked (not usual case)

Uninitialized data (bss)

- `...%d...` initializers
- `malloc.o (lib*.a)`
- `printf.o (lib*.a)`
- `printf.o (lib*.so)`
- `malloc.o (lib*.so)`

Initialized data

- `main.o` (not usually linked)
- `crt0.o (startup routine)`
- `func(72,73)`

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)

- `main()` auto variables
- `main() auto variables`
- `func(72,73)` auto variables
- `func(72,73)` parameters

Uninitialized data (bss)
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
0x600000000000 : mmap'd block1
0x6000000001000 : mmap'd block2
0x7f2244dc44000 : mmap'd file
0x7fff0133b70 : 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
00007fff0114000 132K rw--- [ stack ] STACK
00007fff014d000 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 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