Learning Objectives

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

Getting Started

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

```
$ wget http://www.cs.cmu.edu/~213/activities/vm-concepts.tar
$ tar xf vm-concepts.tar
$ cd vm-concepts
```

1 Memory Addresses: A Lie

Examine the file `addrs.c` using your editor of choice, or the `less` command.

Notice that this program prints the addresses of several different kinds of variables and one function. Then it uses the `fork` system call to create a new process, which also prints the addresses of all the same things. Finally it waits for the child process to finish, using the `waitpid` system call. If you aren't familiar with these functions, you can use the `man` command to read up on what they do: type `man fork` to display the documentation for `fork`. (This works for *any* C library function.)

Once you're comfortable with the program, compile it and run it:

```
$ make addrs
$ ./addrs
```

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

The addresses printed by the two processes are the same.

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

The addresses are the same, so it *seems* like they share memory. But that can't be right, because the `heap` allocation is freed in both processes, the variable `child` has different values in each, and they are making different function calls so they must have two different stacks.

Now read the program `large.c`. It uses the `sysinfo` system call to report how much RAM the computer has, and then it uses the `mmap` system call to allocate as much of this memory as possible. It can, optionally, read from or write to each "page" of each allocation.

Problem 3. Compile this program (`make large`) and run it with no arguments (`./large`). Do you notice something odd happening?

`sysinfo` says the computer has a few dozen gigabytes of main memory, but `mmap` only fails after allocating a much larger amount of memory.

Problem 4. Now run this program in "read" mode (`./large read`). What happens? How is that different from what happened before?

The program runs slower in "read" mode. `mmap` fails after allocating less memory than before, but still quite a bit more than the amount of memory the computer has. (Exactly when it fails will vary from run to run.)

Problem 5. Finally, run this program in "write" mode (`./large write`). What happens? How is that different from what happened before?

The program runs much slower in "write" mode. It gets slower and slower as it allocates more and more memory. And `mmap` fails after allocating somewhat less than the amount of memory the computer has. (Exactly when it fails will vary from run to run.)

Problem 6. What do you think is going on here?

This is called *memory overcommitment*. The operating system pretends to give `large` more memory than the computer actually has. This illusion works best when `large` doesn't access the memory it allocated. If it reads from the memory, the illusion doesn't work as well, and if it writes to the memory, the illusion completely falls apart.

2 Memory Addresses: Timings

We just saw that a memory address in one process usually refers to a *different byte of RAM* than the same memory address in a different process. This is called *virtual addressing*. It's an essential safety feature—it guarantees that one program cannot scribble on another program's memory!

We also just saw that the system calls that allocate RAM (`sbrk` and `mmap`) might be "lazy" and not actually allocate any memory until it is used. This means the operating system can tell when freshly-allocated memory is first used.

These features have some performance overhead. Let's try to measure this overhead. Examine the program `timings.c`, which allocates zero-initialized 100 KB memory regions using two different approaches, then performs a series of writes to each, and measures the time these operations take using a helper function from `benchmark.h`. When you are ready, build and run it.

Problem 7. Both `calloc` and `mmap` allocate a block of zero-initialized memory. Which *call* takes less time?

`mmap`

Problem 8. Which memory region is faster for the application to access, the *first* time it does this?

The `calloc` region

Problem 9. Which memory region is faster for the application to access, the *second* time it does this?

Neither is faster

Problem 10. What do these things imply about the work being done by `calloc` versus `mmap`?

`calloc` is doing more work before it returns, but that work somehow prepares the memory for use by the application, so its first accesses to the memory are faster. However, this only affects the very first access.

Problem 11. Could cache misses alone account for this time difference?

This is an unlikely explanation. We are only accessing one block at a time, in exactly the same pattern, so we should see the same pattern of cache misses for both.

3 Virtual Memory: Page Faults

The memory addresses we've always worked with are known as *virtual addresses*. As we have seen, these are different from the *physical addresses* that actually index the RAM. The operating system maintains a mapping from virtual to physical addresses for each process.

This mapping does not have to be complete. Some virtual addresses might not correspond to any physical address. If a program tries to access a virtual address that has no corresponding physical address, the CPU will trigger a special mechanism called a *page fault* that allows the operating system to intervene. (We'll talk more about this mechanism on Thursday.) One of the things the OS can do when a page fault happens is *create* a mapping for the virtual address and then allow the program to retry the memory access.

