User-level Memory Management

Abhinay Reddy Vongur (av730)
Anmol Sharma (as3593)

Data Structures Used:

➔ Page Directory and Page Tables

- Page directory and page tables are each allocated one page in the physical memory. Base pointers to the respective pages are saved to the variables keeping a reference of page tables and page directory.
- Read and write operations are performed using the index, like in the case of an array.
- The number of entries is restricted based on the page and entry sizes.

➔ TLB

- A linked list is used to maintain the entries of TLB, with a pointer to both the head and tail of the list. A parameter length is used to track the number of entries in the TLB.

```
struct tlb {
    /*Assume your TLB is a direct mapped TLB with number of entries as TLB_ENTRIES
    * Think about the size of each TLB entry that performs virtual to physical
    * address translation.
    */
    struct TLB_Node* head;
        struct TLB_Node* tail;
    int length;
};
```

- Each node is TLB has two parameters, one to hold the virtual address and one to hold the corresponding physical address.

```
typedef struct TLB_Node{
        void *va;
    pte_t pa;
    struct TLB_Node* next;
} tlb_node;
```

➔ Bitmap

- Bitmaps are used to track the free physical pages and page table entries. A character array is used to implement the bitmap.
- Each byte has bits corresponding to 8 pages/page table entries with the left-most bit as the first bit and the right-most bit as the last one.

LRU for TLB

- ➔ All the new translations in a TLB are added to the head of the TLB by default.
- ➔ When a particular translation is accessed, it is moved to the head of the node as it is the most recently accessed node.
- ➔ Thus the node at the tail of the node will be the least recently used translation. When the TLB gets full the node at the tail is replaced with the new entry replicating the LRU algorithm for eviction.

Virtual Memory Implemented Functions:

➔ set_physical_mem():

- ◆ This function is responsible for initializing the physical memory and the required data structures.
- ◆ It is called when the t_malloc() function is called for the first time.
- ◆ Calculates the number of bits for each component in the virtual address, the number of entries in the page directory, and page tables based on the page size and virtual address size.
- ◆ Initialises bitmaps to keep track of virtual and physical pages. TLB to store the frequently accessed translations.

➔ add_TLB():

- ◆ This function will add the translation of a virtual address to a physical address in the structure defined above for TLB.
- ◆ The new entry will be added at the head of the TLB.
- ◆ If the number of entries in the TLB reaches the size limit then the least recently used (LRU) node is evicted from the list and the new translation is saved in its place.

➔ check_TLB():

- ◆ This function will check if the translation of a virtual address is present in the TLB.
- ◆ Each node in the list is traversed and its virtual address is compared to the input virtual address to see if it is the translation required.
- ◆ If the translation is found then the corresponding node is re-arranged as described in the LRU implementation above and the tlb_hit count is incremented and the corresponding physical address is returned.

- If no translation is found then the tlb_miss count is incremented by one and NULL is returned to indicate the absence of translation

➔ translate():

- This function is called when a virtual address has to be translated into the corresponding physical address.
- First, the check_tlb() function is called to see if a translation for the given virtual address exists. The translation process is performed only if there is no matching entry in the TLB.
- The page directory index and page table index of the given virtual address are extracted using bit manipulation.
- Then the page directory and page table entries are checked to see if the given address is valid.
- The pointer to the page table is obtained by getting the entry at the extracted index of the page directory.
- Then the value at the corresponding index of the page table is returned to the calling function. Before returning the translation is also added to the TLB.
- This function returns the base address of the physical page without adding the offset to it.

➔ page_map():

- This function is used to map a virtual to a physical address.
- Similar to the translate function, page directory and page table indices are extracted from the virtual address.
- Then it is verified if the page table at the corresponding page directory index is initialized, if not then a free physical page address is assigned. If no pages are free then an error is returned.
- Finally, it will save the physical address at the calculated page table index of the corresponding page table.

➔ get_next_avail():

- This function uses the virtual bitmap to get the next available free page.
- Num_pages argument is used to inform about the number of free pages needed.
- The virtual bitmap is traversed till the required number of consecutive pages are found.

- Once the free pages are identified, based on the starting page a 32-bit virtual address is formed using page directory index, page table index, and offset as 0. This address is then returned to the caller function.
- Once it finds the location in the virtual bitmap, it will form a 32-bit virtual address by calling a helper function corresponding to start page and page table entries.

➔ t_malloc():

- This function is used to request the allocation of memory.
- Based on the number of bytes requested, the number of pages needed is calculated.
- First, it is verified if the physical memory is set up. If not then the function set_physical_mem() is called to initialize the memory and the data structures.
- Then, it is checked to see if the required number of physical pages is available. If not no memory is allocated and an error is printed.
- Once the physical pages are checked, then the get_next_avail() function is used to check for free page table entries. If not found an error is printed and no memory is allocated.
- After identifying both the physical and virtual pages, the page_map() function is called to create a mapping from the virtual pages to the physical pages. Virtual and physical bitmaps are also updated to reflect the new allocation.

➔ t_free():

- This function will deallocate or free the pages given by the virtual address and the size.
- Based on the size given, the number of pages that need to be freed is calculated and each page table entry is checked to see if it's valid. This is done by extracting the page directory and page table indices from the virtual address.
- For each page that needs to be freed, it is deleted from TLB if present, page table entry is set to NULL, and bits in the virtual and physical bitmap are set to 0.

➔ **put_value():**

- This function copies a value to the location given by the virtual address.
- Based on the size of the data and the offset in the virtual address, the number of pages required to store the value is calculated. Then each of the corresponding page table entries is checked to see if they are all valid addresses. If not an error is printed.
- Then the value is copied into the physical memory byte by byte. get_next_page() function is used to get the base address for next virtual page.
- The translate function is used to get the physical address for each of the virtual pages.

