CS416 Project3: User-level Memory Management

Alfonso Buono (ajb393)
Ali Mohamad (aam345)

Due: April 11th, 2020

Contents

0.1 Introduction

The goal of the project was to create our own implementation of a two layer page table. This was to show all of the caveats and parts to memory virtualization and the obstacles faced by creating an actual memory virtualization. With this, we needed to create the functions associated with memory virtualization in order to allow a user to utilize the memory. Furthermore, we were also told to implement a direct TLB in order to see how it functions and how it can increase efficiency. The complete project also had to be thread safe. The reason for this is that the actual malloc() and free() functions (as well as the memory virtualization) are thread safe as well as the functions are going to be tested by using multi threading. In order to make sure that it was thread safe we had to make sure that any global data structures or information that could be shared between the threads, such as the page directory and page table, were locked and as such no race conditions existed. Overall the project was very informative and made us apply the information that we learned in class. We got to see all of the issues that are typically faced when virtualizing memory and got to see how it is (almost) implemented currently (we are not exactly the same but are close).

Part 1

1.1 Virtual Memory Overview

First starting with the Virtual Memory aspect of the project, we were tasked with implementing a two-layer page table. Here we had to set aside 1 GB of contiguous memory that acted as our Physical Memory. These would be the pages that would be mapped to our virtual address in our tables. We then had to create a variety of different functions listed below in order to create and operate the two-level page table. First, we had a single page directory. In this page directory, the entries contain pointers to their corresponding page tables. We do this in order to prevent wasted memory as we only want to allocate memory when we are using it. Thus, we only want page tables that are in use to be allocated. In each of the page tables each entry corresponds to a page. This implementation allows us to perform the necessary operations. When our a_malloc() is called, we want to allocate however many pages worth of virtual memory, it has to be continuous, and the same amount of memory in the physical address, it does not have to be continuous. In order to do this, we want to map our virtual addresses to our physical address. So, we put our physical address of each of the pages into their corresponding virtual page and then return the FIRST virtual page to the user. In doing this we have to know which pages are in use in order to properly allocate pages. Thus, we have two bit arrays where each bit corresponds to each possible page (both physical and virtual).

1.2 Implementation of VM Functions

1.2.1 set_physical_mem()

The main goal of this function is to allocate the physical memory. The size of the physical memory allocated is dictated by MEMSIZE, a definition given to us. We have to make sure this is only called once (the first time that a_malloc() is called) as to not reinitialize and ruin any information we might already have. Furthermore, in this function we set global variables such as how many bits to use for the page directory index, for the page table index, and for the offset, creating and setting the bit arrays to 0, and creating the page directory. We also want to calculate how many page directory entries there are and how many page table entries there are. This is important and can be utilized in later functions. The TLB is also created here and initialized in order to ready it for use. Here we also mark the first index, index 0, in the virtual bit array to in use as we never want to return the address 0x00000 to the user as a value of 0 equates to NULL. It also helps us error check in a_malloc() and will be discussed more below. This function also This function is important as it sets up our library and gets everything ready for the subsequent library calls.

1.2.2 get_next_avail()

This function's role is to traverse through our page tables and find the next available set of contiguous pages. We need to find however many pages (Found by doing: $ceil(\frac{size}{PGSIZE})$) of contiguous virtual memory. In order to do this we traverse through our virtual memory bit array and find the first set that matches this description. If there is not enough contiguous virtual pages then we return an error value of 0. The virtual address is then calculated by using the VPN (The VPN maps to what bit in the bit array) and then it is returned to the calling function. To calculate the virtual address we do:

$$pageDirectoryIndex = \frac{VPN}{NumberofentriesinaPageTable} \tag{1.1}$$

$$pageTableIndex = VPN \bmod NumberofentriesinaPageTable \tag{1.2}$$

We then use these corresponding numbers and bit shift them over. For example, if the PGSIZE is 4096 then we bit shift over the pageDirectoryIndex over by 22 as it is the top 10 bits. The pageTableIndex is the middle 10 and has to be shifted over as such while the offset is the first 12 bits. We then are able to combine this all into one 32 bit number and this is now our address. We then return this to the calling function.

