USER-LEVEL MEMORY MANAGEMENT

Project 3

Authors: Abhinav Bharadwaj Sarathy (ab2812), Krishna Sathyamurthy (ks2025)

Project 3

Contents

1 Introduction

For this project, we were tasked with implementing our own memory allocation software. We have been given the following details,

```
// Max size of the virtual memory (4GB)
#define MAX_MEMSIZE (1ULL << 32)

// Physical memory size to simulate (1GB)
#define MEMSIZE (1UL << 30)

// Page size considered - (8KB)
#define PAGE_SIZE (1UL << 13) // 2^13 Bytes
```

where, MAX_MEMSIZE, MEMSIZE and PAGESIZE are the size of virtual, physical memory and pages each. The physical and virtual memory will be split into multiple pages depending on the page size, leading to further compartmentalization.

Before diving into how our code works, let us give you a brief overview of what we will be doing here. We are using a virtual memory, something that is unique to each program, to retrieve "our" data from the physical memory. This can be done in a lot of ways, we use paging for this project. To be exact, we use 2 level paging in order to achieve this. In this case, we have 4GB of virtual memory, 1GB physical (/real) memory. This has been further divided into pages of size 8kb, i.e., 8192 bytes. Therefore, we will require $2^{(30-13)} = 2^17$ page entries to store our data. Now, this 2^13 will be our offset, and to link the virtual memory to physical memory, we will require $2^{(32-13)} = 2^19$ entries. This will be further split into two levels, which upon calculation gives us 2^11 for the outer level and 2^8 for the inner level. Finally, using all of these data, we will now start to allocate, translate and process data.

2 Implementation of Virtual Memory System

To begin with, we had introduced a lot of data structures. They are as follows,

```
typedef struct page;
typedef struct bitmap;
typedef struct vm_manager;
typedef struct virtual_page_data;
typedef struct tlb_data;
```

- **page** This is used for storing the physical memory data into multiple pages, where each page has an array of the size PAGE_SIZE/sizeof(unsigned int) for 32bit or PAGE_SIZE/sizeof(unsigned long) for 64bit.
- **bitmap** This is used for keeping track of the bits that are available or used. These bits will be used for identifying which of the pages are free or allocated.
- **vm_manager** This is the main structure of our virtual memory manager. This contains a reference for all the variables and data types that will be required for modifying the virtual to physical memory.
- **virtual_page_data** This is used for storing the current virtual address details, i.e. the one being accessed. This helps us modularize the code and access the details while understanding what's going on at each point.
- **tlb_data** This is used for storing the vpn (virtual page number) and pfn (physical frame number) details in the TLB lookup table.
- **tlb_lookup** This is a structure containing an array of tlb_data that will be used for lookup. Additionally, this is also used for keeping track of the total lookup, hits and misses made by the TLB.

2.1 set_physical_mem

This is usually only invoked one time, and that is performed when the t_malloc is called for the very first time. We are assigning allocating the ”fake physical memory”, storing the page directory in this ”physical memory”, initializing TLB, and finally initializing the 2 level indexes.

2.2 translate

This function takes a virtual address as a value, and uses the vpn to search for a corresponding pfn, and appends an offset to this pfn in order to get the required physical address. This is done in multiple steps,

- Before beginning to ”search” for the physical address, we lookup the TLB to find if we can get the physical address directly from our cache. In case that doesn't work, we register a miss and continue processing.
- We start out by splitting the virtual address into vpn and offset bits.
- The vpn is further split into outer (11 bits) and inner level bits, where the outer level bit points to the address in the physical memory containing the inner level bits address.
- Upon finding out outer level bit, we access the data in memory, and go to the starting posiiton of our inner level page directory.
- The inner level bits (8 bits) is the index that will be used for offsetting to the position where we have stored the pfn.
- To ensure tlb doesn't result in a miss again, we register this data in our tlb, to lookup the data again when it is required.
- Later, we offset this pfn based on the offset we had obtained from the virtual address (trailing 13 bits), to obtain the physical address position.
- Finally we return this physical address as a pointer.

2.3 page_map

This method takes the physical and virtual address as inputs, validates it and creates a mapping between them in the 2-level table.

- With the virtual page address in hand, and we segregate the bits into outer, inner and offset bits.
- We check for existing mapping in the page directory for the outer bits. If there's already a mapping. there's an inner table already present. If not we create inner table for the new mapping in a different page within the physical memory.
- Once that is successfully done, we navigate to the inner level and set the value for the inner level page entry to be the physical address.
- And then, we make a TLB entry for the new mapping.

2.4 t_malloc

The memory allocation method takes the size n, as an argument. Based on this we allocate virtual and physical pages in the respective memory and map them using the above mentioned page_map method.

