PROJECT 3 REPORT:
Tasha Pais (tdp74)
IMPORTANT NOTES:
OLD add_TLB(void* va, void* pa) changed to NEW add_TLB(void* va, pte_t* pte)
OLD struct tlb changed to new tlbe_t

These changes all make sense in this implementation. For example our page directory and page tables are of type pde_t* and pde_t* (arrays of those types) so it makes sense to have our tlb type also be an array of entries (tlbe_t*). Also, since translate returns a pte_t* it makes sense to add that pte_t* directly to our TLB instead of computing the physical address and storing that, after all, page table entries give us more information than a physical address, and doesn't require us to deduce the page number from the physical address when we don't need to.

1. Detailed logic of how you implemented each virtual memory function.
    a. set_physical_mem():
    This method runs once, during the first call to t_malloc(). It's purpose is to initialize data structures needed for our memory manager.
    The logic for this implementation is obvious when we have access to the data listed below, we know how to shift bits to quickly compute things, and we know basic things, like that there are 8 bits in a byte and an unsigned long in 32-bit architecture takes up 32 bits. The special case for this implementation might be the fact that instead of addresses in our PDEs and PTEs, we store the physical page number of the page referred to in an entry. This is done by computing how many bits we need to represent a physical page number, using the number of physical pages which was computed using data we have access to.

    Data we have access to:
        Physical memory size (MEMSIZE)
        Page size (PGSIZE)

    Things we compute with that data (global variables):
        Number of page offset bits (pg_offset_bits)
        Number of page table bits (pt_bits)
        Number of page directory bits (pd_bits)
        Number of physical pages (num_p_pages)
        Number of virtual pages (num_v_pages)
        Number of bits for physical page number in PDEs and PTE (entry_ppn_bits)

    Data structures initialized using those computations (global variables):
        Physical memory (p_mem)
        Physical bitmap (p_map)
        Virtual bitmap (v_map)

b. translate():
We first check to see if there is a translation present in our TLB. If there is, we return the page table entry (pte_t*) that is stored in the TLB, if non, we proceed with the translation. This function takes a virtual address which can be easily used to compute the page directory entry index (pde_index) and the page table entry index (pte_index). With the page directory entry index we can index into the given page directory (pgdir) and retrieve the page directory entry. The page directory entry tells us if the entry is valid, if the valid bit is cleared, we return NULL. If the valid bit is set, we proceed and parse the physical page number of the page table (pt_ppn). Now that we have the physical page number of the page table and we know the base address of our physical memory and the sizes of its pages, we can compute the address of the page table. We do this by using the page size as a constant and the physical page number as a scalar, which gives us an offset we can add to our base address of our physical memory. In the implementation of this library, every physical address that is not in the page directory is computed using this method. We repeat this process with the address of the page table using the page table index, except we don't parse the physical page number within the entry, and instead we return the entire entry if it's valid. As mentioned earlier, if an entry is invalid, we return NULL. Returning the page table entry helps us have access to the entire entry in other functions. Before we return the valid entry, we add it to our TLB.

c. page_map():
This function follows the same logic to parse the page directory index and page table index and to compute physical addresses. While mapping the given virtual address to the given physical address, this function will create a new page table if needed and sets the new page tables physical page number and valid bit as a page directory entry in the page directory using bit shifting and the logical OR (|) operator. Since the physical address is given, we use the base address of our physical memory, the given physical address, and the page size to compute the physical page number (ppn). Again, we use the bit shifting and the logical OR operator to set bits in the new page table entry. Because this is a new page table entry, it doesn't exist in our TLB, so we add the new virtual address to page table entry mapping in our TLB.

d. t_malloc():
At the beginning of the first call to this function, set_physical_mem() is called and our memory is ready to start allocating memory. Given the number of bytes requested, we compute the number of pages needed. For every page needed, we get the next available virtual page and next avaiable phyiscal page. We use page map to set the mapping from the virutal page to the physical page. Once we've mapped all pages necessary, we return the base address of the virutal

address, which according to our logic, is the first virtual address that was mapped to a physical address during that t_malloc() call.

e. t_free():
This function takes a virtual address and number of bytes to free. Using the page size and the number of bytes to free, we offset the virtual address in page size increments while the offset is less than or equal to the allocation size and translate each virtual address. If the virtual address computed corresponds to a valid page table entry, we clear the bit corresponding to that virtual address in the virtual bitmap. We then parse the physical page number using the returned page table entry from translate(), and clear its bit in the physical bitmap. Now that we don't need that page table entry, we use memset() to clear all the bits in that page table entry.

f. put_value() and get_value():
Both of these functions take a virtual address, a pointer to a value, and the number of bytes corresponding to the value pointer. Like t_free(), these functions use the virtual address and the number of bytes to offset the virtual address in page size increments while the offset is less than or equal to the number of bytes to either put or get. Like t_free() we translate() each virtual address and obtain its page table entry, if valid. We use the method mentioned in the translate() description to compute the physical address we are either getting data from and putting data at. We then use memcpy() to copy the values to/from memory from/to the location pointed to by the given value pointer.