1.2.3 translate()

The goal of the translate function is to take in a virtual address and calculate the corresponding physical address. First, we want to use check_TLB() to check our TLB to see if the virtual page to physical page mapping exists. If it does exist then we get this physical address and add an offset (if there is one in the virtual address passed into translate) and then this is returned. If it is in the TLB then we perform the reverse of the operations done in get_next_avail(). To do this we use our private functions in order to get our pageDirectoryIndex and our pageTableIndex. Once we have these values we are able to get into the corresponding page directory entry and get our page table. From there we are able to go into the correct page table entry and find the page we need. This page then contains the physical address that we will use. We add this physical address to the TLB (paired with the virtual address given) and then we add the offset from the virtual address to the physical address and return this.

1.2.4 page_map()

Page map is given a virtual address and a physical address. The goal of this function is to break apart the virtual address in order to find where in the page directory and page table we need to store the physical address. Once we learn of this location (by performing a similar operation to the ones above where we break the VA into the VPN and then find the page directory and page table indices) we then save the value of the physical address. This "links" or "maps" the virtual page to the physical page and allows us to get the physical page whenever given the virtual. If for some reason page_map fails, such as the provided VPN/VA is not not inuse in the Virtual bit array, then I return -1. On success 0 is returned.

1.2.5 a_malloc()

This function is one of the major functions. Here we utilize a lot of the aforementioned functions in order to create a virtual address, link the page(s) corresponding to that va to different physical pages, adding the virtual address(es) and the physical address(es) to the global TLB and then returning the virtual address to the user. For the first part we want to caluclate the number of pages needed. This is done by $ceil(\frac{size}{PGSIZE})$) (we do not have a ceil function but rather divide the provided size by PGSIZE. We then add +1 to the number of pages if there is more memory left (we check by using %)). Once we do this we use get_next_avail() in order to find however many number of contiguous virtual pages. We then check to see if we found virtual pages by checking to see if the va returned is nonzero. This is because the virtual address returned should never be 0 (as we marked the 0th virtual page to in use). If it is 0 then we output saying that there were not enough virtual pages and then return 0 to the user (an error). If it is nonzero then we can continue with our function. The next function that we call is a private function: get_next_avail_phys(). This function is essentially the same as get_next_avail() except for two things. First, it is attempting to find physical pages by iterating through the physical memory bit array. Second, is that it returns an array of physical addresses. This is because the physical pages need not to be contiguous (they can be but are not required to be). Since this is the case we cannot figure out which pages the function marked as in use and subsequently use them. Thus, an array of all of the physical addresses is needed. After this function returns we error check it by seeing if the first index is set to 0 (if in get_next_avail_phys() we do not find enough physical pages). If we do not find enough physical pages inside get_next_avail_phys() then inside the function we must revert all of the physical pages that we marked as in use back to not in use. Furthermore, if this is the case then we need to go back and mark all of the virtual pages that we were going to use back to not in use. We do this by invoking our resetVirtBits()

function which will go to the first virtual page in the virtual bit array and mark it as not in use. We then print an error saying we could not find enough physical pages. However, if this function succeeds then we move onto linking the virtual pages and the physical pages. We do this by iterating through however many number pages that were requested and passing the virtual address (we start with the first one and then for each subsequent loop we add another PGSIZE to get the correct address) and physical address (we get the correct one by iterating through our array of physical addresses). The function of page_map() is described above. Depending on the return value of page_map() (either 0 or -1) we add the virtual address and physical address we just mapped into the TLB by calling the add_TLB() function (discussed below in the TLB Section). If page_map() fails then we print an error and try linking the next virtual address to the next physical address. Once all of this is done we return the first virtual address to the user.

