**Rutgers, The State University Of New Jersey**

DEPARTMENT OF COMPUTER SCIENCE



CS518 - Operating System Design

**Project 3 : User-level Memory Management**

JANKI TRIVEDI(jdt111)

SACHIT PATEL(sp2135)

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

There are few data structures used by us for implementation of project. First is page structure which is an array of unsigned longs having size equal to page size.The physical memory is allocated as an array of pages. Then we have used physical and virtual bitmaps to store availability of physical pages and page table entries(pte) respectively. Lastly tlb structure is used as 2D array. Here the columns store virtual page number and physical page number while rows represent tlb entries.

1 set physical mem

As the name itself suggests set physical memory, here we have performed all the required data structure allocations and initializations. Firstly we have called calculateBitsofLevels() where the number of outer level bits, inner level bits and offset bits are calculated. Here firstly offset bits are calculated as log2(PGSIZE) SO for 4K page size offset bits are 12. So the virtual page number bits are 32-12=20 bits. Now for this 20 bits, as we are using multi-level page table, for the outer level we have used the formula PGSIZE = entries\*size\_of\_pte. So using log on entries we have obtained the outerlevel bits. The inner level bits are vpn - outerlevel bits.

Next we allocated memory for the physical memory using malloc. The pointer will point to array of pages which will have physical memory in form of physical pages. After that we created the virtual and physical bitmap and initialized them. The final function call in this function is assigningPageTable() where the page tables are assigned in physical memory. The last page of physical memory is allocated to page directory(outer page table). After that inside page directory all the entries inside it are assigned to -1 indicating that corresponding inner level page table has not been assigned.

2 translate

This function takes virtual address and page directory’s starting address and performs translation to return the physical address. Firstly we obtain the offset bits. After that as instructed we check in TLB using check\_TLB function to see if the translation already exists. If it returns the ppn then we can add offset to the index it points in physical memory and obtain the corresponding physical address. But if translation is not present, we increment the tlbMiss counter.

Here afterwards we get the index of outer level page table. As it was initialized to -1, using getBit we check whether its -1 or 1. If its -1 then it means translation does not exist so we return. But if its valid, then we obtain the page directory entry by adding page directory address and outer level index. This page directory entry contains the page number of inner level page table. From that page number we obtain the address of it and add the inner level index which gives us page table entry containing the physical page number. This physical page number points to the page address in the physical memory which is added with offset to get the actual physical address. This corresponding translation is then added to the TLB using add\_TLB function.

3 page\_map

In this function we first check if the mapping is already present in TLB. Using check\_TLB function we check if it returns ppn same as passed in argument then it returns or there is a tlb miss and we move ahead. If mapping does not exist we calculate the page directory entry using outer level index. If the value of it is -1 this indicates that mapping does not exist. For this we need to find the page in physical memory for memory. So we get the page from the end and check if its assigned to 0 in physical bitmap. If its 0 then we have found the page. This page number is stored in page directory entry. Using this we can find the address of inner level page table and add index to it through which we can obtain the page table entry. Inside this page table entry we store the value of ppn and hence finally we performed the mapping.We then add corresponding mapping to TLB.

4 get\_next\_avail

This function is used to find the continuous set of num pages available and returns the first available next page. Here we traverse in virtual bitmap and check if the corresponding bit is 0. In the loop we check until our count is equal to num\_pages. We return the index of first page or -1 if the set does not exist. We have changed the return type of function to index type of first page it found for our convenience.

5 t malloc

The first step here is to check if the physical memory is allocated or not.If its not allocated then we allocate using set\_physical\_mem where we allocate physical memory and the bitmaps but before this we acquire lock for safe implementation. After that we calculate number of pages required to allocate in physical memory. We then check using physical bitmap if we have that many available pages, if not return null. For this many pages we then find the corresponding virtual pages available using get\_next\_avail and do mapping of them using page\_map and update both the bitmaps accordingly.After this we release the lock and return the virtual address of first page by performing left shift in the virtual page number.

6 t\_free

This function is responsible for releasing one or more pages using virtual address. First we calculate number of free pages we want to release. Free pages are calculated using size/PGSIZE. Even if some bytes are left using modulo we have increased the number of free pages. We then check if the memory from va to va+size is valid, if its not return null.we then translate the virtual address to the physical address, set bit at index i 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.Here the lock is acquired firstly and released at last for safe implementation.

7 put value

The put function here copies data pointed by val to physical memory pages using virtual address. First we calculate the physical address using translate. Then we check whether all the virtual pages from va to va+size are valid by checking their bits in virtual bitmap. If any bits are 0 we return. Also the virtual address, physical address and val pointer are typecasted to char pointer. If we got virtual pages from bitmap, then using while loop the value of val is copied to 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 update the corresponding physical page to which we can continue copying the data. Lock is hold for this API while copying value into physical memory and release the lock if any error occurs or execution is successfully completed.

8 get value

This function copies the contents of page to val using the given virtual address. The implementation of get value is same as put value. Just in the end byte wise copying is done from physical memory to value pointer passed in function. Lock is hold for this API while copying value into physical memory and release the lock if any error occurs or execution is successfully completed.

9 add\_TLB

Here in this function we get set which acts as TLB row using modulo operation between vpn and tlb entries. For that row we add the corresponding virtual page number and physical page number in 2 columns.

10 check\_TLB

Here we check TLB for a valid translation. Firstly vpn is obtained from the virtual address by performing bit shifting to remove offset bits. We again obtain the set variable as discussed in get value and check if its column corresponding to vpn matches our vpn. If its same then we obtain corresponding physical page number or return -1

11 print\_TLB\_missrate

Here in this function tlb missrate is calculated using tlbMiss and tlbLookup variable globally declared.

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

We have changed the below variables when benchmarking for Part 1 output.

* tlb entries = 512
* thread number = 15

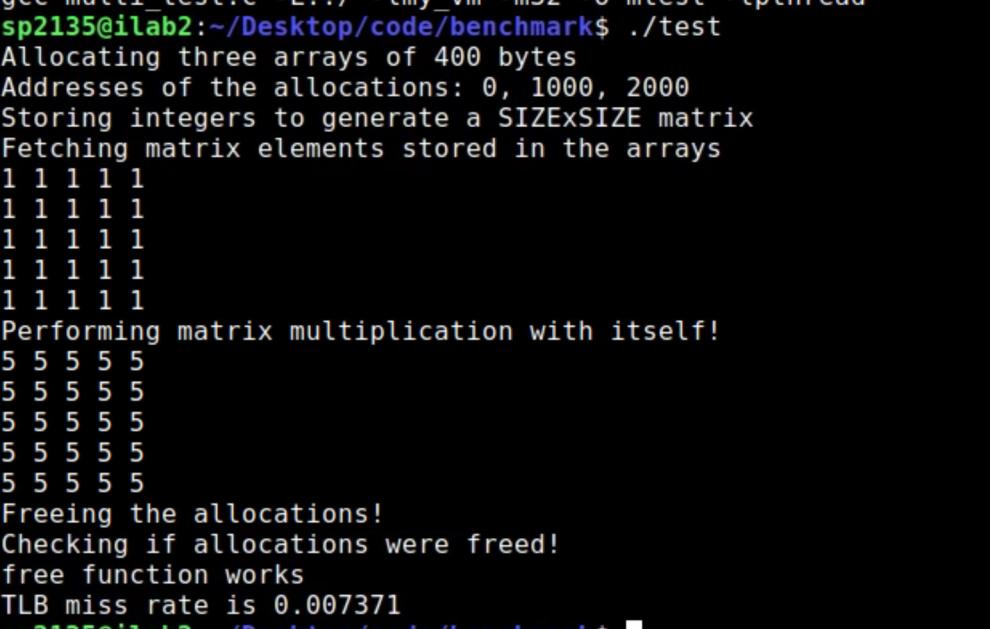


Image 1 test:

