User-Level Memory Management

Names:

Christopher Naporlee - cmn134 Section 3

Michael Nelli - mrn73

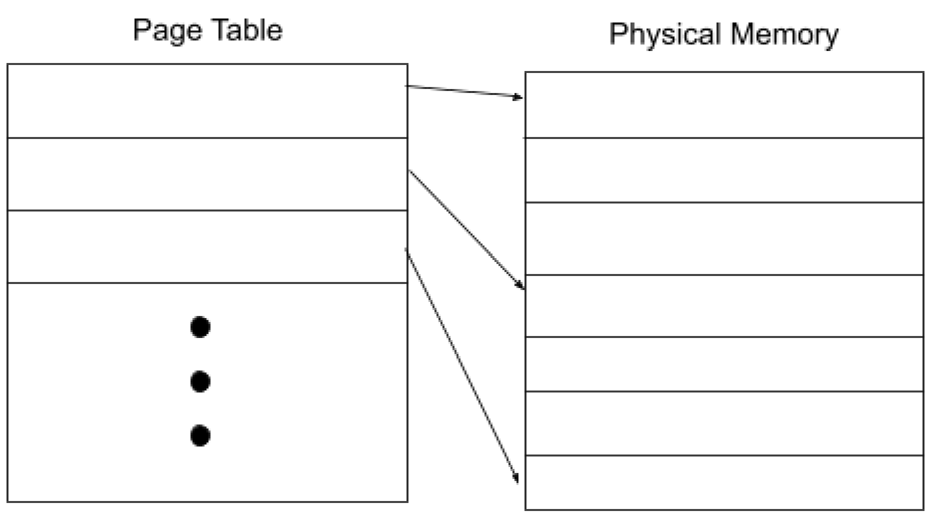
Section 3

Functions:

1. set_physical_mem()
    a. Called upon the first call to a_malloc()
    b. mmap’s memory for physical memory is determined by how large MEMSIZE is.
    c. Sets many globals to keep track of important variables
        i. Number of offset bits
        ii. Number of page table bits
        iii. Number of page directory bits
        iv. Number of total pages
    d. Two bitmaps are initialized
        i. A physical bitmap that keeps track of which physical pages are free
        ii. A virtual bitmap that keeps track of which entries within the page tables are valid
    e. Sets the first page of physical memory to be the page directory for easy access to the directory
2. translate()
    a. Checks TLB for a valid virtual address to physical page translation
    b. Looks up a physical page by traversing through the page directory and it’s tables
        i. Uses macros to access the:
            1. High bits - used to index the page directory
            2. Middle Bits - used to index the page table
            3. Low Bits - saved for the end in which the offset is factored into the final address translation
    c. If no valid entry is found returns 0

3. page_map()
    a. Similar to translate, it goes through the page directory and tables to find the spot in which the virtual address should map itself to a physical address.
        i. Uses the same macros as translate() to access the upper, middle, and lower bits of a virtual address.
    b. If the virtual address requires traversing into an unmapped table, allocates a new page table and traverses to the table
    c. If the entry is invalid, the entry is updated so that the virtual address correctly maps to the physical page.
    d. Adds that address translation into the TLB.
4. get_next_avail()
    a. Simply gets the next available free physical page
    b. Returns the physical page number
    c. This function is usually only called with an argument of 1 so that we can map an arbitrary page to an address

5. a_malloc()
    a. Checks to make sure that memory has been initialized
    b. Uses two locks
        i. Map_lock - mutex lock to enforce coherence between multiple threads looking for valid entries and free pages
        ii. Table_lock - mutex lock to allow only one thread to map to the page table at a time
    c. Allocation process:
        i. Find non valid virtual entries that can be used to map pages by using virtual bitmap.
        ii. If there are no free entries, return NULL
        iii. For each free virtual entry, find a singular physical page to map to it using physical bitmap.
        iv. This process of finding entries and mapping a page to them one by one allows us to have the illusion of contiguous memory, while the true physical pages may be scattered all throughout the memory.
    d. Example (4K Pages with an alloc size of 12000 bytes will call for 3 pages to be allocated):



While the arrows are arbitrary, it is an example that it does not matter where the true pages lie within memory.

