User Memory Report- CS 416

Authors:
Krishna Prajapati kjp189
Rakshay Kanthadai rsk146

Functions:

The following is a brief description of each of the library functions, as well as some helper functions. Some functions such as prints or intermediary functions are not given their own explanation as they are obvious and meant for debugging purposes, or they're explained succinctly in their corresponding function that makes use of them.

SetPhysicalMem():
This is the function that initializes our physical memory as well as our other necessary data structures like our physical/virtual bitmaps. It gets called at the first instance of my_alloc() to set up our physical memory. It has a mutex associated with it so multiple threads cannot initialize the memory at the same time, creating multiple bitmaps and physical memory locations and locks the mutex at the beginning. Then, we malloc our physical memory to the corresponding MEMSIZE definition in the h file and calculate which bits correspond to the offset, page table, and page directory of each virtual address that we will give out. We also calculate the size of the page directory table and a page table. Next, we malloc space for our page directories and set them to 0 to indicate there are no pages allocated yet. Finally, it allocates memory for the physical and virtual memory bitmaps based on the constants defined in the h file, fills them with 0s to indicate all memory locations are open, and initializes our Translation Lookaside Buffer (TLB).

Translate():
This function takes a page directory and a virtual address and returns the corresponding physical address. We already computed the number of bits the offset, page table and page directory take up in the address so we bitshift the virtual address appropriately to generate the page directory, the page table within that directory, and the offset within the page. Then, as long as the corresponding indices are of the appropriate size (i.e. less than the maximum directory and page number size that we calculated in **SetPhysicalMem()**), we know that we can translate the virtual address correctly, so we use an if statement on that condition. We use **check_in_tlb()** to check if we already have a translation for this virtual address and return it if that is the case. If we do not, then we create the entry by indexing the page directory by the corresponding index and then adding in the corresponding page table index to get the correct address from the page table. Finally, we reference what is at this memory address, which should be a physical address in the physical memory location we originally malloc'd in **SetPhysicalMem()**

PageMap():
We take a similar approach to translate and first calculate the index of the page directory and the page table from the virtual address by bitshifting based on the bits that we previously calculated in **SetPhysicalMem().** We then use an intermediary function, checkMap(), to check if the virtual bitmap has a value for this specific page (indicating that we cannot map another physical address to this virtual address). If the bitmap is 1, we return a -1 to indicate a failed mapping. Otherwise, we get the page directory entry corresponding to the virtual address. If the page directory entry is NULL, it means we need to malloc the page table at this new directory entry so we do that. Then, we index to the correct page table entry by starting at this directory entry and adding the corresponding page table index we calculated from the virtual address. Finally, with this page entry, we just point it to the physical address given and return 0 for success

get_next_avail():
This function takes in a number of pages, *n*, and searches through the virtual bitmap. The function will search for the first free page in the virtual bitmap. If found, the function will see if the next *n-1* pages are free. If it is, then the function will return the starting page of the contiguous block of free pages. If not, the function will continue searching until it finds a contiguous block of free pages that satisfy the number of pages specified. If it cannot find any contiguous blocks, then it will return -1, indicating that the number of pages is unavailable.

myalloc():
This function is the first function that the user can utilize. We first check to see if the physical memory has been set. If it has not been set, then we will call **SetPhysicalMem()**. We then calculate the number of pages that need to be reserved for the user. Afterwards, we lock the mutex to make sure no one else touches the memory while we need it. We find the next available page in the virtual memory that can fit the number of pages that need to be reserved. If we cannot find this, we simply unlock and return. Otherwise, we will get the address of the first virtual page and store it as this will be the return value. We then loop through the physical bitmap. Each time we hit a free physical page, we will call **PageMap()** on it to map the virtual address to the physical address. We will keep doing this until we have reserved all the pages needed. We then return the virtual starting address saved earlier.