Unix allows programs to monitor how many times page faults have happened, and new mappings have been created, because of their memory accesses.

Read `faults.c`, a slightly modified version of `timings.c`. It reports page faults instead of timing, using a different helper function from `benchmark.h`. Build and run the program when you're ready.

Problem 12. Which allocation *call* results in more page faults?

`calloc`

Problem 13. Which memory region incurs more page faults upon initial *access*?

The `mmap` region

Problem 14. Compare the numbers you get from `faults` with the numbers you get from `timings`. Do the different numbers of page faults explain the different timings?

Yes. No page faults occur during the `mmap` call, and several occur during the `calloc` call. In the first loop accessing memory, no page faults occur for the `calloc` region but several page faults occur for the `mmap` region—exactly the same number as occurred during the `calloc` call. This is exactly what we were looking for: work being shifted from the first application use of memory, into the `calloc` call.

4 Virtual Memory: Anatomy of an Address

The phenomenon we have just observed—page faults occurring when freshly allocated memory is accessed—is a deliberate operating system design decision known as *demand paging*. Operating systems that implement demand paging wait until freshly allocated memory is first accessed to define a virtual-to-physical mapping for it. This allows them to avoid defining mappings for any memory that is not actually used. Many programs (such as `large.c`) allocate far more memory than they use; demand paging means they only consume the physical RAM they really need.

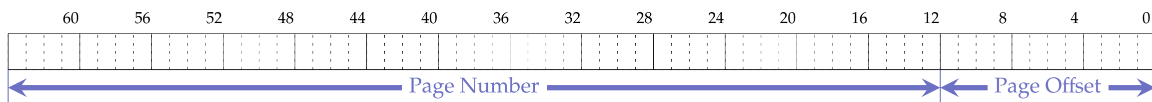
We keep using the term "page." A page is a fixed-size set of virtual addresses that are all mapped to physical addresses as a unit. This is similar to how the memory cache divides memory into blocks. Just like how addresses are divided into cache tag, set index, and block offset, they are also divided into a *page number* and a *page offset*.[1] The page number says which page the address belongs to, and therefore which region of physical RAM it is mapped to. The page offset indicates a specific byte within that page.

Because the operating system on the sharks uses demand paging, if we use `mmap` to allocate some memory, and then access it byte by byte and watch for page faults, we can discover the size of a page, and therefore how many bits of an address are the page offset. Familiarize yourself with `bounds.c`, a program that does this, then build and run it.

Problem 15. 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.

Problem 16. Below is a diagram of the 64 bits of a virtual memory address. Bit 0 is the least significant bit. Label this diagram to show which bits are part of the *page offset* and which are part of the *page number*.



[1] The virtual memory system does not have any equivalent of a set index. As we will see later on, this is because the virtual memory system can be thought of as a *fully associative* cache.

5 Virtual Memory: Program Misbehavior

Sometimes programs access memory addresses they shouldn't. For instance, consider the program `invalid.c`, which has a bug. Compile it and run it—but run it under `gdb`.

Problem 17. What happens when you run it? Which array index is the problem?

It crashes when trying to access element 8192 of the array. This is the element just past the end of the allocation.

Problem 18. What is special about the address of this array (non-)element? *(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 we allocated with `mmap`.

Problem 19. As you've probably already experienced, the OS cannot always detect out-of-bounds memory accesses. There are at least two ways you could change this program that would make it *appear* to run normally but would not actually fix the bug. Can you think of them?

The OS can only detect out-of-bounds memory accesses if they cause page faults. To put it another way, `mmap` always allocates whole pages. If we decrease `LENGTH` to 8191, then `array[LENGTH]` is still outside the bounds of the array, but *within* the page that `mmap` allocated to satisfy the request, and the program will not crash.

You could also allocate the array with `malloc` instead. Then the byte immediately after the allocation will (probably) still be included in the heap!

6 Virtual Memory: Protection

Even if a page of memory is mapped, the program might not be allowed to do just anything to that memory. The hardware maintains *protection bits* for each mapped page which define what operations are allowed on that page. You may have noticed that all the example programs so far passed an argument `PROT_READ|PROT_WRITE` to `mmap`—this asks for memory that is both readable and writable. There are several other possibilities.

Examine the program `protected.c`. It allocates some memory, reads from it, copies a function into it, and tries to execute the code from its new location. You can control how it allocates the memory with command line arguments. Try running it with each of these arguments (one at a time): `""`, `r`, `rw`, and `rwx`.

Problem 20. 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