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

Examine 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 the program, build (`$ make addrs`) and run (`$ ./addrs`) it.

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

2. Do you think the processes share the same memory? Explain why this either must be or cannot be the case.
   The matching addresses suggest they would. But wait: that can't be true, because we know that they have different heaps (otherwise, this program would double-`free()` its allocation) and stacks (otherwise, they'd run into problems when they tried to call different functions at different times)!
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?
   The failure happens after allocating about 65 thousand GB, while the system only has 23 GB of main memory.

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? `calloc()` `mmap()`

5. Which memory region exhibits faster initial *access* times? What does this suggest about how much setup work each of the allocation calls does?
   `calloc()` exhibits faster initial access times, suggesting that `calloc()` does more setup than `mmap()`.

6. Do you suspect that cache misses alone account for the access time difference? Why or why not?
   This is an unlikely explanation: the access patterns are identical. We're only doing a single traversal, so conflict misses shouldn't affect us and any alignment difference will have no effect. You might also notice that the prefetcher should predict these accesses and eliminate even cold misses.
7. Overall, which of the library interfaces was faster to use?
   `mmap()`, but only by 20-30 microseconds.

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.

8. 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.
   `mmap()` incurs more page faults on initial access, which is consistent with `mmap` taking much longer in initial accesses in the previous section.

4 Virtual Memory: Anatomy of an Address

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 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.

[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.

9. Looking at the output, how large is each page? How does this compare to the size of a cache line (64 B)?
Each page is $2^{12} = 4096$ bytes, which is $2^6 = 64$ cache lines.

10. 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	36 page number bits, 12 page offset bits	0

5 Virtual Memory: Program Misbehavior

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

11. What happens when you run it? Which array index is the problem?
It crashes when trying to access element 8192 of the array. Note that this is the element just past the end of the allocation!

12. Thinking about the address of this array element, what distinguishes it from all the others?
(Hint: `mmap()` always returns page-aligned allocations.)
Like elements 0 and 4096, this "element" is located at the start of a page. Unlike them, its address is outside the range communicated to the operating system via the `mmap()` call.

13. As you've probably already experienced, the OS cannot always detect out-of-bounds memory accesses. In fact, there are at least two trivial changes to this program that would cause it to (apparently) run normally without actually fixing the bug; can you think of them?
The OS only detected the problem because the access caused a page fault, so our task is to mask the problem by getting the out-of-bounds address to reside on a mapped, accessible page. One way to do this is to decrease the array size (e.g., to 8191 bytes); another is to switch to allocating via `malloc()`/`free()`, in which case the immediately following byte will be part of the heap!

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).

14. Now that you've seen the reasons the OS might have to intervene in the middle of a memory access, complete this summary table by marking which categories of memory access (*not* allocation call) cause each of the listed outcomes. The first row has been done for you.

Valid access		**Invalid access**		**Outcome**
calloc()'d	**mmap()'d**	**Unallocated page**	**Protection bits**	
		x	x	Segfault
	x	x	x	Page fault
x				No OS involvement
	x			OS maps page