Macauley Pinto, Itamar Levi, Ryan Townsend

NETID: msp194, il166, rrt51

01:198:416

Professor Kannan

8 April 2019

Project 3 Report

Set Physical Mem

Set physical memory is the function responsible for creating initial allocations for our multi-level page tables. Thus, the data structures set up are: a triple pointer for that acts as the different levels for paging and a bitmap for allocated memory and another bitmap for free memory. The function begins by determining outer and inner page bits. The triple pointer labeled pageDirectory is then allocated to size of a page as the reason for using two different bitmaps is the method used to allow for better functionality in the a_free and a_malloc function.

Bitmap

The bitmap is generated as an unsigned char pointer. The allocation is dependent on offset subtracted from the 32-bit virtual address. Since each entry within the pointer has eight bits, accessing a region in the the bitmap utilizes three macro defined functions. The functions are as follows: setBitInMap, clearBitInMap, and checkBitInMap. These functions all take in the same parameter: bitmap and position, but access the bits differently to return the corresponding result. The only operation that is consistent is when the bitmap access the position of the needed page divided by eight to access a bit string in the bitmap. Furthermore, the position % eight determines the location of the bit needed which then right shifts one by the position. This is done

as there are three primary operations for each macro defined function. setBitInMap uses |=, or the bitwise “or” operator, to add the bit to the bit string at the index of the bitstrip. clearBitInMap uses &=, or the bitwise “and” operator, and negates the bits that were shift by the position % 8 to determine which bit in the bit string should be cleared. Lastly, the check bit in map simply checks if the shifted bit also appears in the bitmap at the corresponding bit string index. There are also two bitmaps created: allocated and freed. Allocated indicates if the page has been allocated while free indicates if the page has been freed. This is because there is no actual deallocation of pages by an explicit free call; instead, the program will mark a page freed in the bitmap and clear the bit in the allocated to indicate the page data can be overwritten.

Translate

The translate function performs multiple bit shift operations to the virtual address to determine the page directory and page table index to access within the global pageDirectory pointer. The subsequent access will return a pointer to the page base address.

Page Map

The page map function simply takes a virtual address maps it to a physical address that is set in the global pageDirectory pointer. Thus, the offset, inner and outer page bits are found to determine the directory levels to access. The page directory and page table index is determined based on bitwise operations that shift based on the inner and outer page bits. Once the page directory and page table index are found, the page directory index is multiplied by the amount of entries expected in the page table to find the page table needed for locating the page in the bitmap. Thus, the corresponding result is added to the page table index (inner page bits of virtual address) which determines the location of the page within the bitmap to check if the page has

been allocated. This value is initially zero as it is later set in a_malloc when the pagemap function returns back to a_malloc.

Get Next Avail

The get_next_avail function is critical for generating uniform addresses. The function takes as a parameter the number of pages to be allocated. This value is used to determine the first available index for a contiguous region equivalent in bytes to the number of pages time page size. Thus, the function iterates to through all possible pages that can be generated, or two to the power of the vpn bits (outer page bits + inner page bits). The index checks the allocated and freed bitmap to find the region of memory that can be overwritten or allocated to accommodate the number of pages.

A_Malloc

The a_malloc function allocates pages based off the number of bytes requested by the user in conjunction with page granularity. Thus, the number of bytes divided by the page size determines how many pages need to be allocated. Furthermore, if the number of bytes mod page size equals zero then an extra page will not be allocated. A call to get_next_avail with number of pages needed then determines the first available index that's subsequent indices are contiguous cto allocate all pages. The index returned is bitshifted to generate a virtual address. The inner bits are determined by taking the index returned from get_next_avail and performing the modulo operator on the number of page table entries (two raised to the inner bits). The outer bits are determined by dividing the index returned by the number of page table entries. The for loop then proceeds to map pages if the page has not already been allocated and marks the allocated region in the bitmap to one upon successful completion of mapping.

A_Free

The purpose of the a_free() function is to mark a block of physical memory as ready to be overwritten by a future memory allocation. The function achieves this by first accepting a virtual address, which is a pointer to the memory allocated by an earlier a_malloc() call, as well as the size of the memory block we are freeing in bytes. Next, we determine the location of the region of memory we will be freeing by finding its position in our page table structure. We then calculate the number of pages to be freed, which may vary depending on both the size of the a_malloced space, as well as the PGSIZE. Finally, for each page to be freed, we update that pages respective bit in the bitmap to indicate that memory block is ready to be overwritten. It also indicates that a mapping to that region would not require a new page to be malloced, rather the space that exists there may now be overwritten with a subsequent a_malloc call.

Put_Value

The put value function takes in the virtual address, value, and size. We first compute the appropriate page base address by calling the translate and then store the value passed in the argument val to the memory referenced by the physical address plus the offset from the virtual address.

Get_Value

Similar to the put_value() function, the get_value() function takes in a virtual address, value, and size. The virtual address is translated and subsequently returns the page base address of for the corresponding virtual address. Finally, unlike put_value, the value is instead set to what is pointed at by the physical address plus the offset from the virtual address. Since the value is being retrieved at this already existing physical address location.

Matrix Multiply

The matrix multiply function has 3 for loops that are meant to mimic iterating through two square matrices. The void pointer variables a and b represent the beginning addresses of each matrix in the corresponding page tables, respectively. The variables address_a and address_b will be assigned to the address at each location of the elements in matrices 1 and 2, and will be necessary when calling get_value and put_value. With these variables, we can call assign them to a and b to get the addresses of the starting elements of each row and column, without actually changing the pointers a and b that indicate the starting positions of the rows and columns of matrix 1 and 2. For our most inner for loop, the third for loop, it iterates through the rows of matrix 1 and multiplies that value, received by get_value, by address_b, which is incremented to go to the next column value. Following this, the next loop will increase the second matrix address to the next element, and again will perform this operation with the next corresponding column values. After this, the final variable for that specific element, c, will be returned with put_value, and the operations will continue until complete.

Thread Support

To maintain the intended functionality of our virtual memory system, we had to implement thread safety features to all of our memory read/write operations. Failing to implement thread-safe memory operations naturally would lead to currency issues, specifically the readers/writers problem.

This allows only one thread to write at a single time, while multiple different threads can read the contents of the page tables while no threads are in a current writing session. Any critical operation that allocate or access data on the heap such a_malloc(), a_free(), set_physial_mem(),

put_value(), and get_value() all contain mutex lock and condition variables to indicate which threads are performing writes and reads before entering a critical section.