Projec 3: User-Level Memory Management Library
CS416 OS Design

Paul Kim
NetID: obawolo

November 23rd 2020: tested from 'cp.csrutgers.edu'

FUNCTIONS IMPLEMENTATION

1. SetPhysicalMem
   The function begins by checking that memory hasn't already been initialized, and proceeds to calculate the figures needed to appropriately index a 2-level page table with the defined page size and memory size. It then initializes the page directory, which resides as a separate entity outside our managed memory. The managed memory itself is then initialized, along with bitmaps for physical and virtual addresses, and the page directory.

2. Translate
   The supplied virtual address is split into its vpn and offset. If the function calling translate wants the actual physical translation (which is the case almost all the time unless the calling function is myfree, which uses translate to index the physical bitmap and has no need for the physical address), the TLB is checked. If the translation is present, the function returns with the physical address + the saved offset.

   If not, the supplied virtual address is split into the 1st-level (page directory) index, the 2nd-level page table index, and the offset. The virtual bitmap is checked to make sure the supplied virtual address is valid, and then the page directory bitmap is also checked to make sure that the calculated page directory index is correct. The function then fetched the page table PFN from the page directory, goes to that PFN, and finally fetches the page table entry. If the calling function is myfree, the entry contents (another PFN) are returned as is (plus the offset). Otherwise, the page table entry PFN is used to calculate the physical address, which is then stored in the TLB. Finally the physical address is returned (plus the offset).

3. PageMap
   The supplied virtual address is split into 1st and 2nd level indexes, and offset. Both the virtual and physical bitmaps are then checked using the supplied addresses to make sure they haven't already been assigned. The function then goes to the appropriate location in the page directory using the 1st level index. If it hasn't been assigned (this is checked using the page directory bitmap), then the appropriate amount of contiguous physical pages are allocated with helper function 'get-avail-phys-mult' to store a 2nd-level page table, and their address is stored in the page directory. The function goes to the 2nd-level page table, fetches the appropriate page table entry with the 2nd-level index, and populates it with the supplied physical address (PFN). Both supplied addresses are marked as assigned.

4. myalloc
   If memory hasn't been initialized, function 'SetPhysicalMem' is called. The number of pages needed are calculated, and function 'get-next-avail' is called to locate the appropriate amount of contiguous virtual addresses. The ending virtual address is calculated based on the provided size of memory requested. Then, going from the virtual address found by function 'get-next-avail' to the ending virtual address, each address is mapped to a free physical address by using function 'get-next-avail-phy'.

5. myfree
   The function calculates the ending virtual address with the provided size and starting virtual address. All addresses in between are checked for validity. If this check passes, the corresponding physical addresses (found via Translate) are marked open for use on the physical bitmap, and the virtual addresses are then marked open for use on the physical bitmap. If any of these addresses have an entry in the TLB, that entry is cleared.

6. PutVal
   The function calculates the ending virtual address with the provided size and starting virtual address. All addresses in between are checked for validity. If this check passes, for each address in the range from start to end (incremented bytewise), the corresponding physical address is found via Translate, and the byte at that physical address is populated with the corresponding byte in the supplied val.

7. GetVal
   Exactly the same as put-value, except the bytes at the supplied val are populated with the corresponding bytes at the physical address.

8. MatMult
   The 2 given matrices, mat1 and mat2 are multiplied with each other. The result is stored at the location given by the supplied answer pointer.

9. TLB
   TLB is implemented as an array of structs, each containing a virtual address, the corresponding physical address, and a valid bit. More implementation details are described in Translate and PageMap above.

BENCHMARK OUTPUT: TLB runtime, miss rate with different page sizes

1. 4K page size

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 1000, 2000
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 of 4096 Bytes Page Size : 0.001875
Total Run time of 4096 Bytes Page Size: 1506 microseconds
```

2. 8K page size

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 2000, 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 of 8192 Bytes Page Size : 0.001875
Total Run time of 8192 Bytes Page Size: 781 microseconds
```

3. 16K page size

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 4000, 8000
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 of 16384 Bytes Page Size : 0.001875
Total Run time of 16384 Bytes Page Size: 553 microseconds
```

4. 32K page size

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 8000, 10000
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 of 32768 Bytes Page Size : 0.001875
Total Run time of 32768 Bytes Page Size: 401 microseconds
```

5. 64K page size

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 10000, 20000
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 of 65536 Bytes Page Size : 0.001875
Total Run time of 65536 Bytes Page Size: 223 microseconds
```

6. 1M page size

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 100000, 200000
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 of 1048576 Bytes Page Size : 0.001875
Total Run time of 1048576 Bytes Page Size: 243 microseconds
```

7. 2M page size

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 200000, 400000
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 of 2097152 Bytes Page Size : 0.001875
Total Run time of 2097152 Bytes Page Size: 177 microseconds
```

8. 4M page size

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 400000, 800000
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 of 4194304 Bytes Page Size : 0.001875
Total Run time of 4194304 Bytes Page Size: 218 microseconds
```

9. 8M page size

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 0, 800000, 1000000
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 of 8388608 Bytes Page Size : 0.001875
Total Run time of 8388608 Bytes Page Size: 216 microseconds
```

ANALYSIS

From results above, the run time of the test became faster as the page size increased starting from 4K until 32K. But after page size of 64K, the run time didn't fluctuate much.

DIFFICULTIES

My physical memory was allocated using all malloc since I couldn't understand mmap. Thus, working on this project was an exercise in precision. Every time a bitmasking operation was implemented, its functionality had to be tested in multiple ways. This sort of "unit-testing" benefitted me a lot when it came to building something so intricate from the ground up.