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. Each call to set\_physical\_mem will reset 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 and the 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* is called. MEMSIZE may also not exceed MAX\_MEMSIZE or the behavior is undefined as addresses up to 32bits are only supported.

### Main Functions

*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 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. t\_malloc 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.

***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.

***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 in the bitmap to free and setting the entries to 0. It returns 0 on successful deallocation of all pages or -1 on error. It checks if page tables are empty to free them up if they are unused with ***is\_page\_table\_empty*** which returns 1 if all entries are 0 otherwise returns 0.

***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 as needed. 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 address any offset within a page even if the offset and matrix size in bytes 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 to get the page table, and using the inner index to get the physical page. Finally, it adds the offset to return the physical address. It additionally checks whether the virtual page number is stored within the TLB with ***TLB\_check***. It does this with a simple “hashing” function which is just vpage % TLB\_SIZE. If finds the virtual page number at the index with the hash it takes physical page stored, multiplying it by PAGE\_SIZE, and adding the offset to get the physical address. Otherwise, it calculates the indexes and preforms the indirections as normal storing the both the virtual and physical page number in the appropriate TLB entry with ***TLB\_add***.

### Helper Functions

first\_set\_bit, flip\_bit\_at\_index, get\_bit\_at\_index, bit\_to\_long, index\_to\_va, tu\_malloc, print\_mem, print\_page, print\_va

### Bitwise Helpers

***first\_set\_bit*** returns the first bit in a char that is set to 1 starting at index 0. returns -1 if num is 0

***flip\_bit\_at\_index*** flips a bit changing from 1 to 0 or vice versa at a specific bit index in a bitmap

***get\_bit\_at\_index*** returns the Boolean value of a bit at the specified index in a bitmap

***bit\_to\_long*** converts a series of bits starting at start and of length len and returns the resulting long

### Function Helpers

***index\_to\_va*** converts three unsigned ints representing the indexes into the *outer* page directory, page table, and offset into the physical page, into a single virtual address.

### Debug Functions

***tu\_malloc*** is a wrapper around t\_malloc that converts the void pointer to an unsigned int

***print\_mem*** will walk the *outer page directory* and page tables printing any non-empty entries and reporting possible errors if they are marked as free when not empty.

***print\_page*** prints the contents of an entire page with len columns byte by byte in hexadecimal.

***print\_va*** prints the component parts of a virtual address in plain English and decimal.

***print\_TLB\_missrate*** prints a float representing misses / (hits + misses)

### Compilation, Limitations, & References

The code has been extensively tested with various page sizes and different tests of the various functions. Our test cases are stored in a directory called benchmarks, before building them the library must be compiled first. The steps are listed below.

1. Change directory to project3
2. Run make
3. Change directory to benchmarks
4. Run make all (or specify a specific program to build)

The program uses the math library so any programs compiled with it must use the linker option -lm after linking the library. The program is also 32-bit so the compiler flag -m32 must be used as well.

The program supports any page size from 2^3 to 2^15 bytes. Any others will result in undefined behavior as the either the page table or outer page directory index length respectively would be zero for the given page size. The program also runs slower the smaller the page size due to overhead calls requiring more loops when walking of the *outer page directory* and page tables. For instance, “largemallocs.c” hangs for roughly a minute when allocating the second largest malloc call.

The program was created by Nicholas Dundas Logan Miller and the references consulted were <https://en.cppreference.com/w/c> and <https://man7.org/linux/man-pages> for C and related functions. Project 1 was also used for the bitwise functions found.