Assignment 3 Report

Michael Carlucci (mjc548)
Daniel Bezden (dvb27)

November 2020

1 Virtual Memory Function Logic

1. **SetPhysicalMem**
   We implemented SetPhysicalMem by calculating the number of bits for the directory, page table, and offset. We then malloced everything and used mmap to allocate the physical memory required.

2. **add_TLB**
   We implemented add_TLB by adding a new TLB entry to the end of the linked list and initializing its values. If the TLB is full, we remove the least recently used entry by finding the entry with the highest last_used value. Lastly, we decrement the index of each entry after the entry that was removed.

3. **check_TLB**
   We implemented check_TLB by finding the entry whose page number matches the page number of the virtual address that was passed in, and incrementing last_used for every other entry. We also increment checks, and we increment misses if an entry is not found.

4. **print_TLB_missrate**
   We implemented print_TLB_missrate by calculating and printing the missrate.

5. **Translate**
   We implemented Translate by first checking whether the TLB has an entry for the virtual address. If it does, return the physical address from check_TLB. Otherwise, use bitwise operations to find the directory index, page table index, and offset. We then call add_TLB to add an entry for this mapping. Then we find the corresponding page table entry and return the physical frame plus the offset.

6. **PageMap**
   We implemented PageMap by using bitwise operations to find the directory index, page table index, and offset. If the page table mapped to the directory index has not yet been initialized, initialize it by mallocing it and setting all of its entries to one. Then, find the correct index in that page table and map it to the physical address that was passed in. Instead of mapping it to the actual address, it maps it to the frame number.

7. **get_next_avail**
   We implemented get_next_avail by searching through the virtual bitmap to find a series of consecutive available virtual addresses equal to num_pages. It uses get_virtual_bit to check for availability. Once the available addresses are found, it uses set_virtual_bit to mark them as used. It then returns the virtual address pointing to the beginning of the whole thing.

8. **myalloc**
   We implemented myalloc by first checking if it has been called yet. If it has not then we call SetPhysicalMem. Then, we calculate the number of required pages by dividing num_bytes by the page size and rounding up by one if there is a remainder, This guarantees that it will need at least one page and it will have enough pages for any bytes past the initial page. It then calls get_next_avail to determine the virtual address. Then it checks whether the directory entry at the index specified by the virtual address is already mapped to a page table. If it is not, then loop through all of the page tables to find a table with the necessary number of unused entries and map the directory entry to that page table. Lastly, for each required page, use the physical bit map to find the first available frame and then call PageMap to map the page to the frame.

9. **myfree**
   We implemented myfree by first calculating the number of pages to be freed as well as the directory index, page table index, and offset. We then go through the virtual bit map and clear the bit for each page as well as the physical bit for each frame, and we set the page table entry to negative one.

10. **PutVal**
    We implemented PutVal by first checking whether the value can fit entirely within the page which the virtual address is in. If it does, it performs a simple memcpy and returns. Otherwise, it will memcpy as much as possible to the first page and then find the virtual address of the beginning of the next page, calling Translate to find the corresponding frame, and copying the rest of the value into that frame either until the frame is full or the value has been fully copied. It will keep doing this until all of the value has been copied.

11. **GetVal**
    We implemented GetVal by using an identical process to PutVal except copying from pa to val instead of val to pa.

12. **MatMult**
    We implemented MatMult by having three running for loops. The outermost loop runs through the rows of the first matrix, the middle loop runs through the columns of the second matrix, and the last loop runs through the columns of the first matrix and the rows of the second matrix. At each iteration of the third loop, we find the virtual address of the specified index in both matrices the same way it is done in the test.c file. We then use GetVal to set two variables to the values saved in those indices. We then multiply those two numbers together and add them to a sum variable. Once the third loop is finished running, we get the virtual address of the answer matrix and put the sum value into the needed index. Once all loops are finished running, then the answer matrix should have the correct matrix multiplication values.

2 Benchmark Output

- **Part 1 Output**
  To test our program, we ran the test.c benchmark and looked to see if our output was correct. To our pleasure, our program correctly multiplied both matrices and output a 5x5 matrix with the value 5 in each index. On top of this, the benchmark file output that our free function worked correctly as well, therefore, we are fairly confident in our program's correctness (more on this in the page size support section). Besides correctness, we also wanted to measure the speed of our program. We measured the speeds of GetVal and we arrived at an average time of 186 microseconds without TLB and page size 4 bytes. For page size 2 bytes, 192 microseconds. We also measured the speeds of PutVal and we arrived at an average time of 70 microseconds without TLB and page size 4096. For page size 2 bytes, 38 microseconds.

- **Part 2 TLB Output**
  The observed TLB missrate for 2 bytes were 24.75%. For 4 bytes, the missrate was 0.75%. For 16 bytes, the missrate was 5.25%. For 128 bytes, the missrate is 0.75%. We measured the speeds of GetVal and we arrived at an average time of 169 microseconds with TLB and page size 4 bytes. For page size 2 bytes, 2819 microseconds. We also measured the speeds of PutVal and we arrived at an average time of 70 microseconds with TLB and page size 4096. For page size 2 bytes, 1493 microseconds. Our runtimes with TLB were much slower than without. Our reasoning for this is that translate just utilizes bit operations to go to the right page table and index of said page table. While TLB forces us to go through a linked list to find the correct page table which takes much more time.

3 Page Size Support

- In order to support multiple page sizes, the page size was never hard coded into our program. Instead, it was calculated within the program and that variable was used for the rest of the program. In this way, we were able to have our program support different page sizes even when the default size is 4 bytes.

4 Possible Issues

- One possible issue with our code has to do with page size. If we have really small pages with lots of things being malloced, then problems could arise. This is because values could possibly go between pages, which is not ideal. Besides this, our program works properly in all other cases.

5 Difficulties Encountered

- While most of the project was not overly difficult, there were a few aspects that proved to be more difficult than others. For example, implementing myalloc proved to be difficult. The difficult aspects of it were finding one page table with consecutive open entries in the case where there were multiple pages. In addition to this, working with GetVal to get values that lie on multiple pages and frames proved to be a daunting task.