Kev Sharma (kks107),
Yash Patel (yp315)
April 16, 2023
CS416: Undergraduate OS
Professor Sudarsun Kannan

Project 3 Report

Introduction

Our implementation of the User Level VM features two bitmaps, a two level paging scheme, a TLB cache supporting a LRU replacement policy, multithreading support, and non-contiguous page allocation scheme. During the first incoming t_malloc request, our library initializes these supporting structures. This writeup will assume a 32 bit address space.

set_physical_mem()

This function initializes our paging scheme, frame and virtual bitmap on the heap, cache, MEMSIZE amount of virtual memory, page tables which use this memory, and a single mutex lock. We guarantee that set_physical_mem is called from the t_malloc function at most once with the use of a spin-lock. We found that strange bugs - segfaults / dangling pointers - manifest themselves when we do not preserve the at most once guarantee. Once the library is initialized, we revert to using the mutex lock.

Paging Scheme

Having allocated a contiguous MEMSIZE chunk of memory from the heap, we divy up the first $2^{\text{page directory}}$ number of bits among the page tables while reserving the very first page for the page directory. Our code supports page sizes below 4096 bytes, but we have not implemented support to dynamically adjust the number of levels in our page table. Accordingly, we truncate some bits to represent pages with sizes lower than 4K. Note that sizes above 4K are also supported as long as they are a factor of 2.

Our implementation offers a virtual_address_t type used during the translation process. Helper functions are used to manipulate this type and refactor code such that to extend our two level paging scheme to an arbitrary number of levels would require changing solely these functions. A finer point to be made about our paging scheme is that we restrict the number of virtual pages to match the number of possible frames ($\log_2$(MEMSIZE)).

Bitmaps

As suggested in the prompt, our frame and virtual bitmaps have been allocated using a character array. If we use n bits for our VPN, then our bitmaps contain 2^{n-3} characters as each character represents 2^3 pages. Helper functions for manipulating bitmaps are found at the end of the source code. Our logic to fulfill a t_malloc request is a two-phase protocol to either commit or abort. The first phase uses the helper function n_bits_available in both the frame and virtual bitmaps to determine if n, which is the number of pages necessary to supply the size_t bytes requested, can be accommodated. The second phase aborts if either bitmap lacks n unset bits; otherwise, the request is fulfilled.

Upon library initialization, the first bit of the virtual bitmap is set as its value of 0x0 conflicts with a return value of NULL. The first 1 + $2^{(\text{page dir \# of bits})}$ frame bits are also set for the page directory and page tables respectively.

Malloc & Free

We stated previously that our implementation affords more flexibility in approving t_malloc requests using non-contiguous pages. As a consequence, we incur the overhead of tracking which page succeeds the current page: this state is tracked in the first four bytes of every data frame. That is, for all frames apart from the ones used for the page directory and page tables, each frame of size 4096 is partitioned into two sets: 4 bytes which stores the virtual address of the successor page, and 4096 - 4 bytes which stores the payload containing user data. Let us say a user requests 10,000 bytes from t_malloc with a page size of 4096. Then we allocate 3 pages A, B, and C such that A's first four bytes contain the virtual address of B, B's first four bytes contain the virtual address of C, and C's first four bytes contain the NULL value.

Since the first four bytes are reserved for the virtual address, when t_free is called for 10,000 bytes, we can hop from A to B to C and overwrite the contents of the page. Because the virtual address is stored in the first four bytes, t_free is able to clear out the associated mapping in the page table for that virtual address as well. This implication is convenient for the user as it means they must only track the very first virtual address returned from a successful request to t_malloc. The t_malloc function utilizes the two-phase protocol discussed earlier and the get_next_avail and page_map functions which we examine later. After a successful t_malloc request, we ensure that the appropriate bits in the virtual and frame bitmaps were set.

TLB Cache

We have implemented the template functions check_TLB and add_TLB. Our TLB cache is implemented using a linked list which uses a Least Recently Used (LRU) replacement policy. Upon a cache hit, our check_TLB function calls an auxiliary procedure to search and remove the node in the list containing the virtual address mapped from the cache lookup request and places this node at the front of the list. Such a position shift is predicated upon the fact that our list should be ordered from most recently used to least recently used. It follows naturally, that if the list has reached capacity, that we must evict the least recently used cache entry (the last node), in order to insert the valid but uncached page at the front.

The add_TLB function prepends to the TLB list. Please note that we have altered only the method signature for the add_TLB function: the return value is now a void instead of an int, and the second parameter change is a pointer to a pte_t instead of a void * to a physical address. This parameter change is apt considering check_TLB returns a pte_t * and not a void * pa.

The check_TLB function does an O(n) search through this list to find a matching va : pa mapping; if this search is unsuccessful, then it offsets into the correct index in the page table to derive the mapping, calls add_TLB, and incurs a cache miss. The check_TLB function is motivated by the algorithm found on page 2 of Chapter 19 in OSTEP.

translate, page_map, get_next_avail

The translate function simply calls check_TLB. The default page_map function signature has remained unchanged and is only invoked solely from t_malloc wherein the user requests pages. The page_map function calls the translate function to find the associated page table entry pointed to by the virtual address and writes the physical address to it. And since the translate function calls check_TLB and thus a call to page_map will have the TLB incur a compulsory miss.

