Project 3

Siddhi Kasera(smk339) & Mazaya Rahman(mr1411)

1. Implementation:

SetPhysicalMem: this function allocates physical memory using malloc. It calculates the number of virtual and physical pages needed to be allocated. Virtual and Physical bitmaps are also initialized to 0. We calculate the page directory bits and allocate the directory. Tlb structure is also initialized along with pthread mutexes. Using the PGSIZE, we initialized offset masks to compute page table and page directory indices.

add_TLB: this function adds a virtual to physical page translation to the TLB. Before adding a virtual address entry, it disregards the offset bits. Using the principle of First in First Out a tld is added to a linked list where the virtual and physical address is stored in each node. A count is maintained to check that the number of TLB stored does not exceed the TLB size.

check _TLB: this function traverses the linked list stored in the TLB to find if a virtual address is already stored in the TLB. The total number of requests to access a particular address translation is incremented for every call, and the miss request is incremented in case a virtual address is not found in the TLB.

print_TLB_missrate: this function calculates the total miss rate by dividing the total number of missed requests with total number of requests. We print the output from the above computation.

Translate: This function translates the virtual address passed as a parameter to its corresponding physical address. It first checks if a translation already exists in the TLB. If it exists the function adds the offset bits and returns the corresponding physical address. If a translation does not exist the function first calculates the pade directory index. If an entry is null at a particular page directory index the function returns null, else the page table index is calculated. Offset bits are added to the current physical address at the page table index. This translation is then added to the TLB, and the physical address is returned.

PageMap: this function is similar to translate, it maps a physical address to a virtual address. Given a virtual address, the page directory index is first computed. If the page directory is equal to null at that index, a page table is allocated. The page table index is computed, and the physical address is stored at that index.

Get_next_avail: this function passes the num_pages as a parameter where it is supposed to find the available free pages. It goes though the virtual address bitmap to find contiguous free pages. We used a count variable to keep track of how many contiguous free pages we come across, and return the address of the first free page if count reaches num_pages.

Get_next_avail_phys: is a function we added that is similar to the get_next_avail function where we go through the physical bitmap to find a free page. If the value at an index is equal to 0 the loop breaks and we calculate the physical address. Initialize the bitmap at that index to 1 and return the physical address.

Myalloc: this function is responsible for allocating pages and is called by the user.
We first check if the physical memory is already allocated. If it is not we call the SetPhysicalMem() function and initialize the flag to 1. We then calculate the number of pages that needs to be allocated using the num of bytes and the page size. We set the virtual address of the next available page using the get_next_avail function. We then map the virtual address to physical memory. We first check if the number of physical pages available is greater than or equal to the number of pages required. We then look for the next physical page available one at a time, calculate the virtual address and map it using the PageMap function. We calculate the virtual bitmap index using the virtual address and page size and initialize that index value to 1. We also add the mappings to the TLB.

Myfree: this function releases one or more memory pages given the virtual address and size. We first calculate the number of pages using the size. We iterate through each page, for each virtual address, page directory and page table indexes are computed. If we find a physical page stored at the page table index, it is set to null so that the mapping is removed. The corresponding physical bitmap is set to 0 to indicate a free page. The corresponding virtual bitmap index is also updated to 0 and total count of physical pages is incremented.

PutVal: We are given a virtual address (va), the address of the value to copy (val), and the number of bytes to copy (size). We first used the virtual address to compute which virtual page it is from, and at which address that page ends. Using this we found the number of bytes from va to the end of the current virtual page.
Subtracting these number of bytes from the total size, we get the number of bytes (if any) that would need to be written in the following virtual pages after the current one. We divide by PGSIZE to get the number of pages. We then iterate as many times as the number of pages, and each time we translate the virtual address to get the physical address, and copy the number of bytes we computed from val to va. After each iteration, we update va and val by the number of bytes just written.
We then update the total number of bytes still needed to be written. If it is greater than PGSIZE, then the next number of bytes to write is PGSIZE, otherwise it would just be the total bytes left.

GetVal: Same logic as putval, but here we copy bytes from va to val.

MatMult: In this function we get the values of the two matrices using getval and putval functions, using the size of int to get the correct addresses for each element in the matrix. We then performed matrix multiplication with the elements stored.

Making the code thread safe: We initialized a mutex in our code, locked and unlocked all the functions that the user calls from our library such as myalloc, my free, putval and getval. Using mutexes for these functions ensures that only one thread accesses the data structure and values used in them.

Analysis and Support for different Page Sizes:

Page Size: 4096
TLB size: 120
Matrix Size: 5
Total requests: 653
Missed Requests: 3
TLB miss rate: 0.004594

Page size: 8192
TLB size: 120
Matrix Size: 5
Total requests: 653
Missed Requests: 3
TLB miss rate: 0.004594

Page size: 8192
TLB size: 240
Matrix Size: 5
Total requests: 653
Missed Requests: 3
TLB miss rate: 0.004594

In the data above we observe that the missed requests remain the same for different page and tlb sizes if the memory allocations don't exceed the page size. This could be happening because all the matrix inputs are fitting into a page.

Page Size: 4096
TLB Size: 120
Matrix Size: 38
Total requests: 228155
Missed Requests: 6
TLB miss rate: .000026

Page Size: 8192
TLB Size: 120
Matrix Size: 38

Total requests: 228155
Missed Requests: 3
TLB miss rate: .000013

But if we allocate memory that exceeds the page size, we can see the missed rate is higher for smaller page sizes. The missed rate is halved when we increment the page size.

Challenges in the project:

In the beginning we had trouble understanding the complete structure of the project and how the page tables and the page directories were designed to store the addresses.

We also had trouble getting the values of the matrices correctly in the matmul function. Earlier while trying to access the values of the matrices we were indexing it as [i*size +j] which was giving us a strange matrix output. The correct way to index was ((unsigned int) answer + ((i*size * sizeof(int))) + j*sizeof(int))).

It also took us some time to figure out the correct way to mask to get the correct number of bits.

We changed our approach to the getval and putval functions to properly copy data that continues to more than 1 page..