User-level-Memory-Management

By: Kirollos Basta (knb93) and Manav Patel (mjp430)

We have included functions to replace our usage of the math library which can be activated via defining "SEPERATE_MATH". We could not directly include a makefile with the math library linked due to the library being compiled with the -c flag (signaling stop before linking). We also assume whenever threading is used -lpthread is linked in as well (to make our mutexes work).

General Functionality

In order to ensure concurrency, we take the very simple approach of locking all globals whenever they are used. This does not allow for maximum concurrency but seeing as that was not the main point of the project we did not see it as an important point to improve. Improving concurrency could simply be done with a separate lock for all globals and distinguishing locks within those via semaphores for reads and writes (as multiple reads can be allowed at one time but not multiple writes).

We use a struct to represent our inner and outer page tables instead of the predefined objects.

void set_physical_mem();

This function is responsible for allocating the memory buffer which will create an illusion of physical memory. In the first run, it initializes an array which will mimic our memory. The function calculates the page offset, the number of page table entries, the number of inner and outer bits, the number of physical pages, and the number of page directory entries. Using this information, an outer page table and physical bitmap are allocated (according to the exact amount of space needed). A virtual bitmap is allocated on a need by need basis per inner page table entry. A hashtable is also allocated for storing TLB entries.

pte_t *translate(pde_t* pgdir, void *va);

This function is given a page directory (address of the outer page table) and a virtual address, and returns the corresponding physical address. First, the

given virtual address is checked to ensure it isn't a NULL value. Once this is confirmed, the function will check the TLB to see if the virtual address has already been inserted; if it has, we can return the translation with the proper offset. If it's not in the TLB, we check if the page directory is NULL and if it is we return NULL as an error case. We then find the page directory index return NULL if the index is false in the Outer Table's Bitmap. If the value found is true, we get the inner table's data and find the proper page table index. Just like with the Outer Table's Bitmap, we return NULL if the index is false in the Inner Table's Bitmap. If the index is true, we get the physical address from the Inner Table, add it to the TLB, and return the physical address with the proper offset.

int page_map(pde_t *pgdir, void* va, void *pa)

This function walks the page directory to see if there is an existing mapping for a virtual address. If the virtual address is not present, then a new entry will be added, otherwise, if it is, replace the old physical address mapping with the new one. We first do our error check by seeing if NULL. After we get the page directory index with the given virtual address, we check the index in the Outer Table's Bitmap. If it's false, the location has not been mapped to, so we initialize the inner page table for the index. Once the Outer Table is set up and confirmed to exist, we get the page table entry index and check the index in the Inner Table's Bitmap. If it's false, we set the physical address, flip the bit and return success. If the it's true, the virtual to physical allocation given has already been mapped so we return and error (-1).

void *a_malloc(unsigned int num_bytes)

This function takes the number of bytes to allocate and returns a virtual address. Our version of my_vm.c includes a global boolean called "memory_initialized". If this is false, we call "set_physical_memory()" and set "memory_initialized" to true. We calculate the number of pages required for this memory allocation by taking the ceiling of the number of bytes divided by the page size. We then find the initial virtual address by calling the function "get_next_avail()" with the number of pages we need ("get_next_avail()" is a function that finds the virtual address with enough free space for the number of pages we need using the virtual address bitmap). If that function returns NULL, we don't have enough space for the allocation requested, so we return NULL. Assuming we got a virtual address from "get_next_avail()", for every single page, we need to allocate and create a page map. Finally, we return the initial virtual address.

void a_free(void *va, int size)

This function takes a virtual address and the number of bytes (int), and frees pages starting from the page representing the virtual address. We first test if the virtual address plus the size is a valid translation. If it's not, we do not perform the free operation. If we wished to full mimic the OS, we could instead raise a SIEGSEGV signal here. Once we confirm a valid translation, we find the physical address and free it. In order to free the physical address, we need to find the next index to free. We find the next index to free and set the bit in the physical bitmap to 0. After, we set the virtual address to be free by getting the page directory index and page table index and setting the index in the Inner Table's Bitmap to 0. We do this for every item that needs to be freed. We also remove the virtual address from the TLB.