The get_next_avail function disregards the template parameter. Our logic instead opts to return the first available position within the virtual bitmap which refers to the first unused page table entry. We can get away with this while maintaining correctness because of our non-contiguous scheme. This scheme reverts to a contiguous scheme in the event that n consecutive unset bits in the virtual bitmap is found but still uses linking.

get_value and put_value

The structure of both of these functions is isomorphic. When a request to either get_value or put_value is made, the function first ensures whether the virtual address contains a mapping to a valid frame. Additionally, suppose the request must write to n data pages, an auxiliary function ensures that n non-contiguous or contiguous pages are actually linked together by doing a depth first search using the first four bytes of every frame pointing to the next frame's virtual address (not physical address).

After this valid linking is confirmed and the request is found to be valid, get and put call their own auxiliary recursive functions to write the amount of bytes possible to the payload part of the data page (PGSIZE - 4 bytes reserved for link).

Let us dive into an example: assume our page size is 4096 bytes. The first four bytes are reserved to reference the succeeding page. The size of our payload is thus 4096 - 4 = 4092 bytes. If the virtual address pointed to in the request parameter contains a non-zero offset, we must read starting from 4 + offset amount. If this offset is 100 say, we begin reading the size_t amount of bytes beginning at 104 bytes from the address of the frame. Should the user request be 4096 but our payload is only 4092 bytes, we would recurse to the next page and read the 4 remaining bytes.

The t_malloc, t_free, get_value, and put_value all use the initialized library lock to preserve thread-safety.

Extra Credit

We have implemented neither part 1 nor part 2 of the extra credit. Our paging scheme does however support lower page sizes with the downside of truncating page bits.

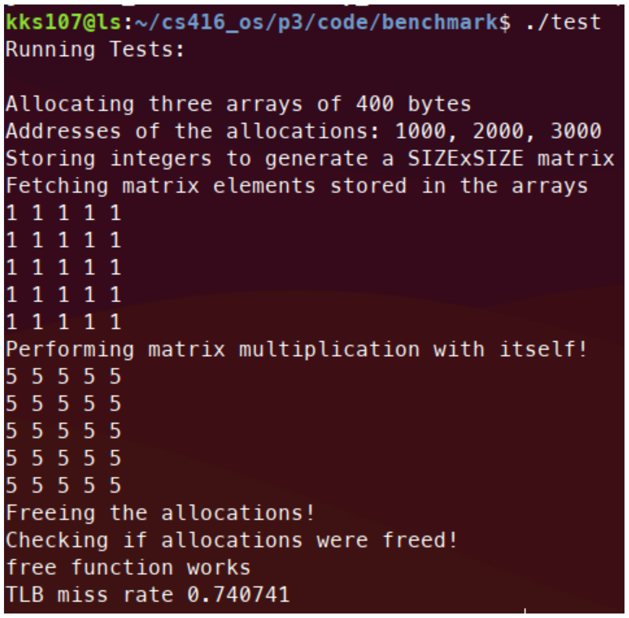
Benchmarks

TLB with 512 entries:
For test.c, we observe a 0.74% miss rate. For mtest.c, we observe a 2.41% miss rate.

TLB with 8 entries:
No changes observed for test.c which mathematically checks out.
For mtest.c, we observe a 14.19% miss rate.

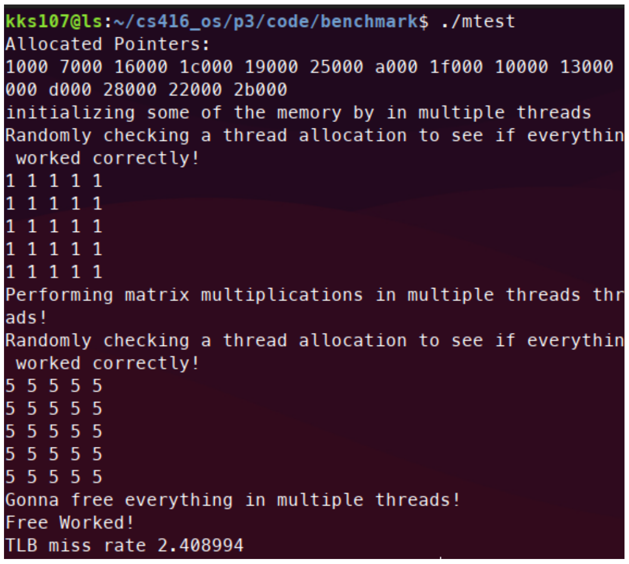
We were able to push our mtest.c number of threads up to 339. For an undetermined reason, when we pushed our num_threads value to 342, it ended with a segmentation fault.

These attached pictures are for the TLB with 512 entries running default parameters for both tests:



```
kks107@ls:~/cs416_os/p3/code/benchmark$ ./test
Running Tests:

Allocating three arrays of 400 bytes
Addresses of the allocations: 1000, 2000, 3000
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 0.740741
```



```
kks107@ls:~/cs416_os/p3/code/benchmark$ ./mtest
Allocated Pointers:
1000 7000 16000 1c000 19000 25000 a000 1f000 10000 13000
000 d000 28000 22000 2b000
initializing some of the memory by in multiple threads
Randomly checking a thread allocation to see if everythin
 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 thr
ads!
Randomly checking a thread allocation to see if everythin
 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 2.408994
```