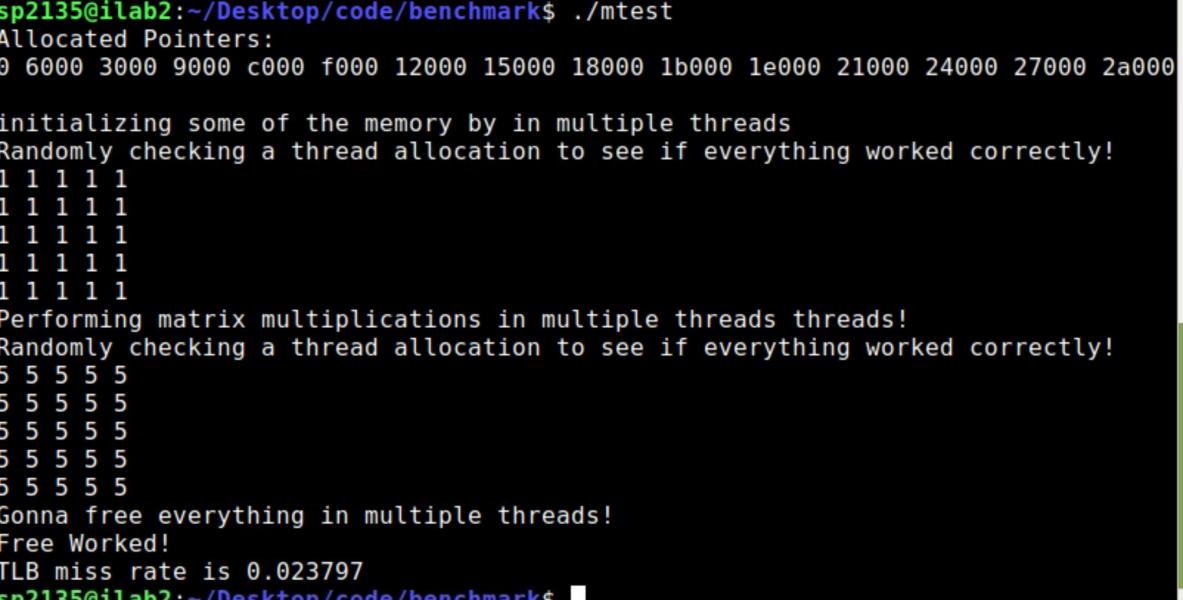


Image 2 multitest:

* tlb entries = 512
* thread number = 25

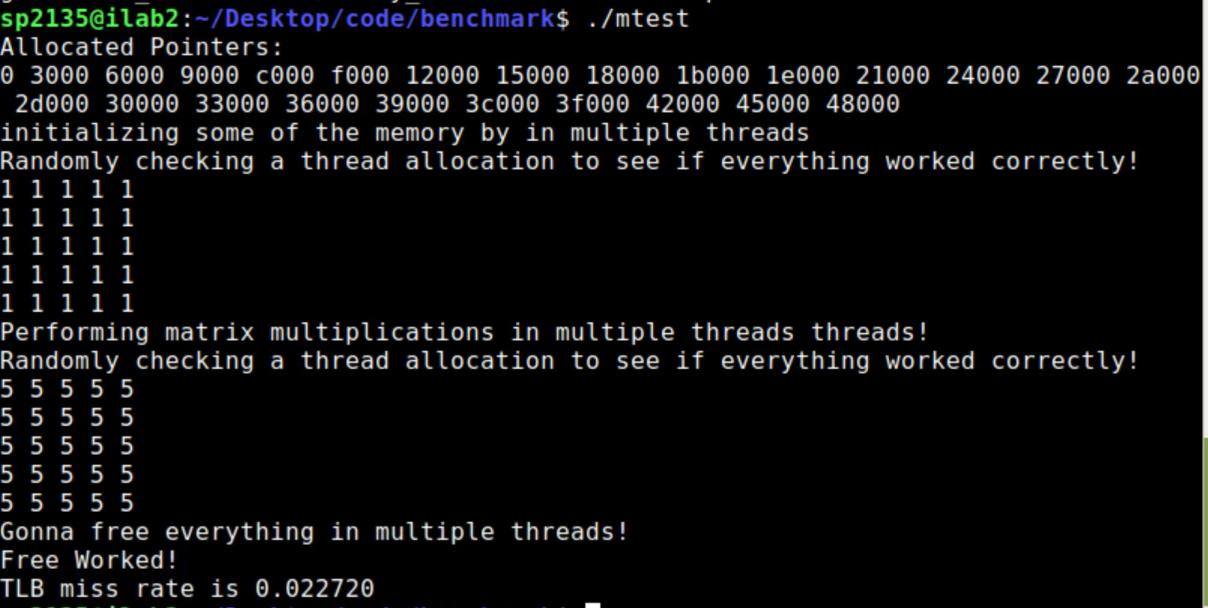


Image mutlitest:

* tlb entries = 512
* thread number = 35

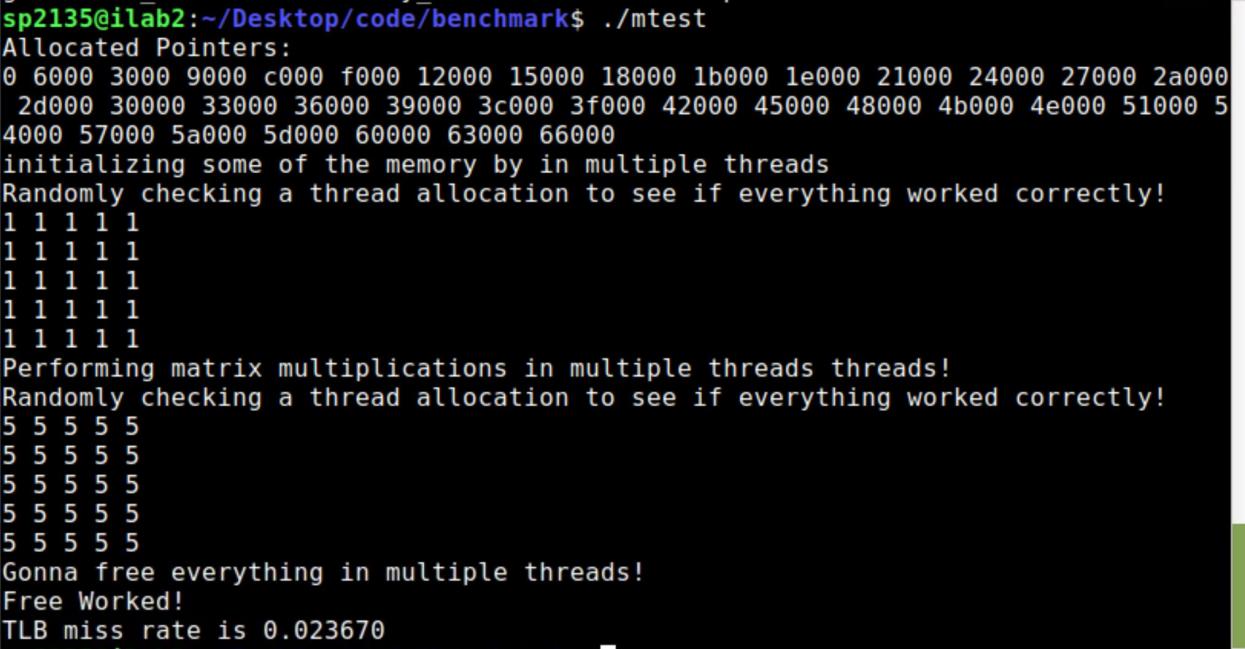


Image multitest:

