Project 3 Report

Nikoloz Chichua

April 16, 2023

1 Introduction

This is an implementation of an n-level page table where n is determined by three parameters:

(1) the architecture of the system, either `32-bit` or `64-bit`,

(2) the size of the address space, defined as the `ADDRESS_SPACE` macro in *my_vm.h* and initialized to 32 by default,

(3) and the configured size of each page defined by the `PGSIZE` macro in *my_vm.h*.

The size of the page directory will always be less than or equal to the sizes of the rest of the page tables, depending on the configuration, but the sizes of the page tables, not including the page directory, will always be the same. For example, a `32-bit` system with a page size of 4096 bytes and `ADDRESS_SPACE` set to 32 bits will have 10-10 distribution with 2 levels, while a `64-bit` system with a page size of 8192 bytes and `ADDRESS_SPACE` set to 48 bits will have 8-9-9-9 distribution with 4 levels.

2 Implementation

2.1 Part 1: Implementation of Virtual Memory System

2.1.1 `void set_phyisical_mem()`

This function handles the initialization of the data structures and global variables that will be later used for subsequent methods for handling allocation, storage, retrieval, and deallocation of memory. The method is called only once, when `t_malloc` is called for the first time.

Notable data structures and variables initialized by this method include:

(1) `void *physical_memory` - maintains the simulated physical memory of the system and is allocated with `MEMSIZE` amount of bytes.

(2) `uint8_t *physical_bitmap` - records the status of each physical page.

(3) `uint8_t *virtual_bitmap` - records the status of each virtual page.

(4) `int max_levels` - maintains the number of levels the current configuration has as calculated through the required parameters.

The method also initializes other needed variables which are self explanatory in the code.

2.1.2 `int allocate_pages(uint8_t *bitmap, int start, int num_pages)`

This method takes in a bitmap, which simulates the allocation status of pages, a start index, and the number of pages it needs to mark as allocated, sets the bitmap in that range, and then returns the starting page number of the allocation. The global variables `vpn_pointer` and `ppn_pointer` can be used to determine the starting index, so that it is not necessary to traverse the whole bitmap on every allocation.

2.1.3 `Int map_page(Int vpn)`

This method iteratively goes through each level and adds an entry for each page table based on `Int`[1] `vpn` argument. It allocates 1 page per page table and another one for the page that needs to be mapped to with `allocate_pages()` method through the physical bitmap.

2.1.4 `void *t_malloc(unsigned int num_bytes)`

This function allocates the number of bytes provided and then returns the generated virtual address.

To do this, first `t_malloc()` makes sure, for thread safety, that a thread that enters it acquires the mutex lock. Then, if physical memory is not initialized, meaning it is being called for the first time, it calls `set_physical_mem()` and initializes the page directory and the Translation Lookaside Buffer using `init_directory()` and `init_tlb()`, respectively.

When everything is initialized, the method calculates the number of pages it will need based on `num_bytes` and sets the virtual bitmap using `allocate_pages()` method which then returns the Virtual Page Number (`VPN`) that needs to be mapped to a physical page. If there are no contiguous virtual pages available it returns `NULL` and prints an error message.

Then it maps each `VPN`, using `map_page()` function, starting from the returned `VPN` up until the range of all `VPN`'s that need to accommodate `num_bytes`, to each appropriate Physical Page Number (`PPN`). If there are no physical pages availalbe it returns `NULL` and prints an error message. Otherwise, before returning the virtual address determined by the `VPN` and the offset of 0, it releases the mutex and is done.

2.1.5 `void unmap_page(Int vpn)`

This method works similarly to `map_page()` but instead of mapping pages, it removes a mapping from the page tables.

2.1.6 `bool clean_tables(Int *page, int level)`

This function, starting from the page directory, recursively traverses the radix tree of page tables and checks if each of the page tables are empty. If any one of them is empty it will make sure to remove the entry mapping to it from the upper level, recursively checking each case, and then, if necessary deallocates the page table, marking it 0 in the physical bitmap. This method makes sure that all unneeded page tables are not allocated.

2.1.7 `void t_free(void *va, int num_bytes)`

This method frees the amount of pages required, based on `num_bytes`, starting from the index provided by the extraction of the `VPN` from the `va` virtual address.

The method first makes sure that the pages allocated are indeed contiguous through the range provided by `num_bytes`. If this is not the case it fails and outputs a failure message. Otherwise, it continues and calls `unmap_page()` for each virtual page through the range determined by the `VPN` and `num_bytes` parameters. Afterwards it calls the `clean_tables()` method and returns.

2.1.8 `Int translate_page(Int vpn)`

Translates a virtual page to a physical page by traversing the page tables, returning the physical page number.