myfree()
This function frees the memory that was reserved by **myalloc().** We first make sure that the virtual address and size do not exceed the maximum address space. We then calculate the number of pages that need to be freed, lock the memory mutex, and then check that all the pages (starting from the given virtual address) are marked as used. If these 2 checks pass, then we proceed with freeing the memory. Otherwise, we unlock and return. If we are freeing, we use the virtual address to calculate the indexes of the page table entry within the page directory and page table. We then simply get the physical address, calculate the physical page index, and mark that the page is free in the physical bitmap. We then clear the entry from the page table. Finally, we mark the virtual page as free in the virtual bitmap. We repeat this bitmap cleaning and page entry cleaning for all pages that need to be freed.

PutVal():
This function copies data pointed to by a value at the physical address designated by a virtual address. First, we calculate the number of pages this data will take up, then we initialize and calculate some other variables that will be necessary to use, such as an index for the page number that the virtual address corresponds to, the number of pages we have filled so far, and the offset from the virtual address. If we require more pages than we currently have consecutive space for in the virtual address space, we return immediately as the function cannot fulfill its role. We then lock the mutex for the virtual bitmap and loop through the bitmap to see that the virtual address is already allocated according to the bitmap. If we find that it is not, then we unlock the mutex and return as we need the memory to be allocated before we fill it. Then, we loop over the number of pages we have to set values for, translate the virtual address and memcpy() the appropriate number of bytes of the value pointer to the translated address. Then we increment to the next page and the next virtual address and continue to do this until we fully put the value into the memory location. We finally unlock the bitmap mutex and return.

GetVal():
This function operates exactly the same as **PutVal()** except that the memcpy() functionality is backwards: it copies the value at the physical address pointed to by a virtual address to the value pointer provided and increments all the pointers in the loop.

MatMult():
This function takes two size by size matrices and multiplies them. For a matrix multiplication, we need to traverse the rows of the first matrix and the columns of the second matrix, which we do with a nested for loop. We start our indices off at the appropriate row/column and find the index in the answer matrix that we will be putting the answer from each iteration. Then we loop over the size of the array, incrementing our first matrix index to the next column (getting each element in the row), incrementing our second matrix index to the next row (getting each element in the column) and using **GetVal()** to retrieve the value at those indices from the matrix. Then we add their multiplication to a sum variable and continue through the loop. Finally, after this loop is complete, we use **PutVal()** to insert this sum into the correct index for the answer matrix, which we calculated just before the loop began. We index the matrix as A[i*size + j] = A[i][j], and using this definition, skipping through rows and columns is easily accomplished by adding an index (skipping through columns) or adding an index*size (skipping through rows).

check_in_tlb():
This function checks if a given virtual address is in the TLB. It increments a global variable indicating the number of calls to the TLB first. Then it loops the TLB looking for a virtual address that maps to a non 0 physical address (since 0 physical address indicates the TLB entry has not been initialized) and returns the corresponding physical address. If it finishes the loop, then it did not find the virtual address in the TLB so it increments a global variable indicating the number of TLB misses and returns NULL.

put_in_tlb():

This function puts a virtual and physical address into the TLB. If the number of elements in the TLB is less than the size, it loops through the TLB for the first empty array slot and fills it in with the corresponding physical address and virtual address. In addition, the TLB struct holds a time field which keeps track of how much time each TLB struct has been in the TLB array. This time field is set to 0 upon putting the TLB struct into the array. Finally, all the times are incremented as we loop through indicating how long they have been in the TLB. Finally we increment the number-of-elements variable and return. If the number of elements in the TLB is equal to the TLB_size, then we simply loop through the array looking for an element with time value equal to 1 less than the max TLB size (since that must be the element with the longest time in the TLB, otherwise we would have a free slot). Then, we replace that TLB struct with the new physical and virtual address and a new time of 0, and finally increment all other times.

print_tlb_missrate()
This function simply takes the number of times check_in_TLB() was called and the number of misses from checking the TLB and prints the miss rate: the number of misses divided by the number of calls. If no calls were made, it prints that no calls have been made instead.

removeFromTLB()
This function searches for a virtual address in the TLB and removes it by setting the TLB struct's values all to 0 and decrements the number of elements in the array. This is used in **myfree()** to remove virtual addresses that we free from the TLB since they will never be hit after freeing.