* tlb entries = 512
* thread number = 45

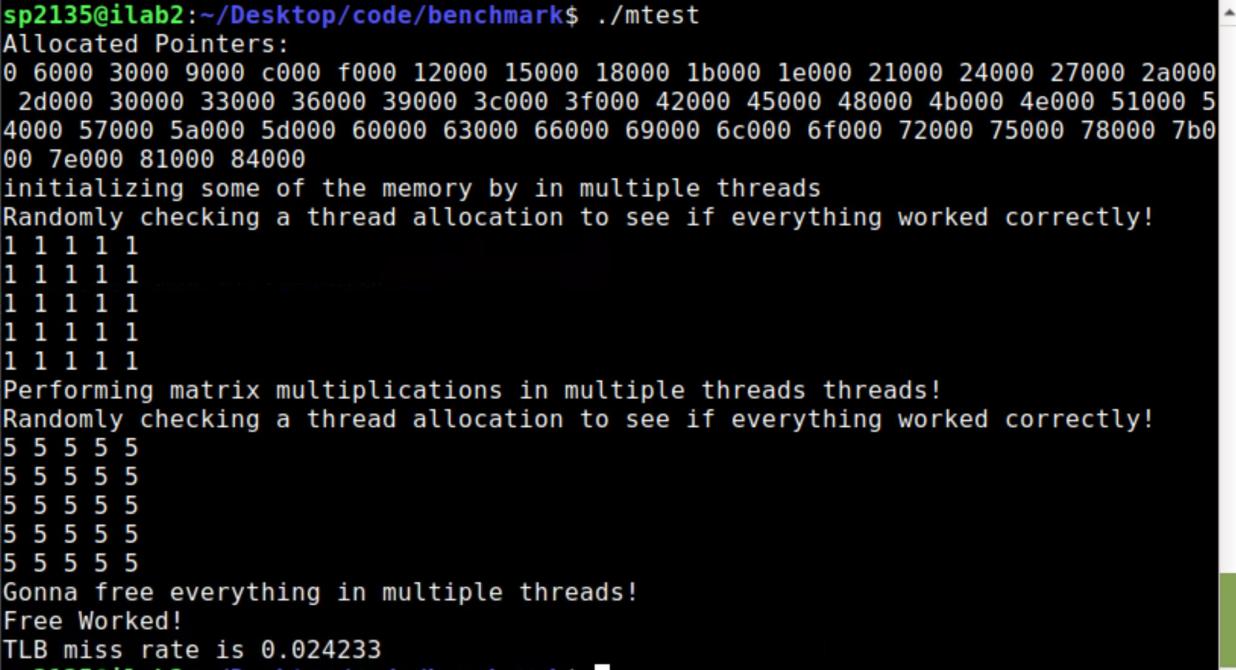


Image multitest:

* tlb entries = 512
* thread number = 50

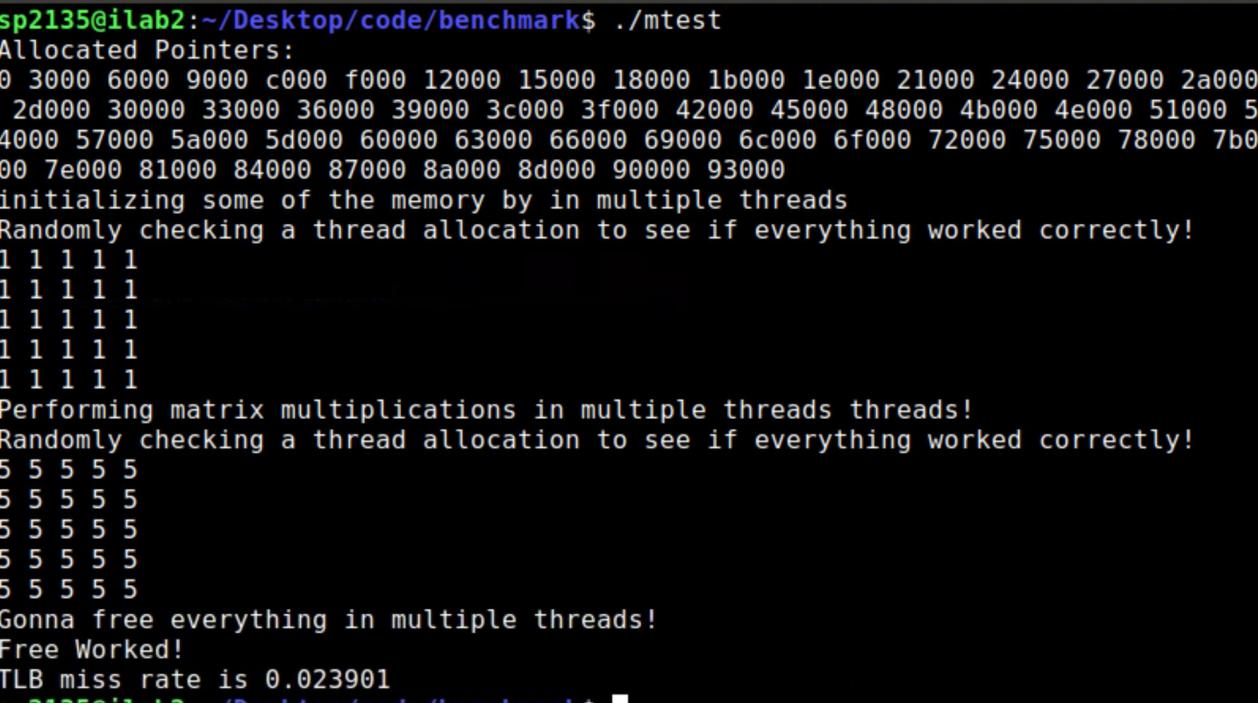


Image multitest:

We have changed the below variables when benchmarking for Part 1 output.

* tlb entries = 256
* thread number = 15

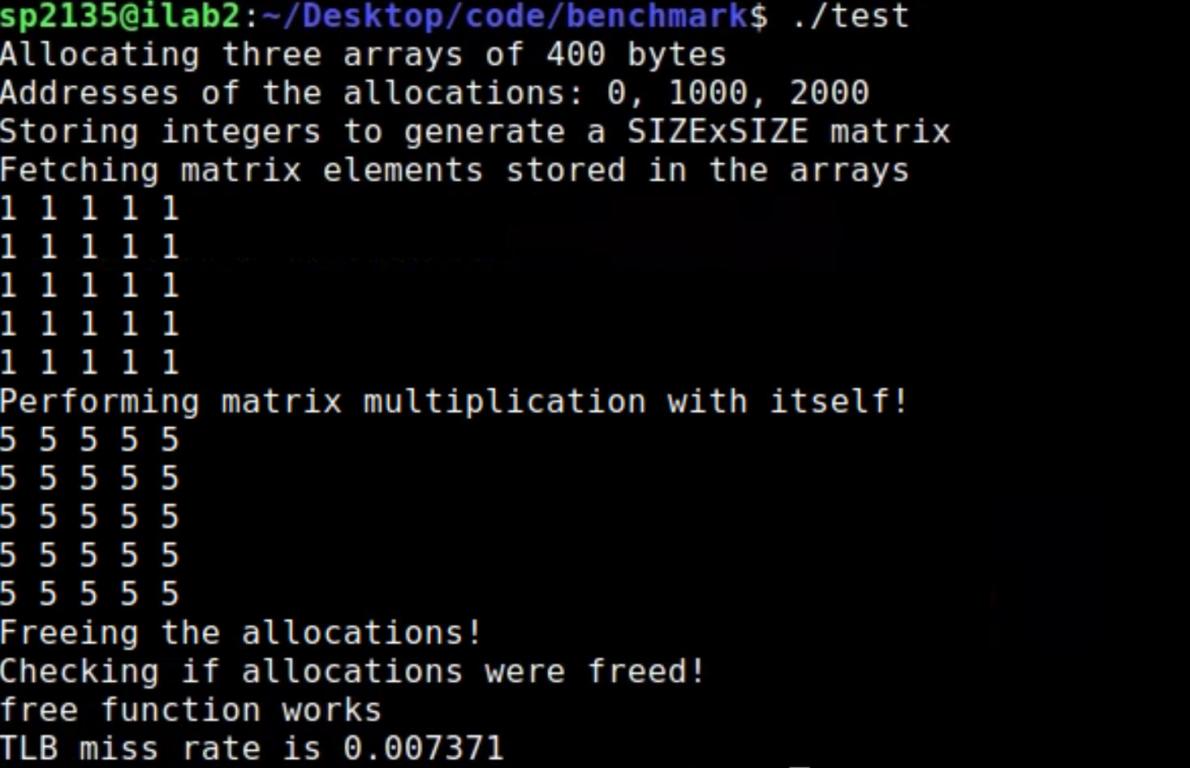


Image 1 test:

