CS416 Project 3

### User-Level Memory Management

By Nicholas Dundas & Logan Miller

### Layout

*set\_phsyical\_mem, cleanup*

The first function called is ***set\_physical\_mem***, this can either be explicitly called one or more times via the user or automatically called once during the first *t\_malloc* call. with each call resetting all memory and any global variables within my\_vm.c. Whenever it is called it called it calculates the total number of pages in physical memory, number of bits required for the offset, page tables, and page directory. These are all stored in the appropriately named global variables. Next the memory and the corresponding bitmap are both allocated. The bitmap is a direct mapping of bit to status of a page where 1 indicates free and 0 indicates used. Next, we calculate the number of required pages for the tlb and outer page directory. These will depend on the PAGE\_SIZE and TLB\_ENTRIES macros respectively, thus they will never shrink or grow. The corresponding bits in the bitmap are set to 0 and memset is used to zero out all page directory and tlb entries. Finally, *atexit* is called with a function called ***cleanup*** to free memory and the corresponding bitmap on program exit. Once the initial setup is done the memory layout can be thought of as broken into three distinct chunks. As shown by the diagram below.

|  |  |  |
| --- | --- | --- |
| OUTER  PAGE  DIRECTORY | TLB | DATA |

(Sizes are not always reflective of actual memory but are to show average scales between them)

The *outer page directory* will contain entries that point to some page in *data* which represents a *page table*. A *page table* will contain entries that point to various pages containing the actual data a user may store. A zero in either a *page table* or the *outer page directory* represent the empty entry indicating nothing as the zeroth page is always taken up by the *outer page directory*. This setup creates a two-level page table where we need not worry about contiguous physical pages in *data,* but only logical ones contained within the *outer page directory* and *page tables* themselves. The management of traversing through the two levels of indirection are handled in functions that are to be explained.

The *translation look-aside buffer* or *TLB* is stored in the pages directly after the *outer page directory*. Its size is determined by TLB\_ENTRIES \* sizeof(tlb\_ent) where tlb\_ent is a struct containing two unsigned ints representing the virtual page and physical page. The *TLB* uses this struct to create the direct mapping from virtual to the physical pages and bypass the indirection required by the lookup of physical pages.

MEMSIZE must be large enough to handle the *outer page directory* and *TLB* otherwise the behavior is undefined when set\_physical\_mem and by extension t\_malloc and its family of functions are called. MEMSIZE may also not exceed MAX\_MEMSIZE or the behavior is undefined as addresses up to 32bits are only supported.

### Logic

*translate, page\_map, t\_malloc, t\_free, get\_next\_avail, findContSpace, is\_page\_table\_empty, mat\_mult, TLB\_check, TLB\_add*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| OUTER  PAGE  DIRECTORY | TLB | PAGE TABLE | MAPPED DATA | PAGE TABLE | MAPPED DATA | UNUSED DATA |

When ***t\_malloc*** is called and assuming *set\_physical\_mem* has already been run, it will first calculate the number of required pages for the *t\_malloc* call. Next ***find\_cont\_space*** will walk the *outer page directory* and subsequent page tables to find enough entries to find the required number of pages. It returns the first virtual address that meets the criteria. If it fails it will return 0 and by extension, make *t­\_malloc* return 0. If it successfully finds enough it stores the resulting virtual address to return to the user and calls ***page\_map*** on all contiguous virtual pages. *page\_map* automatically creates page tables and page table entries if they do not exist and marking them as used in the bitmap. *page\_map* internally calls ***get\_next\_avail*** which finds the next free physical page in memory via searching the bitmap. It also automatically calls translate subsequently adding an entry to the TLB. Multiple calls to *t\_malloc* may result in a memory layout like shown below.

(Note that despite the page tables and data being separated the user only sees contiguous mapped data)

***t\_free*** takes a virtual address and frees n number of bytes after it. It does this by walking the *outer page directory* and page tables, flipping each used page to free. If any are already free it returns -1, otherwise returns 0 on successful deallocation of all pages. Internally it also checks if page tables are empty to free them up if they are unused with ***is\_page\_table\_empty***.

***put\_value***and***get\_value*** work very similarly to each other by both walking the *outer page directory* and the page tables moving data in and out of the memory. They also implement memmove instead of memcpy internally to allow for overwriting itself although this may be unsafe and impossible to implement from a user standpoint if the number of bytes + the offset exceed PAGE\_SIZE but regardless is kept in as a safety feature.

***mat\_mult*** is a built-in function to showcase how to use the put and get functions via matrix multiplication. You can easily address any offset within a page even if the offset may exceed the total size of a page as *put\_value* and *get\_value* automatically handle the abstraction of walking the page directory for all the data IO operations.

***translate*** directly maps a virtual address to a physical one by finding the index in the outer page table, taking the page value it contains pointing to a page table. Finally using that page table to find the actual physical page stored and using that with the addition of the offset to return the physical address. It additionally checks whether the virtual page number (all the bits left of the offset) is stored within the TLB with ***TLB\_check***. It does this with an extremely simple “hashing” function which just takes the modulus of the virtual page number. If finds the virtual page number at the index with the hash it takes physical page stored multiplying it by PAGE\_SIZE and adds the offset to get the physical address. Otherwise, it calculates as normal and stores the both the virtual and physical page number in the appropriate TLB entry with ***TLB\_add***.