CS 416: Project 3 Report

Sanjana Pendharkar (sp1631) & Vaishnavi Manthena (vm504)

April 12, 2021

1 Part 1: Detailed Logic of how we implemented each virtual memory function.

Important data structures we used:

- Struct page: this just has an array of unsigned longs and the total size of the array is equal to the page size. It is used to allocate a physical memory in the form of an array of pages.
- pBitmap and vBitmap which are used to store the availability of physical pages and page table entries respectively
- struct tlb (tlb_store): This just has a 2D array with 2 columns and 'TLB_ENTRIES' number of rows. First column stores virtual page numbers. Second column stores corresponding physical page numbers.

Description of TLB functions:

- int addTLB(int vpn, int ppn): We find the set (tlb row) into which we need to insert this translation by getting the value of vpn % TLB ENTRIES. In this row we insert the translation of vpn (virtual page number) to ppn (physical page number).
- int check TLB(void * va): We get the virtual page number by using bit shifting to remove offset bits from va. We then index into that particular set (by using modulo calculation as mentioned before). If the virtual page number in that set does not match, then we return -1. Else, we return the sorresponding physical page number.
- void print TLB missrate(): We calculate miss rate by decimal division of global variable tlb misses by global variable tlb lookups.

Overall idea of using locks: We only had one pthread mutex t variable called 'mutex.' For the purpose of making the implementation thread safe we put locked and unlocked the mutex at the starting and ending of (respectively) almost every function that can be called by the user (a malloc, a free, put value,

get value). I made sure to do unlock before any return statements in the middle of the function.

Note: whenever we use the term 'virtual page number' we are actually just referring to the particular page table entry.

1.1 set physical mem

In this function, we perform all the required data structure allocations and initializations. Firstly, we set calculate all the important global variables through the function setGlobals(). This includes the variables 'offsetBitNum,' 'innerLevelBitNum,' and outer level bitnum.
Calculation for offsetBitNum: $log_2(PGSIZE)$.
Calculation for innerLevelBitNum: $log_2 \frac{PGSIZE}{sizeof(pte_t)}$
Calculation for outerLevelBitNum: $32 - innerLevelBitNum - offsetBitNum$

However in case the PGSIZE id: $128K$
Calculation for innerLevelBitNum: $32 - outerLevelBitNum - offsetBitNum$

Calculation for outerLevelBitNum: $\frac{headerBitNum}{2}$

Next, we allocated memory for the physical memory using malloc. The pointer to this allocation is 'physicalMemory.' As a result this pointer will point to an array of struct pages with a length equal to the number of physical pages.

Next we allocated memory for the virtual and physical bitmap. Required calculations: Number of bits needed for virtual bitmap: Number of virtual pages = MAX_MEMSIZE/PGSIZE.
Number of bytes for virtual bitmap: (Above value)/8
Number of bits needed for physical bitmap: Number of physical pages = MEMSIZE/PGSIZE.
Number of bytes for physical bitmap: (Above value)/8
Based on the above calculations we allocate (through malloc) character arrays of appropriate lengths for the global variables 'pBitmap' and 'vBitmap.' Finally, we call a helper function 'settingUpPT().'

Basic functionality of settingUpPT(): Reserve the last physical page (by setting the last bit in pBitMap) for the page directory. In the page directory initialize all entries as '-1,' meaning the corresponding page table has not been reserved yet.

1.2 translate

Our translate function takes a virtual address and the page directory's starting address and performs translation to return the physical address. We first get

