Detailed logic of how you implemented each virtual memory function.

- **set_physical_mem()**
  - This function is meant to set up all global data structures and memory regions, and also establish any metadata needed to perform future calculations. To be more specific, the global two-level page table, TLB, physical memory bitmap, and actual physical memory structures are allocated in this function. General metadata was also allocated within the Metadata struct as well, with this data including the amount of virtual and physical pages, amount of bits in each portion of the virtual address, and the amount of entries per second level page table. All of this Metadata will come in handy when analyzing provided virtual addresses.
  - The amount of physical and virtual pages were calculated using the predefined constants MEM_SIZE and MAX_MEM_SIZE and dividing them by the PAGE_SIZE #define that was added in my_vm.h.
  - The amount of pages in the second level table was taken by taking the total memory in the virtual address space and dividing that by the page size, and then the amount of entries per page in the second level was determined by taking the log base 2 of that value (calculateLog() function). The amount of bits in the second portion of the virtual address was based on the amount of entries per page in the second level page table, and the amount of bits in the first portion of the virtual address was determined by subtracting the offset and second level bits from 32.
  - The size of the physical memory bitmap was calculated by taking the amount of physical pages and dividing that by 32, since we wanted to maintain an array of uint32.
  - The actual physical memory was made a character array of size MEMSIZE+1 to account for any delimiter issues.
  - A final calculation that was done was taking the bit portions for each portion of the virtual address and left shifting them so the raw value could be obtained.
- **translate()**
  - The goal of the translate() function summarized is to take a virtual address and return the corresponding physical address for that virtual address.
  - This was done in a series of steps. The first step was to take in the virtual address and extract it into the 3 parts (outer directory portion, second level portion, offset portion)
  - After this, a check was done to ensure that the given virtual address was one that actually existed.
  - After this check, a TLB check was done by taking the first two levels and combining them into one group of bits and passing this into the check_TLB() function. This is done so that if there is a TLB hit, extra memory accesses are not needed in the page table and can be skipped. However, in the case of a TLB miss, the corresponding PTE is added to the TLB and the physical page number is extracted. All of these checks are done to extract the physical page number
  - After the checks are done, the physical address corresponding to the initial physical page number is returned.
- **page_map()**

- ➢ The goal of this function summarized is to take in a virtual address and amount of pages (this parameter was added by us), and map that virtual address space to physical pages.
- ➢ This function is where the bitmap comes in handy, since in order to get the first amountOfPages amount of free physical pages, bitmap operations can be used for fast access.
- ➢ We once again extract the 3 portions of the virtual address, and call an assignPhysicalPageNum() function to take in the portions of the virtual address and assign that PTE physical pages depending on the amount needed.
- ➢ If there already exists an entry for this virtual address, no further action is needed.

❖ **t_malloc()**

- ➢ The goal of t_malloc() is to take in a specified amount of bytes and return a virtual base address corresponding to the requested allocation. This implementation is done using page_map().
- ➢ A two-level page table walk is done to retrieve the first outer level directory which has a corresponding page table that has an available entry. After this, depending on the location of that PTE, the separate parts of the virtual address are created, and these parts are merged to create a final virtual address. Then page_map() is called to map this virtual space to actual physical pages, and the virtual address is returned.

❖ **t_free()**

- ➢ The goal of this function is to take a virtual address returned by t_malloc(), a size, and free up to that many bytes.
- ➢ The physical page collection (or physical pages corresponding to the given virtual address, we want to keep track of this because of the possibility of having non-contiguous physical pages) is iterated through, and each bit corresponding to the physical page is reset back to 0 in the physical memory bitmap. The PTE state is also reset back to available for further use. There are many error checks done in this function to check whether the size specified in the function correlates to the entire physical space of the virtual address. If the amount of bytes requested to be freed is less than the total size, then -1 is returned.

❖ **put_value()**

- ➢ This function takes in a virtual address, void* val, and size of val and inserts the contents of val into the virtual address.
- ➢ Because our implementation allows for non-contiguous pages, the current page we were at had to be kept track of. Initially, however, the virtual address is split into the 3 portions and edge cases are checked.
- ➢ After this, there are 3 possible cases with put value. The first case is that the amount of bytes that want to get copied is less than one page size, or less than the page size - offset value. If this is the case, then only one iteration of copying of information is needed. If the amount of bytes to copy exceeds one page, then there are 2 more cases, with these cases being either one whole page of data will be copied or if we are at the last page, the remaining amount of data left in the buffer will be copied. We are able to do this by extracting the physical page collection for the given PTE associated with the virtual address and iterating accordingly.

❖ **get_value()**

- ➢ Similar to put_value(), this function takes in a virtual address, dst void*, and size and writes the specified amount of bytes into the dst.
- ➢ There are error checks done initially, and then a similar approach to put_value() is used. The only difference here is that the destination is now the dst pointer, and the source is the location within the actual physical memory. All of the logic remains very similar to put_value() however.

❖ **mat_mul()**

- ➢ This function takes in 3 matrix virtual addresses (that are allocated by the user using t_malloc() and put_value()) and performs matrix multiplication with them.
- ➢ Using a triple for loop to iterate through matrix a and perform dot products with matrix b, we also use a multitude of looper values to keep track of current row, column, etc.
- ➢ One bug we noticed with the project pdf was how when explaining how to index the matrix, instead of multiplying the r value by the number of rows, we had to multiply it by the number of columns. If we used the number of rows there would not be any actual increase in the position value for the matrix.
- ➢ All of the dot product results are copied into matrix c and put_value() and get_value() are used as well.

Support for different page sizes (in multiples of 8K).

In terms of support for different page sizes, we handled this in the setPhysicalMem(). By setting certain values in the Metadata structure, we were able to keep track of, based on the page size, how many bits should be in the first level, second level, and offset. One certainty is that we always know that the amount of offset bits in the virtual address will be dependent on the page size. Whatever the page size is, the offset will be log base 2 of that, since we want the offset to be able to iterate within one page. Another certainty is that the amount of bits in the second level is related to the amount of pages per page table in the second level page table, so we calculate the log of this value to get the amount of bits in the second level. The first level is calculated by determining the remaining bits (since we are looking at 32 bit address spaces), and this approach allows us to account for any page size efficiently.

Possible issues in your code (if any).

While there are not any glaring issues in the code, some interesting observations can be described. One concept we noted is that the virtual address is split into 3 parts, first level, second level, and offset. However, the interesting observation is that the offset is within one page only. Because of this, if you have multiple pages for your allocation, then all of the allocation for those multiple pages has to be done in one put_value(). Our put_value() function accounts for multiple pages, but this is if the size of the val pointer passed in is multiple pages. We think that this makes sense since the offset bits are only meant to increment within one page, and to get outside of that page is not possible within the bounds of the offset bits. Besides this feature or bug, we do not believe there are any other interesting observations.

If you implement the CS518 part, description of the 4-level page table design and support for different page sizes.

Did not implement.

Collaboration and References: State clearly all people and external resources (including on the Internet) that you consulted. What was the nature of your collaboration or usage of these resources?

Sanjay Kethineni (sk2425), Adhit Thakur (at1186) => Partners on this project
Our good friend Visual Studio.