You are building a new startup for Cloud Infrastructure (Amazing Systems); a competitor of Amazon Cloud Services. In the class, we discussed the benefits of keeping memory management in hardware vs. moving them to software. As the CEO of your company, you decide to move page table and memory allocation to software. In this project, you will build a user-level page table that can translate virtual addresses to physical addresses by building a multi-level page table. For extra credits, to reduce translation cost, you will also design a TLB. For evaluation of your code, we will test your implementation across different page sizes.

1. Description

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. The physical memory refers to a large region of contiguous memory which can be allocated using mmap() or malloc() and provides your page table or memory manager an illusion of physical memory. For simplicity, we will use a 32-bit address space which 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). Feel free to use malloc for allocating other data structures.

translate(): This function takes as an 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 malloc 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 ny 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)]

BONUS (10 points): If you have finished the page table design, implement a direct-mapped TLB. The length for this TLB would be configurable and should be specified in the my_vm.h file as a TLB_SIZE constant. <u>You will get bonus points only if the page table design works correctly with threading support and your TLB works correctly.</u>

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

2. 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.

3. Compiling and Benchmark Instructions

Please only use the given Makefiles for compiling. 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 a hint for testing your code.

4. Machines with a 32-bit library (-m32 in your makefile)

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