Project 3 Report: User-level Memory Management

CS 416/518 – OS Design
Professor Srinivas Narayana

Project Partners
Kush Patel (kp1085)
Pavitra Patel (php51)

Tested ilab: ilab: kill.cs.rutgers.edu

Q1. Detailed logic of how you implemented each virtual memory function.

A1.

`void set_physical_mem()`

- This method performs three important tasks:
    - Firstly, it allocates physical memory using malloc. A data structure named "page" has been defined that has an array whose size is equal to size of one page. Physical memory is malloced using (total physical pages possible * sizeof("page" data struct)).
    - Based on the page size, now we calculate number of inner bits, outer bits, and offset bits based on page size.
        - Offset bits can be found by log2(PGSIZE)
        - Page directory bits or second level bits can be found using log2((PGSIZE)/sizeof(pte_t))
        - Page table bits or first level bits can be found by (32 – offset bits) – second level bits
    - Now we call initialize_physical_virtual_bitmaps() method to initialize physical and virtual bitmaps.

`void initialize_physical_virtual_bitmaps()`

- Now we `(char *)malloc(tot__physical_pages/8)` malloc for physical bitmap. We perform the operation (total physical pages/8) because we only need 1 bit to represent a physical page and 1 char has 8 bits. Similarly, we malloc bitmap for virtual bitmaps as well.
- We initialize each bit of both bitmaps by '\0'.
- Now we reserve the last page in physical memory to represent page directory. We set the bit of that index in physical bitmap to 1. As a page represents a data structure "page", that has an array of size PGSIZE. We iterate over that array and initialize each index to -1. In other words, there is no page table associated in any of the page directory indexes as of now.

`void set_bit_at_index(char *bitmap, int index)`

- This function sets the bit to 1 at specified index in a given bitmap. It calculates the byte index using the formula index/8, bit offset within that byte using the formula index%8, and then performs the operation `bitmap[byte_index] |= (1 << bit_offset)` to set value to 1 at that index.

`int get_bit_at_index(char *bitmap, int index)`

- This function retrieves the value of a specific bit at the specified index in a given bitmap. It calculates the byte index using the formula index/8 and the bit offset within that byte using the formula index%8.

The result is obtained by performing the operation `(bitmap[byte_index] >> bit_offset) & 0x1`, extracting the targeted bit.

`void reset_bit_at_index(char *bitmap, int index)`

- This function resets (sets to 0) the bit at the specified index in a given bitmap. It calculates the byte index using the formula index/8 and the bit offset within that byte using the formula index%8. The result is obtained by performing the operation bitmap[byte_index] &= ~(1 << bit_offset), which sets the value to 0 at that index.

`int add_TLB(int va, int pa)`

- For TLB, we have a data structure that has a 2D array. In the array, element array[i][0] represents virtual page number that will be stored for a particular entry that needs to be saved. The element array[i][1] represents physical page number associated with that entry. The method add_TLB adds such entry into the 2D array of TLB data structure. The function makes sure if the address are valid and then calculates index using the formula `(size_t)va % TLB_ENTRIES.` At this index, virtual to physical translation will be saved.

`int remove_TLB(int st, int en)`

- Similar to how we add_TLB, here we remove_TLB. We remove the virtual pages to physical pages translation from the TLB.

`unsigned long check_TLB(void *va)`

- The check_TLB function looks up for the provided virtual address (va). The function calculates the virtual page number based on the given virtual address, taking into account the offset bits. Then, it determines the TLB index checker_t by applying the modulus operation to the virtual page number with respect to the number of TLB entries. The function then examines the TLB entry at the calculated index to verify whether the stored virtual page number matches the calculated one. If a match is found, the function retrieves and returns the associated physical page address from the TLB. If no valid translation is present in the TLB for the given virtual address, the function returns -1.

`void print_TLB_missrate()`

- The print_TLB_missrate function computes and displays the miss rate of the TLB. It is calculated by dividing the number of TLB misses miss__TLB by the total number of TLB lookups look__TLB. The result is then formatted to double precision floating-point value representing the miss rate. Then, the function outputs this miss rate using the fprintf function.

`pte_t *translate(pde_t *pgdir, void *va)`

- The translate function is responsible for translating a virtual address (va) to a physical address. The translation process involves two main steps: TLB lookup and page table lookup.
  - TLB Lookup:
    - First, we find the translation in the TLB by calling the check_TLB function.
    - If a translation is found in the TLB (TLB_pn != -1), then we get the corresponding physical address from the TLB.
  - Page Table Lookup:
    - In case of a TLB miss, we perform page table lookup.
    - First, we get the virtual page number (get_vpn) from the virtual address (va), and then we calculate the corresponding page table index.

