**CS 518 Project 3**

Members : Abhilash Kolluri(ak2048), Srikar Gona(gs943)

iLab Machine : kill.cs.rutgers.edu

PART 1

Firstly, we will describe the important data structures which we used for this project.

* The most important abstraction we have created is PAGE. It is a structure that has an array of unsigned longs. The array size is determined by simply dividing the PAGE\_SIZE with size of PTE(Page Table Entry) in this case it is unsigned long. All the access to physical memory is made via this abstraction.
* As suggested, we have used 2 bitmaps to know what are valid/free pages are present in our Memory Management Unit (MMU). p\_bit\_map and v\_bit\_map. Since 1 byte = 8 bits,

The memory allocated to them is corresponding pages in memory/ 8. This gives us the memory allocated for bitmaps which are used to keep an eye on all the pages.

* Another important data structure is TLB. We have implemented tlb\_store as 2d hashmap. This contains the mapping of vpn to ppn. We have changed this API signature to take vpn and ppn instead of VA and PA for the ease of programming.
* Finally, we have used mutex from pthread library for locking purpose. We have used this mutex for malloc and free functions.

Description of TLB Functions :-

Int add\_tlb() function :- In this function we add the virtual to physical page translation to the TLB . It has two input parameters vpn and ppn. We find the row in tlb which we could insert the translation (vpn->ppn), We call that row as index (vpn % TLB ENTRIES) . Then we update the translation in that row.

Int check\_tlb() function :- In this function we check if the vpn->ppn translation is present in the TLB. If that translation is present we return the physical address else we return -1. To find that we get the virtual page number using bit manipulation on the virtual address. Thereafter, we check that in the tlb.

Void print\_tlb\_missrate() :- In this function we print the tlb miss rate. To calculate that we do the division of two variables i.e., num\_tlb\_misses and num\_tlb\_lookups.

**set\_phyical\_mem** : This API is responsible for allocating memory. We have split the API into 3 parts by making use of 2 helper functions, split\_virtual\_bits and initialize\_PT. Initially we divide our 32 bit address space into various bits and store the result. The offset\_bits is used to store number of bits needed for offset which is equal to log2(PAGE\_SIZE). We calculate the inner level bits by considering page size = no of entries \* size of PTE. This way we get the bits used for inner level page table. The outer level bits is simply 32 – (offset\_bits + inner\_level\_bits).

Once we have split our 32 bit address space, we allocate physical memory using malloc. The memory for bit maps is also performed at this step. Once we are done with all memory initializations, we call initialize\_PT. Here, we reserve the last page in physical memory for the page directory. This page directory will have pow(2, outer\_level\_bits) number of 2nd level page tables. We mark all the 2nd level page table info to -1 in this step. This completes our setup for the Memory Management Unit (MMU).

**translate** : This API is used to translate the given Virtual Address(VA) and return the Physical Address. First, we check whether there exists a mapping in the TLB. If there exists a mapping we get a PPN. Once we get the PPN, we get the actual address of page by accessing physical\_memory[PPN]. The final PA is returned by adding the offset. If it’s a TLB miss, we calculate the outer index and inner index value. This is calculated using VA and bit masking operations. With the help of the outer index we get the 2nd level page table starting address. Once we have that, we find the VPN present in this 2nd level page table by adding the inner index. Once we deference it, we get the value present in this location which is PPN. Once we have PPN we follow a logic similar to above in TLB hit case and compute the actual physical address. Since, this lead to TLB miss, we increment the counter of num\_tlb\_miss to denote it’s a TLB miss.

**page\_map** : This API is used to map the Virtual Address(VA) to the Physical address(PA). Similar to translate we calculate the outer index and inner index value by using bit masking and bit shifting operation on the VA. We also calculate the VPN and PPN by right shifting the corresponding VA and PA by offset\_bits value. Before we procced in the API we again if there exists a mapping in the TLB. If there already exists a mapping we simply return from the API. If not, we compute the 2nd level page table value by accessing the page directory with outer index we computed. If this value is -1 that means we need to find a page for 2nd level page information. We do this by moving from last but 1 page in actual memory, once we find an empty page in this direction, we assign it for 2nd level page table say index. Finally, we get the address of this page table(&physical\_memory[index]) and add inner index value to get the actual page table entry location where we store the ppn. The computed PPN is stored at this found address. Lastly, this mapping is added to the TLB by calling add\_TLB function and passing VPN and PPN to this.

**get\_next\_avail** : This API is responsible for finding continuous free pages in the virtual space. We have changed the API signature to return the first index of the found-out page in virtual space for ease of coding. A simple while loop was executed for this purpose. It iterates through all the virtual pages, see if the page is free by using get\_bit API. If the bit is 0 it implies it’s a free page and we can keep continuing else if bit is 1, we need to break and start the search again as we are not able to find continuous pages requested to this API.

**t\_malloc**  : This API is responsible for allocating the requested memory by user. The very first time it is called we call the set\_physical\_memory and then we acquire a lock on this API before proceeding for thread safe implementation. The first thing we do is we compute how many pages are required for the requested memory. Once we know how many pages are required for requested memory, we try to find those many pages in both virtual address space and physical memory. We call get\_next\_avail for this purpose to know if we can find the required pages in virtual space. This returns us the start page in virtual space. Now, we find the required pages in the physical memory. NOTE: these may not be continuous. Once we are able to find pages both in virtual space and physical space, we set the pages as marked by calling the set\_bit API. Now we are ready to perform the mapping. For each page we map the corresponding virtual page to the physical page. We make use of the page\_map function to achieve this. The page\_map expects VA and PA. These are computed easily by left shifting both the virtual page and physical by offset\_bits. Once the page\_map returns we are done with the implementation. Finally, we release the mutex and return the virtual address which is the first page we found in the virtual address space left shifted by offset\_bit value.

