Computer Systems
Project 6
Implenting a simulated MMU

1 Simulating virtual memory

Virtual memory is typically implemented within the operating system kernel (which maintains page tables and handles page faults), and with the support of the *memory management unit (MMU)* in hardware (which uses the page tables to translate virtual addresses to physical addresses). We want to experiment with the implementation of this abstraction, but modifying real kernel code is fraught—its code is large and deeply complex, and any errors in the kernel can be exceedingly difficult to diagnose. Therefore, working within the kernel is not desirable for our first encounter with virtual memory mapping.

Instead, we will use a *simulated* virtual memory abstraction. We will use programs that allocate and use addresses that are virtual, and mapped to different addresses within a region of the program's memory. **Our project will be to write a simulated MMU** that performs the translation—the mapping—of those addresses.

1.1 *Simulated* and *real* spaces

In order to avoid confusion (and perhaps risking the creation of more confusion), we will not use the terms *virtual* and *physical* to describe the memory that we are managing in this project. For the kernel and MMU, the main memory (RAM) is literally *physical*, making the space used by each process *virtual*. The spaces managed in this project are different, although analagous. Specifically, our code will be creating two memory spaces:

Real: A single, contiguous block of memory allocated within the process and managed by our code. The size of this real memory is determined when the process begins, and may vary from one run to the next. This space is analogous to the *physical memory* managed by the kernel/MMU, where the size of RAM is determined when the system boots.

Simulated: The space used by our process, but whose addresses are mapped to *real* addresses by our simulated MMU. The size of this memory appears to be as large as the address space, and pages are mapped to underlying real memory as they are used. This space is analogous to the *virtual memory* provided by the kernel/MMU, where the size of the abstracted space is constant, and not tied to the size of the underlying memory.

In short, we will use programs that store data into, and retrieve data from, *simulated* addresses. However, those addresses will be translated automatically (by our code) to *real* addresses, at which the given data will really be stored.

2 Getting started

In the usual way, create a new GitLab repository and grab the starting code…

1. **Login to the server** via `ssh`.

2. **Login to GitLab** in your browser.

3. **Start a new project:** Set the *Project name* to be `sysproj-6`.

4. **Clone the repository onto the course server:**

```
$ git config --global user.name "Your Name"
$ git config --global user.email "yourusername@amherst.edu"
$ git clone git@gitlab.amherst.edu:yourusername/sysproj-6.git
$ cd sysproj-6
```

5. **Download the source code:**

```
$ wget -nv -i https://bit.ly/cosc-171-21f-p6
$ ls -l
```

6. **Add/commit/push the source code to the repository:**

```
$ git add *
$ git commit -m "Starting code."
$ git push
```

2.1 The `vmsim` library

To make it possible for programs to use simulated memory, our code will be contained within a *library*—a pre-written collection of functions that other code may use.[1] This library will create the *real* memory, create and maintain a page table of mappings from *simulated* to *real* addresses, and translate the *simulated* addresses into *real* ones on demand.

The interface: The `vmsim` library provides the following functions:

- ```
  void vmsim_read (void* buffer,
                   vmsim_addr_t sim_addr,
                   size_t size)
  ```

  Read `size` bytes from a *simulated* address (`sim_addr`) into the `buffer`.

[1] We have used this idea for our heap allocators, e.g., `libpb` and `libbf`. A library provides existing functions and provides an interface for calling those functions, much as a Java class contains pre-written methods and public methods that other code can call.
  ```

- ```
  void vmsim_write (void* buffer,
                    vmsim_addr_t sim_addr,
                    size_t size)
  ```
  Write `size` bytes from the `buffer` to a *simulated* address (`sim_addr`).

- ```
  void vmsim_read_real (void* buffer,
                        vmsim_addr_t real_addr,
                        size_t size)
  ```
  Read `size` bytes from a *real* address (`real_addr`) into the `buffer`. This function **should not** be called by normal programs using the `vmsim` library; but it does **need** to be called by the simulated MMU in order to access the *real* memory space.

- ```
  void vmsim_write_real (void* buffer,
                         vmsim_addr_t real_addr,
                         size_t size)
  ```
  Write `size` bytes from the `buffer` to a *real* address (`real_addr`). This function **should not** be called by normal programs using the `vmsim` library; but it does **need** to be called by the simulated MMU in order to access the *real* memory space.

- `vmsim_addr_t vmsim_alloc (size_t size)`

  Allocate a block of at least `size` bytes of simulated space. A simulated address is returned.[2]

- `void vmsim_free (vmsim_addr_t sim_addr)`

  Deallocate the block of simulated memory at `sim_addr`. This block must have been allocated using `vmsim_alloc()`.

The `vmsim` library also defines the following types:

- `vmsim_addr_t`: A 32-bit unsigned integer that stores a single *simulated* or *real* address.
- `pt_entry_t`: A 32-bit unsigned integer that stores a single page table entry. (See Section 3 for more information on the `vmsim` page tables.)

[2]A program is **not** required to use this allocator to obtain simulated memory—it may simply use any simulated address—but the allocator may be useful to for imposing an organization on the simulated space.
  ```

Writing a program to it: Included in the vmsim directory are a pair of programs that use simulated memory and rely on vmsim to provide it. Here, we will examine the code in iterative-walk.c.