        - Next, we check if the virtual page is present in the virtual memory bitmap (my_virtual__bit_map). If not, return NULL.
        - If the virtual page is present, we calculate the page directory entry (get_pde) and the page table entry (get_PTE).
        - The physical page number (pg_number) is obtained from the page table entry, and the physical address is calculated by combining the page base address with the offset from the virtual address.
        - The translation is then added to the TLB using the add_TLB function.
        - The final result is the physical address corresponding to the provided virtual address.
        - If the translation was successful, we return the physical address; otherwise, return NULL.
        - We also keep track of TLB misses and uses the look__TLB and miss__TLB counters to calculate and print the TLB miss rate.

```
int page_map(pde_t *pgdir, void *va, void *pa)
```

- The page_map function sets a page table entry in the provided page directory (pgdir) based on the given virtual address (va) and physical address (pa).
- TLB Lookup:
    - The function first tries to find the translation in the TLB by calling the check_TLB function, using the virtual address (va).
    - If the translation entry is found (found_pg == get_ppn), which indicates that the virtual address is already mapped to the physical address. We return as the mapping already exists.
- Page Table Entry Setup:
    - Now, we set up the page table entry as there is no existing mapping for the virtual address.
    - It calculates the page table index based on the virtual address and checks if the corresponding page table entry exists in the page directory.
    - If the page table entry is not present, it searches for a free page in physical memory to map the page table entry.
    - Once a free page is found, the function updates the page directory entry and sets the page table entry to the physical page number (get_ppn).
- TLB and Bit Map Updates:
    - Now, we call add_TLB function to map the new mapping by providing the virtual page number (get_vpn) and physical page number (get_ppn).
    - The physical memory bitmap (my_physical__bit_map) is also updated as we have allocated the physical page for the page table entry.
- We return -1, if the mapping is incomplete.

```
unsigned long get_next_avail(int num_pages)
```

- The get_next_avail function returns the starting index from where consecutive virtual pages are available.
- To find consecutive virtual pages, we loop through the virtual page bitmap. The outer loop iterates through the virtual pages, and the inner loop finds consecutive free pages.
- Next, we check if the current virtual page at index checker_i is unset (bit value is 0).
- If a free page is found, inner while loop is executed to see if consecutive free pages are available or not.
- If we find free pages (checker_t == num_pages), the function sets start_p to the index checker_i and returns this starting index.
- If free pages are not found, we return the original value of start_p (initialized to 0), indicating that there are not enough consecutive free pages in the virtual address space.

```
void *t_malloc(unsigned int num_bytes)
```

- The t_malloc function is allocates pages requested by a user.
- First, we checks if physical memory is initialized; if not, set_physical_mem function is called to allocate and initialize physical memory.
- Next, we find the number of pages (npages) required based on the specified number of bytes requested by the user.
- Page Allocation:
  - We use a loop to find consecutive available physical pages by iterating through the physical page bitmap (my_physical__bit_map).
  - Now the temp_pp_arr array is set to have indices of available physical pages.
  - We return NULL, if free pages not found.
- Virtual Page Allocation and Mapping:
  - We execute get_next_avail function to get the starting index of virtual pages found.
  - Now we map virtual pages to physical pages by updating the virtual and physical bitmaps, and then we call the page_map function to set page table entries in the page directory.
- Return Value:
  - If the allocation was not possible, we return NULL.
  - If successful, we compute and return the starting virtual address (va) and return it.
- Multi-threading Consideration:
  - Pthread_Mutex is also used (pthread_mutex_lock and pthread_mutex_unlock) to ensure thread safety during memory allocation.

```
void put_value(void *va, void *val, int size)
```

- As soon as we enter the put_value we call for the mutex_lock, to ensure the thread safety, and at every return we unlock the mutex. Moreover, we assign source and destination char *. The source and destination declares the pointer to character.. Making sure of the virtual page that it is been mapped. The function is operating the copying data from the source to destination virtual addr. It will be perform in loop util all the data has been occupied. The copying is performed using memcpy. At the end of the loop, we increment the source, destination, and decrement the size to flow with the correct index to copy.

```
void get_value(void *va, void *val, int size)
```

- The get_vlaue function is the counter part of just like put_value. Also, we have implemented the mutex lock and unlock for the thread safety. It take the data from the source virtual addr (va) and copies to destination to (val). We check before going into the loop, that the virtual pages respectively source adr and its size are been mapped properly. If not then we return. Now, moving to loop where memcpy is been used to copy the data from physical page to destination addr until the loop finish iterating through the data size passed parameter.