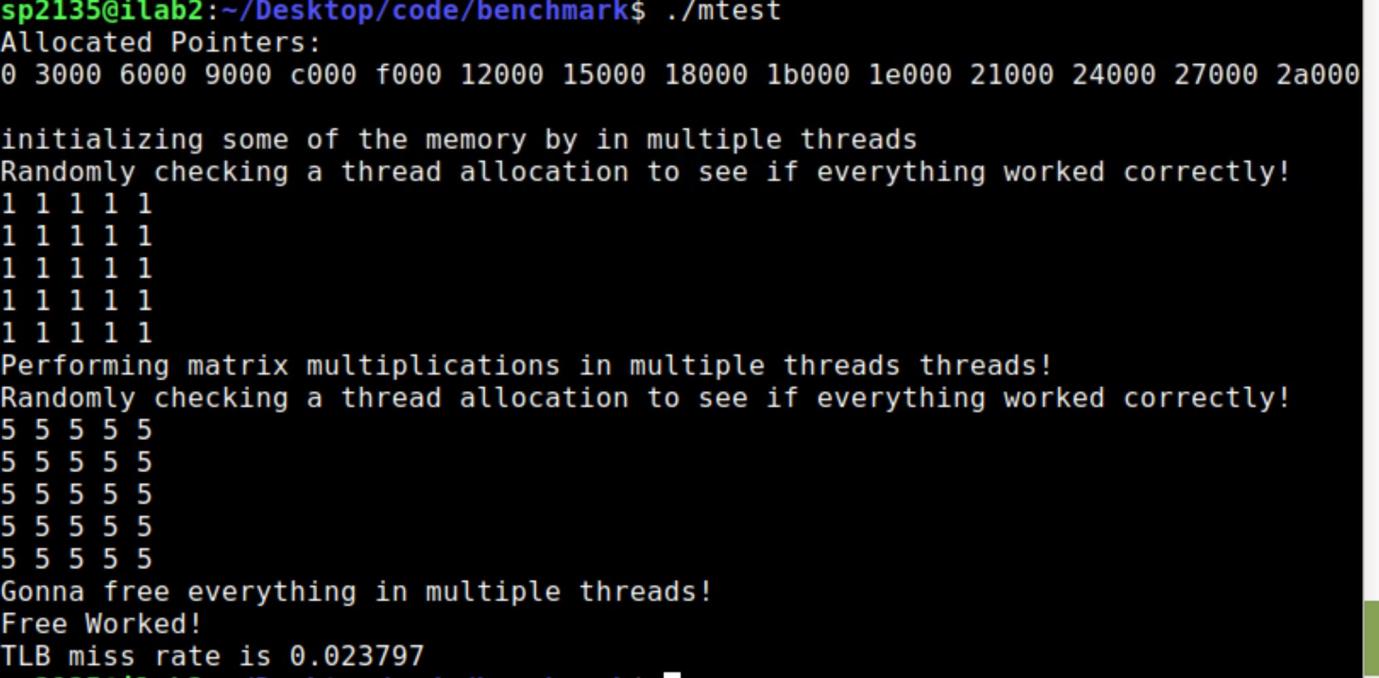


Image 2 multitest:

* tlb entries = 256
* thread number = 25

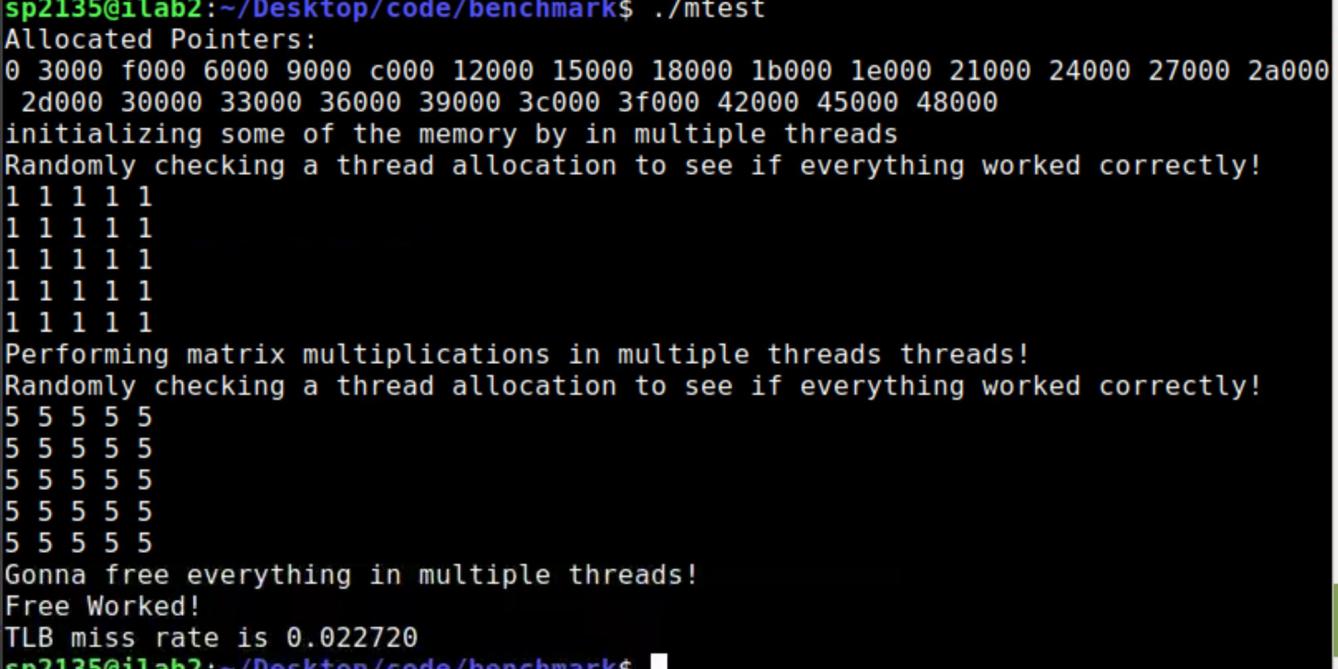


Image multitest:

