Project 3: User-Level Memory Management Library
CS 416 OS Design
Deadline: 8th November 2019 [23:55]

You are building a new 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.
Extra Credit: Also add a TLB for reduced translation cost.

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

You can do this project in groups of 2.

Description:
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.

The goal of this project is to implement "m_alloc()" 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 one's your OS provides.]. This provides your memory manager an illusion of Physical Memory.

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.

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. **m_alloc()**: This function takes the number of bytes to allocate and returns a virtual address. To make things simple, assume that all allocations are at a page granularity.
   So, even if user program asks for 10 bytes, you allocate a complete page if it is smaller than the size of your page.
5. **a_free()**: This call takes a virtual address and releases memory allocated at this virtual address.
   Note: a_free() returns success only if it is able to release all the pages of this address.

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

7. **MatMult()**: 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. Hint: A[i][j] = A[(i * Number of Rows) + j]

BONUS: If you have finished the page table design and it works fine, implement a direct-mapped software TLB. The length of this TLB would be configurable and should be specified in the as a Macro Definition (#define TLB_SIZE 120).
You will get bonus points only if your page table design works correctly with threading support and your TLB works correctly too.

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 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 m_alloc(), a_free(). (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 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.

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 <netid>.tar.gz project3 ##netid of the person submitting on sakai

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

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

For further questions, please visit the TA in his office hours or recitations.