First, notice the inclusion of the library's *header file*, vmsim.h. This file (which you can open and examine) provides the declaration of the types, constants, and functions that can be called. The #include directive must be used in any program that uses vmsim.

Second, notice in the functions populate() and traverse() how the vmsim_read() and vmsim_write() functions are used. Let us take, as an example, the following lines:

```
uint64_t current;
vmsim_read(&current, addr, sizeof(current));
```

We create a space, current, that holds a 64-bit unsigned integer (uint64_t). We then call the vmsim_read function, passing it the following information:

- &current is a *pointer to the space of that name*. That is, the ampersand (&) is the *reference operator* in *C*; it is the inverse of the more familiar *dereference operator* (*). Instead of passing the value of current itself, we are passing a pointer to the space named current.
- addr is a simulated memory address (determined by code that precedes this example).
- sizeof(current) is the number of bytes in the space named current. Given that current is a 64-bit value, the value passed here is 8.

The result of this call is that the 8 bytes stored starting the simulated address addr are copied in current itself. By writing code the reads bytes from and writes bytes into the simulated space, we can use that simulated space to store arbitrary data.[3]

Compiling and running: In order to compile vmsim and the test programs that use it (iterative-walk and random-hop), use the make command, simply, like so:

```
$ make
```

This command reads the Makefile (which you can examine) in order to know how to compile the pieces of this project. Learning how to use this command is highly recommended, so Google for make command tutorial, or just start with this seemingly decent tutorial on it.

You will see that make compiles vmsim to create libvmsim.so (a *shared library*), then compiles and links the two test programs. Once the make command is done, you will also have two executable files, one each for the test programs. If you try to run, say, random-walk, you would invoke it like so, but see the following error:

[3] This approach to reading and writing data is not elegant, but it is the price we pay for defining and implementing the simple vmsim interface. This interface is quite like the one used for reading data from and writing data to *files* using standard file system functions.

```
$ ./random-hop
./random-hop: error while loading shared libraries:
  libvmsim.so: cannot open shared object file:
  No such file or directory
```

What does that error message mean, and how do you fix it? Any interesting program depends on library functions. Most of these libraries, such as the *standard C library* (known as `libc`) are stored their own set of pre-determined directories. The compiler and the shell use these directories automatically to find and link the correct libraries to a program when it runs.

Our test programs use `libvmsim`, which is **not** a standard library in one of thse pre-determined directories. We have to set an environment variable so that shell can find `libvmsim`, which is also within our current directory. Thus, we need first to use the following command (and we need to use it only once):

```
$ export LD_LIBRARY_PATH=${PWD}
```

This command sets the *environment variable* named `LD_LIBRARY_PATH` to include the *current directory*. Once we have done so, we can run one of the test programs:

```
$ ./random-hop
$ USAGE: ./random-hop <space size (bytes)>
$ ./random-hop 100000
```

This little program randomly selects simulated addresses from 0 to (in this case) 100,000. At each address, if the value is 0, the value is then set to 1; if the value is already 1, then the program ends, reporting the number of addresses is visited. However, **this program won't work properly**. Initially, the MMU always returns the real address 0; the mapping of simulated to real address has not yet been properly implemented. That leads us to Section 3...

3 Writing the MMU

Your task is to implement the simulated MMU, making it translate simulated addresses to real ones using page tables created and managed by other `vmsim` code. Before you do that, though, in Section 3.3, there are things you need to know about the page tables and how to manipulate their entries in C.

3.1 Address and page table format

Page tables and address spaces for `vmsim` mimic the format used for the *32-bit Intel ia32* (a.k.a., *x86*) ISA. Specifically, addresses are 32-bits each, with those bits divided as follows:

- **[31-22]:** The most significant 10 bits of each address are the *upper page-table index.*

- **[21-12]:** The next 10 bits of each address are the *lower page-table index*.
- **[11-0]:** The least significant 12 bits of each address are the *byte offset within the page*.

Each block of the multi-level page table is 4 KB that contain 1,024 entries each. Thus, for a given simulated space, there is a single *upper page-table (UPT)*, stored at some *real* address. For a given address, the UPT index specifies one entry ($2^{10} = 1,024$) in that UPT. That entry contains the real address of a *lower page-table (LPT)*. The LPT index specifies one entry within the LPT.