For both put value and get value, if there is an invalid VA given, we do not put or get the value and a SIEGSEGV may occur. A SIEGSEGV from a bad value here makes sense from a regular OS perspective as normally when you try to dereference an object you do not have access to you receive a segfault anyway.

void put_value(void *va, void* val, int size)

This function takes a virtual address, a value pointer, and the size of the value pointer as an argument, and directly copies them to physical pages. We first find the physical address by performing a translation on the virtual address. We then use a helper method called "get_space_left" to get the amount of space left in a page given a virtual address. This helper method calculates the space left by returning the difference between the page size and offset. Now we want to copy the given data to the address space one page at a time. To do this, we call another helper method called "copy_data" which takes an input of a destination address, source address, the size amount to copy and amount of space left in the page. This helper method will keep track of the amount we've written. It checks if the size amount to copy is less than or equal to the space left in the page and if it is, we can do a memcpy and update the amount written to equal the size amount to copy. If the size amount to copy is greater than the space left in the page, we can only memcpy the maximum we can and update the amount written to the space left. We then reduce the amount we have to write by the amount we wrote and return it back to "put_value". Once we are back in "put_value", we check the returned size from "copy_data" and if it isn't equal to zero, we did not finish copying all of the data and have to move over the virtual address by a page to finish copying. This repeats until we have finished copying.

void get_value(void *va, void* val, int size)

This function reads the data in the virtual address to the value buffer. It works very similarly to "put_value" but is done reversely in copying data. We start

by looping through while the given size is greater than zero. Once again, we translate the physical address and get the amount of space left in the page with the virtual address. We call "copy_data" again, but this time we swap the destination and source addresses from when we called the helper method in "put_value". We continue looping through all the data by moving over to the next page to copy if necessary.

void mat_mult(void *mat1, void* mat2, int size, void *answer)

This function receives two matrices mat1 and mat2 as an argument with a size argument representing the number of rows and columns. After performing matrix multiplication, copy the result to answer array. This function uses a double for loop to go through the indices of each point in the matrices. At every point in the matrices, we loop once again to calculate the sum for that index in the answer matrix. We do this by calculating the proper addresses and using "get_value" to get the value at those addresses. We then multiply and add the values in order to get calculate the correct product that belongs in the index of the answer matrix. After the loop, we calculate the correct address of the answer matrix and use "put_value" to place the calculated product in that address. This loop continues for every index of the square matrix.

int add_TLB(void *va, void* pa)

This function adds a virtual to physical page translation to the TLB. It first verifies that the virtual address isn't NULL. Then, we get the virtual address bits and use it as a key to insert into our TLB hashtable.

pte_t * check_TLB(void *va)

This function checks the TLB hashtable for a valid translation and returns the physical page address. We get the key by getting the virtual address bits and using it as a key to search the TLB hashtable.

void print_TLB_missrate()

This function is called to calculate the TLB missrate. We do this by allocating matrices with "a_malloc" and using "mat_mult" to calculate the matrix to the power of fifty. Essentially, we are using three matrices to do this with the limitations of the "mat_mult" function, an original matrix, a copy of the original matrix, and the resulting matrix. A helper method is also used here called "perform_mat_exp" which takes as input of the size of the matrix, the original matrix, a copy of the origin matrix, the result matrix, and the power to raise it to. It is a general function to perform matrix exponentiation on a matrix. After

repeating this multiplication to the power of ten, three times. After that is done, we calculate the miss rate by the formula (miss_rate = misses / (misses + hits)). The miss rate is then displayed on the screen.

TLB Functionality

This functionality implements a direct-mapped TLB through the use of a hashtable. The hashtable is created with the number of entries equal to the defined (TLB_ENTRIES) in "my_vm.h" as default. The hashtable holds an array of unsigned long long keys as well as an array of void ** translations. The hashtable was implemented with a hash function which is the (given key % TLB_ENTRIES). Using this, we can map to the proper translation. The hashtable supports multiple functions. Recognize we opted not to use the given tlb_store but rather made our own pointer `struct tlb *tlb_entries;` that gets allocated during set_physical_mem for the hashtable.

struct tlb * table_create()

This function creates the TLB hashtable. It callocs an array TLB structs as well as callocs the arrays for the keys and translations.

void table_insert(struct tlb * ht_head, unsigned long long key, void * translation)

This function inserts a translation into the TLB hashtable. The table is passed in through "ht_head". We first ensure that the hashtable itself is not NULL before proceeding. The index for the insertion into the table is calculated through the given "key", with the formula (key % TLB_ENTRIES). Using this index, we appropriately set the key and translation into the hashtable.

void * table_get(struct tlb * ht_head, unsigned long long key)