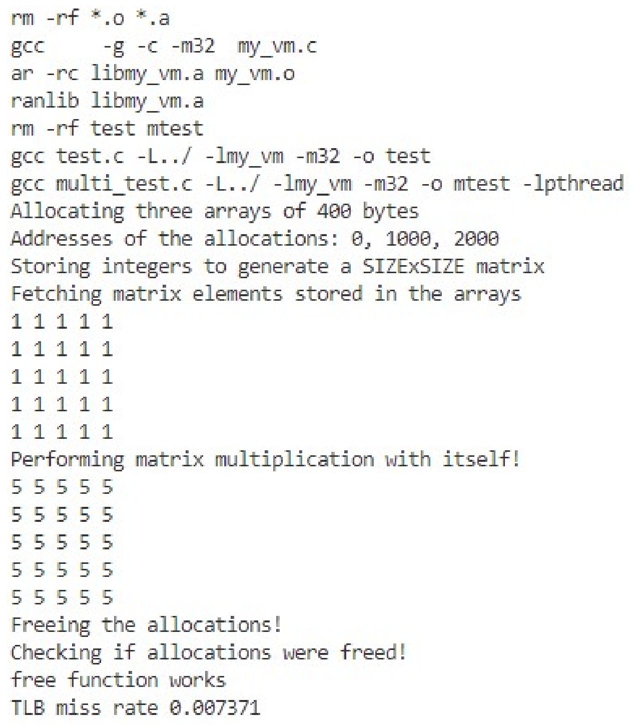
```
void t_free(void *va, int size)
```

- The t_free is called upon to deallocation of memory with the given parameters f virtual addr (va) and size. We are implementing, the mutex lock and unlock for thready safety. Calculating starting page and free pages need to be are npage. Making sure, the entire memory is been checked by virtual bitmap, we make sure that any pages is not allocated then we exits without doing the free. To make sure the integrity of memory free validity. Then unset the bit at the index from the addition of start_p and npages. We are iterating through the pages in the range and reriving the physical addr using translate. Also, clearing the content of the physical memory using the memset. Also, calling the remove_TLB method to update the TLB pages.

Q2. Benchmark output for Part 1 and the observed TLB miss rate in Part 2.

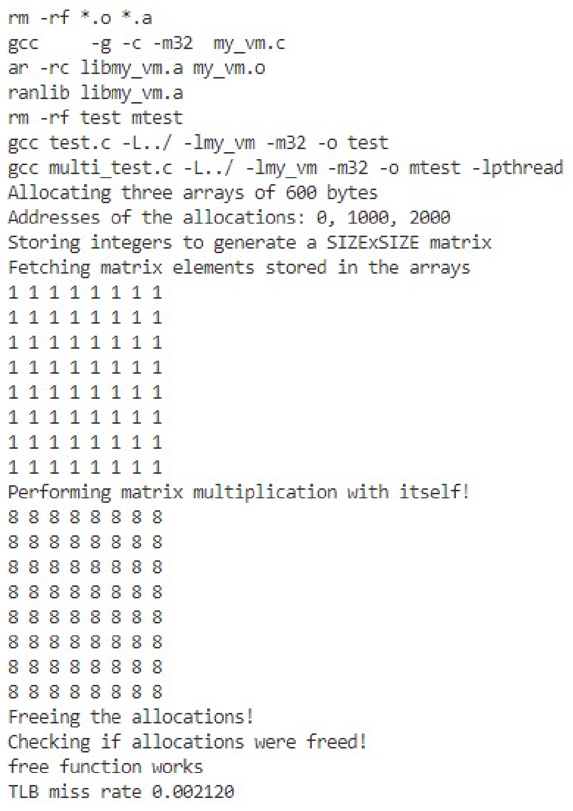
A2.