1.2.6 a_free()

This function is another major function in the library. Here we first calculate the number of pages needed (Using the ceil() described previously) in order to know how many pages we are going to be freeing. We then calculate the VPN of the first Virtual address. Using this VPN we check to see if all of VPNs (from the first VPN to the last one, which is found by adding the number of pages being freed to the VPN) were allocated. This is a check to make sure that the user is freeing data that we gave to them. If any of those pages are not in use, we check this by looking at the virtual bit array, then we print an error and return. Next we want to loop through each of the VPNs and their corresponding physical pages and mark them as not in use anymore. This is done by using the calculated VPNs and calling our translate() function. If the physical address returned by translate is 0 then we print an error and return. However, if the physical address is not 0 we then calculate its corresponding PPN by doing:

$$PPN = \frac{pa - physicalMem}{PGSIZE} \tag{1.3}$$

where pa is the physical address and physicalMem is the start of our physical memory. We then remove the virtual address and its physical address from the TLB by calling our private function removeTLB() (this function works very similarly to checkTLB/addTLB and zeroes out the VPN and pa stored in the TLB entry). We then set the bits in both the physical and virtual bit arrays corresponding to the PPN and VPN to 0. Once all of the pages have gone through successfully we then return.

1.2.7 put_value()

This function is also important as it gives the user the ability to save/put data into the memory that we allocated to them. They cannot just put it into the virtual address that we gave them as the va we gave them would map to memory not allocated by the program (can be memory saved for the OS or other data for the system). Thus, we have to take the virtual address we gave them, find the physical page associated with it, and then store the data there. To start, we first calculate the number of pages needed (this is done just like previously). We then want to figure out the offset inside the virtual address passed to the function. This offset is important for later when we are looping through the physical page(s) to store the data. We subtract the offset from the virtual address so that we just have the address to the START of the first virtual page. We do this because our TLB stores only the start of both the virtual pages and physical pages in each entry. We then want to loop the number of pages that the user is trying to write. We call translate() as:

$$translate(NULL, (void*)(virtAddr + offset + (counter * PGSIZE)))$$

We separate the offset from the virtual address because if the user is askng to put more than 1 page worth of data into the physical memory and they give an offset then the offset only applies to the first physical page. We also include counter* PGSIZE as if there is more than one page that needs to be written to, we need to increment our VA by a size of a page in order to get the next physical address. Once we get the physical address from translate() we error check in order to make sure that there exists a virtual to physical page mapping. If there does not we print an error and return (stopping the put call). Next we are going to be actually copying the data the user gave into the physical memory. There exists two cases:

1. The current amount of memory that needs to be written is less than or equal to the current amount of memory left in the physical page. If that is the case, then we call:

   $$memcpy((void*)pa, (void*)(val + (size - tempSize)), tempSize)$$

   We add (size - tempSize) to the address of the value that the are passing us as we want to make sure we are copying off from what we have copied so far. Size is the amount of memory that they have passed to us while tempSize is the amount of remaining so far. So, if we subtract the two we have the amount of memory that we have copied so far. We then want to copy tempSize amount of data left as we are filling in the rest of the data into the page (we would not go outside of the current physical page as we checked to see if this amount of memory is less than or equal to the amount of memory left).

2. The next case is if the amount memory that wants to be written is greater than the memory left in the page. If this is the case then we call:

   $$memcpy((void*)pa, (void*)(val + (size - tempSize)), PGSIZE - offset)$$

   The first two parameters are the same as the previous call. However, the last parameter is different. Instead of passing just tempSize (the remaining amount of memory that we need to copy) we pass PGSIZE - offset. This is because the amount that we memcpy() is based off of two things. It is based on where we are in the physical page (denoted by offset) and the current PGSIZE. If we are at the start of a physical page then offset will be 0 and we will just copy a full PGSIZE worth of memory. If we start say halfway through the physical page then we only want to memcpy() the remaining half. We then subtract how much we copied from the remaining size (tempSize) so we can know how much left we need to call for the remaining iterations.

Once either of these have been called we set the offset to 0. This is because the offset only applied to the first physical page and from now on out (if there is more memory to copy) we go from the start of the next physical page. This is iterated however many pages of memory that they called put_value() on and once we have memcpy()'d all of the memory we return.

