John Quinn, jjq23

A Report on my Virtual Memory Interface

1. The virtual memory API worked as such
    a. The set_physical_mem() function is called upon the first call of t_malloc(). What it does first is declare the number of frames, pages, and tables necessary for the physical and virtual memory blocks. Afterwards, the physical memory block itself, comprised of an array of pte_t arrays of size PGSIZE, is allocated and initialized. The virtual and physical bitmaps are then after allocated and initialized. The directory and tables are set up as well, with the very first entry in virtual memory set to 0xAAAAAAAA to declare that it is off limits. The directory and first table are then put in physical memory, with the appropriate bits set up in the bitmaps. Finally, the TLB is initialized with its values and blocks of addresses set up.
    b. The translate() method simply calculates the physical address by manipulating the virtual address. The function takes the vpn and calls check_TLB(), where it checks the entry corresponding to the vpn and sees if it has a mapping. If it does, the hit counter on the TLB increases and the corresponding physical address is sent back. Otherwise, the miss counter on the TLB increases and an error value is sent back. Upon receiving the error value, the indexes for the directory entry and table entry are used to pick out the correct physical address. Once calculated, the function calls add_TLB(), where both the virtual and newfound physical addresses are added into the TLB. The function either returns the value stored in the TLB, or the value calculated inside the method.
    c. The page_map() method works similar to translate(), making out the proper page table entry for the virtual address, and it checks if any value is in there, and if not, the passed in physical address is added. The corresponding bitmap values are set to 1.
    d. The t_malloc() function works as such. Upon the first call, a call to set_physical_mem() is made. After this, the get_next_avail() function is called, where the virtual bitmap is iterated through to find the first empty page table entry to be used. After this, the same is done to the physical bitmap, thus obtaining the corresponding frame index. Next the function goes into a while loop, where the page frame address is retrieved and page_map is called. Upon completion is a check of the size of the allocated data. If the size exceeds PGSIZE, the while loop continues, otherwise the loop ends. The function then returns a pointer to the newly allocated memory.
    e. The t_free() function works very similar to t_malloc. Inside of a while loop, the page frame address and frame Index are obtained, the bitmap entries are set to 0, the data inside the frame is nullified, and the corresponding table entry is set to 0.

Once again is a check for the size of the data, with the loop iterating again if the size exceeds one page.

f. The put_value() and get_value() functions are near identical. Like t_malloc() and t_free(), there is a while loop that iterates based on the size of data being accessed. For put_value() the data from the val pointer is copied into the physical frame, and for get_value() the data is copied from the frame into the val pointer.

2. Here are the observed TLB miss rates observed across various tests based on different thread counts and page sizes.

(TLB miss rates)	**test (1 thread)**	**mtest (3 threads)**	**mtest (15 threads)**	**mtest (30 threads)**	**mtest (51 threads)**
4K PGSIZE	0.007444	0.022005	0.024390	0.024725	0.024866
8K PGSIZE	0.007444	0.014778	0.016393	0.016620	0.016716
16K PGSIZE	0.007444	0.007444	0.008264	0.008380	0.008428

3. The only major issue with the code is that the output for the multi_test benchmark becomes unstable for page sizes of 32K and up, as either the matrix data becomes corrupted, or an address becomes unproperly freed, however all page sizes under it produce the same, correct output every time in both single threaded and multi-threaded benchmarks.