<u>Benchmark Output:</u>

The benchmark successfully ran and first created a SIZExSIZE array of 1s using **myalloc(), PutVal(),** and **GetVal()**, and then multiplied it by itself using **MatMult()** and printed the correct SIZExSIZE array. In this specific case, SIZE was 10, so we produced a 10x10 matrix of 1s and that matrix squared correctly produced a 10x10 matrix of 10s. The benchmark also successfully uses **myfree()** to free the matrices and then proceeds to compare the newly allocated address with the old value to see if the free works as intended. Since it printed that the function works, the free operates as intended. Finally, it prints the TLB missrate as .001154, which is very good. Since there were less than 120 addresses used, the TLB should have cached all of them and only missed them upon their first translation.

<u>TLB Miss Rate and Runtime Improvement:</u>

To test the miss rate and runtime improvements, we kept to a procedure of changing only one variable (page size, TLB size, etc) at a time to see how the TLB affects runtime for different cases. To actually count the time, we used the timer from sys/time.h similar to the first project. Our baseline was the benchmark running with a 4096 page size, 1gb of memory, and a TLB size of 120. The following table shows our results:

Page Size	TLB Size	Runtime with TLB (μs)	Runtime without TLB (μs)	Miss Rate
4096	120	1640	2159	.001154
512	120	7487	8196	.001154
2	120	1385111	1304032	.690962
2	600	1326937	1303570	.115385

Thus, based on this chart, we can see that under some conditions, the TLB does improve runtime significantly and other times, it is detrimental. We can see that if the size of the TLB is large or the number of pages being used is much greater than TLB size, we will actually increase our runtime rather than improve it. If the TLB size is large, our linear search through it takes more time and the many TLB checks start to negatively affect the runtime. If there are a lot of pages being used (i.e. greater than the TLB size) then our miss rate goes up drastically (2 orders of magnitude), which also slows down the runtime significantly. Thus, for the TLB to be useful, we need to find a balance of these conditions.

Support for Different Page Sizes:

We can change the page size to different values using the PGSIZE definition in the h file. As we can see in the chart above above, either with or without the TLB, there is successful compilation and execution of the benchmark for varied page sizes and that does affect runtime. Since the page size can be any power of 2, the page size will also work for any multiple of 4K.

Possible Issues:

We do not support a page size of 1. If we try to run with that small of a page size, we run into the problem of trying to allocate more than 4GB worth of bitmaps, which we cannot address with 32 bit addressing.

Difficulties in Solving the Project:

There were a few difficulties in this project. First, we initialized a lot of indices, size constants, and numbers of bits as integers, and used them as such. However, for the sizes of the maps and directories we are using, an int does not store enough bits. We learned this the hard way as when we tried to run the benchmarks initially, we were floored with many negative numbers - a sign of overflow errors. Upon this realization, we fixed our indices and changed them to unsigned longs so that they would hold the necessary amount of information (up to 2^32 instead of the 2^32-1 for the int). We also struggled to understand the purposes of some parts of the project such as the TLB. This is because we thought that bit-shifting and converting memory addresses was not too costly of an operation compared to array looping; however we found that for smaller array sizes like the TLB, it is actually faster than converting the address with a calculation and thus the TLB is important. Finally, our last problem was with threads entering **SetPhysicalMem()** at the same time and initializing our mutex multiple times, so we had to do some research and found that to avoid this case, we needed to use the PTHREAD_MUTEX_INITIALIZER to globally initialize that mutex upon compilation, before **SetPhysicalMem()** is called.