1. SIZE: 5 ARRAYSIZE: 400



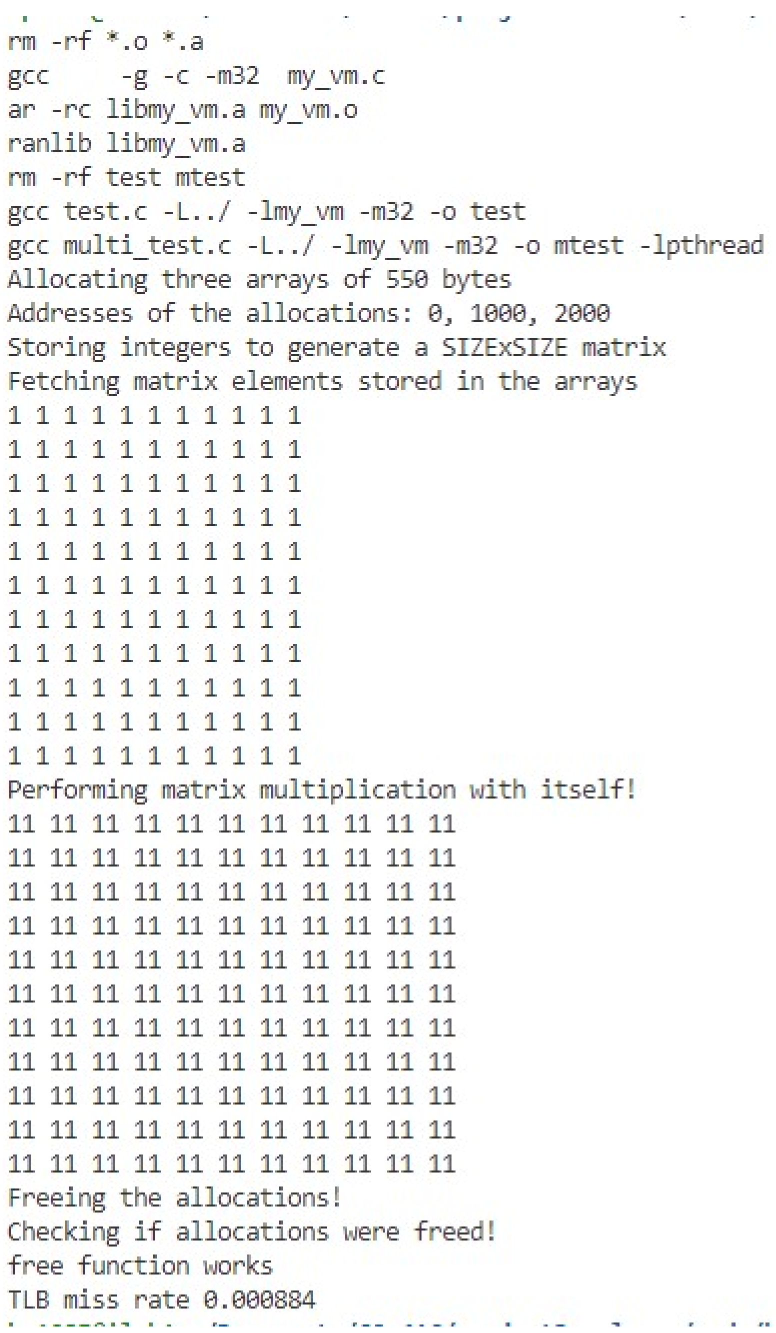
```
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest
gcc test.c -L../ -lmy_vm -m32 -o test
gcc multi_test.c -L../ -lmy_vm -m32 -o mtest -lpthread
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 1000, 2000
Storing integers to 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
```

2. SIZE: 8 ARRAYSIZE: 600



```
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest
gcc test.c -L../ -lmy_vm -m32 -o test
gcc multi_test.c -L../ -lmy_vm -m32 -o mtest -lpthread
Allocating three arrays of 600 bytes
Addresses 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 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
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!
8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8
Freeing the allocations!
Checking if allocations were freed!
free function works
TLB miss rate 0.002120
```

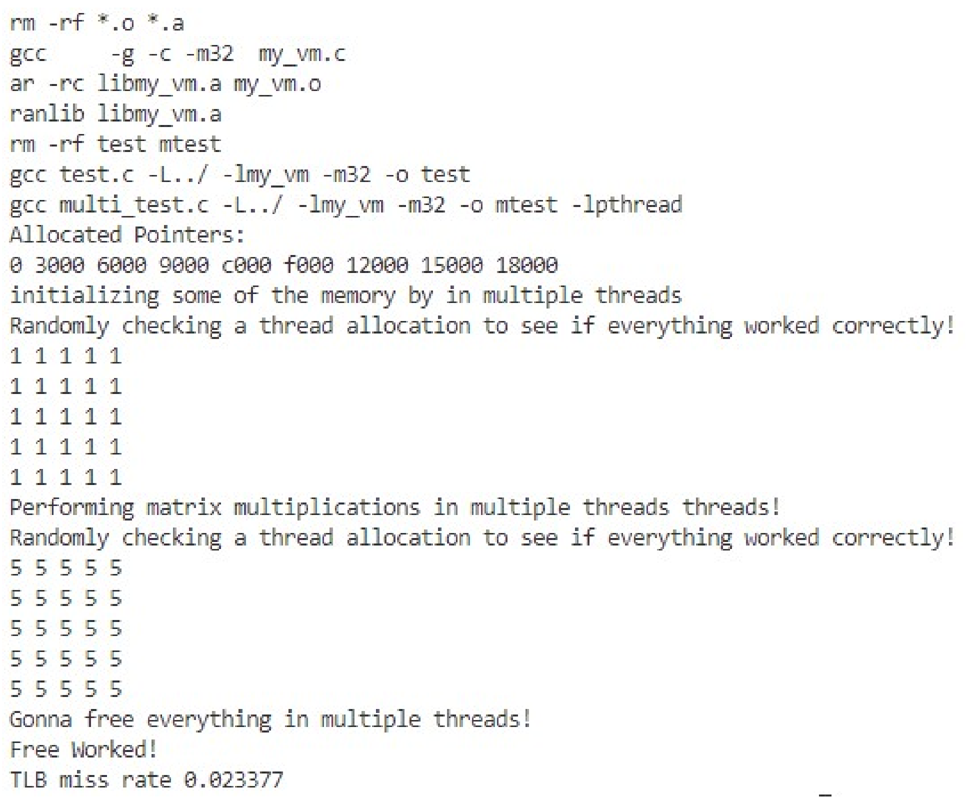
3. SIZE: 11 ARRAYSIZE: 550



```
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest
gcc test.c -L../ -lmy_vm -m32 -o test
gcc multi_test.c -L../ -lmy_vm -m32 -o mtest -lpthread
Allocating three arrays of 550 bytes
Addresses 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 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 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!
11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11
Freeing the allocations!
Checking if allocations were freed!
free function works
TLB miss rate 0.000884
```