- With the size, we evaluate the number of pages which is necessary to allocate
- This will constitute for both, the physical and the virtual address space. However the difference lies in allocating those pages.
- The virtual address space is allocated contiguously, by pages while the physical memory allocation happens in non-contiguous way.
- Accordingly, the bitmap for those allocated pages are set for both the space.
- For each address pair, we iterate and map the addresses in the page table by calling the page_map method.
- Finally we will return the starting virtual address.

2.5 t_free

The memory de-allocation/free method takes the virtual address and size n, as an argument. With this we de-allocate the virtual and physical pages in the respective address space.

- Similar to t_malloc, we calculate the pages to de-allocate.
- Before doing that, we do a validation if the pages are allocated in first place, if not we return error code -1.
- For each address pair, we iterate the virtual pages and find their respective physical addresses using translate function.
- And then, the bitmap for those allocated pages are reset to 0.
- Finally we also remove any TLB mappings those address pair have stored.

2.6 put_value

Put value accepts, a virtual address vp, a pointer variable and size n. It updates the value in the physical address page(s) bytewise based on the value in the variable.

- With the virtual address, we find the physical address by calling the translate method.
- To update values bytewise, we type-cast all the addresses and variables to char *, so on iterating we navigate through each byte $(sizeof(char) = 1byte)$
- Once validating the pages to update, we begin iterating through each byte until we reach the size n.
- In each iteration, we update the value in the physical address to the value from the variable. Once an offset is zero, we reset the physical address based on the current virtual address.
- In case of any failure, we return -1, else 0.

2.7 get_value

Get value accepts, a virtual address vp, a pointer variable and size n. It fetches the value in the physical address page(s) bytewise and updates the variable.

- The steps are quite similar to what we did in put_value, except here on each iteration, we fetch the value from current physical address and update the variable bytewise.
- In case of any failure, we return -1, else 0.

2.8 mat_mul

mat_mul takes in the respective virtual address of matrices a, b and c. The size of matrices a, b and c, l, m and n, where l*m is the size of a, m*n is the size of b and l*n therefore will be the size of c.

- We iterate through the rows and columns of the matrix a and b, for each element, we calculate the address in the virtual space.
- We then get the value on matrix a and b by using get_value()
- For each of these values, we multiply and add them to the resultant entry in the matrix c.

3 Support for multiple page sizes

3.1 How is this handled?

This has been handled by using a **#define PAGE_SIZE**, that takes the size of the page as an argument. So, while compiling if you want the size to be modified, it is suggested to change it from "13" to any other value of your preference.

Even if the page size were to vary, since the outer and inner level indices are calculated based on the current size of the PAGE_SIZE, they are always allocated dynamically as follows,

```
int outer_level_bits = log2(PAGE_MEM_SIZE);
int inner_level_bits = (virtual_page_bits - outer_level_bits);
```

where outer_level_bits is the size taken by each page in our unsigned int array (which should increase further if we use int_64 for instance), and inner_level_bits is calculated based on the total leftover bits.

4 Possible issues in your code (if any).

We believe, some issues come up when we try to allocate a very large memory space. Although we were not able to reproduce this issue often, we think there could be some issues when you allocate rather large chunks of memory. We also think this might be some issue in our testing code rather than on the project code.

Another noticeable thing was about the TLB missrate. Upon discussing with our peers, we came to know the TLB missrate was surprisingly higher than what we observed in general. We tested this out by testing mat_mul and t_malloc/t_free on several large range scenarios, however we were able to see missrates in range of 0.0025. We didn't know the expectation that was there for the TLB missrates.

5 4-Level Page Table

5.1 How was this implemented?

We decided was to draw some inspiration on how x86_64 has implemented the page tables in their architecture. They consider the first 48 bits to be the most meaningful bits, and the rest of them would be unused most of the times. They have a 4-level table, each of size 9-bits, and a offset of 12 bits, thereby a 4KB page size. Due to challenges in handling custom cases, we weren't really able to handle for cases where the page size is dynamic, i.e., based on the given page size, each level will be assigned a specific value.

We were able to implement the 4-level table by extending the code from our 2-level table, and since our code was already modularized, it was easy to extend and support more levels just by changing the levels to arrays and including a few while loops to check if the page directory was set or not.

References

[Bla] David Black-Schaffer. *Virtual Memory*. URL: https://www.youtube.com/playlist?list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX.

[Col] Casey Cole. *Operating Systems*. URL: https://www.youtube.com/playlist?list=PLWCT05ePsnGww5psXWHRLG7p30eKKt1Pd.

[Deva] OS Dev. *Memory Allocation*. URL: https://wiki.osdev.org/Memory_Allocation.

[Devb] OS Dev. *Paging*. URL: https://wiki.osdev.org/Paging.

[Devc] OS Dev. *TLB*. URL: https://wiki.osdev.org/TLB.

[Devd] OS Dev. *Writing a memory manager*. URL: https://wiki.osdev.org/Writing_a_memory_manager.

[Lun] Benjamin David Lunt. *The FYSOS Operating System*. URL: https://github.com/fysnet/FYSOS/blob/master/bucket/Section20.pdf.