6. a_free()
    a. Upon entering, it calculates the amount of pages the user is requesting to free.
    b. Then it checks using the virtual bitmap, that all the entries they are asking to free are valid mappings and can actually be free'd
    c. Once the previous check is complete and working, it works through the bitmap, translating each virtual address to their corresponding physical address, clearing the valid bit of entries and marking the pages of the physical pages as free.
    d. Also invalidates the entry if it is in the TLB
7. put_value()
    a. Copies data from the virtual address into a physical frame.
        i. Obtains the physical location of the virtual address by calling translate().
8. get_value()
    a. Given a virtual address, copies data from its corresponding physical translation into a given value.
        i. Obtains the physical location of the virtual address by calling translate().
9. mat_mult()
    a. Multiplies two matrices.
        i. Stores values in a third matrix using put_value()
        ii. Obtains values from matrices using get_value(), which we can then perform operations on.

TLB Functions

1. add_TLB()
    a. Takes a virtual address and makes a “tag” out of the upper bits of the address
    b. Then maps the address to an index in the TLB using the modulo of the number of entries (tag % TLB_ENTRIES)
    c. Maps it to the entry and sets it to valid
2. check_TLB()
    a. Checks the TLB for a virtual address to physical translation and increments the total amount of TLB lookups
    b. If one exists, returns the physical page number and returns
    c. If one does not exist, it increments the total amount of TLB misses.
3. print_TLB_missrate()
    a. Simply prints out the TLB miss rate by doing: Total Number of TLB misses / Total TLB lookups.

Benchmark Outputs:

test.c with a matrix size of 5x5

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 1000, 2000, 3000
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 0.000000
```

test.c with a matrix size of 15x15

```
Allocating three arrays of 4000 bytes
Addresses of the allocations: 1000, 2000, 3000
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 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 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
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 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 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
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 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 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!
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
Freeing the allocations!
Checking if allocations were freed!
free function works
TLB miss rate 0.000000
```

multi_test.c with a size of 5x5

```
cmn134@snow:~/Assignments/CS416/project3/benchmark$ ./test
Allocated Pointers:
9000 1000 f000 3000 11000 5000 7000 b000 d000 13000 15000 19000 17000 1b000 1d000
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
Freeing everything in multiple threads!
Free Worked!
```

IMPORTANT NOTE: The Tlb miss rate reports 0 for both tests because there are significantly more entries in the TLB than how many pages are allocated. This mechanism of putting the most recently mapped entries into the TLB provides for extremely fast address translation for recently malloc'd pages.

Support for different page sizes

- Support for different page sizes is done by calculating at physical memory setup how many bits to use for the directory
  - Page bits = log_2(PGSIZE)
  - Page Table bits = log_2(PGSIZE / sizeof(entry))
  - Page directory bits = # bits in addr - page table bits - offset bits
- Above is useful for most page sizes. However, for pages that are really big (like 128k) we use a different algorithm.
  - Page bits = log_2(PGSIZE)
  - Page Table bits = (# bits in address - offset bits)/2
  - Page directory bits = Page Table Bits + 1
- We use the above for extremely large pages, because if we don’t we will have little to no bits for our page directory index.
  - While this will cause page tables to not be completely filled with entries, it is the easiest way to implement extremely large page sizes without complicating code.

Possible Issues

- A possible issue with the code is that to ensure that the user can have continuous pages without having the actual pages be next to each other, the page table entries must be contiguous. This allows for the virtual address + Page_size to automatically address the next malloc’d page by accessing the next entry in the table. However, if the user ends up having serious fragmentation and wants to allocate a new huge pool of memory, this constraint stops the user from accessing the free pages in between the already allocated pages.