the offset by using bit manipulation and the input 'va'. We then increment tlb lookups because we are checking to see if the virtual address is present in our tlb. If it is present, we add the offset to our tlb's physical address (which we get by using the physical page number stored in the tlb) and return the value casted as a page table entry. We then use outer level bits to index into the outer level. If there is a tlb miss, we continue. We get the outer level index by finding the unsigned int corresponding to outer level bit num bits of virtual address through right bit manipulation. To get the inner level index, we get the unsigned int corresponding to the middle inner level bit number bits of virtual address. We then remove the lower order bits, and find inner index. Now, we will check that this page table entry is valid using bitmap. The bit that we need to check is $outer\_Index \cdot innerLevelBitNum^2$, so this is out index. Then, we get the bit in virtual bit map at the position of index. If bit does not equal to 1, this means the page table entry is not valid, so we return null. If the page table entry is valid, then we need to walk through the page table to get the valid mapping. We will use the outer and inner index to get the physical page number. We do so by finding the page directory entry to be pgdir and outerIndex sum. We know that the entry of page directory has the page number of the inner level page table. Using this entry we find the address of the inner level page table. This is basically the address of the struct page in the physical memory array at location equal to the page number of the inner level page table. Next, we find the page table entry by adding the address of the inner level page table and the innerIndex. The value at this entry is the page number of the physical page 'pnum'. We find the address of the physical page by finding the address of the entry within physical memory array at pnum. We now know that our physical address is the address of our physical page plus the offset, casted into an unsigned long. Finally, we are adding this entry (virtual page number to physical page number) to tlb, and increment tlb_misses. We then return the physical address (physicalAddress).

1.3 page map

First, we check if this mapping is already present in the TLB incurring a lookup. This is done using the check_TLB function. If it is present in TLB the function returns, else the number of misses is incremented by 1 and we proceed forward.

We first use bit manipulation to get the outer index (top bits of virtual address) and inner index (middle bits of virtual address). Then we use the outer index to index into the particular entry of the page directory. We check the page number of page table entry stored here. If the number here is 0, then that page table has not been reserved in physical memory yet. So, in this case we reserve this page, by traversing through the physical bitmap backwards and setting the first free bit. The index of this free bit (a physical page number) is now used as page number of page table corresponding to this page directory entry. So, we insert this page number into the page directory entry. In case, the page directory entry was originally not -1 this means that the entry has the

page number of the corresponding page table which has already been reserved previously.

Now we go to the page table address. Here, using inner index we index into a particular page table entry. In here we insert the physical page number of 'pa'. According to our implementation 'pa' is not a valid physical address. It is actually just: Physical page number left bit shifted by offsetBitNum. So, to get physical page number from pa we do: pa right bit shifted by offsetBitNum. Similarly, to get virtual page number we do 'va right bit shifted by offsetBitNum.'

Now we insert this mapping (virtual page number to physical page number) into the TLB using add_TLB() function.

1.4 a malloc

Initially, we would check if the physical memory and required data structures like bitmaps have been allocated. The global variable 'initialized' keeps track of whether this has happened or not. If initializing these structures has not yet happened we call the set_physical_mem() function which takes care of this and update the 'initialized variable.'

Next, we calculate the number of pages required based on the PGSIZE and parameter num_bytes. Number of pages needed will be num_bytes/PGSIZE and 1 will be added to this if there is any remainder. We store this in a variable 'numPages.'

Then, we check if there are 'numPages' number of free physical pages and record these page numbers. To do this we traverse across each bit of the physical bit map and record the indexes of the bits which are currently free until we find numPages free pages. These free physical page numbers are stores in an array called 'physicalPages.' If numPages number of physical pages were not found then we return NULL. If we did find them, we move forward.

Then, we check for the availability of 'numPages' contiguous page table entries by using the get_next_avail function and sending 'numPages' as the argument. If we get an invalid virtual page number (-1) from this function then we return NULL, else we move forward. We record the virtual page numbers (page table entry number) of the first page of allocation (the one returned by get_next_avail) and last page of allocation (firstPage + numPages -1).

Later through a for loop (to simultaneously traverse through the continuous page table entries and also the free physical pages numbers stored in physicalPages) we complete the following tasks:

- Set the corresponding bits in vBitmap and pBitmap.
- We call the page map function to establish the mapping between the virtual address and physical address. Virtual address is 'virtual page num left shifted by number of offset bits'. The physical address 'physical page num left shifted by number of bits'. We know this manipulation to obtain

physical address does not give the actual physical address. However, it was easier for the overall logic to send the value 'physical page num left shifted by number of bits' to page map. Over there we obtain physical page number by taking the input physical address and doing 'physical address right shifted by number of offset bits.'

- Finally by getting the virtual address from the first virtual page number we do 'firstPage left shifted by number of offset bits.' We return the void * version of this address.

