Project 3 - User-level Memory Management (100 points + Extra credit 20 points)

CS 416/518 - OS Design

Deadline: April 12th, 2022, 11:59pm

Please read the project description carefully.

Assume you are building a new startup for Cloud Infrastructure (Terrific Systems); a competitor of Amazon Cloud Services. As the CEO of your company, you decide to move the page table and memory allocation to the software. In this project's part 1, you will build a user-level page table that can translate virtual addresses to physical addresses by building a multi-level page table. In part 2, you will also design and implement a translation lookaside buffer (TLB) cache to reduce translation cost. We will test your implementation across different page sizes to evaluate your code. We will mainly focus on the correctness of your code, the page table, and the TLB state. For the extra-credit (see part 3), you will implement a 64-bit 4-level page table design. Note that this project would require bit manipulation that you learned in Project 1.

Part 1. Implementation of Virtual Memory System (70 pts)

While you have used malloc() in the past, you might not have thought about how virtual pages are translated to physical pages and how they are managed. The goal of the project is to implement "t_malloc()" (Terrific malloc), which will return a virtual address that maps to a physical page. For simplicity, we will use a 32-bit address space that can support 4GB address space. In this project we will emulate our RAM (i.e., physical memory) by allocating a large region of memory and vary the physical memory size and the page size when testing your implementation.

set_physical_mem(): This function is responsible for allocating memory buffer using mmap or malloc that creates an illusion of physical memory (Linux http://man7.org/linux/man-pages/man2/mmap.2.htm). Here, the physical memory refers to a large region of contiguous memory that is allocated using mmap() or malloc() and provides your page table and memory manager an illusion of physical memory. Feel free to use malloc to allocate other data structures required to manage your virtual memory.

translate(): This function takes the address of the outer page directory (a.k.a the outer page table), a virtual address, and returns the corresponding physical address. You have to work with two-level page tables. For example, in a 4K page size configuration, each level uses 10-bits with 12-bits reserved for offset. For other page sizes X, reserve log_2(X) bits for offset and split the remaining bits into two levels (unequal division is possible in this case).

page_map(): This function walks the page directory to check if there is an existing mapping for a virtual address. If the virtual address is not present, a new entry will be added. Note that you will be using this in t_malloc() to add page table entry.

t_malloc(): This function takes the bytes to allocate and returns a virtual address. Because you are using a 32-bit virtual address space, you are responsible for address management. To make things simple, assume for each allocation, you will allocate one or more pages (depending on the size of your allocation). So, for example, for t_malloc(1 byte) and t_malloc(1 byte), you will allocate one page for each call. Similarly, for t_malloc(4097 bytes), you will allocate 2 pages when the page size is set as 4KB.

If the user allocates a memory size that is larger than one-page size (e.g., 8KB bytes for 4KB page size), the multiple pages you allocate can be either physically contiguous or non-contiguous in the physical memory depending on the availability.

Note that this approach would cause internal fragmentation. For example, to allocate 1 byte of memory, we must allocate at least 1 page. For simplicity, you don't have to handle internal fragmentation in this in this project (unless you are aiming for extra credits). We describe an approach to reducing internal fragmentation found in Part B of the extra credit section.

Keeping Track: You will keep track of physical pages that are already allocated or free. To keep track, use a virtual and physical page bitmap that represents a page (you **must** use a bitmap). You can use the bitmap functions from Project 1. If you do not use a bitmap, you will lose points (for example, using an array of `chars` with 'y' or 'n' or an array of ints using each index for a page).

The get_next_avail() (see the code) function must return the next free available page. You must implement bitmaps efficiently (allocating only one bit per page) to avoid wasting memory (https://www.cprogramming.com/tutorial/bitwise_operators.html). Also, see the C Review resources on Canvas for some simple examples.

Why use physical and virtual bitmaps? With the physical bitmap (1 bit for each page), you can quickly find the next available free physical page, which can be present anywhere in the RAM.

Next, the use of a virtual bitmap is optional. The virtual bitmap expedites finding free page entries. Alternatively, you can walk the page table/page directories to find free entries. However, you might need to maintain some extra information within each entry to do this.

t_free(): This call takes a virtual address and the total bytes (int) and releases (frees) pages starting from the page representing the virtual address. For example, when using 4K page size, t_free(0x1000, 5000) will free two pages starting at virtual addresses 0x1000. Also, please ensure t_free() isn't deallocating a page that hasn't been allocated yet! Note, t_free returns success only if all the pages are deallocated.