1.2.8 get_value()

get_value() is another important function as it allows the user to copy/get the data that they have saved in their allocated physical pages. This is done exactly like put_value() except the memcpy call is different. It follows the same two cases and the memcpy() calls are as follows:

1. The current amount of memory that needs to be written is less than or equal to the current amount of memory left in the physical page.

$$memcpy((void*)(val + (size - tempSize)), (void*)pa, tempSize)$$

2. The next case is if the amount memory that wants to be written is greater than the memory left in the page.

$$memcpy((void*)(val + (size - tempSize)), (void*)pa, PGSIZE - offset)$$

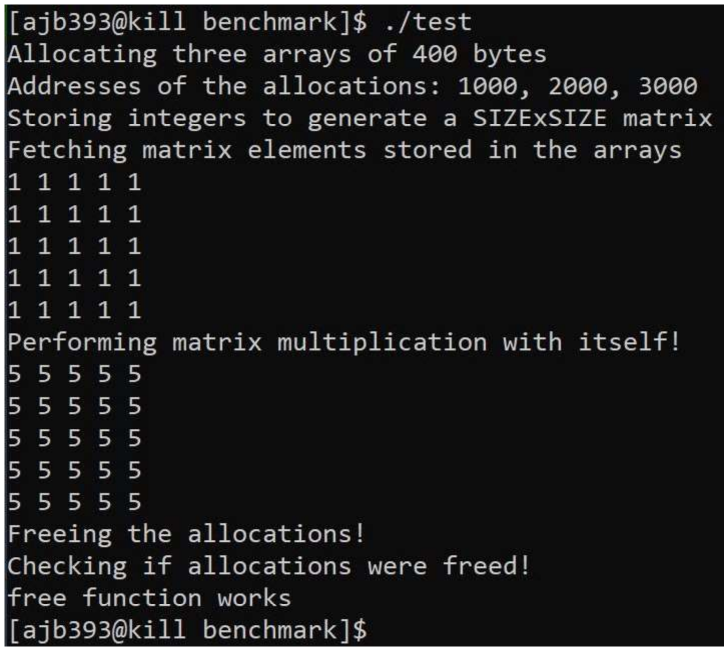
This is because we are now copying the memory from the physical pages to the variable val that they passed to us.

1.2.9 mat_mult()

Matrix multiply is a function that given two matrices and a resulting matrix it calculates the product of the two matrices and stores it in to the answer matrix. This is done by looping through the size of the matrices and calculating the current addresses. We do this because the matrices have the rows stored in them linearly (as in row 0 is the first n*sizeof(int) bytes in the physical page. Row 1 is the next n*sizeof(int) bytes immediately following row 1, etc). Once we calculate these addresses we do basic matrix multiplication (the elements of the row of the first matrix multiplied by the elements of the col of the second matrix and add them together). Here we then store the resulting sum into the correct address in the resulting matrix (calculated similarly to how we found the data in the first two matrices). Once this loops through all of the data it returns and the user should now have the resulting matrix stored in the answer matrix passed into the function.

1.3 Benchmark Output

When compiling and running the benchmark the following output is given:



```
[ajb393@kill benchmark]$ ./test
Allocating three arrays of 400 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
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
[ajb393@kill benchmark]$
```

Here we see that it allocates the 3 arrays of 400 bytes each. It then prints the corresponding address of those pages in hex. We see that the address are 1000, 2000, and 3000. This makes sense as we mark the zeroth page as in use (the reason being we do not want to give an address of 0 to the user as this is how we denote an error for the user). Thus, the 3 pages that were allocated are page 1, 2, and 3. Next we see that the program stores the values into the first two matrices and then prints one of them as a test. Following this the mat_mult() function is called and then the resulting matrix is printed. This matches the result when multiplying two matrices of 5x5 with all 1's in each entry. It then frees the matrices and sees that the free correctly works.
We have also created our own test cases which can be found in the test.c file.

1.4 Different Page Sizes

In dealing with different Page Sizes we tested our code (using the test functions in test.c) with different Page Sizes. It seemed that our library would adapt to the changes as we set up the page directly and tables by using the PGSIZE definiton in the header file. When handling this we have to decide (if there are an odd number of bits for the VPN) if the page directory or the page table would have more bits. We decided on giving the extra bit to the page table. So, in testing our functions with differing page sizes we noticed that we do not allocate new pages unless we cross the "boundary" of memory given by the new PGSIZE.