1.5 a free

Our a free function first finds the firstPage to free and the number of pages to free. Then, we check if we exceed the first page and increment as necessary. Next, we want to make sure that memory from our virtual address to the sum of virtual address and size is valid. We do this by getting the bit at the index of virtual bitmap for the amount of pages indexed by that entry and check if the bit is valid (meaning that it equals 1). If it returns 0, we then return. If all the virtual pages to free are valid, we loop through the virtual pages to perform the following functions:

- We translate the virtual address to the physical address, set bit at index i (this loops variable) in virtual bitmap to 0, and set bit at index of physical page in physical bit map to 0.
- We then remove the virtual pages from TLB structure by setting all entries to -1 (invalid).
- The main purpose of this function is to make sure that the entry exists in our virtual address, and then removing the entry from physical bit map, virtual bit map and TLB to make sure that the memory allocation for it does not exist.

1.6 put value

Our put value function copies the content of a value 'val' into a physical page using the translate function, which takes a virtual address and turns it into a physical address.

First we make sure that all the virtual pages corresponding to 'va' to 'va+size' are valid. To do this we check if the bits are set in vBitmap for the virtual pages corresponding to va to va + size. If any of these bits are 0, we return. Now, after this point, we know all virtual addresses from virtual address start to virtual address and size sum are valid. Now we type casted our virtual address, physical address (we got this from translate) and val pointer to a char pointer. Then through a while loop we copy the value of 'val' to the physical address byte

by byte. During this loop at any point if we encounter a new virtual page (indicated by a 0 offset) we use translate again to find the corresponding physical page to which we can continue copying the data to.

1.7 get value

Our get value function has the purpose of putting the values pointed to by virtual address inside the physical memory at the val address.

The implementation of this is function is identical to put value except this time we copy data in the other direction (from physical memory to val). The rest of the ideas including checking validity of virtual pages and using translate whenever necessary is the same.

In-depth Explanation of our function:
We first find the physical address using our translate function, which takes a virtual address and transforms it into a physical address. We then check for the validity of the virtual pages from virtual address to virtual address and size sum. We set two vpns, one for the virtual address at the start, and one for the virtual address at the end value (which corresponds to virtual address + size). Then, we iterate through vpn1 until vpn2 to get the bit at the index vpn from our virtual bit map. If bit is equal to 0, this means the virtual address is not valid, so we just return. Once we pass this loop, we know that all virtual page numbers from virtual address start to virtual address and size sum are valid. Then, we iterate from 0 to size, increasing virtual address T, physical address and virtual address by 1.

Note:
For both set value and get value, all the tlb related calculations (modifying lookups, misses, checking tlb, adding entries to it) are taken care of the translate function itself.

1.8 get next avail

This function tries to find a contiguou set of num_pages (the input) page table entries.
We traverse through the virtual bitmap to find a continuous set of bits which are 0 (indicating they are free and can be allocated). We return the first bit index (i.e, the first available virtual page number) of this set if one exists. If not we return -1.

1.9 mat_mult

Our matrix multiplication function receives two matrices, mat1 and mat2, as an argument with size argument representing number of rows and columns. After performing the multiplication, we copy our result to the answer parameter.

We traverse through the answer array using the variables 'i' and 'j.' To fill the value at answer[i][j] we do the following:

We use a variable 'k' to multiply the ith row of mat1 with the jth column of mat 2. To get the value at any location in mat1 and mat2, we use the get value function. To put the result in answer[i][j] we use put value. To index into a particular location of any array we use the formula given in project description.

2 Part 2: Benchmark output for Part 1 and the observed TLB miss rate in Part 2.

While benchmarking our output for Part 1, we changed the following variables: (i) MEMSIZE, (ii) PGSIZE, (iii) TLB_ENTRIES, (iv) Thread number in case of multi test.