The contents of the LPT entry is the *real address of a page*—that is, the page to which the simulated address's page number is mapped. If the real page's address is combined with the 12 offset bits, the result is a specific byte address, in the real address space, to which the simulated address maps.

3.2 Handy bit-manipulation operators in *C*

Given a 32-bit value that needs to be decomposed as described above, in Section 3.1, how do you isolate and use each component? To do so, you need to use the *bitwise operators*, which allow you to manipulate values at the bit level. Here is a listing, where you should assume that `x` and `y` are such a 32-bit unsigned integer values.

- `x >> y` *(shift right)*: Shift the bits of the value in `x` to the *right* by `y` positions, inserting `y` 0 values at the most significant positions.
- `x << y` *(shift left)*: Shift the bits of the value in `x` to the *left* by `y` positions, inserting `y` 0 values at the least signficant positions.
- `~x` *(bitwise logical NOT)*: Invert the bits, making each 0 into a 1, and each 1 into a 0.
- `x & y` *(bitwise logical AND)*: For each pair of bits at each position in `x` and `y`, perform the logical AND operation.
- `x | y` *(bitwise logical OR)*: For each pair of bits at each position in `x` and `y`, perform the logical *inclusive* OR operation.
- `x ^ y` *(bitwise logical XOR)*: For each pair of bits at each position in `x` and `y`, perform the logical *exclusive* OR operation.

Used together, these operations allow you to isolate any group of bits in a value. For example, in order to isolate the offset bits of an address, we can do the following:

```
uint32_t offset = addr & 0xfff;
```

First note that the constant `0xfff` is 20 0's, followed by 12 1's, composing a complete 32-bit value. The 1's are all in the positions associated with the offset in the address. This constant is being used as a *bit mask*—a special value used to isolate some bits of a value. By applying this bit mask to `addr` with the bitwise AND operator, we achieve two things:

first, the upper 20 bits of the result must be 0 (since any value AND 0 = 0); second, the 12 lower bits of the result will be a *copy* of those lower bits in `addr` (since any value AND 1 = itself). And thus, we keep the lower 12 bits and clear the upper 20, giving us exactly what we wanted—the offset of the address is isolation.

3.3 Where to write your code

Open `mmu.c` in order to get started in earnest. You will see that very little is defined. The module variable `upper_pt_addr` is declared and, via a call to `mmu_init()` (which is called from within `vmsim`), set. This variable contains the real address of the upper page-table that the MMU should use.

You will also see the `mmu_translate()` function. **Your task** is to fully implement this function. It is passed a simulated address, and your code should traverse the page tables in order to translate that simulated address into a real address. That real address is what `mmu_translate()` must return.

It is important to note that the page table, initially, is composed solely of the UPT. All of the 1,024 entries of the UPT are 0, and no LPT's exist. Thus, the MMU must handle all three of the following possibilities as it tries to translate `sim_addr`:

1. `UPT[upper_index] = 0`: There is no LPT to which the address's UPT entry leads. Call `vmsim_map_fault()` and then re-attempt the translation.
2. `UPT[upper_index] != 0 && LPT[lower_index] = 0`: There is no real page to which the address's LPT entry leads. Call `vmsim_map_fault()` and then re-attempt the translation.
3. `UPT[upper_index] != 0 && LPT[lower_index] != 0`: The is a real page backing this simulated page, so complete the real address with the address's offset bits and return it.

Notice that `vmsim_map_fault()` is already written to update the page table to ensure that a given simulated page is properly mapped to some real page.

3.4 How to test your code

Once you have attempted to write `mmu_translate()`, you should compile it with the `make` command and run either of the test programs. However, it is deeply likely that you will have bugs with which to contend. What to do?

First, *add some debugging output* to your MMU code. What simulated address is being passed? What are the upper and lower indices being extracted? What happens when your code tries to look up the UPT and LPT entries? If `vmsim_map_fault()` is called, then what happens when your code tries the translation again?

Second, *use the source level debugger* `gdb`. Set breakpoints in your MMU code and step through it. Is it behaving as you expected? Do you actually know what you expect? Are the values in the page tables what you expected?

Third, *write your own test program.* Write something even simpler than the two provided—one where you know exactly what should be stored in a simulated space and then read back from it. Add your debugging code to that simple test program.

In short, poke and prod the behavior of the code and figure out what is going on. **Good luck!**

4 How to submit your work

First, be sure that the most recent versions of your work are up-to-date on the GitLab server by performing an *add/commit/push* with `git`. Then, go to GitLab with your browser, and add me (*sfkaplan*) as a *Developer* to your repository.

This assignment is due on Tuesday, Nov-02, 11:59 pm.