Jason Dao(jnd88)

Notes: I changed the makefile to include -pthread flag and -lm for math.h in the makefile for my_vm.c and my_vm.h and also included the -lm flag at the end for the makefile for the benchmarks.

ASSUMPTIONS:

In the instructions, it says to add to TLB after mapping, however, this makes it so for the first TLB_ENTRIES entries, the answer is 0% missrate and I don't think that is the output that is expected, (for example, test.c shows a 0% miss rate since it only uses 3 pages and nothing ever gets cleared or added). So for my implementation I DO NOT add to TLB after mapping because it seems like an error. However, I commented out on line 347 to add it to the TLB so when testing and you guys want to add it to there, just uncomment line 347.

1. To implement set_physical_mem, first I allocated the memory of the physical memory using malloc and then memset everything to 0. Next, I created the bitmaps of length MEMSIZE/PGSIZE for the physical bitmap and bitmaps of length MAXMEMSIZE/PGSIZE since those are the number of pages that can be put into the memory. I divided that by 8 because each char can represent 8 bits and made that the length of the bitmaps using malloc.

   To implement add_TLB, I first put a mutex so no more than 1 thread can access the tlb. I then found the index using the virtual address modulus the number of TLB entries. Next I set the virtual value to the virtual address inputted and physical value to the physical address in the parameter. After this I unlocked the mutex.

   To implement check_TLB, after a mutex grab, I first increment a variable totallookups. Next, I get the TLB by doing virtual address modulus tlb entries. Next, if the virtual addresses match up, a hit variable is incremented and the physical address returned. If not, a miss variable is incremented and NULL is returned.

   To implement printTLBmisrate, I divided the number of misses/totallookups and print it.

   To implement translate, after getting a lock, first I check if the entry is in the TLB using check TLB, if it is, I return it. If it is not, then I get the middle bits for the pte and check the table at the given page directory with that pte and if the number is not 0 (shows there is no entry there) I return it.

   To implement page_map, after locking with the lock for the page table I found the index of the pde passed in. Next, using bit manipulation, I found the middle bits for the page table entry and then checked to see if the pagetable with that pde and pte value had value 0(this means it has no address for the physical page). If it does, then I set that value to the physical page address passed in. I then unlock the lock and return 1 if is successful, -1 if not.

   To implement get_next_avail, after locking with the mutex for the bitmaps, first I searched the virtual bitmap to see if there were any 0s to mark an open page. If yes, check the next num_pages consecutive pages to see if there are any of that amount if the user wants more than 1 page. If it happens that the pages need a new page directory not previously allocated, a new page will be added for the pte of the new page directory and this process would restart since the page directory will use up a new value in the bitmap and so another page would be needed. After finding a page I then check the physical memory to see if there are that many pages left, if not it breaks. If there are for every page, I would get the pde and pte number using bit manipulation and call page map with the pde page as well with the virtual page bitmap index converted into a hexadecimal address(shifted for pde, pte and offset) as well as the next physical address available using the physical bitmap index of the next one. Once a page has successfully been created in the table, the physical and virtual bitmaps would be updated for all the virtual pages and physical pages created. Then I unlocked the mutex. Lastly, I return the address of the first page allocated.

To implement t_malloc, after setting a mutex for t_malloc, I checked to see if it was the first time running. If it was, I set the physical memory, as well as set up the page table(put page directory table in first page and first page table entry in second page) and stored the pointer to the physical memory as a global variable. Next, I found the number of pages needed based on bytes (numpages/pgsizes) and if numbytes % pgsize is not 0 add another. I inputted that into get_next_avail and that returned back the virtual address for this allocation. Next, I inserted into a data structure the virtual address and the size allocated to keep track of if when someone frees or puts value, they do not overstep the boundaries allocated. Then I returned the address after unlocking the mutex.

To implement t_free, first I found the number of pages it would need to free it. Next, I checked the size in the data structure where I inserted the virtual address and num_bytes to see if it fit that bound, if not then I return and print an error statement. If the address if not in that structure as well, it would return and print an error statement. Next I locked it using the lock for bitmaps, and for every page needed to free, I get the pde by shifting by pte+offset bits and then call translate with the address and that pde. After I get the address, I find the virtual bitmap index of it in the virtual address and using the physical address returned, get the index in the physical bitmap and set it to 0. Next, I call delete TLB where it finds that entry, if the index has the same virtual address, it sets that tlb entry to 0. After that I increment by the next virtual page size to get any other pages the user wants to free and repeat. After getting out of the loop, I remove the entry where it stores the first virtual address and the num_bytes.

