Project 3: User-Level Memory Management Library CS 416 OS Design
Deadline: 16th November 2020 [23:55]

You are building a startup for Cloud Infrastructure (Amaze.Me); a competitor of Google Cloud. You remember your class 416 where we discussed the benefits of keeping memory management in hardware vs on a software. As the CTO of Amaze.Me you decide to move page table and memory allocation to software.

In this project, you shall build a user-level page table that translates virtual addresses to physical addresses by building a multi-level page table.

For evaluation, we shall test your implementation across different page sizes.

You can do this project in groups of <=2.

Part 1: Implementing a Virtual Memory System (70 pts)

Presuming you have used malloc in the past, programmers take addressing as a black box. Virtual pages are translated to physical pages and there are a whole lot of book-keeping mechanisms involved in the OS/hardware.

The goal of this project is to implement "myalloc()" which will return a virtual address that maps to a physical page. Here, physical memory is a large region of contiguous memory which can be allocated using mmap() and malloc() [The functions your OS provides.]. This provides your memory manager an illusion of Physical Memory in software.

For simplicity, we shall use 32-bit address space which can support up to 4GB of address space. Make sure your library works with different memory sizes and page sizes (incrementing in powers of 2). Also, there is a need for your library to be thread safe; your library should be able to work without errors when multiple threads request and access memory at the same time.

The following are the APIs your library shall have:

1. **SetPhysicalMem()**: This function allocates memory buffer using mmap or malloc that creates an illusion of physical memory.

2. **Translate()**: This function takes a page directory (address of the outer page table) and a virtual address as input and returns the corresponding physical address. You have to work with a two-level page table. For example, in a 4K page size config, each level uses 10 bits with 12 bits reserved for offset (10 + 10 + 12 = 32 bits).

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

4. **myalloc()**: This function takes, as input, the number of bytes to allocate and returns a virtual address. To make things simple, assume that all allocations are at a page granularity, ie. for a page size X, myallocs with <X sizes will still allocate a complete page of size X.

5. **myfree()**: This call takes a virtual address and releases memory allocated at this virtual address. Note: myfree() returns success only if it is able to release all the pages of this address.

6. **PutVal()** / **GetVal()**: These functions take a virtual address, a value pointer and the size of the value pointer as an argument, and directly copy/read them to/from physical pages respectively. You have to check the validity of the library's virtual address. You cannot write/read to/from a place which was not allocated first. If you decide to implement a software-TLB, check the TLB first before proceeding forward. TLB hits makes accesses very fast since page walks are averted.

7. **MatMul()**: This function is your user function. This program includes your memory manager library. This function receives two matrices mat1 and mat2 and their sizes (number of rows and columns) as arguments. After performing the matrix multiplication, copy the result to an answer array. You do not need to have a 2D array since that increases Memory manager's complexity. Hint: A[i][j] = A[(i * Number of Rows) + j]

Part 2: Implementation of a TLB (20 points):

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 *put_in_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. You must implement *check_in_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. 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 *put_in_TLB()*.

Number of entries in TLB:

The number of entries in the TLB is defined in my_vm.h (TLB_SIZE). However, your code should work for any TLB entry count (modified using TLB_SIZE 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 *put_in_TLB()* function.

Expected Output:
You must report the TLB miss rate. See the class slides for the definition of TLB miss rate.

Important NOTE: You code should be thread safe. Testing will be done with multi-threaded benchmarks. While discussing at a theoretical level is encouraged, refrain from sharing source codes with other groups. We shall run a plagiarism checker on your source files.

Suggested Steps:

1. Design basic data structures for your memory management library.
2. Implement SetPhysicalMem(), Translate(), PageMap(). (Make sure they work)
3. Implement myalloc(), myfree(). (You must keep track of already allocated virtual addresses)
4. Test your code with Matrix Multiplication.
5. Implement a direct-mapped TLB if steps 1-4 work correctly.

Compiling and Benchmark Instructions:

Please use the given Makefile for compiling. Before compiling the benchmark, you have to compile your project code first. Also, the benchmark would not display correct results until you implement your page table and memory management library. The benchmark provides a hint for testing your code.

For 32-bit addressing, compile your code with **-m32** flag on gcc. For ease of testing later on, keep your page sizes as a macro definition (#define PAGETABLE 4096) in your library.
For this assignment, you can use either ilab machines (some of them do not support 32-bit compiler) or one of the following:

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

In the report add all the relevant details about the implementation of this project. Also, add what parts were done by each group member.

Report (10 pts):

Besides the VM library, you also need to write a report for your work. Mention your team members in report.

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 / runtime improvements in Part 2.
3. Support for different page sizes (in multiples of 4K).
4. Possible issues with your code (if any).
5. What were the most difficult parts of solving this project?

Report should be named **report.pdf**. Pl. adhere to the naming convention.

Submission Format:

You need to submit your project code and report in a compressed tar file on sakai. ONLY ONE SUBMISSION PER GROUP IS REQUIRED.

You MUST use the following command to archive your project directory.
`$ tar zcvf <netid1>_<netid2>.tar.gz project3` ##netid1 of the person submitting on sakai

NOTE: Your grade may be reduced if your submission does not follow the above instruction. For students working alone, netid2 == netid1.

PS: Start early, this project is NOT a cakewalk.

For further questions, please visit the TAs in their office hours and/or recitations.

Other Noteworthy Points:

1. **MAX_MEMSIZE vs MEMSIZE**. Within the header file (my_vm.h), you will see two definitions: 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 (4 * 1024 * 1024 * 1024) which is 2^32 bytes or 4 GB, the amount of virtual memory a 32-bit system can have theoretically. On the other hand, MEMSIZE is defined as (1024 * 1024 * 1024) or 2^30 bytes or 1 GB, which is how much memory you should mmap/malloc to serve as your "physical memory".

2. Be mindful of values above 2^32. Notice that 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 arch, an int/unsigned int/long/ are all 4 bytes, meaning they can only represent values from 0 to 2^32 -1 different values. So when dealing with MAX_MEMSIZE or other large values, make sure to use unsigned long long to avoid any value truncation (and hours of debugging).

3. If you are using a personal computer for the project and getting the following error, then refer to this link:
   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: *** ...
```