Assignment 3 Report

By: Mark Pirella map577 and Alec Bakholdin amb672

Implementation of Each Virtual Memory Function

SetPhysicalMem() - This function is called upon the first invocation of the library (through a call to myalloc()). It initializes important global variables including our slice of physical memory, virtual and physical bitmaps, a mutex, the total number of physical pages, the total number of virtual pages, the numbers of bits used in an address for offset, outer index of page table directory, and inner index of page table directory, and the page table directory itself.

Translate() - Given a virtual address, this function returns the corresponding physical address by first checking for the virtual address in the TLB. If found, it will simply return a pointer to the value of the physical address found (quickly) in the TLB. If not found, then it accesses the page table directory to obtain a pointer to the value of the corresponding physical address.

PageMap() - We determined the number of bits for the first and second layers of the page directory in SetPhysicalMem(). Here, we use those values to find the pageDir pageDir indices in the virtual address. We then simply take the physical address passed as a parameter and insert that into the location defined by the two indices.

myalloc() - Calls SetPhysicalMem() if the library has yet to be initialized. Will look for the next x number of available consecutive pages in the virtual bitmap, based on the number of bytes the user requested to allocate. Will also look for the same number of available pages (doesn't have to be consecutive) in the physical bitmap. Then, it will add appropriate virtual->physical mapping entries to the TLB and use PageMap to add similar mapping(s) in the page table directory.

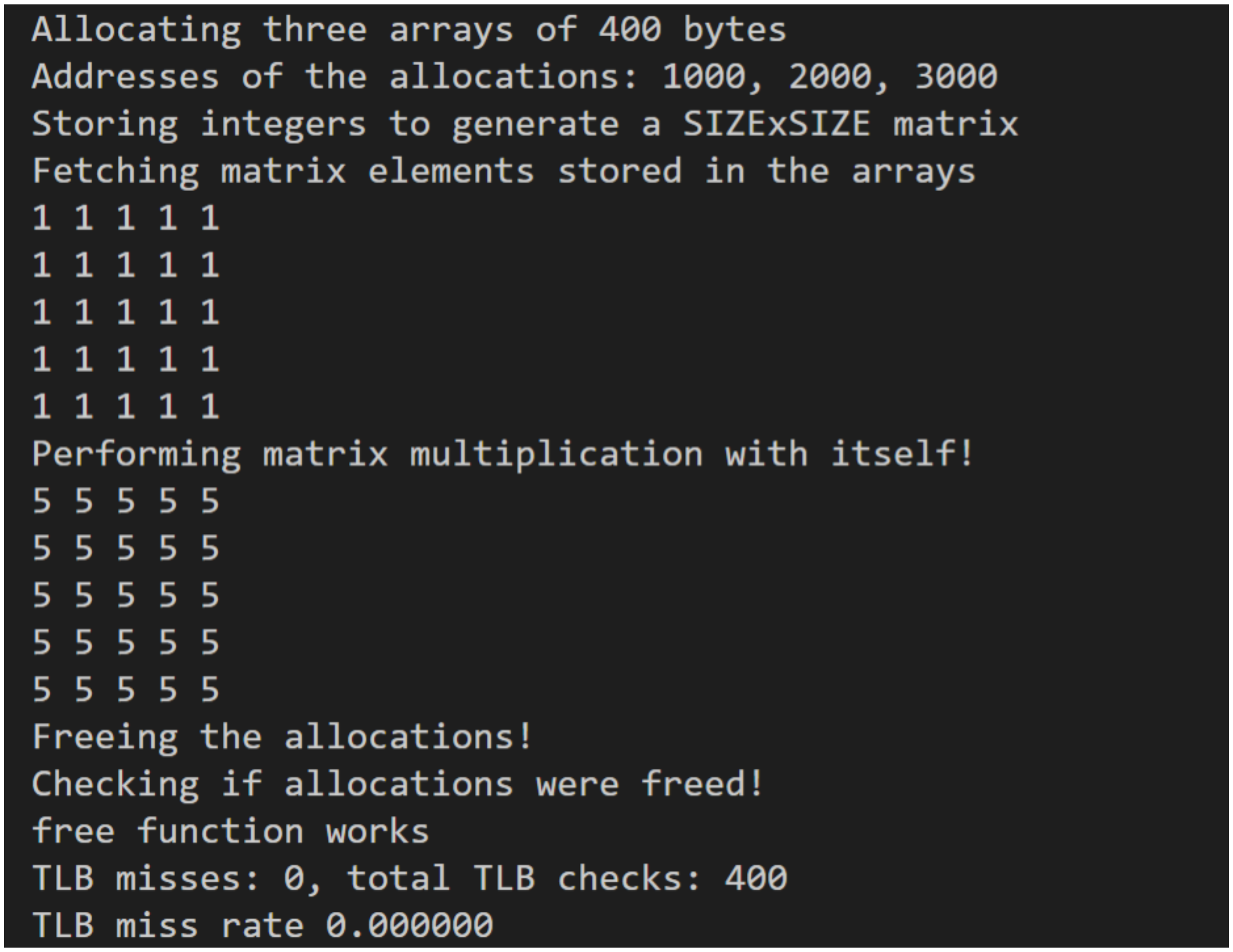
myfree() - This function calculates the number of pages with size/(PGSIZE), then iterates over that many pages (the index of the virtual page is the first [numInnerbits + numOuterBits] bits), and calls Translate() to get the corresponding physical address of each page. These page indices are stored in a secondary array, and as long as all the pages are accessible by the user, we can go on to free them. Otherwise, we throw a segfault and return. Finally, it uses the virtual and physical page indices to clear the corresponding bits in physBitmap and virtBitmap, which define which pages are available to be allocated.