To implement put_value, I first put a mutex and check if the address+size is within the bounds of something malloced early, if the address is not, it returns, if the size is not, then I make the size the last address in that allocation. For example, the user wants to put something 10 bytes, but there is only 8 left, I make the size 8. After that, I check to see if the first address and last are in the same page, if they are, I copy the whole thing to the physical address and return, if not I increment byte by byte in case the pages are not contiguous. Then I release the lock.

To implement get_value, I first put a mutex and check if the address+size is within the bounds of something malloced early, if the address is not, it returns, if the size is not, then I make the size the last address in that allocation. For example, the user wants to put something 10 bytes, but there is only 8 left, I make the size 8. After that, I check to see if the first address and last are in the same page, if they are, I copy the whole thing from physical memory to the address provided and return, if not I increment byte by byte in case the pages are not contiguous. Then I release the lock.

Mat mult was implemented for us.

2. THIS ASSUMES NOT ADDING TO TLB AFTER PAGE MAP OR ELSE THE OUTPUT WILL BE 0

Test.c:

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 2000, 3000, 4000
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 miss rate 0.007444
```

Mtest.c:

```
Allocated Pointers:
5000 2000 8000 e000 b000 1d000 2c000 23000 1a000 29000 11000 14000 17000 20000 26000
```

```
initializing some of the memory by in multiple threads
Randomly checking a thread allocation to see if everything worked correctly!
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 multiplications in multiple threads threads!
Randomly checking a thread allocation to see if everything worked correctly!
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
Gonna free everything in multiple threads!
Free Worked!
TLB miss rate 0.024390
```

3. Test.c
   32768 page size: Same as above except it says addresses of the allocations: 10000, 18000, 20000. Same TLB miss ratio as well.
   65536 page size: Same as above except it says addresses of the allocations: 20000, 30000, 40000. Same TLB miss ratio as well.
   1024 TLB: Same as above
   2048 TLB: Same as above

   Mtest.c
   15 threads 10000 mem alloc, 32768 page size, 512 TLB entries: Same as above except it says the addresses are: 10000 18000 20000 30000 28000 38000 40000 50000 48000 58000 60000 68000 80000 70000 78000 and the TLB ratio is 0.008264.
   15 threads 10000 mem alloc, 65536 page size, 512 TLB entries: Same as above except it says the addresses are: 80000 20000 90000 a0000 f0000 c0000 d0000 30000 e0000 100000 40000 70000 50000 60000 b0000 and the TLB ratio is 0.008264.

15 threads 10000 mem alloc, 4096 page size, 1024 TLB entries: Same as question 2 except ratio is 0.008264.

54 threads 10000 mem alloc, 8192 page size, 512 TLB entries, 20 matrix size: TLB ratio is 0.000340, and a 20 by 20 matrix with 1's and also with 20's appears

54 threads 1000000 mem alloc, 8192 page size, 512 TLB entries, 20 matrix size:

```
Allocated Pointers:
4000 1154000 fa000 3dc000 2e6000 4d2000 5c8000 6be000 7b6000 8ac000 9a2000 a98000
b8e000 c84000 d7a000 e70000 f68000 105e000 1f0000 124a000 1340000 1436000 152c000
1622000 171a000 1810000 1906000 19fc000 1af2000 1be8000 1cde000 1dd4000 1eca000
1fc2000 20b8000 21ae000 22a4000 239a000 2490000 2586000 267c000 2774000 286a000
2960000 2a56000 2b4c000 2c42000 2d38000 2e2e000 2f26000 301c000 3112000 3208000
32fe000
initializing some of the memory by in multiple threads
Randomly checking a thread allocation to see if everything worked correctly!
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 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 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 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
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 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 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 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
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 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 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 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
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 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 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 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 multiplications in multiple threads threads!
Randomly checking a thread allocation to see if everything worked correctly!
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
```

```
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
Gonna free everything in multiple threads!
Free Worked!
TLB miss rate 0.020639
```

From these miss rates and outputs, it shows that the benchmarks work because with a different number of matrix sizes, TLB sizes, thread numbers and page sizes it still works. From the ratios, they are low because the matrix multiplication tends to use only the same few pages to store the data in. In addition, in my getvalue and putvalue function, when it is near the boundaries, it uses byte by byte in order to keep track of noncontiguous memory, furthering lowering the ratios. With a higher allocation size, the ratio goes up because of free using the function translate to find all the pages to free. With more threads though the number goes down because each thread does the same thing, so the number is further divided by the number of threads.

4. No issues that I am aware of