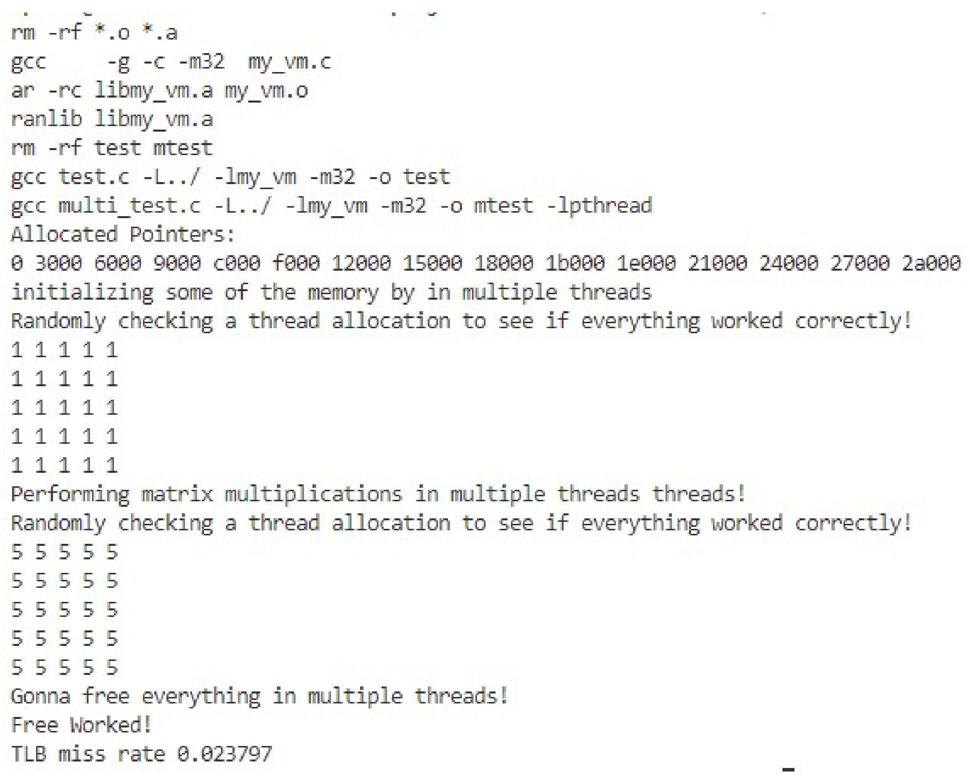
PART: 2

1) Num of threads: 9



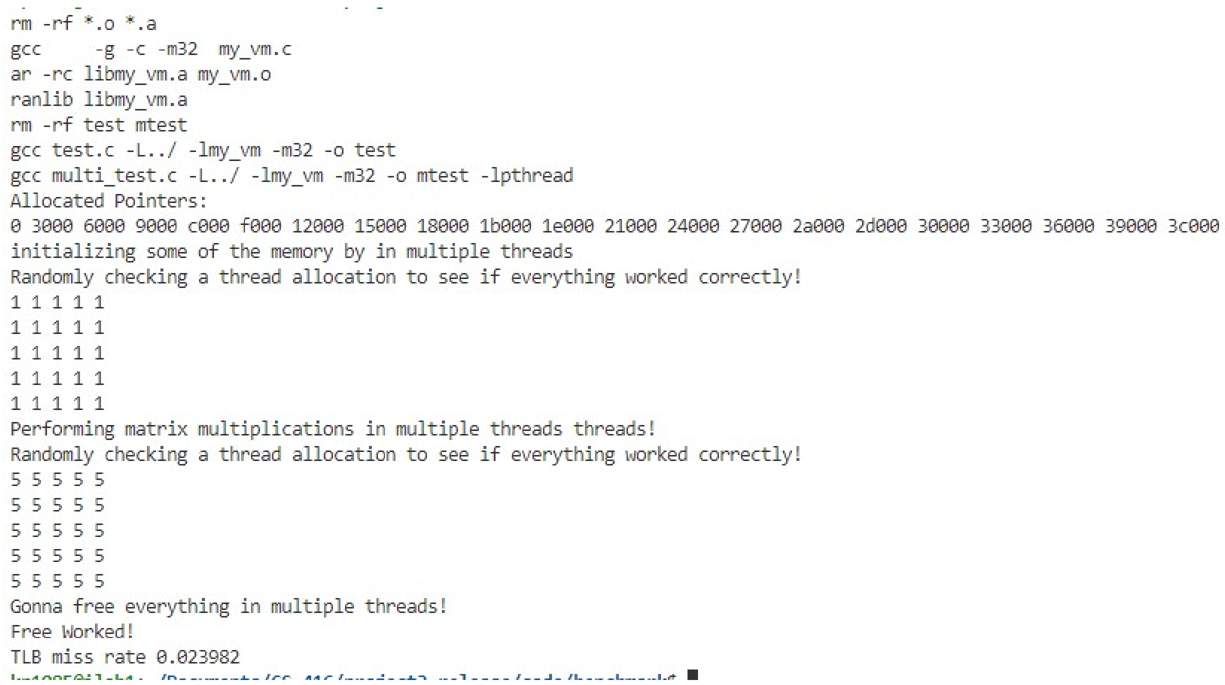
```
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest
gcc test.c -L../ -lmy_vm -m32 -o test
gcc multi_test.c -L../ -lmy_vm -m32 -o mtest -lpthread
Allocated Pointers:
0 3000 6000 9000 c000 f000 12000 15000 18000
initializing some of the memory by 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 0.023377
```

