Andrew Park (ap1614), Elysia Heah (eyh24)

CS416: Operating Systems Design

Fall 2020

Project 3 Report: User-Level Memory Management Library

Part 1

In the my_vm.c file, we defined several functions and a couple global variables in order for our memory management library to function properly. Of note are the macros that easily allow the user to manipulate the size of the page they want, as well as the structure for the translation lookaside buffer. The TLB was made into an array of node structures so that it was easy to access the virtual address, corresponding physical address, and position of the node in the LRU eviction queue all at once.

bitMask was first written so that we can easily have a function which extracts a specific sequence of bits from values that are passed into it.

SetPhysicalMemory was simple to write. We first created the physical memory block by malloc'ing a block of size MEMSIZE. We then created 2 sets of state tables, one for physical page states and one for virtual page states.

In Translate, we used our bitMask function which we defined earlier to separate the address into 2 parts. These are the first level index and the second level index, which we used to access the pagedir and return the physical address.

In PageMap, we take the pgdir and va, which we then use as the first and second level indices on pagedir. We check if it is NULL, and map the physical address when necessary.

In get_next_avail, we were able to easily find the next available page by looping through the vpage_states and seeing which one is available.

In myalloc, we first check if the memory section was allocated. If not, we call SetPhysicalMem in order to initialize it. Then we find the next available page using get_next_avail and then calculate how many pages we will need. We then set the pages in virtual memory. If the location in pagedir is NULL, then we would set it to the address that we compute. Otherwise, the mapping already exists. We then return the address.

In myfree, we take the va and use Translate to get the physical address. We use the address and figure out how many pages need to be freed and where. Then we loop through the vpage_states and set them as available. We then take the address and separate it using bitMask, then set the location in pagedir as 0.

For PutVal and GetVal, we simply took the virtual address and translated it using Translate. The only difference is in the memcpy, where we would just reorder the arguments.

In MatMul, we first allocate space using malloc for 3 matrices. We then use 3 sets of for loops. The first set of for loops will use GetVal and store them in the first 2 matrices. The second set of for loops will then calculate the actual multiplication of the two matrices and store them in a resultant matrix. The last set of for loops will then use PutVal to store the result in the answer.

Part 2

Once everything in the first part of the project was set up, we could now implement the TLB. This is a simple cache structure that is modeled by an array of structures as noted in the header file. Each node has two void pointers for each type of

address and an integer that attaches a position in the eviction order when the cache reaches its limit.

In add_TLB, we assess whether or not the tlblist has empty space. If the first index is empty we can immediately fill it in; otherwise, we loop through the array and keep track of the minimum position that we can find. If we reach a null index it means that there is still enough space and we can add the entry there, otherwise we exit the loop and replace the lowest priority entry with the new one to be added. We allocate memory for each *brand new* entry in the array.

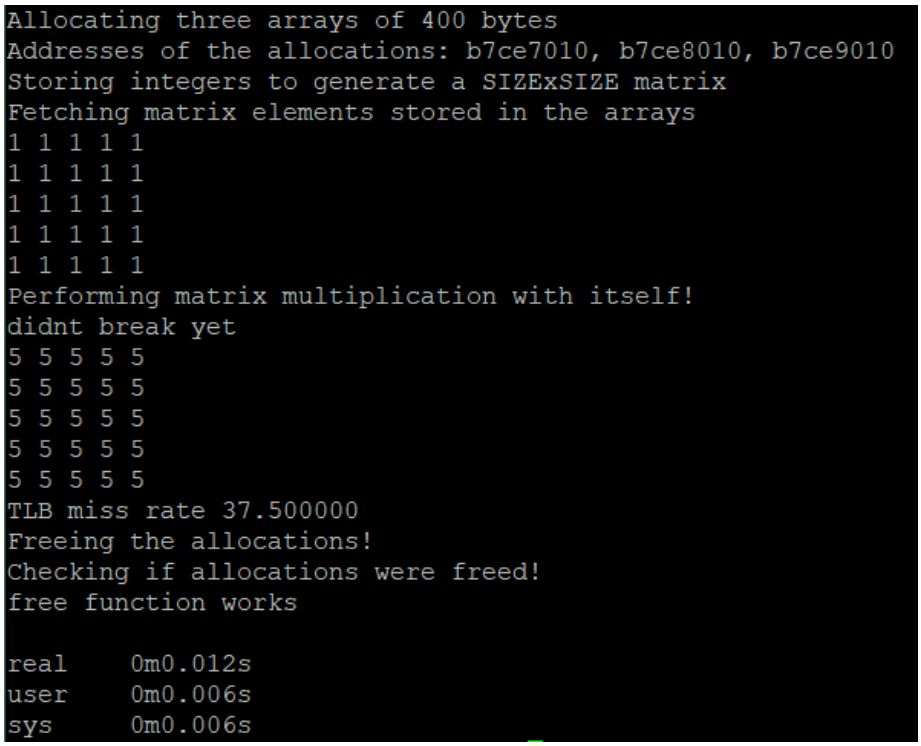
In check_TLB we determine whether or not an entry exists in the cache. We loop through the array and check if we hit either a null entry, or the one we are looking for. In the former case we exit and return null because the list ended before we found what we needed, and in the latter case we simply return the physical addresses from the entry that we needed.

