Rutgers University CS416 Fall 2023 Operating Systems Project 3 Report
November 22nd 2023
Nivesh Nayee: nn395
Rohith Malangi: rgm111

1 - Implementation

Part 1: Virtual Memory System

void set_physical_mem()

This function will be called once the first time the user calls t_malloc.
We first calculate the number of bits for the outer page, inner page, and offset of the virtual address based on the constants given in the my_vm.h file.

Number of bits for the offset is simply log2(PGSIZE) as we need to be able to index into any byte of a page.

The remaining bits should be split between the inner and outer bits.

Since the inner bits are used to index into the page table, we set the inner bits to the number of page table entries we can fit in one page (log2(PGSIZE/size of PTE)). Then whatever remaining bits in the VA are given for the outer page.

We then create our fake physical memory of size MEMSIZE as defined in the .h file. We use malloc to allocate the memory.

We then want to create our physical bitmap. We need a bit to represent each physical page so we calculate the number of physical pages in our physical memory (PM) which is simply (PM size / PGSIZE). Since we are keeping our bitmap in a char array and one char is 8 bits, we divide the numPhysicalPages by 8 to determine how many characters to allocate.

We then set everything in the bitmap to 0 to start off.
We also initialize the entire PM to 0.
We set the bit at index 0 in our bitmap to be 1 since we are reserving the first physical page for our page directory.

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

To translate the VA to PA, we first call check_TLB to check if our virtual address is stored there. If it is, we can directly return the PA from there.

Otherwise, we will calculate the PD index, and PTE index in our physical memory based on the virtual address passed in. the PD index is calculated by right shifting the VA by the number of inner bits + offset bits which leaves us with just the outer bits. The PT index is calculated by right shifting by number of outer bits to truncate them to the left, and then right shifting by number of outer bits + offset to get the PT index bits in the correct position to which we can cast it to an unsigned long.

The offset value stored in the VA is calculated in similar way to how the PT index was calculated. Doing 2 shifts to truncate the bits that are not the offset bits (left shift) and then shift the bits down (right) to get their value.

We then check if the page directory entry in our physical memory corresponding to the PD index from the VA maps to a page table. Since we are storing our page directory in the 1st physical page of our PM, we can directly index into our PM with the PD index calculated earlier. If the value at the PDE is 0, that means that it does not map to a page table. If it is 0 we return NULL.

Otherwise if the PDE is not zero, it is holding the index in our PM of the start of a page table. So we get that PT_start value and add on the index of the PTE which we had calculated earlier. We then check if that PTE is storing a PA. If the PTE value is 0, there is no PA and we return NULL. If there's a valid PA in the PTE, we get that as that will be the start of the physical page we want to return in the PA. We then add onto it the offset value from the VA passed in that we had calculated earlier.

Since this PA was not found in the TLB initially, we add it to the TLB now along with its VPN which we calculated earlier using bit operations.

int page_map(pde_t *pgdir, void *va, void *pa, unsigned long *arr)

Explain why we changed the function signature.

The goal of this function is to put the PA passed in into the correct PTE based on the VA passed in.

Similar to the translate() function, we calculate the PD index and the PT index based on the VA passed in. We then check if there is mapping at the PD index for a page table. If there's not (the PDE contains value 0), then we have to create a page table.

If need to create a page table, we search through all the physical pages to find one that is free. We use our get_bit_at_index helper function and our physical bitmap (physical BM) to determine if a physical page is free. If a page is free (the bit in the physical BM is unset) we get the byte index of the start of this PT in PM. We then make sure this is not the same location as one of the addresses passed in through the "arr" parameter of the page_map function. We are doing this check because the page_map function is called in t_malloc and prior to it being called, we call the get_next_avail() function which returns some available physical pages in PM for the users data. So we want to make sure we are not using the space that we had already allotted for the user to store our new page table. If they are the same address, we use the "continue" keyword to break out of that iteration of the for loop and try the next physical page. If the address has not been allotted from the "arr" passed in, we can then assign that address (start of the page table) to the PDE which originally stored a value of 0. Then, using the same (PT_start_index + PT index) logic as before, we assign the PA passed in to the PTE which we calculated from the passed in VA. We then set the correct index in the physical BM which corresponds to the page we have just given for the new page table. We then calculate the PPN of the passed in PA and set that bit in our physical BM. We then add the PA and its VPN to the TLB.

If a page table was already created for the PD index, we will do similar steps as above to get the PTE index in the page table and then set the contents of the PTE to the PA passed in. Then set the bit in the physical BM. and then we add it to the TLB.

void *get_next_avail(int num_pages)

At the beginning we create an array of addresses to keep track of the available PAs we have found.

