Pedro Torres (pt397) and Ivan Zheng (iz60)

Project 3 Writeup

Code Breakdown

`Set_physical_mem`

In implementing the `set_physical_mem` function, we allocated all the resources we needed for the other virtual memory functions, including all our locks for thread safety, bitmaps checking page allocations, and our physical page itself. We also set and calculated all necessary variables, including the page directory bits, the offset bits, and the page table bits.

`TLB_add`

This function takes care of adding items to the TLB. The TLB is an array of `tlb_entry`, which is a struct that contains a virtual address, a corresponding physical address, and a valid bit. The `TLB_add` function relies on a helper method called `TLB_hash`, which will hash the entry based on its virtual address and return an index to put this translation into the TLB.

`TLB_check`

This function checks the TLB to see if the virtual address being passed in corresponds to a valid physical address. It also relies on the `TLB_hash` function, checking the TLB entry at the index of the hash and returning the `pte_t` page frame number if it exists already.

`print_TLB_missrate`

This function uses two global variables, `tlb_miss` and `tlb_total` to calculate the number of misses, which are incremented when `TLB_check` returns `null`.

`Translate`

The `translate` function takes in the virtual address and strips it apart into its individual bits that correspond to the page directory bits, the page table bits, and the offset bits. We then check at the correct place in the page directory for the location of the page table, and then go to the page table and use the page table bits to find the actual location of the page frame. Once we have found the actual location, if it exists, we recombine that address with the offset to regain the physical address translation.

`Map_page`

This function is responsible for the mapping of our virtual address to a corresponding physical address. It first strips the virtual address apart like it does in `translate`, and moves through the levels, checking if any of the entries are null and allocating them if it is. The allocation makes use of `get_next_avail`, which has a physical address version too. Its task is to check the bitmap and find the next available page by checking the bits at each index of the bitmap until one that can be used is found.

`N_malloc`

This function is how the user allocates and is given a virtual address. The function gets the next available free page frame, and promptly passes it to the map_page function in order to map the page, and once it has been mapped and validated in the bitmap, the address is returned to the user.

`N_free`

The function takes in the virtual address and checks the bitmap to see that the one calling free is that originally allocated the memory and then frees that memory if it is within the bounds. After all the checks, it is possible to set the bits back to 0 in the bitmaps.

`Put_data`

The function takes the virtual address and the source and first translates the virtual address into a physical address using `translate`. It then writes the amount of bytes available on that page starting from the offset, and continues moving to the next pages to write more by adding the amount copied to the virtual address and the src address and recalculating the physical address to get the next physical address. That way, we can keep copying, and copy data over multiple pages.

`Get_data`

Get data applies a similar concept to put data, translating the addresses and instead, writing the data in the page starting from the offset to the given address location. It also jumps to the next page by incrementing by the amount copied once it reaches the limit of the page frame.

Benchmark
Our output for test.c:
“Addresses of the allocations: 1000, 3000, 4000
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 0.964286
”

Output for mtest.c with 51 threads:
“Allocated Pointers:
1e000 5000 3000 22000 4000 1000 1b000 21000 1d000 f000 b000 6000 27000 14000 9000 24000 7000 20000 8000 e000 c000 a000 23000 32000 d000 10000 26000 2d000 13000 12000 2b000 1f000 34000 11000 16000 15000 31000 2a000 2c000 2f000 28000 30000 1c000 19000 17000 1a000 29000 25000 33000 2e000 18000
initializing some of the memory by in multiple threads
Randomly checking a thread allocation to see if everything 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 threads!
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!
TLB miss rate 0.875927
”

Output for 4KB * 17 for both test and mtest:
Test.c:
“Allocating three arrays of 400 bytes
Addresses of the allocations: 10000, 30000, 40000
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 0.964286
”

Mtest.c:
“Allocated Pointers:
150000 40000 10000 30000 230000 1b0000 50000 200000 60000 70000 80000 90000 b0000 c0000
140000 a0000 1c0000 d0000 e0000 1f0000 1e0000 100000 110000 f0000 2c0000 290000 330000
130000 120000 2a0000 170000 2f0000 2b0000 160000 240000 2e0000 280000 250000 190000 1a0000
2d0000 300000 260000 270000 320000 310000 180000 220000 340000 210000 1d0000
initializing some of the memory by in multiple threads
Randomly checking a thread allocation to see if everything 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 threads!
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!
TLB miss rate 0.875817
”

Extra Credit

In order to implement the 64-bit, 4-level version, we first changed all the data types to fit a 64-bit design. For example, our TLB, which originally used a `uint32_t`, was changed to `uint64_t` design, along with any other data types that did not convert automatically. After this, we changed how we parse through the virtual address, dividing it to include 2 more levels of page tables, and adjusting all of the methods

that relied on the bit calculations to match. Once all of this was done, the 64-bit version was able to work properly.

For implementing fragmentation, we had to look into adding an inline data structure so we can manage mallocing fragments out. This required each fragment of memory to be prefixed with a `header_t` object, which has its top 15 bits holding the size of the current block, and the last bit holding if it was free or not. If a call to n_malloc was of bytes close to the page size or larger, we would allocate however many pages were needed as normal. If the number of bytes was less than that, we would fragment a page, finding the next big enough block in a page or using a new page as a fragmented page.

- Note, the free test at the end of test.c and mtest.c is not friendly with our implementation of fragmentation, since an address can be malloc with x bytes, freed, and never appear again if we repeatedly malloc with different bytes y. (The test currently assumes that a freed address will be malloced again)