Details:

- mem size = 1 gb
- page size = 4k
- tlb entries = 512
- thread number = 15

picture_1_test:

```
Allocating three arrays of 400 bytes
completed malloc for acompleted mallocsAddresses of the allocations: 0, 1000, 2000
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 0.007371
```

picture_1_multi_test:

```
Allocated Pointers:
0 3000 6000 9000 f000 c000 12000 1b000 15000 18000 1e000 21000 24000 27000 2a000
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 0.023797
```

Picture 3: (use same set)

Details:

- mem size = 1 gb
- page size = 4k
- tlb entries = 512
- thread number = 3

picture_3_multi_test:

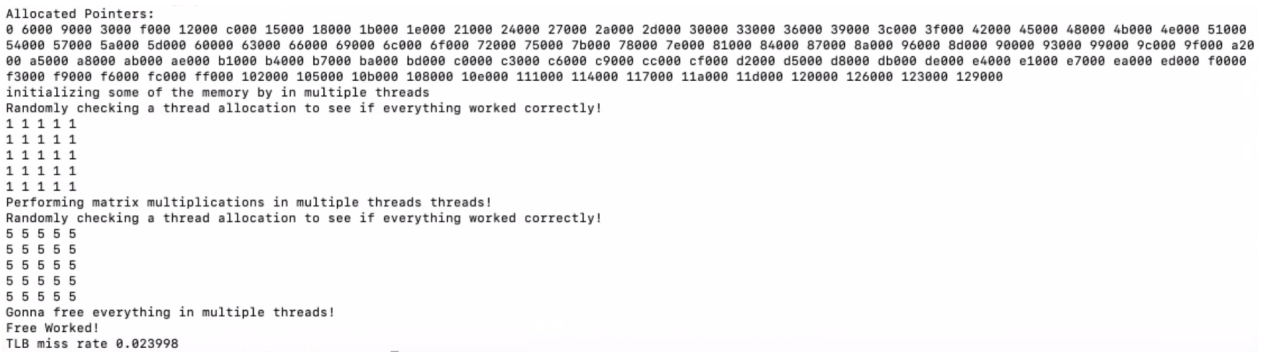
```
Allocated Pointers:
0 3000 6000
initializing some of the memory by in multiple threads
Randomly checking a thread allocation to see if everything worked correctly!
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Performing matrix multiplications in multiple threads threads!
Randomly checking a 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 0.021480
```

picture_4_multi_test: (only thing that changed is thread number, given in pic)

Details:

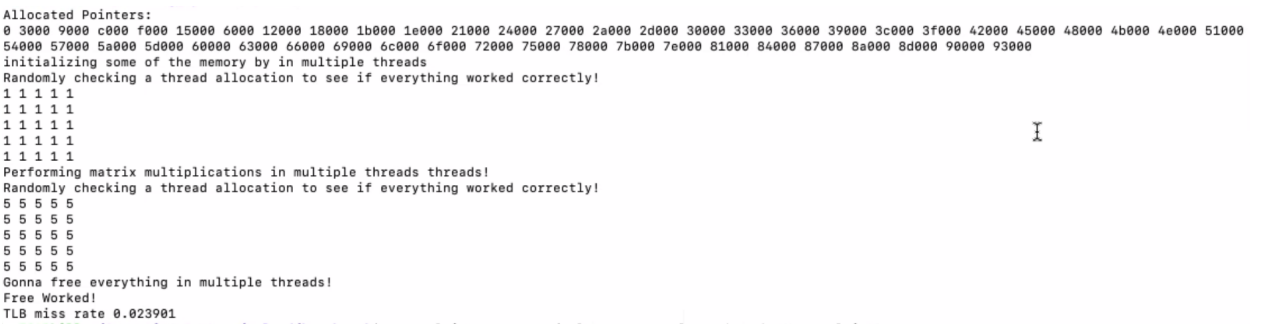
- mem size = 1 gb
- page size = 4k
- tlb entries = 512

- thread number = 100



picture_5_multi_test:
Details:

- mem size = 1 gb
- page size = 4k
- tlb entries = 512
- thread number = 50



Picture 6:
Details:

- mem size = 1 gb
- page size = 4k
- tlb entries = 512
- thread number = 15
- alloc size for multi test = 20000

```
Allocated Pointers:
3c000 a000 14000 2d000 0 32000 23000 19000 1e000 5000 f000 28000 46000 37000 41000
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!
Some Problem with free!
TLB miss rate 0.038442
```