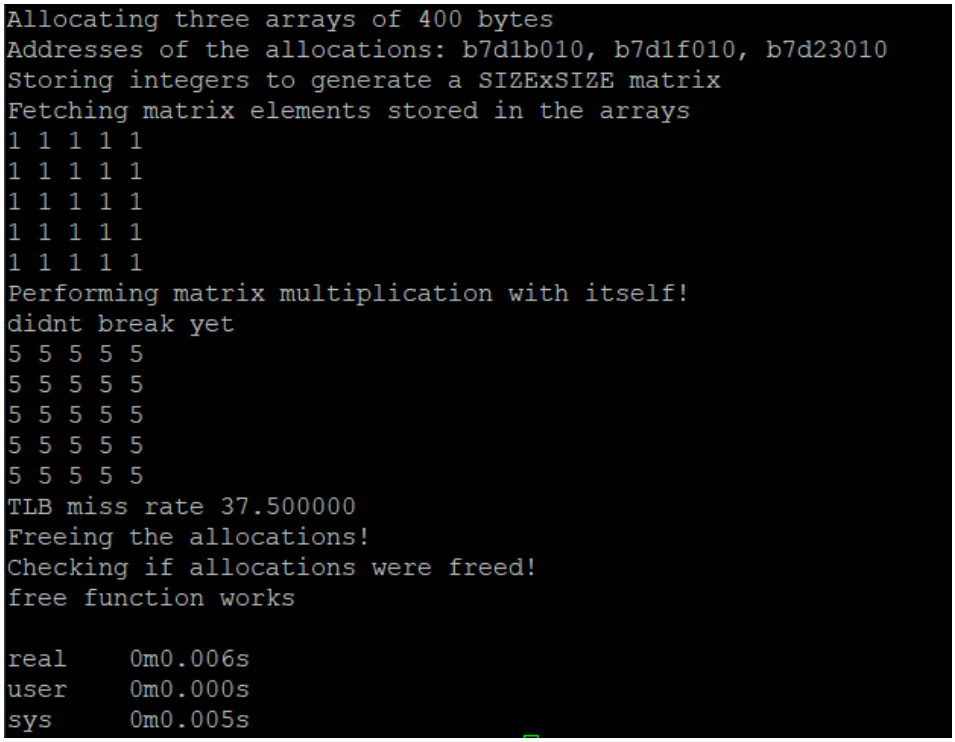
Note that when we call check_TLB in the translate function, we also have to update our *misses* and *count* counter variables. The former keeps track of all the times that we have a cache-miss, and is incremented every time we search for an entry that does not exist in the TLB. The count variable keeps track of the number of all searches that are performed. This does two things: first, it allows us to update each entry's position appropriately. Every time an entry is searched it will be assigned the value of count, so entries that are newly added or are revisited (cache-hit) get their position updated as a more recently-used node. Every time we need to evict a block, we simply look for the entry with the lowest priority (lowest position) because that is the oldest-updated block in the cache.

Finally, keeping the count variable also makes counting the miss rate very easy. In print_TLB_missrate, all we have to do is cast the *misses* and *count* variables to doubles and calculate missed/total*100 to get the miss rate of all the searches.

Benchmarks and TLB

Overall we fulfilled the benchmarks well, and we observed that the TLB made a marked improvement as well. We had a miss rate of around 37.5%, which is decent for our purposes. Timing the program shows that it can run in 12ms with the standard 4K page size. Bumping the page size up by 2-3 times produced similar results.





Some of the trickier components of the project included getting myalloc to work, as handling the memory allocations required some thinking. Structuring the physical and virtual addresses was a bit tough as well, but we managed to get it down. The bitmasking was a little complicated but it helped out a lot and made everything a lot smoother. As we were able to fulfill everything, we hope that there are no glaring issues or faults in the program.