CS 416 Project 3: User-level Memory Management

Names: Michael Liu msl196

Project Implementation Description

To begin, our implementation contains the following core structs:

(Note that more detailed descriptions of these can be found in the code files)

- **vm_t** – Stores all vm data needed — number of physical/virtual pages, bitmaps for virtual and physical, the head of the pgdir and the pointer to the physical memory
- **tlb** – Stores all data for the TLB, includes all entries as an array and statistics for miss rate.

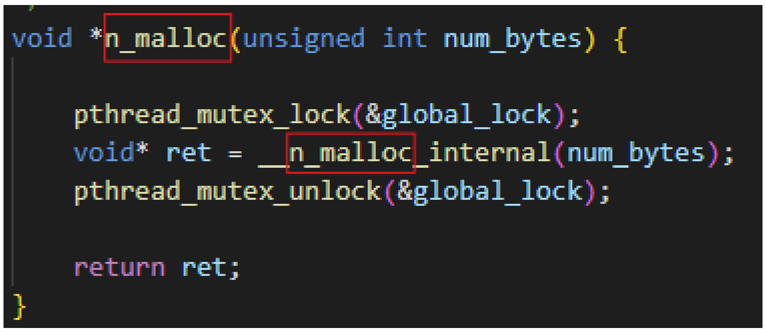
1. Detailed Logic of Each API Function

Part 1 – Virtual Memory System

Thread Safe Implementation

Now I would first like to explain my design to keep everything in my library thread-safe since the multithread test uses multiple threads to run the operations.

- In my header file, we can see that I've defined "**internal**" functions for all core operations: **n_malloc, n_free, put_data** and **get_data**
- The reason I've made these internal functions is so I could mutex lock the entire function, as we can see from the actual library functions, all they do is call the corresponding internal function that has the actual implementations and mutex lock the entire function. This makes it easier for me since I can just mutex lock once instead of doing it multiple times inside the function. And effectively making sure that the function is also thread-safe.



```
void *n_malloc(unsigned int num_bytes) {

    pthread_mutex_lock(&global_lock);
    void* ret = __n_malloc_internal(num_bytes);
    pthread_mutex_unlock(&global_lock);

    return ret;
}
```

- This might be redundant and can be inefficient but in my opinion for these crucial functions that directly modify the memory or the corresponding page table, the **WHOLE** function IS the **critical section**. Therefore, I chose to sacrifice efficiency for safety.

Memory Allocation (n_malloc)

Definition:

This function allocates virtual memory for the user. This process includes:

1. When called for the first time, initialize vm using **set_physical_mem**
2. Next, calculates the total number of pages needed based on the number of bytes requested.
    - Note that this rounds up, so for example if the page size is 4096, a request of 4097 bytes will need 2 pages.
3. Then, finds the next available virtual page and gets it's address using **get_next_avail_virtual**
4. Now, for EACH page:
    - Find the next available physical page and calculates it's address using get_next_avail_physical
    - Map the current virtual address to that physical address using map_page
    - Set both virtual and physical bitmaps, indicating that this page has been allocated and used
    - Move on to the next page
5. Finally, return the virtual address for the original page.

Memory Deallocation (n_free)

Definition:

This function frees memory for the user. This process includes:

1. Similar to malloc, calculates the total number of pages needs to be freed, rounded up.
2. For EACH page:
    a. Gets the physical address based on the virtual address provided
    b. Finds the corresponding location in the page table and sets it to NULL, clearing the memory
    c. Clear the corresponding bit in the virtual and physical bitmap
    d. TLB – Remove the cache for the current virtual address

Data Operations (put_data)

Definition:

This function simply puts the given data into the location of the virtual memory address given

- This is done not at once but in steps where in each step PGSIZE bytes is memcpyed to the actual physical memory location (calculated using the virtual address given)
- Note that if a part or the last part of the data doesn't fit in one page, still copy that amount of bytes into the page, but the page will not be full

Data Operations (get_data)

Definition:

This function simply gets the data at a given virtual address and copies it to the destination given.