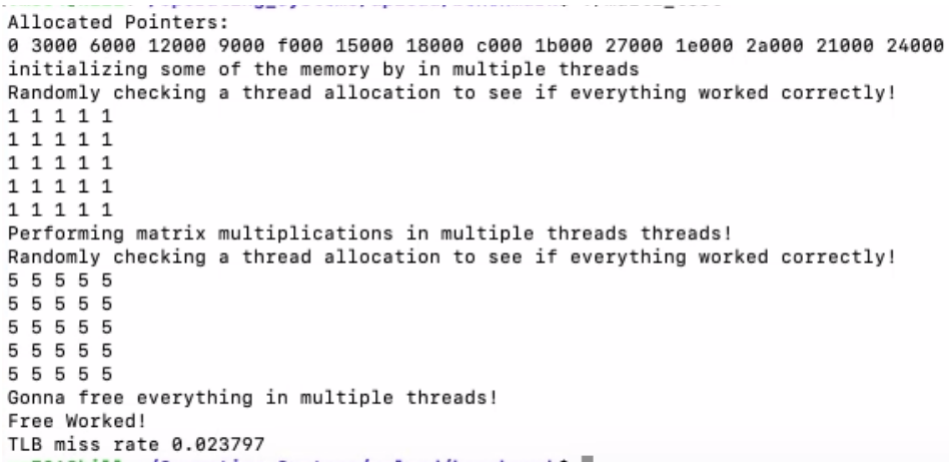
Picture 7:
Details:

- mem size = 2 gb
- page size = 4k
- tlb entries = 512
- thread number = 15

picture_7_test:

```
Allocating three arrays of 400 bytes
completed malloc for acompleted mallocsAddresses of the allocations: 0, 1000, 2000
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 0.007371
```

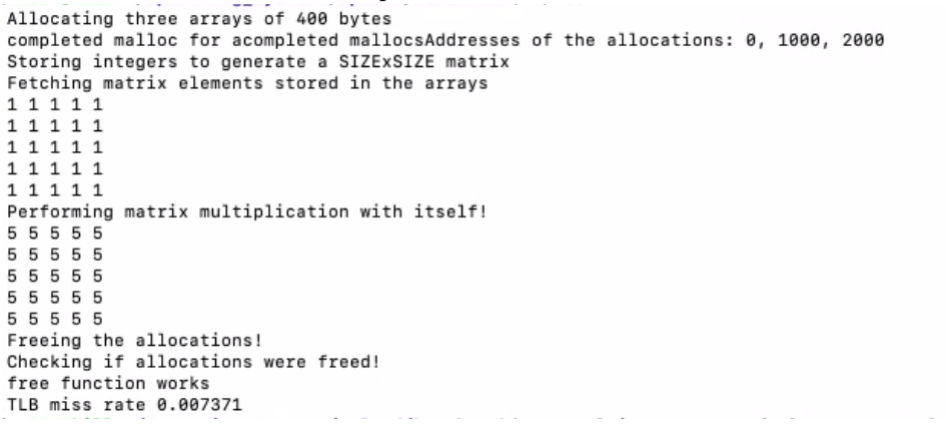
picture_7_multi_test:



Picture 8:
Details:

- mem size = 1 gb
- page size = 4k
- tlb entries = 256
- thread number = 15

picture_8_test:



picture_8_multi_test:

```
Allocated Pointers:
0 3000 9000 f000 12000 c000 18000 6000 2a000 15000 1b000 24000 27000 21000 1e000
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 0.023797
```

3 Part 3: Explain how your code provides support for different page sizes (in multiples of 4K).

We supported different page sizes by considering the PGSIZE as a general value that is a multiple of 4K. We did not specialize any part of the implementation towards any particular value(s) of PGSIZE. The below results show our support for different page sizes.