2.1.9 `int put_value(void *va, void *val, int num_bytes)`

This method first makes sure that all the required pages determined by `num_bytes` and the `VPN` from `va` are contiguously allocated and then it goes through each physical page mapped from those virtual pages and stores the values

[1]Since, for each entry, we are using 4 bytes in a `32-bit` system and 8 bytes in a `64-bit` system, the `Int` struct is defined as either a 4 byte number or an 8 byte number, based on the system. This is done because, when iterating through a page table we need to go through a variable amount of bytes (4 or 8 bytes depending on the system) at a time to view each entry. This way, when trying to retrieve a pointer to a page table, it is not necessary to cast it to the appropriate size to iterate through them.

inside each page at the given offset from `va` from the virtual page translation using `translate_page()`. If pages are not contiguous it returns -1 and prints an error message.

2.1.10 `void get_value(void *va, void *val, int num_bytes)`

Behaves the same way as `put_value` but instead of storing, it retrieves the values from `va` into `val`. If pages are not contiguous it returns and prints an error message.

2.2 Implementation of the TLB

For the Translation Lookaside Buffer, the following methods were implemented:

2.2.1 `void init_tlb()`

This method initializes the TLB and is called only once, on the first call of `t_malloc()`.

2.2.2 `void add_TLB(int vpn, int ppn) )`

This method adds a new translation to the TLB from the virtual page `vpn` to the physical page `ppn` and is called every time a new translation is added in `map_page()` function.

2.2.3 `void remove_TLB(int vpn)`

This method removes a translation given from `vpn` from the TLB and is called every time a translation is removed via the `unmap_page()` function.

2.2.4 `int check_TLB(int vpn)`

This method is only used in the `translate_page` function and checks whether a translation exists and returns the translation if that is the case. Otherwise it returns -1 and notifies `translate_page()` to look for the translation.

3 Report

3.1 Detailed logic of how you implemented each virtual memory function.

This is described in section 2.

3.2 Benchmark output for Part 1 and the observed TLB miss rate in Part 2.

These test were done on a `32-bit` system with `ADDRESS_SPACE` macro set to 32 and `PGSIZE` set to 4096.

3.2.1 Benchmark output for Part 1

(1) Benchmark output from `test.c` with `SIZE` macro set to 6 and `ARRAY_SIZE` macro set to 1000:

```
Allocating three arrays of 1000 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
Performing matrix multiplication with itself!
```

```
6 6 6 6 6 6
6 6 6 6 6 6
6 6 6 6 6 6
6 6 6 6 6 6
6 6 6 6 6 6
6 6 6 6 6 6
Freeing the allocations!
Checking if allocations were freed!
free function works
```

(2) Benchmark output from `mtest.c` with 15 threads, `matrix_size` set to 5, and `alloc_size` set to 1000:

```
Initiating!
Allocated Pointers:
1000 2000 3000 4000 5000 6000 7000 8000 9000 a000 b000 c000 d000 e000 f000
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!
```

3.2.2 Observed TLB miss rate in Part 2

(1) TLB miss rate on `test.c` with `SIZE` macro set to 200, `ARRAY_SIZE` set to 1000000, and TLB size of 512:

```
Number of total translations 16240000.000000
Number of TLB misses 241792.000000
TLB miss rate 0.014889
```

(2) TLB miss rate on `mtest.c` with `matrix_size` set to 100 and TLB size of 128:

```
Number of total translations 10220000.000000
Number of TLB misses 6254208.000000
TLB miss rate 0.611958
```

3.3 Support for different page sizes (in multiples of 4K)

Supports all logical page sizes that are a power of 2.

3.4 Possible issues in your code (if any).

I was not able to discover any.

3.5 If you implement the extra-credit part, description of the 4-level page table design and support for different page sizes.

Instead of using `MAX_MEMSIZE` for determining the address space, I added a new macro `ADDRESS_SPACE` to determine how many bits the page table should support. Since this paging system was designed as a dynamic multi-level page table, meaning that based on the parameters and how `ADDRESS_SPACE` macro is changed and whether `32-bit` or `64-bit` system is used, it will dynamically adjust to the appropriate amount of levels.

(1) Meaning that, if one wants to test a 2 level page table then the parameters that need to be used are a `32-bit` system, with `ADDRESS_SPACE` set to `32`, and page sizes of `4K`, `8K`, `16K`, or `64K`, or any other appropriate parameters.

(2) Or to test 4 levels: `64-bit` system, with `ADDRESS_SPACE` set to 48, and `PGSIZE` macro set to `4K`, `8K`, or `16K`, or any other approriate parameters.

(3) One can even test 5 levels: `64-bit` system, with `ADDRESS_SPACE` set to 64, and `PGSIZE` macro set to `16K` giving a 10-10-10-10-10-16 distribution.

This was designed by making sure that mappings were done dynamically, based on the number of levels that were calculated beforehand. Instead of statically going through a fixed amount of page tables and mapping them, it instead is based on that `max_levels` variable and the mapping is adjusted appropriately. Of course, the other functions had to be implemented accordingly, like the `translate_map()` and `unmap_page()` functions, and cleaning of the tables had to be done recursively, going through each of the page tables and their children tables, etc.