* tlb entries = 256
* thread number = 35

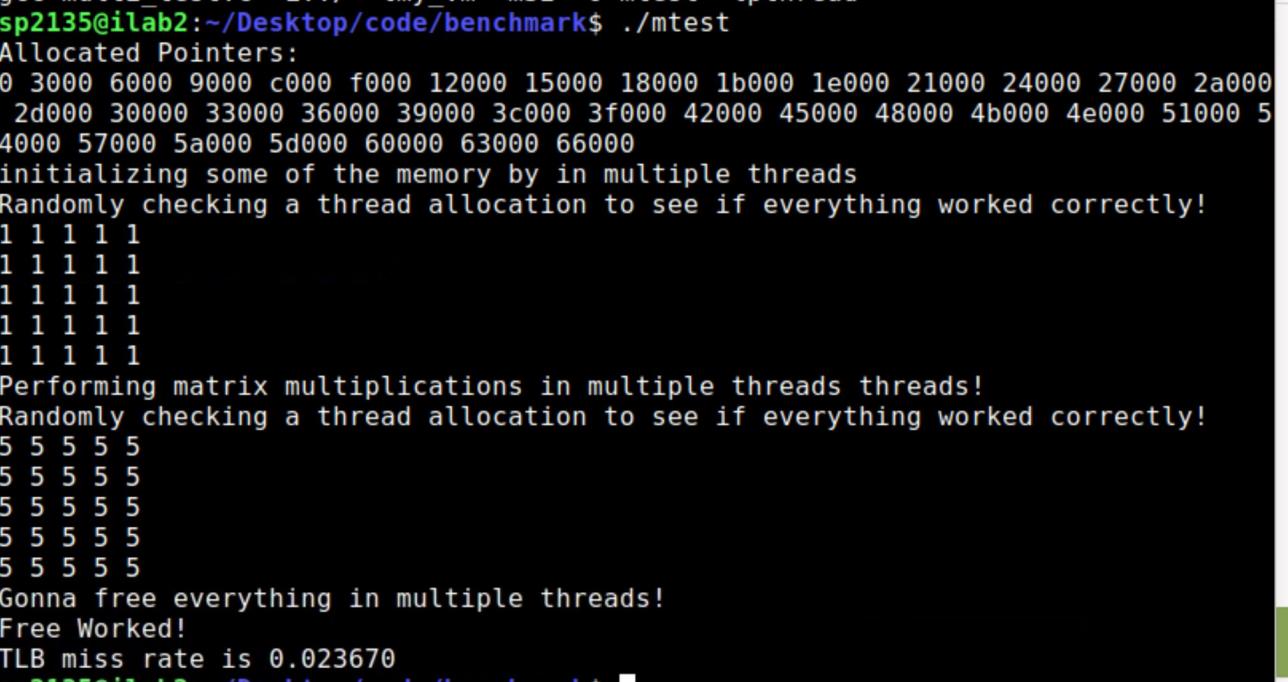


Image multitest

* tlb entries = 256
* thread number = 45

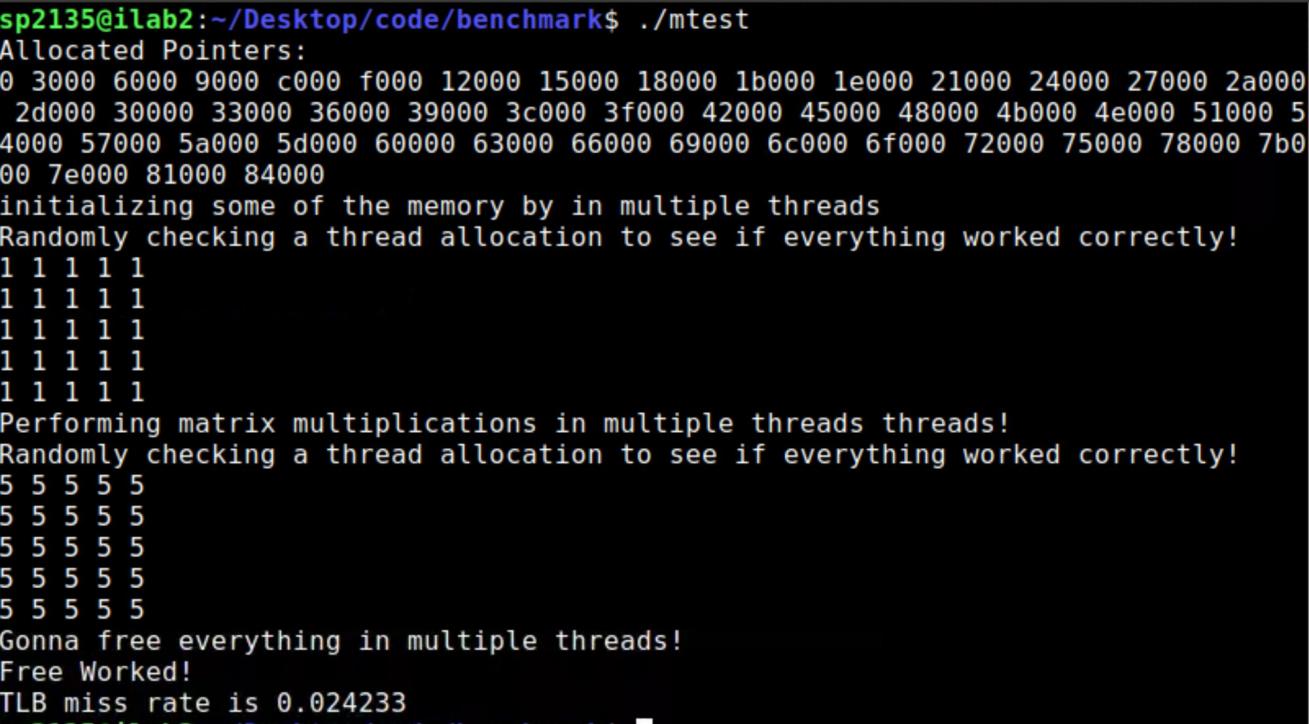


Image multitest

We have changed the below variables when benchmarking for Part 1 output.