This function retrieves a translation from the TLB hashtable. The table is once again passed in through "ht_head" and we are given the key for finding the appropriate translation. We calculate the stored key using the same formula (key % TLB_ENTRIES). If the hashtable is NULL or the passed in variable key does not equal the newly calculated stored key, we return NULL due to errors. If the stored key is equal to the passed in key, we find the appropriate translation from the hashtable and return it.

int table_remove(struct tlb * ht_head, unsigned long long key)

This function removes a translation from the TLB hashtable. The table is passed in through "ht_head" and we are given the key in order to delete the appropriate translation. We calculate a stored key with the formula (key % TLB_ENTRIES). This function maintains the same error conditions as "table_get" where we return an error (-1) if the table is NULL or if the stored key is not equal to the passed in key. If the keys match, we have found the correct entry and set the key in the table to NULL and set the translation to 0. We then return success (0).

Outputs

Benchmark Output

The following is the output as given by "test.c" :

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 1000, 2000, 3000
Storing integers to generate a SIZExSIZE matrix
Fetching matrix elements stored in the arrays
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Performing matrix multiplication with itself!
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
Freeing the allocations!
Checking if allocations were freed!
free function works
```

As you can see, the free function works as well as the matrix multiplication function as they both generate the desired outputs. What is interesting to note is the addresses returned from our calls start from outer page index 0 and inner page index 1. We skip inner page index 0 as that maps to NULL and we use NULL as a form to return errors.

TLB Miss Rate Output

The following is the output as given by "print_TLB_missrate()" in "my_vm.c" : For with prefetching in a_malloc:

```
TLB miss rate 0.000801
```

and without:

```
TLB miss rate 0.000802
```

As you can see, the miss rate generated via our print_TLB_missrate() function is quite low. This is because we are reusing the space pieces of memory multiple times to perform the matrix exponentiation. If, instead, we were using over 512 pages and were switching between them the miss rate would end up being quite high. What we made was more of an ideal scenario.

It is also interesting to note that adding the translation to the TLB immediately within a_malloc() improves the hit rate slightly. Going further, we tested the TLB miss rate on the page sizes listed in the following section and the varying page sizes had no impact, mainly because we are not crossing over pages of data with our array sizes.

Please recognize the above miss rates take into account "cold misses", adding to the TLB within the maximum TLB size and not forcing any extra misses that are not necessary. As per a piazza post, if we disregard these cold misses during prefetching we get a miss rate of 0. For this reason, we have made a function that purposely forces entries out by allocating a 2D pointer via a function named print_TLB_missrate_2. The miss rate for this function is 0.167736 when disregarding the cold misses.

Unlike the original TLB miss rate function, we did not test it with multiple page sizes but it would be expected due to the large initial allocation, it would actually benefit slightly from a larger page size.

Support for varying page sizes

We use a calculation to determine the number of bits and pages we wish to use to see the result of that calculation for specific page sizes, just change the header file to define BASIC_INFO. We choose the number of inner bits to use for the page table by performing log2(Page Size/ (the size of one pte_t)). We also choose the number of outer bits to be whatever is leftover in the virtual address space and the page offset to be the standard log2(Page Size).

For specific page sizes we have:

4096 bytes:

```
The page offset is: 12
The number of pde is: 1024
The number of outer bits is: 10
The number of pte is: 1024
The number of inner bits is: 10
```

4096 * 2 bytes:

```
The page offset is: 13
The number of pde is: 256
The number of outer bits is: 8
The number of pte is: 2048
The number of inner bits is: 11
```

4096 * 4 bytes:

```
The page offset is: 14
The number of pde is: 64
The number of outer bits is: 6
The number of pte is: 4096
The number of inner bits is: 12
```

If you wish to see different outputs feel free to change the page size manually and uncomment BASIC_INFO. You will also notice the address outputs will change as well in the benchmark files. This is due to the page size changing as the first inner page will be at a different virtual address start position and for specific page sizes (smaller than what is noted here) there may not be enough space within one page to hold everything. For example, using the last page size we get:

```
Addresses of the allocations: 4000, 8000, c000
```

within the given benchmarks/test.c. This corresponds still to the inner indices 1, 2 and 3 but since there is a 14 bit page offset the binary corresponds to:

```
0100 0000 0000 0000
1100 0000 0000 0000
```

and so on, which is different from the prior outputs given earlier in the readme.