2) Num of threads: 15



```
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest
gcc test.c -L../ -lmy_vm -m32 -o test
gcc multi_test.c -L../ -lmy_vm -m32 -o mtest -lpthread
Allocated Pointers:
0 3000 6000 9000 c000 f000 12000 15000 18000 1b000 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
```

3) Num of threads: 21

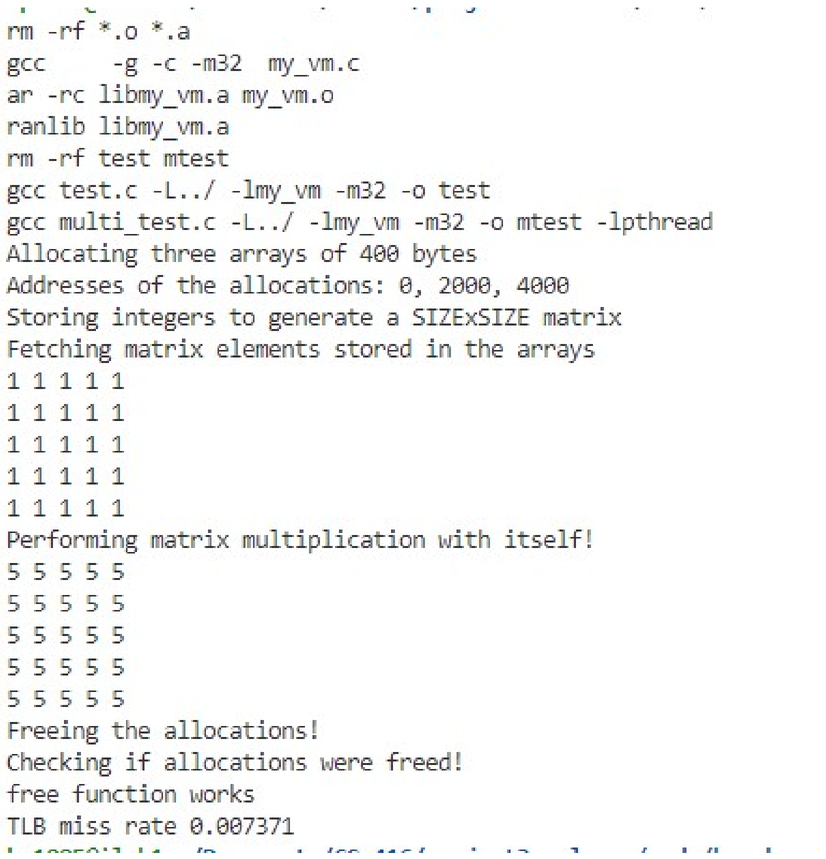

```
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest
gcc test.c -L../ -lmy_vm -m32 -o test
gcc multi_test.c -L../ -lmy_vm -m32 -o mtest -lpthread
Allocated Pointers:
0 3000 6000 9000 c000 f000 12000 15000 18000 1b000 1e000 21000 24000 27000 2a000 2d000 30000 33000 36000 39000 3c000
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 0.023982
```

Q3. Support for different page sizes (in multiples of 4K).