You should be able to free non-contiguous physical pages using the information from your page tables, and the virtual address is given as an argument in t_free(); otherwise. When a user frees one or more pages, you would have to update the virtual bitmap, the physical bitmap (marking the corresponding page's bitmap to 0), and clean the page table entries.

Beyond *t_malloc()* and *t_free()*, you will also develop two additional methods that directly use virtual addresses to store or load data.

put_value(): This function takes a virtual address, a value pointer, and the size of the value pointer as an argument and directly copies them to physical pages. Again, you have to check for the validity of the library's virtual address. Look into the code for implementation hints.

get_value(): This function also takes the same argument as put_value() but reads the data from the virtual address to the value buffer. If you are implementing TLB, always check the presence of translation in TLB before proceeding forward.

mat_mult(): This function receives two matrices mat1 and mat2 as input arguments with a size argument representing the number of rows and columns. After performing matrix multiplication, copy the result to the answer array. Take a look at the test example. After reading the values from two matrices, you will perform multiplication and store them in an array. For indexing the matrices, use the following method:

```
A[i][j] = A[(i * size_of_rows * value_size) + (j * value_size)]
```

Important Notes: You cannot change the function arguments/signature of the following: t_malloc(), t_free(), put_value(), get_value(), mat_mult(). Your code must be thread-safe and your code will be tested with multi-threaded benchmarks.

We have included one sample test code (benchmark/multi_test.c) to check your code for thread safety. This is just a sample; feel free to use (by adding to Makefike), extend, increase thread count to verify the correctness.

Part 2. Implementation of a TLB (20 pts)

In this part, you will implement a direct-mapped TLB. Remember that a TLB caches virtual to physical page translations. This part cannot be completed unless Part 1 is correctly and fully implemented.

Logic:

Initialize a direct-mapped TLB when initializing a page tab. No translation would exist for any new page that gets allocated in the TLB. So, after you add a new page table translation entry, also add a translation to the TLB by implementing *add_TLB()*.

Before performing a translation (in *translate()*), lookup the TLB to check if virtual to physical page translation exists. If the translation exists, you do not have to walk through the page table for performing translation (as done in Part 1). You must implement *check_TLB()* function to check the presence of a translation in the TLB.

If a translation does not exist in the TLB, check whether the page table has a translation (using your part 1). If a translation exists in the page table, then you could simply add a new virtual to physical page translation in the TLB using the function *add_TLB()*.

Number of entries in TLB: The number of entries in the TLB is defined in my_vm.h (TLB_ENTRIES). However, your code should work for any TLB entry count (modified using TLB_ENTRIES in my_vm.h).

TLB entry size: Remember, each entry in a TLB performs virtual to physical page translation. So, each TLB entry must be large enough to hold the virtual and physical page numbers.

TLB Eviction: As discussed in the class, the number of entries in a TLB is much lower than the number of page table entries. So clearly, we cannot cache all virtual to physical page translations in the TLB. Consequently, we must frequently evict some entries. A simple technique is to find the TLB index of a virtual page and replace an older entry in the index with a new entry. The TLB eviction must be part of the *add_TLB()* function.

Expected Output: You must report the TLB miss rate by completing the *print_TLB_missrate()* function. See the class slides for the definition of TLB miss rate.

Important Note: Your code must be thread-safe, and your code will be tested with multi-threaded benchmarks.

Part 3. Extra Credit Parts

(A) Implementing a 4-level page table (10 pts)

For the first extra credit, you are required to implement a 4-level page table for 64-bit addressing by extending the 2-level page table in Part 1 for 32-bit CPUs. You will get points only if your other parts of the project are correct and your extra-credit part is correct.

1. Before attempting to implement the extra-credit part, you must complete Part 1 and Part 2 of the project.
2. We cannot evaluate Part 3 unless Part 1 and 2 are implemented correctly.
3. Your 4-level design must be thread-safe.
4. Note that for the 64-bit design to work, you will have to compile the code without the -m32 flag. Therefore, we suggest creating "my_vm64.c" and compiling the code without the 32-bit flag.
5. Note that the TLB entries will also store 64-bit addresses.
6. For more questions, email us.

