Name: Jack Lin
Net ID: jjl327

CS416 Project 3 Report

1. FUNCTION LOGIC

set_physical_mem():

I first determined the split in the 32-bit virtual address space (e.g. 9 bits for the outer level, 10 bits for the inner level, and 13 bits for page offset). Then I used malloc() to allocate the physical memory.

As for the bitmaps, I initialized them by allocating (num_of_pages / 8) to figure out how many bytes I needed. Then I used memset() to set every bit to 0 to mark every page as unallocated.

Finally, I initialized the outer level table, the inner level tables, and the TLB.

translate():

Using the split I determined from set_physical_mem(), I performed bit operations on the 32-bit virtual address space to obtain the page directory index (pdi), page table index (pti), and page offset bits.

From there, I first checked the TLB to see if the page mapping already existed. If it did exist, I would perform the simple calculation (start of physical memory + (physical page number * page size) + page offset) in order to return the physical address. If not, I resorted to traversing the page tables and calculating the physical address using the page table entries.

page_map():

First, I calculated the pdi, pti, and page offset bits that are necessary for traversing the page tables. Then I used these values to check if the inner page tables mapped to a physical page. If they didn't, I created the mapping, set the corresponding bit in the bitmap, and returned the physical page. If they did, I just returned the physical page that was already mapped.

t_malloc():

I first calculated the number of pages I needed based on the requested size and the page size. Then I checked the physical bitmap to see if there were any physical pages that were free for me to allocate.

If there was, I would find a free virtual page and map the virtual page to the physical page using page_map() and then set the corresponding bit in the bitmap. This process loops for as many pages as I need to allocate the correct amount. I also declared a variable that held the address of the first page's virtual address to return.

t_free():

This function essentially reverses what t_malloc() does. First, I used translate() to check if the virtual address actually had a mapping to it. I could have also done this with page_map(), but the writeup did not specifically say that I had to use page_map().

If it did have a mapping, I used bit operations to calculate the pdi, pti, and page offset to traverse the page tables. I then unlinked the mappings in the page table entries and unset the corresponding bits in the bitmaps. In order to increment through the pages necessary to free, I incremented the virtual_address by the page size.

put_value():

To start, I used translate() to check if the mapping for the virtual page existed before moving on to the logic.

Using the virtual address provided, I made sure to modify the address to align with the start of the page it is in. This way, wasted memory is less likely. From there, I used memset() to copy and paste the code in increments of pages. I used a variable remaining_size to calculate how much data is left to copy since memset() requires a specific amount of memory. I also incremented the virtual address by the page size to move to the next page.

get_value():

My logic for get_value() is identical to the logic in put_value(). The only difference is the order of the parameters in the memset() function. This way, instead of copying and pasting the values from the value pointer to the memory, I do the opposite.

mat_mult():

I used 3 loops to iterate through the rows of matrix A, the columns of matrix B, and the dot product. In the dot product loop, I calculated the virtual addresses of the elements in matrices A and B, obtained their values by translating the virtual addresses and dereferencing them, and added them to the dot product sum. Then I simply stored the sum in address C by calculating the virtual address, translating it into a physical address, and storing it there.

add_TLB():

First I checked the entry count of the TLB to see if it was at full capacity. If it was, I evicted an entry using the modulo operation: vpage % TLB_ENTRIES. Using this method is a simple and memory-efficient way of evenly distributing the virtual page numbers throughout the TLB. Then I added the entry using the vpage and ppage arguments.

check_TLB():

This function is very simple. At the very top, I incremented tlb_check_count to keep track of how many times the TLB was accessed. This comes in handy for the next function. Then I iterated through the entries in the TLB to see if the virtual page numbers matched. If they did, I returned the physical page number. If they didn't, I incremented the tlb_miss_count global variable and returned -1.

print_TLB_missrate():

To calculate the miss rate of the TLB, I used tlb_miss_count / tlb_check_count and stored it into a float. When printing the miss rate, I rounded the percentage to 2 decimal places.

2. DIFFERENT PAGE SIZES

By default, I chose an 8 KB page size to test my build with. But after implementing and testing everything, the build seems to work with virtually any page size. At the end of development, I ran tests on 4K, 8K, 16K, and 32K page sizes.

3. POSSIBLE ISSUES

Overall, development went pretty smoothly with the bugs not taking too much time to fix. My biggest issues while developing and possibly now might be due to the way I indexed my paging system:

1-index: Throughout development, I constantly had issues due to my use of a 1-indexed virtual and physical paging system. Rather than starting at page 0, I chose to start at page 1 because it made more sense to me. This made my implementation harder than it would have been if I used a 0-indexed system because of the way arrays are naturally 0-indexed.

5. COLLABORATION AND REFERENCES

Piazza: I looked on Piazza to find clarification for a few of the functions, mainly page_map() due to the writeup being vague.

Internet: As for resources on the internet, I searched for tutorials on how multi-level page tables worked since I had some confusion on that topic and how to use bitwise operators to manipulate my bitmap, including this on the writeup: cprogramming.com/tutorial/bitwise_operators.html