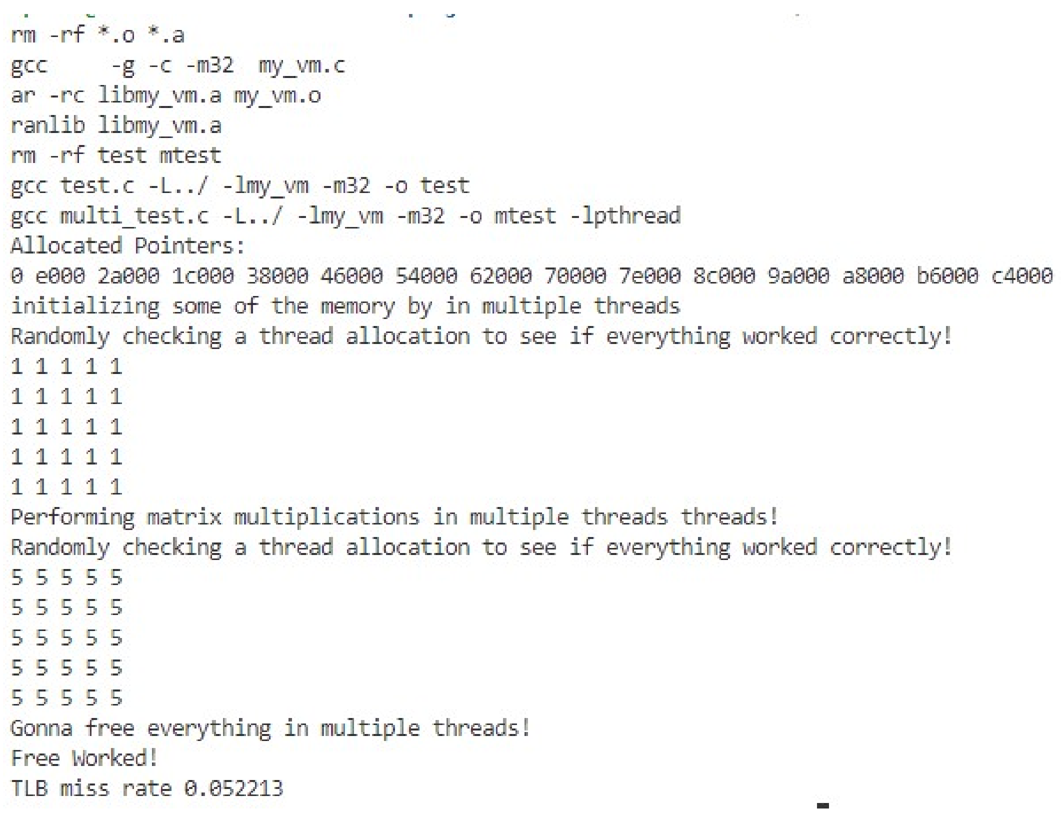
A3.

1) 4096*3

for ./test

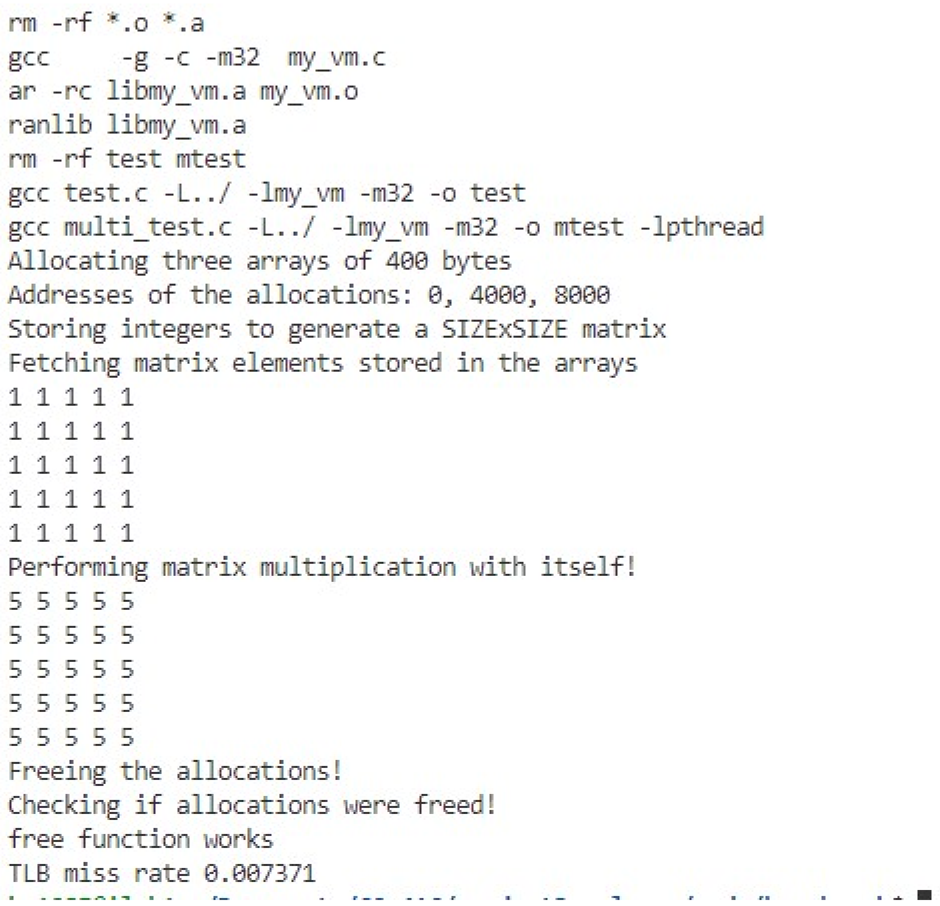

```
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest
gcc test.c -L../ -lmy_vm -m32 -o test
gcc multi_test.c -L../ -lmy_vm -m32 -o mtest -lpthread
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 2000, 4000
Storing integers to 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
```