(B) Reducing Fragmentation (10 points):

For the second extra-credit part, you will find ways to reduce the internal fragmentation.

One way is to combine multiple small allocations to one page. For example, the first call of t_malloc returns an address 0x1000. When you call t_malloc again, you can return 0x1001 (if the application asked for 1 byte) or any address within the page size boundary. The next call to t_malloc (if the size requested ends up not fitting within the first page) would return 0x2000 or higher. Additionally, this would also complicate freeing up memory. For example, if you free 1 byte, you cannot always release the page because the page could contain data used for other allocations. So, you must separately keep track of what is allocated in a page, and release the page only after allocations in the page have been deallocated.

4. Suggested Steps

- Step 1. Design basic data structures for your memory management library.
- Step 2. Implement set_physical_mem(), translate() and page_map(). Make sure they work.
- Step 3. Implement t_malloc() and t_free().
- Step 4. Test your code with matrix multiplication.
- Step 5. Implement a direct-mapped TLB if steps 1 to 4 work as expected.

5. Compiling and Benchmark Instructions

Please only use the given Makefiles for compiling. We have also provided a matrix multiplication benchmark to test the virtual memory functions. Before compiling the benchmark, you have to compile the project code first. Also, the benchmark does not display correct results until you implement your page table and memory management library. The benchmark provides hints for testing your code. Make sure your code is thread-safe.

We will focus on testing your implementation for correctness.

6. Report (10 pts)

Besides the VM library, you also need to write a report for your work. The report must include the following parts:

1. Detailed logic of how you implemented each virtual memory function.
2. Benchmark output for Part 1 and the observed TLB miss rate in Part 2.
3. Support for different page sizes (in multiples of 4K).
4. Possible issues in your code (if any).
5. If you implement the extra-credit part, description of the 4-level page table design and support for different page sizes.

Because we are using a 32-bit page table (for Part 1 and 2), the code compiles with the -m32 flag. Not all iLab machines support -m32. Here's a list of them that you could use.

```
kill.cs.rutgers.edu
cp.cs.rutgers.edu
less.cs.rutgers.edu
ls.cs.rutgers.edu
```

7. Submission

Submit the following files to Canvas, as is (**Do not compress them**):

1. my_vm.h
2. my_vm.c
3. my_vm64.c (only if attempting the extra-credit Part 3)
4. A report in .pdf format, completed, and with both partners' name and NetID on the report
5. Any other support source files and Makefiles you created or modified

Please note that your grade will be reduced if your submission does not follow the above instructions.

8. FAQs

1. **Storing page tables** The page tables should be stored in the fake physical memory (the buffer that we are using as "physical memory"). This does not apply to globals and other data structures (that aren't the page tables or t_malloc calls) that you might need.
2. **MAX_MEMSIZE vs. MEMSIZE.** Within the header file (*my_vm.h*), you will see two defined values: (1) MAX_MEMSIZE and (2) MEMSIZE. The difference between the two is that MAX_MEMSIZE is the size of the virtual address space you should support, while MEMSIZE is how much "physical memory" you should have. In this case, MAX_MEMSIZE is defined as (4ULL * 1024 * 1024 * 1024), which is 2^32 bytes or 4GB, the amount of virtual memory that a 32-bit system can have. On the other hand, MEMSIZE is defined as (1024 * 1024 * 1024) or 2^30 bytes or 1GB, which is how much memory you should mmap() or malloc() to serve as your "physical memory."
3. **Be mindful of values 2^32 and up.** Notice that the definition of MAX_MEMSIZE is cast as an unsigned long long. This is because the library is compiled as a 32-bit program. With a 32-bit architecture, an int/unsigned int/long/unsigned long are all 4 bytes, meaning they can only represent values from 0 to 2^32 - 1. So when dealing with MAX_MEMSIZE or other values that equal or larger than 2^32, make sure to use *unsigned long long* to avoid any value truncation.
4. If you are using your personal computer for development and getting the following error, then refer to this link: https://www.cyberciti.biz/faq/debian-ubuntu-linux-gnustubs-32-h-no-such-file-or-directory/

```
gnu/stubs-32.h: No such file or directory compilation terminated. make: *** ...
```