1.5 Possible Issues

When working on the project there were some issues that we believed we resolved. First, when using put_value() and get_value() we did not account for when the user would pass a virtual address that was not at the start of that virtual page. This would cause the data to start at that written portion in the physical page but then could overwrite data in the direct next physical page. This is wrong if the physical pages are not contiguous. We changed this by modifying our put and get functions by including how much memory is left in the page.
From here there are two parts that could be improved on for our system. First, we could reduce the critical sections in order to make the library faster. We were having trouble with reducing the areas in order to make it faster and were thus running into issues of deadlocks and other issues that can occur because of multi-threading. Second, we did not attempt the reducing fragmentation improvement. This improvement would not just allocate new virtual and physical pages but instead would fill in the pages with data depending on how much data was needed. These extra steps would change some of our library as we would have to really focus on the offset and making sure that data is not overwriting previously written data.

Part 2

2.1 TLB Overview

The second part of the project consisted of us having to create a direct TLB. A TLB is a cache-like object that is used to make use of temporal locality and space locality. The steps in translating a virtual address to that of a physical address can be long and slow down the program. So, to combat this a TLB is made. Here the virtual addresses and their physical addresses are stored, but only a limited amount. A simple calculation is done in order to find it in the TLB:

$$TLBIndex = VPN \bmod TLB_ENTRIES$$

This allows us to quickly store or find a VPN in the TLB. Our approach to creating a TLB was usin the tlb_store struct that was given to us. In it we placed our TLB hits, TLB misses, and an array of a new struct tlbEntry. A tlbEntry contains two integers, the VPN and a physical address. The reason for storing the VPN is because when we hash the VPN by the TLB_ENTRIES in order to find the index, different VPNs can hash to the same index. Thus, we want to compare the current VPN to the one in the entry. This will be discussed more below.

2.2 TLB Functions

2.2.1 add_TLB()

This function does what the name implies. It is given a virtual address and a physical address and it adds it to the TLB. To do this it calculates the VPN from the given virtual address and then hashes that number to get an index in the TLB. At this index it then replaces whatever VPN and physical address values in that entry and then returns 0 success.

2.2.2 check_TLB()

This function also does what its name implies. In order to speed up translations check_TLB() exists in order to check if the translation exists and if it does return the physical address. To do this check_TLB() receives a virtual address which is then turned into a VPN. From there the VPN is hashed into the TLB. At that TLB entry we check to see if the VPNs match. If they do, then the physical address stored in that entry is returned. If it is not, then null is returned. Furthermore, we want to keep track of all of the hits and misses that our TLB does. So, in this function if the VPN does not match the entry in the TLB then our misses counter in the tlb_store struct is incremented by one. However, if there is a match then our hits counter is incremented. This is used in the next function to calculate the miss rate.

2.2.3 print_TLB_missrate()

TLB miss rate is calculated by dividing the total number of misses by the sum of the misses and the hits. So, our function does just that. We store the TLB miss rate and print out the amount of hits, misses, and the miss rate.

2.3 TLB Miss Rate

Our TLB miss rates that we calculated are typically quite small. At first it was larger, but once we decided to add the virtual address to physical address translation to the TLB after the memory is a_malloc'd, then the miss rate dropped significantly. This is because typically the memory that is just allocated will usually be used soon after. The reason for such a low miss rate is that most of the testing functions would not create enough unique pages that would fill in each of the TLB entries. Because of this, once the TLB entry is filled with the translation after malloc then there are no misses until it is evicted. However, the way we evicted in our direct mapped TLB was by overwriting whatever currently existed in that index. Thus, each of the translations stayed inside the TLB and if we had a miss rate it was very low. However, we did test the miss rate by changing the size of our TLB to very very small. Here we saw that the TLB miss rate shot up because each time malloc'd there were more and more VPNs that would index to the same entry in the TLB. Thus, more and more collisions and evictions. This then caused our program to miss a lot more and have to perform the normal translate() function and then add the translation to the TLB (which resulted in evicting the current VPN in the entry).

2.4 Possible Issues

For the most part I do not think there are any potential issues that could arise from the TLB. At first I thought there was an issue in regards to returning NULL on failure to find a translation in check_TLB(). I thought this because

I wondered what if the physical page being returned was the zeroth physical page. Would its address not be 0 and then it would return NULL? But then I recalled that the zeroth page in physical memory does not correspond to physical address 0. It corresponds to the address of the physicalMemory that we created in set_physical_mem().