➔ **get_value():**

- This function is implemented similar to the put_value() function defined above, the only difference being the source location of the value and the destination.
- Here, it will copy the value pointed by the virtual address byte by byte to the location obtained from the val pointer.

Support for different page sizes

➔ The function calculate_entries_bits() calculates the number of bits required for each component of the address and the number of entries per page directory and page table dynamically based on the page size.
➔ Thus page size macro can be changed as needed. It should not affect the library's behavior.

Results:

➔ Output from test.c

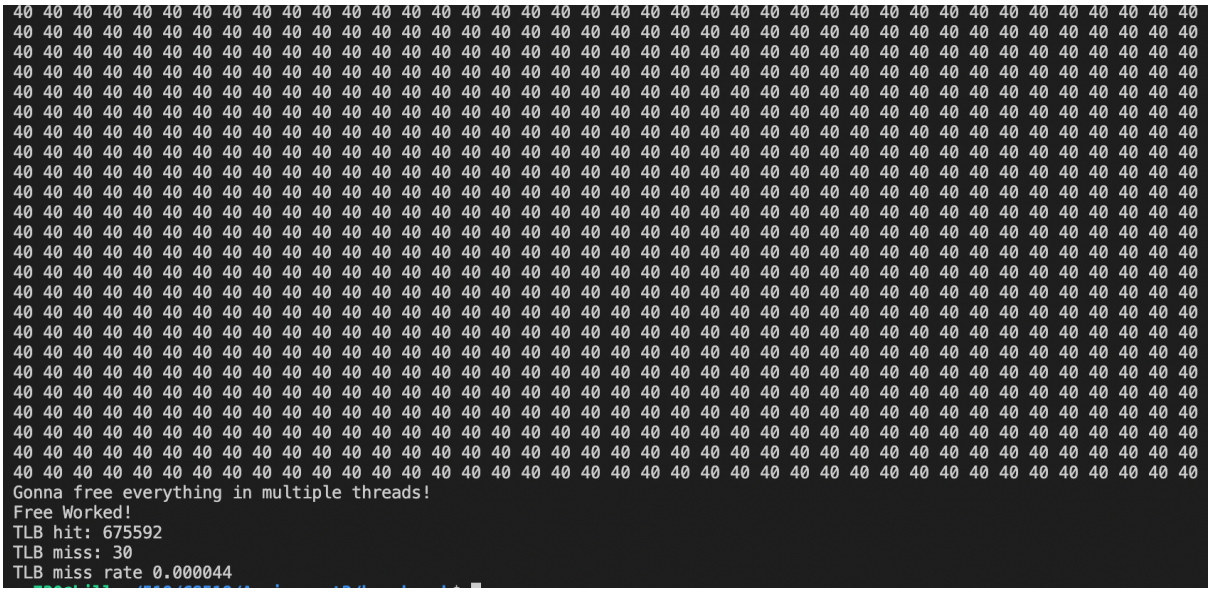
```
as3593@ilab3:~/Desktop/Project-2 OS/Project-3OS/Assignment3/benchmark$ ./test
Allocating three arrays of 400 bytes
Setting up memory
Addresses of the allocations: 0, 10000, 20000
Storing integers to generate a SIZExSIZE matrix
Fetching matrix elements stored in the arrays
1 1 1 1 1
1 1 1 1 1
1 1 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
as3593@ilab3:~/Desktop/Project-2 OS/Project-3OS/Assignment3/benchmark$
```

➔ Output from multi_test.c

```
as3593@ilab3:~/Desktop/Project-2 OS/Project-3OS/Assignment3/benchmark$ ./mtest
Setting up memory
Allocated Pointers:
0 10000 20000 30000 40000 50000 60000 70000 80000 90000 a0000 b0000 c0000 d0000 e0000
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!
as3593@ilab3:~/Desktop/Project-2 OS/Project-3OS/Assignment3/benchmark$
```

Support for Different Page Sizes and TLB Miss Rate:

➔ Miss rate with 4kb page size, 40x40 matrix (multi_test.c)



➔ Miss rate with 8kb page size, 40x40 matrix (multi_test.c)

Extra Credit

Internal Fragmentation

- ➔ Memory allocation is done in such a way that multiple applications can take up blocks of memory from the same page instead of letting the space be unused, thus avoiding internal fragmentation.
- ➔ When an application needs only a part of the total page size, the rest of the page is marked as available and allotted to the application whose memory requirements can be satisfied with the available space.
- ➔ This was implemented with the help of two data structures, one linked list containing the details of all the pages with some available space, called partial_pages_list. Another array of linked lists, where the list for each page consisted of details of all the allocations done on the page, is called allocation_lists.
- ➔ When a malloc call is received for a size less than the page size, partial_pages_list is checked to see if any of them can cover the requirement. If yes, then the application is given the virtual address with offset as the address and the page's space available is updated in the partial_pages_list. A node is also added to the corresponding page's list in allocation_lists with the start and end offset of the current allocation.
- ➔ When a page's space available goes to 0, it is removed from the partial_pages_list.
- ➔ In the t_free, put_value, and get_value functions, the address range is validated by checking if the address+size falls exclusively in any of the allocated ranges of the page saved in allocation_lists. If not, it is treated as an invalid address and no processing is done.
- ➔ When free is called on any such partial allocation, the page entries and bits in the bitmap are not cleared till all the allocations on the page are free. This is verified with the help of the free_status flag of each allocation node in the allocation_lists.
- ➔ **Addresses without fragmentation support**

```
Allocating three arrays of 400 bytes
Setting up memory
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
```

➔ **Addresses with fragmentation support**

```
Allocating three arrays of 400 bytes
Setting up memory
Addresses of the allocations: 0, 191, 321
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
```