* tlb entries = 256
* thread number = 50

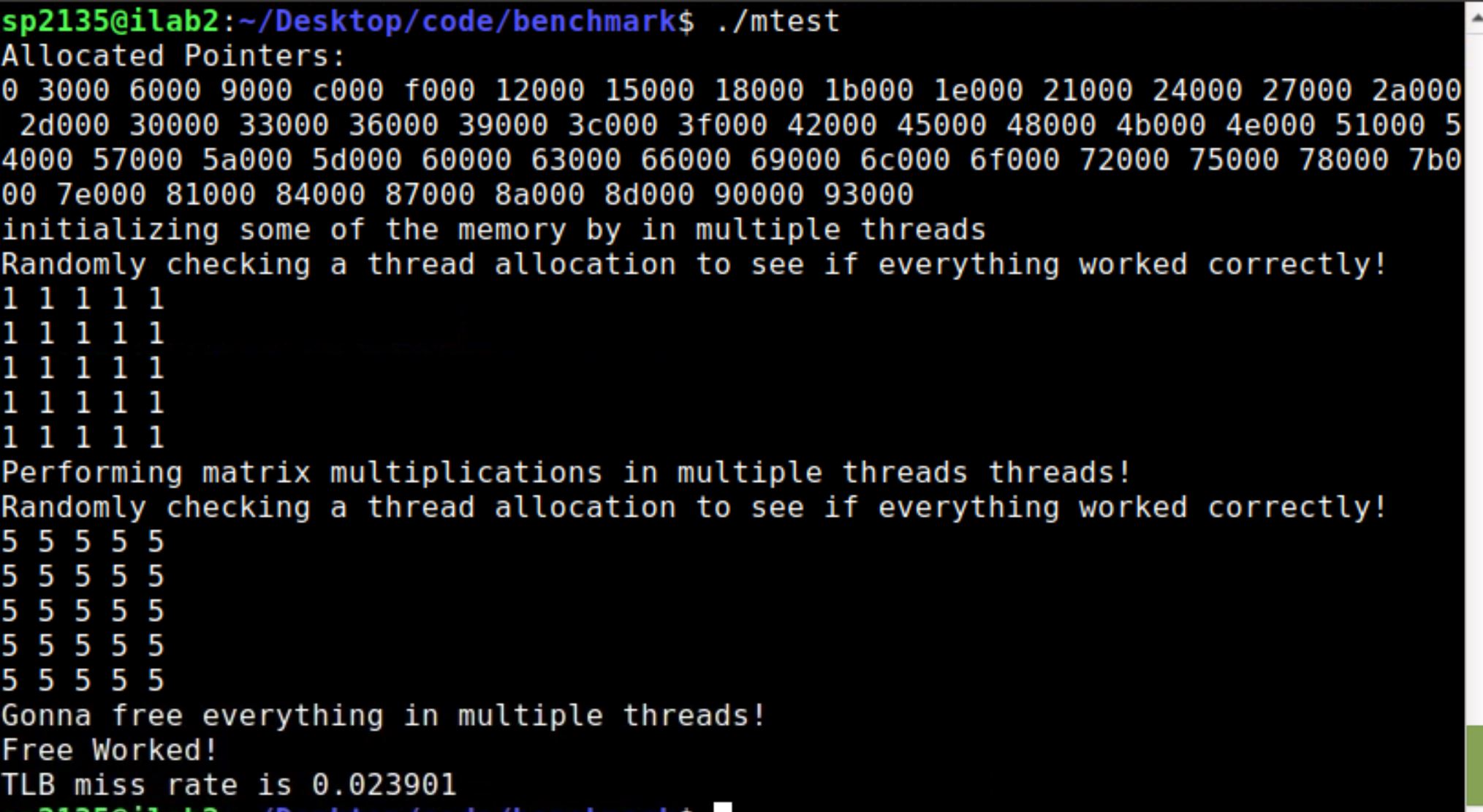


Image multitest:

**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.

* page size = 8k
* tlb entries = 512
* thread number = 15

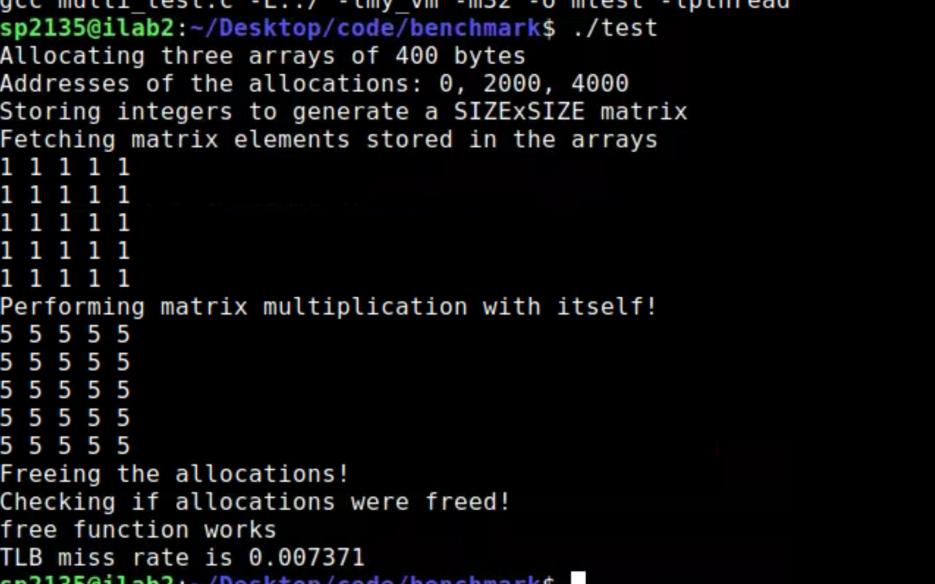


Image1 test

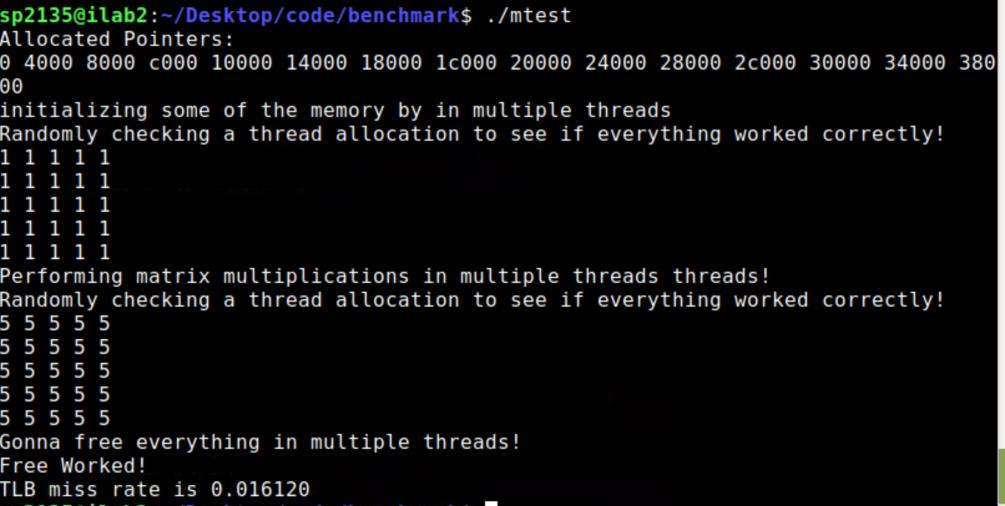


Image 2 Multitest

* tlb entries = 512
* thread number = 15
* Page size = 16k

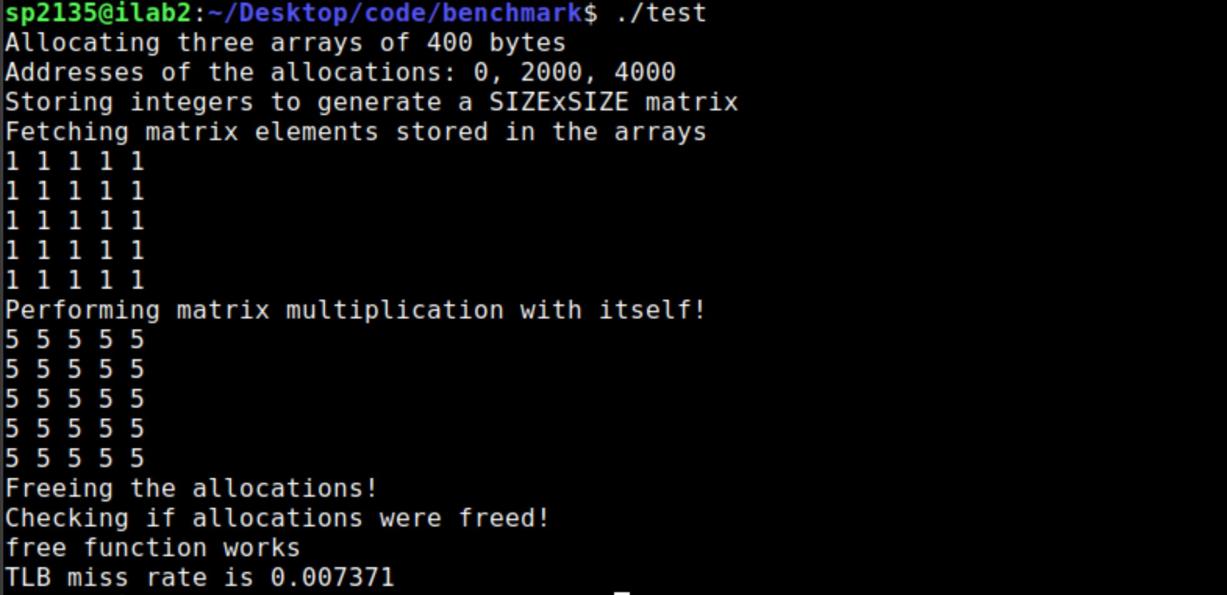


Image 1 test:

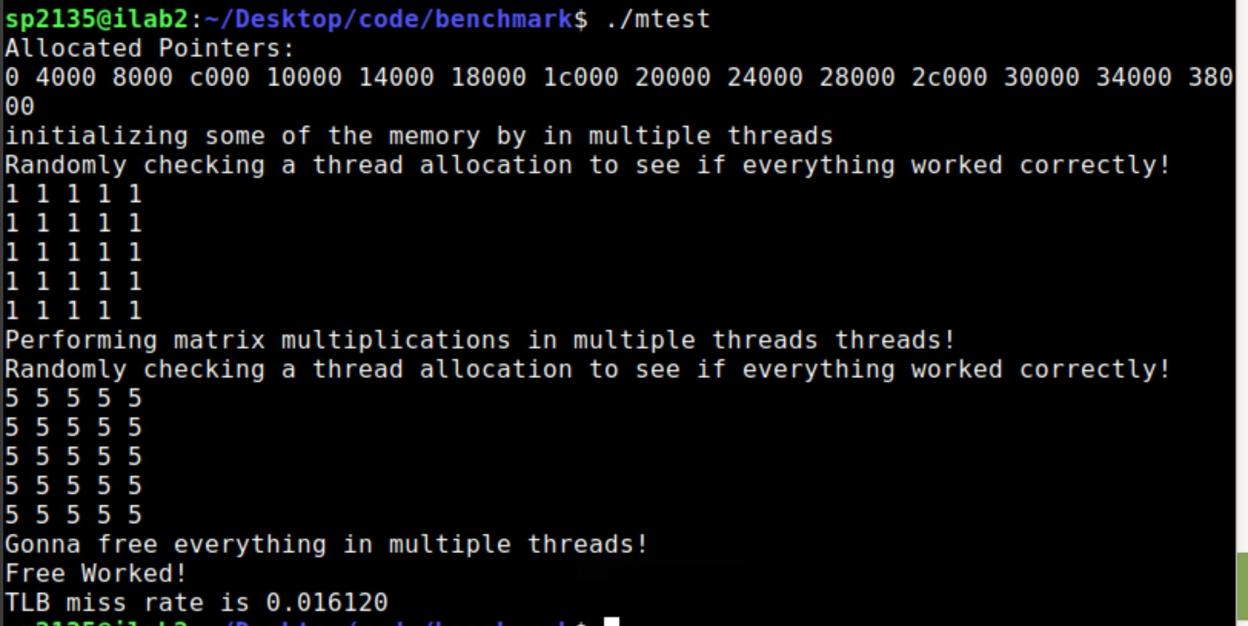


Image 2 multitest:

* tlb entries = 512
* thread number = 15
* Page size = 32k

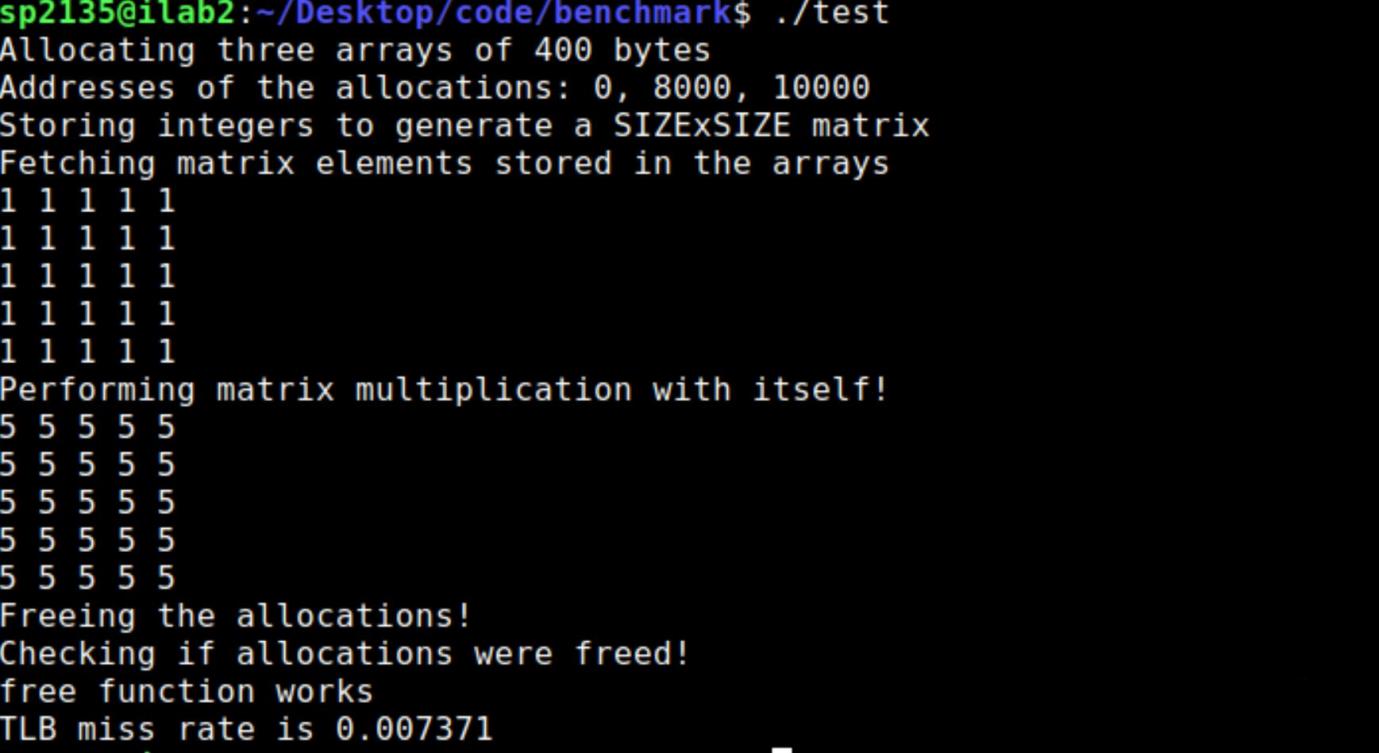


Image 1 test:

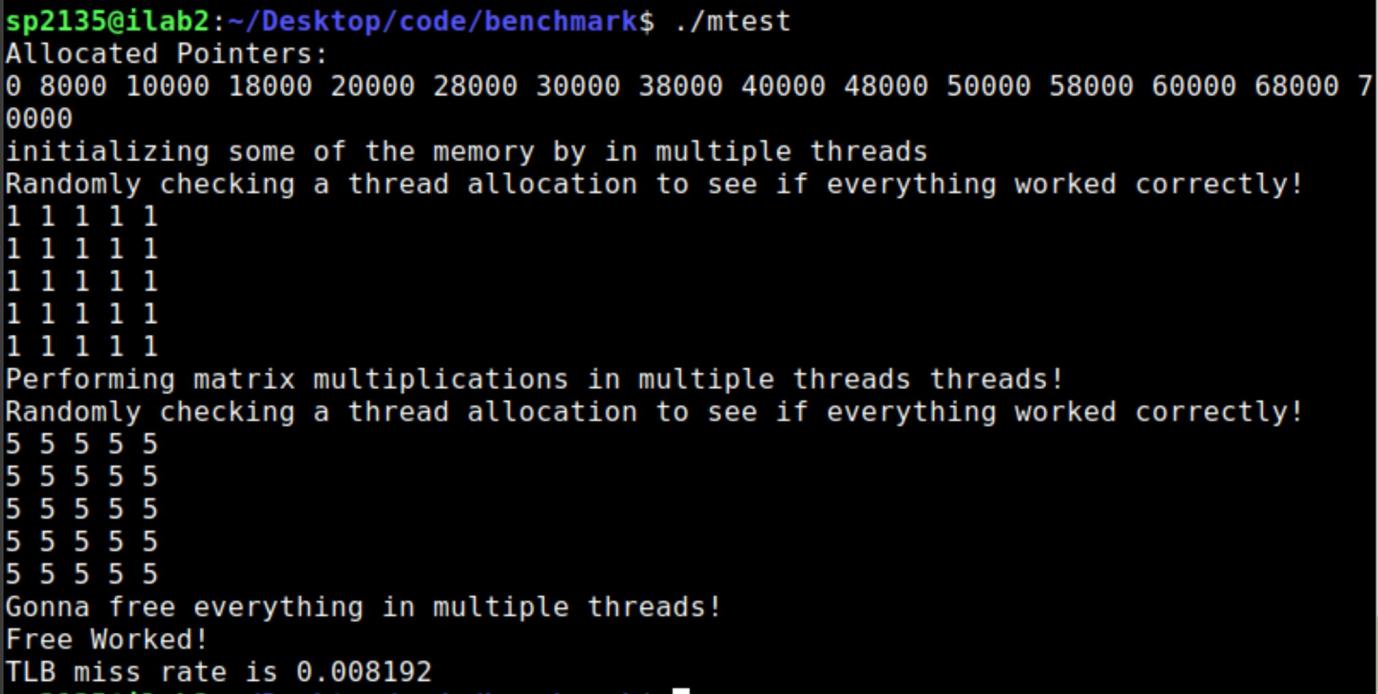


Image 2 multitest:

* tlb entries = 512
* thread number = 25
* Page size = 64k

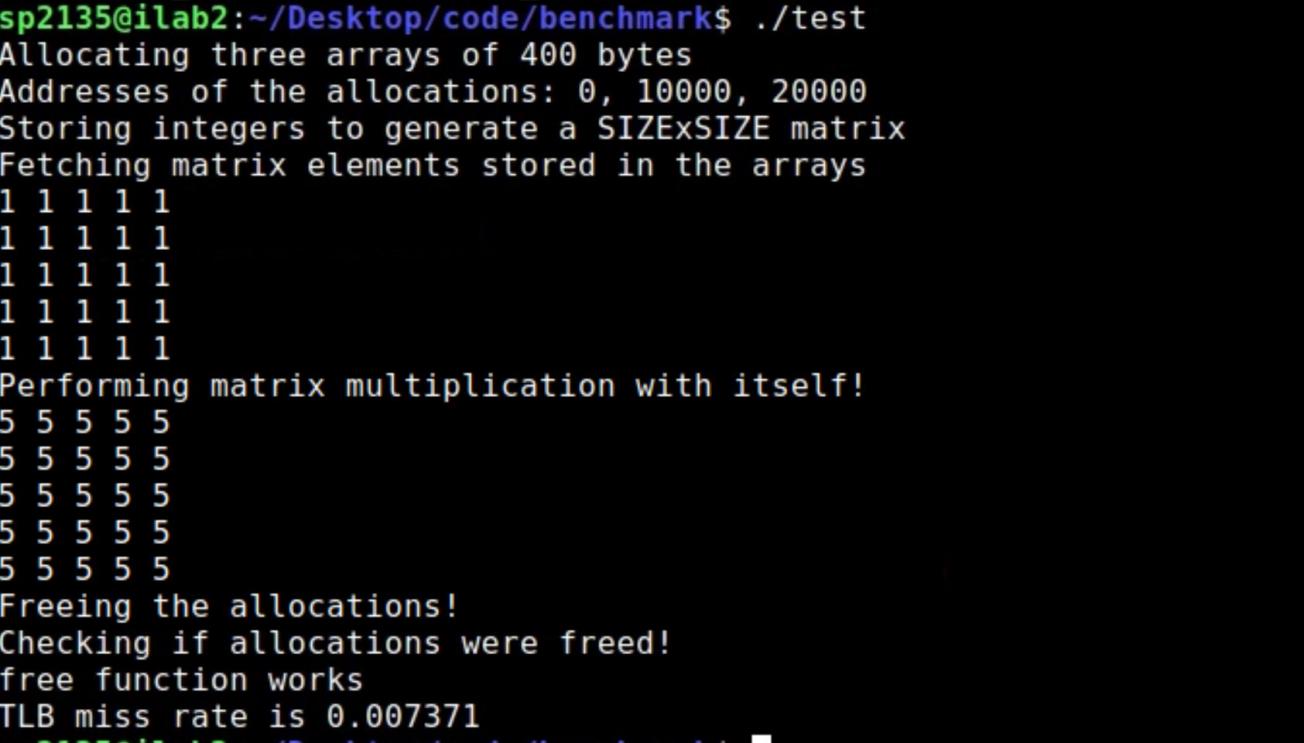


Image 1 test:

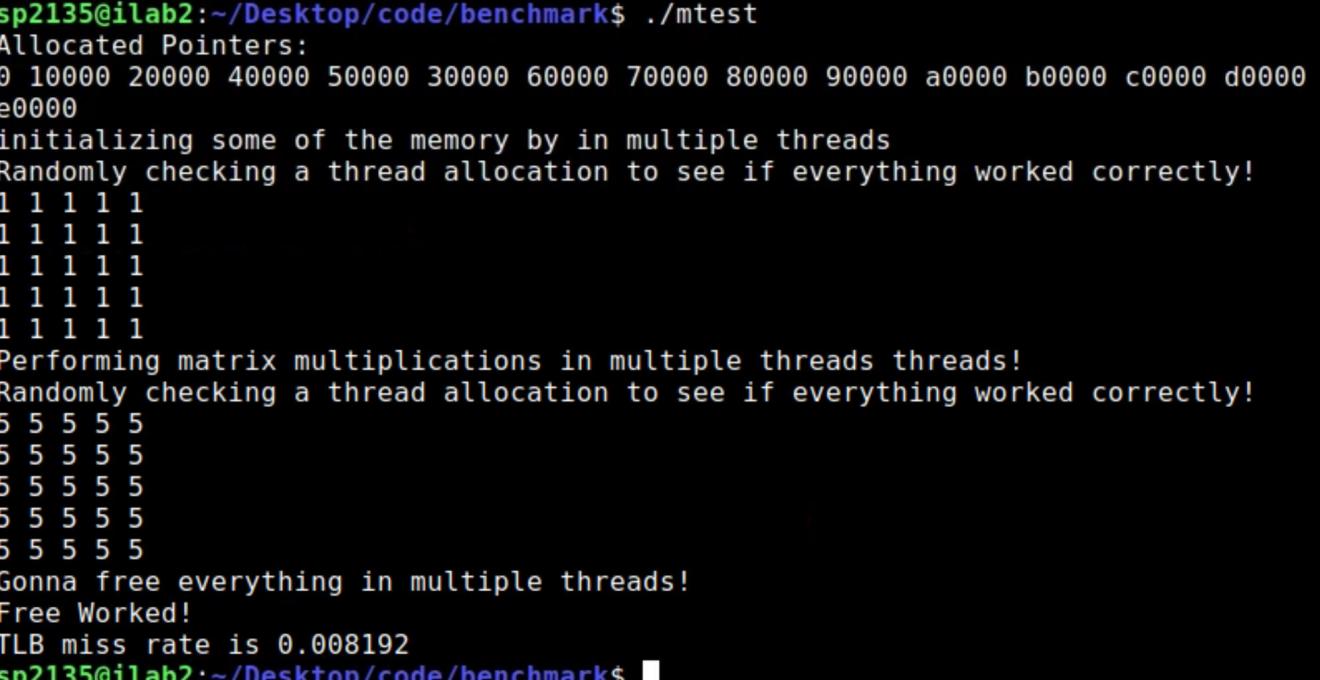


Image 2 multitest:

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

There are some issues in our code. Firstly we did not take care of internal fragmentation as the physical pages were not allocated contiguously. Also we think that the critical section can be reduced further as we have taken almost all operations under critical section for this project.