3_Picture 7_test:

Details:

- mem size = 1 gb
- page size = 8k
- tlb entries = 512
- thread number = 15

```
Allocating three arrays of 400 bytes
completed malloc for acompleted mallocsAddresses of the allocations: 0, 2000, 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 0.007371
```

3_Picture 7_multitest:

```
Allocated Pointers:
0 4000 8000 c000 10000 38000 14000 18000 1c000 20000 24000 28000 2c000 30000 34000
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 0.016120
```

3_Picture 8_test:

Details:

- mem size = 1 gb
- page size = 32k
- tlb entries = 512
- thread number = 15

```
Allocating three arrays of 400 bytes
completed malloc for acompleted mallocsAddresses of the allocations: 0, 8000, 10000
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 0.007371
```

3_Picture 8_multitest:

```
Allocated Pointers:
0 8000 20000 10000 18000 28000 30000 38000 40000 48000 50000 58000 60000 68000 70000
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 0.008192
```

3_Picture 9_test:

Details:

- mem size = 1 gb
- page size = 64k
- tlb entries = 512
- thread number = 15

```
Allocating three arrays of 400 bytes
completed malloc for acompleted mallocsAddresses of the allocations: 0, 10000, 20000
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 0.007371
```

3_Picture 9_multitest:

```
Allocated Pointers:
0 10000 20000 30000 40000 50000 60000 70000 80000 e0000 90000 a0000 d0000 b0000 c0000
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 0.008192
```

3_Picture 10_test:

Details:

- mem size = 1 gb
- page size = 128k
- tlb entries = 512
- thread number = 15

```
Allocating three arrays of 400 bytes
completed malloc for acompleted mallocsAddresses of the allocations: 0, 20000, 40000
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 0.007371
```

3_Picture 10_multitest:

```
Allocated Pointers:
0 60000 20000 40000 80000 a0000 c0000 e0000 100000 120000 140000 160000 180000 1a0000 1c0000
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 0.008192
```

Some explanations about benchmark results:

- The allocations are being set as expected. Consider picture 2 or question 2. We changed the size of allocations to 20000. $20000/4096 = 4.88 = 5$ pages per allocation.
- We can see that 5 continuous virtual pages are being allocated for each allocation. This can be seen by the allocated addresses in the output which look like 0 5000 a000 etc...
- Also, as can be seen by the pictures in question 3, the TLB rate tends to go down as we increase the page size. This is because with increasing page sizes a tlb translation will be useful for a longer amount of time.
- As can be seen by some of the pictures in question 2, our code works while changing the memory size and number of tlb entries as well.

4 Part 4: Explain possible issues in your code (if any).

- We did not take care of the optional fragmentation part in case this would cause any issue.
- Our code may not work so well for more than 2GB, since this was not required for the case of this project as specified by the instructor.
- We customize some of the signatures for the TLB related functions except for print TLB, which may be problematic if the correct type is not returned to or passed within them.
- The critical sections of our mutex locks may be reduced further for real life applications, but for the sake of this project we though our performance was pretty fast anyways.

Extra Credit Explanation:

```
Allocating three arrays of 400 bytes
a_malloc: 400
These are the number of pages I am looking for:1
current BitMap[00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000]
virtualPageNum: 1048576
This is the first virtual Page that I got from get next avail: 0
This is the last virtual Page that I will allocate: 0
completed malloc for aa_malloc: 400
These are the number of pages I am looking for:1
current BitMap[10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000]
virtualPageNum: 1048576
This is the first virtual Page that I got from get next avail: 1
This is the last virtual Page that I will allocate: 1
a_malloc: 400
These are the number of pages I am looking for:1
current BitMap[11000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000]
virtualPageNum: 1048576
This is the first virtual Page that I got from get next avail: 2
This is the last virtual Page that I will allocate: 2
completed mallocsAddresses of the allocations: 0, 1000, 2000
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!
Segmentation fault
```

Above is our extra credit implementation. So far, we have got everything right except for the free part, which gives a segmentation fault. The reason for our segmentation fault is from the free bit part, in which we are actually freeing our bit. We think this may be an error in the allocation of the bit memory that we are trying to free - maybe the memory of the bit is not allocated for by our program due to some miscalculations with 64 bit addresses.

Note that for our Makefile for the Extra Credit, on the iLabs account, we used the same Makefile for the regular part by keeping both files as my_vm.c. For the extra credit we just removed the m32 flag. However, in exporting the code into our own laptop, we renamed the file to my_vm64.c. The makefile we submitted is just for the regular project and not the extra credit.