Project 3 - User-level Memory Management (100 points)

CS 416 - OS Design

DEADLINE: April 9th, 2020, 11:59 pm

Assume you are building a new startup for Cloud Infrastructure (Amazing Systems); a competitor of Amazon Cloud Services. In class, we discussed the benefits of keeping memory management in hardware vs. moving them to the software. 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, to reduce translation cost, you will also design and implement a translation lookaside buffer (TLB). For evaluating your code, we will test your implementation across different page sizes. We will mainly focus on the correctness of your code and the page table and TLB state.

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 "a_malloc()" (Amazing 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. We will 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 can be allocated using mmap() or malloc() and provides your page table and memory manager an illusion of physical memory. Feel free to use malloc for allocating other data structures required for managing your virtual memory.

translate(): This function takes input a page directory (address of 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.

page_map(): This function walks the page directory to see if there is an existing mapping for a virtual address. If the virtual address is not present, then a new entry will be added.

a_malloc(): This function takes the number of bytes to allocate and returns a virtual address. Because you are using a 32-bit virtual address space, you are responsible for address management. You must also keep track of already allocated virtual addresses. To make things simple, assume that all allocations are page granularity.

For example, the first call of a_malloc returns an address 0x1000. When you call a_malloc again, you can return 0x1001 (if application asked for 1-byte) or any address within the page size boundary. The next call to a_malloc (if the size requested ends up not fitting within the first page) would return 0x2000 or higher. In summary, your virtual addresses are page aligned.

You will keep track of which physical pages are already allocated and which pages are free; use a virtual and physical page bitmap that represents 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 each page) to avoid wasting memory (https://www.cprogramming.com/tutorial/bitwise_operators.html).

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

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 hints.

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

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

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

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

Part 2. Implementation of a TLB (20 pts)

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

Logic:

Initialize a direct-mapped TLB when initializing a page table. For any new page that gets allocated, no translation would exist 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 are 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.

3. 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 a_malloc() and a_free().
- Step 4. Test your code with matrix multiplication.
- Step 5. Implement a direct-mapped TLB if steps 1 to 4 works correctly.

4. 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 towards testing your implementation for correctness.

5. Report (10 points)

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).

Because we are using a 32-bit page table, the code compiles with -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
```

6. Submission

Submit the following files to Sakai, as is (**Do not compress them**): 1. my_vm.h 2. my_vm.c 3. A report in .pdf format, completed, and with both partners names and netids on the report 4. Any other support source files and Makefiles you created or modified

Please Note: Your grade will be reduced if your submission does not follow the above instruction.

7. Other Things To Note

1. **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".
2. **Be mindful of values 2^32 and up.** Notice that the definition of MAX_MEMSIZE is casted 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 cal only represent the 0 to 2^32 - 1 different values. 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.
3. 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: *** ...
```