15213 Lecture 17: Virtual Memory Concepts

Learning Objectives

- Understand the distinction between virtual and physical addresses, and which are visible to processes.
- Describe page faults and give at least two situations where one might occur.
- Compare and contrast pages with cache lines.
- Identify what the OS does to programs that perform illegal memory accesses.

Getting Started

The directions for today's activity are on this sheet, but refer to accompanying programs that you'll need to download. To get set up, run these commands on a shark machine:

1. `$ wget http://www.cs.cmu.edu/~213/activities/lec17.tar`
2. `$ tar xf lec17.tar`
3. `$ cd lec17`

1 Memory Addresses: A Lie

Start by examining the `addrs.c` file using your editor of choice or the `cat` command. Notice that the program prints example addresses from a number of its sections, then forks a child process that does the same. Once you're comfortable with what the program is doing, build (`$ make addrs`) and run (`$ ./addrs`) it.

1. What do you notice about the addresses printed by the two processes?

2. Do you think the processes share the same memory? Explain why this either must be or cannot be the case.

3. Now consider the `large.c` program, which performs a number of 1-GB memory allocations. Building and running the program, do you notice anything about its output?

2 Memory Addresses: Timings

We just saw that memory addresses are *remappable*, and that the system has been silently managing the mappings for us all along! Let's try to observe the performance overhead of this management. Examine the `timings.c` program, which allocates zero-initialized 100-KB memory regions using two different approaches, then performs a series of writes to each. When ready, build and run it.

4. Both the `calloc()` and `mmap()` library calls allocate a block of zero-initialized memory. Which *call* takes less time?

5. Which memory region exhibits faster initial *access* times? What does this suggest about how much setup work each of the allocation calls does?

6. Do you suspect that cache misses account for the access time difference? Why or why not?

3 Virtual Memory: Page Faults

The memory addresses we've always worked with are known as **virtual memory addresses**, and usually differ from their corresponding **physical memory address** in DRAM. Any time the system attempts to translate a virtual address, there's a possibility that no valid mapping is present: such a situation is called a **page fault**, and handling it is one of the operating system's main jobs. One common OS response is to simply map the address to some unused portion of physical memory, then allow the program to retry; we'll focus on this for now and look at the other possibilities later.

Unix exposes counters that reveal how many page faults the OS has handled behind the scenes. Take a look at `faults.c`, a slightly modified version of the last program that uses a helper function from `benchmark.h` to report these counts. Build and run the program when you're ready.

7. Which allocation *call* results in more page faults? Which memory region incurs more page faults upon initial *access*? Now compare against the results from the previous section.

4 Virtual Memory: Address Translation

The difference between the two regions' page fault patterns is the result of an approach known as **demand paging**, whereby the OS defers mapping virtual addresses until they are first accessed (faulted). Whereas `mmap()` merely informs the OS that the locations may be accessed at some point, `calloc()` goes further by **prefaulting** all requested pages before it returns[1][2].

Much like the CPU cache, the virtual memory system breaks memory into a huge number of fixed-sized regions analogous to blocks; however, this time they are called **pages**. Also like cache, a portion of the address (the **page number**) determines which page it occupies, and a different

[1] Technically, `calloc()` performs an additional task before returning: The OS only zeros a page if a page fault actually occurs. However, because `calloc()` allocates from the process's heap, it sometimes reuses pages that the process has previously faulted (e.g., allocated then `free()`'d). Thus, `calloc()` must explicitly zero such pages.

[2] As an optimization, the GNU versions of `malloc()` and `calloc()` perform allocations larger than a certain size threshold (typically a little larger than the 100 KB we allocated) using `mmap()` instead of allocating heap space.

portion (the **page offset**) describes the location within that page. Thanks to demand paging, we can reveal the system page size using `mmap()` by observing `when` within our traversal loop the page faults occur. Familiarize yourself with `bounds.c`, a program that does this, then build and run it.

8. Looking at the output, how large is each page? How does this compare to the size of a cache line (64 B)?

9. Given the following depiction of a virtual memory address, draw two vertical lines to split it into the *page number*, *page offset*, and *unused bits*, labeling each part and the bits between.

63	0

5 Virtual Memory: Program Misbehavior

Of course, sometimes an application is wrong to attempt a certain memory access. For instance, consider the simple program `invalid.c`.

10. What happens when you run it? Do you think the OS is able to detect *all* out-of-bounds memory accesses to handle them this way? If so, why? If not, which ones go undetected?

11. How do you think the OS decided to handle the situation in this way? *(Hint: think about how this access differs from those of the previous program, which* also *used unmapped pages.)*

6 Virtual Memory: Protection

The hardware stores some metadata for each mapped page. One important such entry is a **valid bit** to indicate whether that page is mapped; however, there are also **protection bits** that control which operations are valid on that page, useful for preventing some of the attacks you saw earlier in the course. Examine `protected.c`, a program that allocates some memory, reads from it, copies a function into it, and tries to execute the code from its new location. Try modifying the memory mapping by invoking it once with each of the arguments `""`, `r`, `rw`, and `rwx` (one at a time).

12. Some of the accesses cause the program to crash. Compare and contrast this category of access with the others we've seen by indicating what does (not) happen in each case:

Property	**Valid access**		**Invalid access**	
	calloc()'d	**mmap()'d**	**Unallocated page**	**Protection bits**
Page fault?				
OS maps page?				
Segfault?				
________________				
________________				