- Very similar to put_data with only the different of copying from the physical memory to the destination provided (also done in steps)

Utility Functions

Address Translation (translate)

Definition:

Translates a virtual address to its corresponding physical address in the following process

- **Page Table Walk:** Calculate the page directory and page table index and index the head of the page directory to find the corresponding physical address.
- **TLB:** If the virtual to physical mapping is already cached, find it and return it, otherwise, cache it by adding it to the TLB

Page Mapping (map_page)

Definition:

This function will walk the page directory to see if there is an existing mapping for a virtual address. If the virtual address is not present, then a new entry will be added, this is done in the following order:

1. First, calculate the page directory index by extracting the first part of the virtual address and try to index the page directory, if it doesn't exist, create the page directory using calloc (which will set all pages in that directory to 0 – unused)
2. Next, calculate the physical page number by simply removing the offset from the physical address provided, this is because the offset does not need to be stored (the va contains the offsets).
3. Finally, store the physical page number into the corresponding location in the page table we indexed earlier.
4. TLB – Cache translation found inside TLB if not already inside

Finding Pages

get_next_avail_virtual - gets the next available virtual address (must be contiguous)

get_next_avail_physical **-** similar to above, this gets the next available physical address (does not have to be contiguous)

Address Manipulation Functions

extract_data_from_va

- this function extracts the three different parts of an address (for a two-level page table)

- example - for a 4k page size: the page directory index (first 10 bits), the page table index (second 10 bits) and the offset (last 12 bits)
- This function extracts these three parts from the address and stores it into the pointers provided

extract_page_number_from_address

- Similar to the function above but only extract the page number from an address, in other words, it removes the offset from an address and returns the first part of it which is the page number

Bit Manipulation Functions

- **Set, Clear, and Get Bits from Project 1**
- **Extract Bits:**
    - Extracts specific ranges of bits from an address for use in page table indexing.

Part 2 - TLB Implementation

Core Structure

tlb – This is the core data structure that stores all data for the TLB, includes all entries as an array and statistics for miss rate.

- Where each entry is a **tlb_entry_t** that contains:
    - Virtual Page Number (VPN)
    - Physical Page Number (PPN)
    - Valid Bit (to mark the entry as valid or invalid).

Functions

TLB Initialization (TLB_init)

- Initializes all entries to invalid.

TLB Add (TLB_add)

- Adds a virtual-to-physical translation to the TLB based on the addresses provided.
- The index is calculated using the following formula: $(vpn \,\%\, TLB\_ENTRIES)$

TLB Lookup (TLB_check)

- Checks if a translation exists for the given virtual address.
- We determine if the address is cached or not by first finding the tlb entry based on the index, and then checking to see if it's vpn is the same as the one given as well as if the valid bit is set or not.
- Returns the physical address if found; otherwise, increments the miss counter.

TLB Add (TLB_remove)

- Extra function I added for removing the mappings when freeing memory, simply sets valid bit to 0

TLB Miss Rate (print_TLB_missrate)

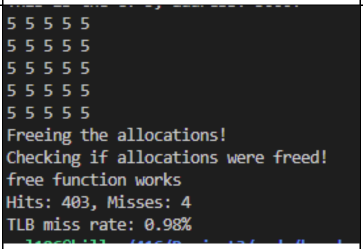
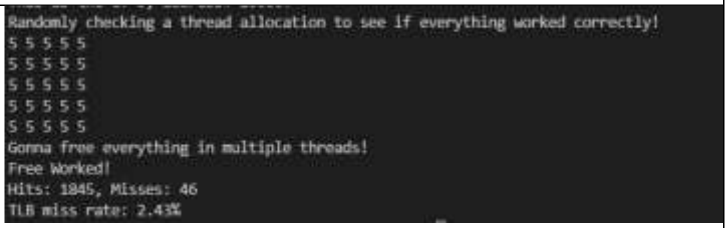
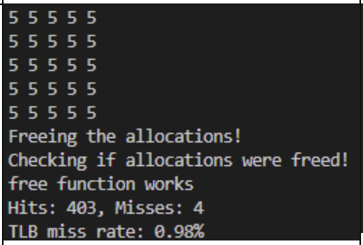
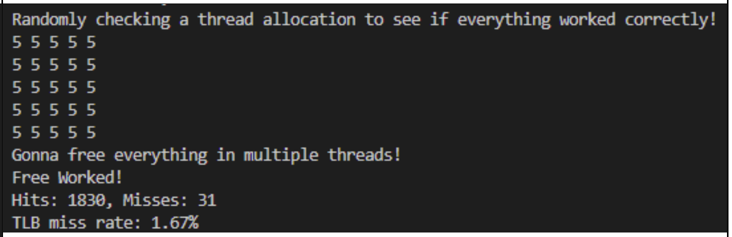
- Calculates and prints the TLB miss rate, where