2) Benchmark output for Part 1 and the observed TLB miss rate in Part 2.

a. Output from ./test

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 400, 800, c00
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 0.009926

b. Output from ./multitest
Allocated Pointers:
400 2c00 5400 7c00 a400 cc00 f400 11c00 14400 16c00 19400 1bc00 1e400 20c00 23400 25c00 28400 2ac00 2d400 2fc00 32400 34c00 37400 39c00 3c400 3ec00 41400 43c00 46400 48c00 4b400 4dc00 50400 52c00 55400 57c00 5a400 5cc00 5f400 61c00 64400 66c00 69400 6bc00 6e400 70c00 73400 75c00 78400 7ac00
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 0.077375

3) Support for different page sizes
This is primarily handled in the set_physical_mem() function. The entire memory management scheme in our code revolves around the value of PGSIZE. Changing PGSIZE will alter the memory layout, including the number of pages, the structure of page tables and directories, and the configuration of the TLB.

- Calculating the Number of Pages:
  For physical pages: The code calculates the number of physical pages (num_p_pages) by dividing the total size of physical memory (MEMSIZE) by the page size (PGSIZE). This division determines how many pages of size PGSIZE fit into the physical memory.

  For virtual pages: The calculation starts with determining the pg_offset_bits, which is the number of bits needed to represent the PGSIZE. This is achieved by right-shifting PGSIZE until it reaches zero, effectively counting the number of binary digits. Then, the number of virtual pages (num_v_pages) is calculated using the formula 1 << (32 - pg_offset_bits), assuming a 32-bit address space.

This formula calculates the number of pages by raising 2 to the power of the difference between the total address bits (32) and the bits needed for the page offset (pg_offset_bits).

- Memory Allocation for Bitmaps:
  The code creates bitmaps (p_map for physical pages and v_map for virtual pages) to keep track of which pages are in use. The size of these bitmaps is proportional to the respective number of pages, with each bit in the bitmap representing one page.

- TLB Configuration:
  The Translation Lookaside Buffer (TLB) is also configured based on the page size. The TLB index bits (tlb_index_bits) are calculated based on the number of TLB entries, and the TLB tag bits (tlb_tag_bits) are derived using the formula 32 - pg_offset_bits - tlb_index_bits. This calculation partitions the virtual address into parts used for indexing and tagging in the TLB.

- Page Table and Directory Bits:
  The code calculates the number of bits needed for page table entries (pt_bits) and page directory entries (pd_bits). These calculations are vital for determining how the virtual address space is divided among the page directory index, the page table index, and the offset within each page.

4) Possible issues in your code
Error Handling in Memory Allocation:
When allocating memory for p_mem, p_map, and v_map, there is no error checking to ensure that the memory allocation was successful. If malloc fails (for example, due to insufficient memory), it will return NULL, which our code does not currently check for. This can lead to undefined behavior or segmentation faults when attempting to access these pointers.

Concurrency Issues with Global Variables:
Our code uses global variables (like tlb_lookups, tlb_misses, etc.) which are accessed and modified in multiple functions. If these functions are called concurrently in a multithreaded environment, it could lead to race conditions. While we have mutex locks for some functions (t_malloc, t_free, etc.), the accesses and modifications to these global variables are not always protected by these mutexes.

TLB Miss Handling in add_TLB Function:
The add_TLB function increments tlb_misses each time it is called. This approach might not accurately reflect the actual number of TLB misses, as the function seems designed to add a translation to the TLB regardless of whether it's a miss or not. A more accurate approach would be to increment tlb_misses only when a miss is detected.
Return Value of add_TLB Function:

The add_TLB function always returns -1. If this return value is meant to indicate success or failure, it should be adjusted to reflect the actual outcome of the function (e.g., return 0 for success and -1 for failure).

Translation Lookaside Buffer (TLB) Size and Indexing:
The TLB is allocated based on TLB_ENTRIES, but there is no check to ensure this number is adequate or optimal for the simulated system. Additionally, the calculation of tlb_index_bits and tlb_tag_bits might not properly handle all corner cases, especially if TLB_ENTRIES is not a power of 2.

Handling of Page Directory and Table Entries:
In functions like translate and page_map, our code assumes that the page directory and table entries follow a specific format and structure. If the format of these entries changes or if they need to handle more complex scenarios (like larger address spaces or additional attributes), the current implementation may not be sufficient.

Efficiency of Bitmap Operations:
In get_next_avail and get_next_ppn, our code iterates over the entire bitmap to find the next available page or page number. This linear search could be inefficient, especially for larger bitmaps. Optimizing these operations to more quickly find free pages could significantly improve performance.

Memory Leak in set_physical_mem:
If set_physical_mem is called multiple times (e.g., if init is mistakenly reset), it will allocate new memory for p_mem, p_map, v_map, and tlb without freeing the previously allocated memory, leading to a memory leak.

5) Extra Credit
N/A

6) Collaboration and References
GNU C Manual: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf
Recitation questions with TA Adithya