PutVal() - This copies chunks of size PGSIZE into memory at each corresponding page location. We first calculate the number of pages in the value, then for each page in the value, we copy over PGSIZE bits, until we get to the last page, at which point we only copy over n bits, where n is the remaining bits in the value.

GetVal() - Works similar to PutVal(), but in reverse. Copies over chunks of size PGSIZE from physical memory [size/PGSIZE] times (number of pages), until we get to the last page. At this point, we copy over n bits, where n is [size % PGSIZE].

MatMul() - Does basic matrix multiplication. The only thing remarkable here is that it uses GetVal() and PutVal() instead of traditional bracket access [i][j]. Otherwise, each matrix is simply a flattened two-dimensional array.

Part 1 Benchmark Output



```
Allocating three arrays of 400 bytes
Addresses of the allocations: 1000, 2000, 3000
Storing integers to generate a SIZExSIZE matrix
Fetching matrix elements stored in the arrays
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Performing matrix multiplication with itself!
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
Freeing the allocations!
Checking if allocations were freed!
free function works
TLB misses: 0, total TLB checks: 400
TLB miss rate 0.000000
```

Part 2 Observed TLB Miss Rate and Runtime Improvements

(look at above screenshot for observed TLB miss rate when running benchmark with all default values: TLB size 120, matrix size 5, page size 4096)

If we keep the above values but change TLB size to 2 (to ensure we get some misses), we now see these results:

```
TLB misses: 175, total TLB checks: 400
TLB miss rate 0.437500
```

Using our own test program to calculate the latency of memory accesses through page table directory vs. TLB:

```
TLB_SIZE: 0     Number of Accesses: 100000      Average access time: 0.103620 usec
TLB_SIZE: 200   Number of Accesses: 100000      Average access time: 0.070160 usec
```

I suspect the change is not as dramatic as it could be because normally the TLB is on the CPU chip, and hardware speed plays a big role in reducing access time. However, here we still have to go to memory with our TLB, then iterate over a linked list, etc.

Support for Different Page Sizes

In order to ensure our program can support different page sizes, we made sure that we would grab the proper number of inner index bits, outer index bits, and offset bits from a given address based on the values given to MAX_MEMSIZE, MEMSIZE, and PGSIZE, and NOT using fixed, specific values. The number of virtual pages is obtained by calculating MAX_MEMSIZE / PGSIZE, the number of physical pages is obtained by calculating MEMSIZE / PGSIZE, the number of offset bits per address is obtained by calculating log-base-2(PGSIZE), the number of outer index bits is obtained by calculating (total number of bits per address - number of offset bits) / 2, and the number of inner index bits per address is obtained by calculating total number of bits per address - (number of outer index bits + number of offset bits).

Possible Issues with Our Code

The TLB is a linked list, which proves to be problematic when the TLB_SIZE exceeds certain numbers. For the example above, the length of the data we were accessing was 20,000 bytes, or 5 pages long. For larger page counts, the time it takes to access data actually goes up when the TLB is implemented. If we were to improve on our code, I think we would use a tree structure, perhaps a red black tree, that allows for O(logn) access instead of O(n).

Most Difficult Parts of the Project

The most difficult part of the project had to be understanding what it was we needed to do. It took several sessions looking over the slides and comparing them to the project instructions to figure out what our plan was going to be, but once we started coding, everything went pretty smoothly.