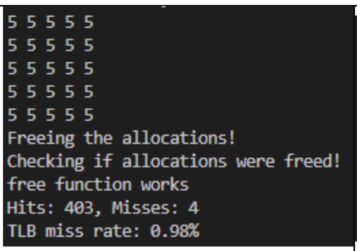
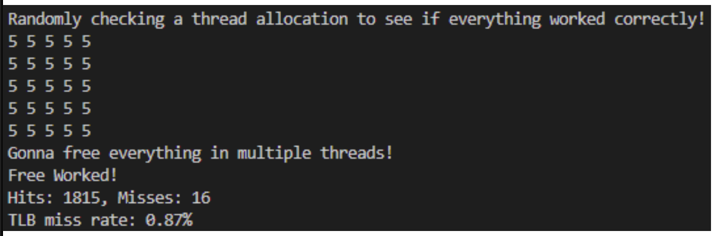
$$Miss\ Rate\ (\%) = \frac{Misses}{Hits + Misses} \times 100$$

2. Benchmark Results

In this section, I will benchmark the results of my vm library with different configurations of page sizes (in multiples of 4k). I will be testing my library with the following page sizes: 4096, 8192, 12288 (4096 * 3)

For both benchmarks: test, multi_test, I will screenshot the result logs for each page size

Page Size	test	multi_test
4096	 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 Hits: 403, Misses: 4 TLB miss rate: 0.98%	 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! Hits: 1845, Misses: 46 TLB miss rate: 2.43%
8192	 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 Hits: 403, Misses: 4 TLB miss rate: 0.98%	 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! Hits: 1830, Misses: 31 TLB miss rate: 1.67%

12288		

```
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
Hits: 403, Misses: 4
TLB miss rate: 0.98%
```

```
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!
Hits: 1815, Misses: 16
TLB miss rate: 0.87%
```

As we can see from the output logs above, for all page number tests, malloc, free both worked and the results were correct, the TLB miss rate were also acceptable around 1%, and it seems to decrease for multi_test as the page size increases.

Extra Credit Implementations

Note – instructions and file descriptions of the extra credits implementations can be found inside the **README** file, I chose to make both extra credits as separate files from the default files of Part 1 and 2 as suggested.

Part 1 – Implementing a 4-level Page Table

Modifications:

- Since we now have a 4-level page table, assuming 4096 page size, meaning 12 bit offset, we now have 64 – 12 = 52 bits left for 4 levels, splitting them evening making 13 bits per level. Many of my utility functions for calculating page directory and table addresses only works for two levels, so all of them have to be modified.
- All the unsigned int (32 bit) was changed to unsigned long (64 bit)

Key Function Modifications

translate – Before in two levels, we only had to traverse from the page directory to the second level which is the page table, now since we have 4 levels, I used a loop to walk down the page table until the final 4th table has been reached, which is where the physical address is stored, and returned that.

extract_data_from_va – The 2 level 32 bit version of this just splits up the address into three parts, but now we have to split it up into 5 parts, so I modified it to make it take an array of indices that represents the index for each level, each will be calculated and set.

map_page – Similar to translate, this function now have to walk down the 4 level page table until reaching the actual table, and it sets that table's corresponding entry to the physical address passed in.

malloc, free, put_data and get_data – these core function all used the utilitiy functions to calculate virtual addresses, all I did was to make sure the arguments were correctly passed into these utility functions.

tlb functions – These functions and structs were modified to support unsigned long instead of unsigned int.

Part 2 – Reducing Fragmentation

Modifications:

- I followed the implantation described in the pdf instructions, to do this, I decided to use a struct to store the page metadata since we now need to know how much space in the page is actually used instead of just if the page was used or not.
- For each page, I stored how much space does it have left and where exactly does that free space start within the page (offset)

Key Function Modifications

n_malloc:

For malloc, instead of just mallocing one or more new pages when called, now we have to actually try to find a page where remaining size isn't 0, and once we've found one, we get the offset and return that.

n_free:

As the instructions described, reducing fragmentation does indeed complicate free a lot, instead of just freeing one or more page as a whole, now we have to making sure that the whole page is unused before actually freeing it, we do this by checking first to see if the remaining space is equal to the page size, if it is, free it, if not, we just add to the remaining space the amount we're freeing, basically freeing it.