For ./mtest


```
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest
gcc test.c -L../ -lmy_vm -m32 -o test
gcc multi_test.c -L../ -lmy_vm -m32 -o mtest -lpthread
Allocated Pointers:
0 e000 2a000 1c000 38000 46000 54000 62000 70000 7e000 8c000 9a000 a8000 b6000 c4000
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 0.052213
```

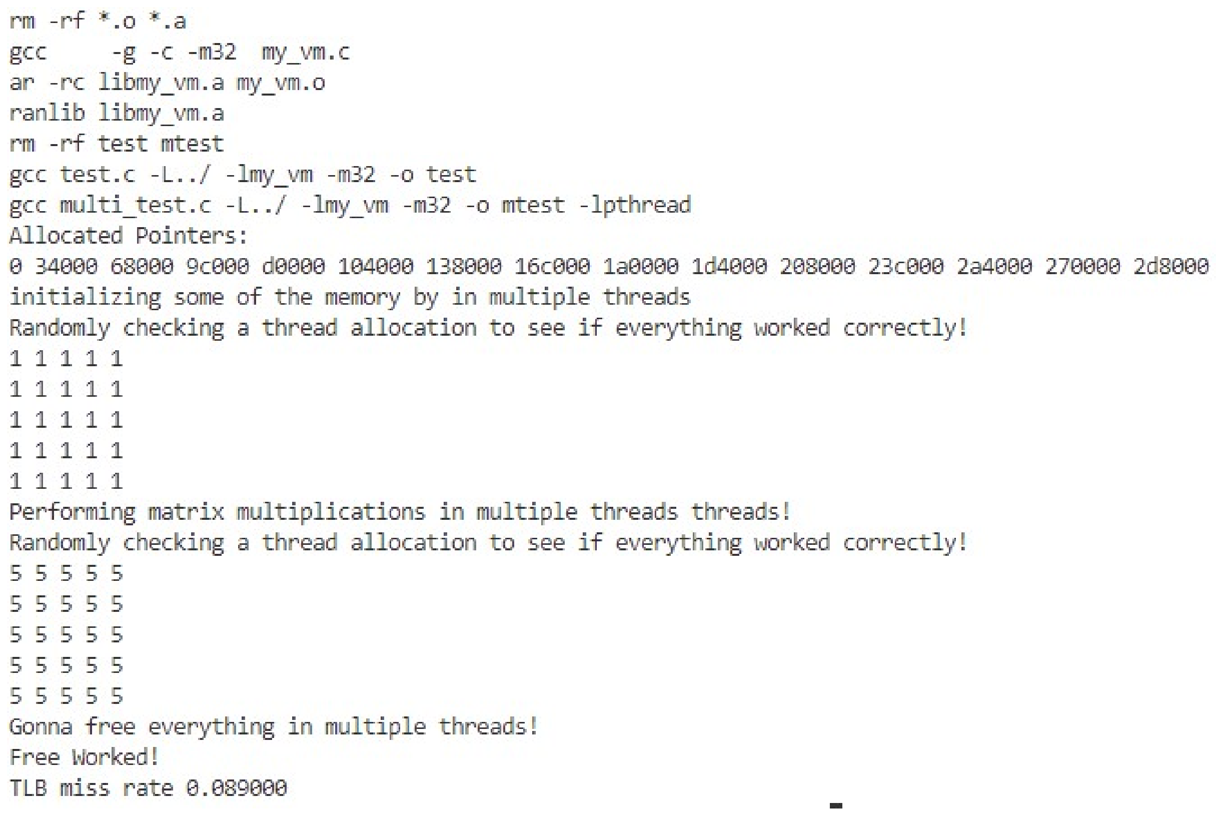
2) 4096 * 6

for ./test



```
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest
gcc test.c -L../ -lmy_vm -m32 -o test
gcc multi_test.c -L../ -lmy_vm -m32 -o mtest -lpthread
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 4000, 8000
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
```

For ./mtest



```
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest
gcc test.c -L../ -lmy_vm -m32 -o test
gcc multi_test.c -L../ -lmy_vm -m32 -o mtest -lpthread
Allocated Pointers:
0 34000 68000 9c000 d0000 104000 138000 16c000 1a0000 1d4000 208000 23c000 2a4000 270000 2d8000
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 0.089000
```

Q4. Possible issues in your code (if any).

A4. No

Q5. If you implement the extra-credit part, description of the 4-level page table design and support for different page sizes.

A5. Not implemented extra-credit

Q6. Collaboration and References: State clearly all people and external resources (including on the Internet) that you consulted. What was the nature of your collaboration or usage of these resources?

A6. Consistently, discussed implementation ideas with TA's. Referenced several textbooks and tutorials to understand the concepts related to this assignment.