**t\_free** : This API is used to free the number of bytes that we have previously allocated. Similar to t\_malloc we acquire a mutex lock before proceeding in the API. We then calculate the number of pages to free. We also require the information of the start page from which we must free these pages. This information is extracted from the virtual address that is passed to this API. Now we know the start page, we then validate if the page from the start page to the end page(this is computed using the passed size, end \_page = start\_page + (size/page\_size)) is valid. We again make use of get\_bit API to know if these pages are valid ones. If they are valid, we un-mark them both in virtual space and physical space. The corresponding physical pages information is found out by calling the translate API. Once both the virtual and physical pages that correspond to the freeing of requested memory is un-set, we remove the corresponding mapping from TLB too, if it exists. Finally, we release the mutex.

**put\_value** : This API is responsible for copying a given value into actual physical memory. We first check if all the pages are valid by examining the virtual page map. If any of the page is not valid, we return. Using translate we compute the physical address from the virtual address as well. At this point, we type cast our physical address, virtual address and val to a typecast pointer. This was done so we do a byte wise copy. This is performed by a simple while loop and if offset becomes 0, it means we have encountered a new virtual page and we update the physical address by calling translate again. Also, we hold a lock for this API while copying the value into the physical memory and if any error occurs wrt pages being invalid we release the lock.

**get\_value** : This API is responsible for getting the value from a physical address. This API is very similar to put\_value. We first check if all the pages are valid. Once physical address is calculated from translate function, we do byte wise copy from physical memory and store it in the value pointer passed to the API. Also, we hold a lock for this API while copying the value into the physical memory and if any error occurs wrt pages being invalid we release the lock.

Other Lookup functions :-

Set\_bit() :- In this function we set the bit, we take two parameters bitmap and index. At first we calculate the starting location where index is present, then we find the exact bit to set. Finally we set the bit at that index to 1.

Get\_bit() :- In this function we take two parameters, pagemap and index. First we calculate the starting location where index is present. Then we get the bit at required index ( which is a function parameter). Then we return that bit.

Reset\_bit() :- We use this function to reset a bit value. After calculating the starting location where index is present, we find the exact bit to be replaced. Then we reset that bit.

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

While benchmarking the output for Part 1 we changed the variables such as pageSize, No. of tlb entries, no of threads etc..

mem size = 1 gb

page size = 4k

tlb entries = 512

thread number = 15

Text

Description automatically generated

Text

Description automatically generated

mem size = 1 gb

page size = 4k

tlb entries = 512

thread number = 20

Text

Description automatically generated

Text

Description automatically generated

mem size = 1 gb

page size = 4k

tlb entries = 512

thread number = 50

Text

Description automatically generated

Text

Description automatically generated

mem size = 1 gb

page size = 4k

tlb entries = 512

thread number = 5

Text

Description automatically generated

Text

Description automatically generated

mem size = 1 gb

page size = 4k

tlb entries = 512

thread number = 40

Text

Description automatically generated

Text

Description automatically generated

mem size = 1 gb

page size = 4k

tlb entries = 512

thread number = 30

Text

Description automatically generated

Text

Description automatically generated

mem size = 1 gb

page size = 4k

**tlb entries = 256**

thread number = 15

Text

Description automatically generated

Text

Description automatically generated

mem size = 1 gb

page size = 4k

**tlb entries = 256**

thread number = 50

Text

Description automatically generated

Text

Description automatically generated

**Part 3: Explain how your code provides support for different page sizes (in multiples of 4K).**

We Supported different page sizes in our project, through the variable called PGSIZE. We could configure that variable accordingly to run our project for different page sizes.

mem size = 1 gb

page size = 8k

tlb entries = 512

thread number = 15

Text

Description automatically generated

Text

Description automatically generated

mem size = 1 gb

page size = 16k

tlb entries = 512

thread number = 15

Text

Description automatically generated

Text

Description automatically generated

mem size = 1 gb

page size = 32k

tlb entries = 512

thread number = 15

Text

Description automatically generated

Text

Description automatically generated

mem size = 1 gb

page size = 64k

tlb entries = 512

thread number = 15

Text

Description automatically generated

Text

Description automatically generated

**Some Explanation about Benchmark results** :-

1. Our code runs for different page sizes and for different number of threads**.**
2. As we can observe TLB rate tends to decrease as we increase the pagesize . This is because as increasing pageSize , TLB entries would be more helpful for a longer amount of time.

**Possible Issues with Code**

* We did not take care of the internal fragmentation part. If page size is greater and we require minimal amount of memory to be allocated, we would still be allocating one whole page for it.
* The placing of mutex lock can be optimized further. In this project, we have used locks for complete malloc, free, put\_value and get\_value. Identifying Critical Section more carefully would make code more efficient.
* We could optimize our code while searching for free physical pages. We keep traversing from index 0 to find free pages. Probably having a variable that stores the last allocate physical page and then if we search from here, it would improve the performance. We can do a full search again if we were not able to find free physical page this way first.