We run a for loop to go through all the physical pages, starting from index 1 because we have reserved the 1st physical page for the page directory. The loop will end once we have found the number of pages passed in by parameter. If we find a free page, we will get the PA of this page and store it in our array of found pages.

If we were unable to find the number of pages passed in, we return NULL to indicate failure. Otherwise, we return the array of physical addresses that point to the physical pages we have found.

void *t_malloc(unsigned int num_bytes)

When t_malloc is called, this is a critical section of the code so we use mutex lock to make sure it is not interrupted until it's finished.

If it's the first time t_malloc is called, we call set_physical_mem to initialize all our data structures because this should be the first time the user has interacted with our virtual memory.

We calculate the number of physical pages we need to find for the user by dividing the number of bytes requested by the PGSIZE and rounding up if there's a remainder (using the ceiling function). We then use the get_next_avail() function to find the free physical pages for the user which is returned to us as an array of physical addresses. If the array is NULL, it means we couldn't find enough free physical pages for the user so we return an "ERROR" and t_malloc fails.

Our next job is to find enough consecutive virtual pages in a page table of which we will return the first one to the user.
First we will search through all the PDEs to find one that has a mapping to a page table. In this outer for loop we also keep track of the PDE index where there is a free PDE just in case we can't find space in an already allocated page table. Once we find a PDE that maps to a page table, we go into that page table and attempt to find enough consecutive PTEs for the user based on the number of physical pages they had requested. In our for loop that searches through all the PTEs in a page table, we keep track of what the first free PTE that we found is and if we do not find enough consecutive virtual pages after that one, we set it to 0. It will be set again when we find another free virtual page.
If we were unable to find enough consecutive pages in any of the already allocated page tables, we will use the free PDE we had kept track of in the outer for loop.
We then create a VA using the PD index and PTE index we had found in our search and with some bit manipulation. This VA translates to the first physical page we had found for the user. So we then have a for loop that maps the remaining physical pages the next sequential virtual pages using our page_map() function. If we could not map the PAs to corresponding PTEs we return an error.

At the end of the function, we unlock the mutex lock and return the pointer to the first virtual address in the set of consecutive VAs.

void t_free(void *va, int size)
We use mutex lock so that we are not interrupted while we are modifying important data structures in our VM such as the bitmap and PTEs.

We use similar logic as before to calculate the PD index and PTE index of the VA passed in. We calculate the number of pages using similar logic as before. We then have a conditional that checks whether the number of PTEs the user requested to free goes beyond the end of the page table in our PM. If it does, we return an error.

Otherwise, we translate the VA passed in to a PA and if the PA is valid, we unset the bit in our physical BM that corresponds to the page of the PA. We then also set the PTE that stored that PA to 0 which means that that PTE no longer holds a PA and can be used by another t_malloc call. We also remove the PA and it's tag from the TLB.

For any remaining pages that need to be freed, we modify the original VA by adding 1 to the inner bits and then use that new VA to get the next physical page to free. We repeat the same steps as before: unset the bit in the physical BM, set the corresponding PTE to 0, and remove the VPN and PA from the TLB.

int put_value(void *va, void *val, int size)
This is a critical section, so we are using mutex lock to make sure this function isn't interrupted. We don't want to be interrupted while we are writing to our PM.

We find the number of pages we need to write data into and then translate the VA passed in by parameter to a PA. If the VA passed in doesn't map to a valid PA, we return -1.
If we only need to write into one page we simply use memcpy() function to write the data stored in the value pointer to the physical address in our PM.
Otherwise if we need to write to more than one page, we will get the PA from the VA and after writing to the physical page, we will increment the inner bits of the VA by 1 because the virtual pages should all be allocated sequentially for however many pages the user is requesting to write to. Then we will get the next PA from the incremented VA and keep doing this until we've written to all the pages requested to write to.
The conditions in the for loop check as to how much size is left over to be written to physical memory. If there is more than one PGSIZE length remaining, then we will write one PGSIZE worth of memory to the page. If there is less than one PGSIZE worth of memory remaining, we will just write that much, and if there is nothing remaining, we will break out of the loop.

void get_value(void *va, void *val, int size)
This function follows the same logic as the put_value function, but instead of doing memcpy() from the value into pointer into our PM, we do memcpy() from our PM into the val pointer.

Part 2: TLB

TLB struct has two unsigned long variable : tag (VPN) and pa. Created an array of tlb_entries of size TLB_ENTRIES

int add_TLB(unsigned long vpn, unsigned long *pa)

This function is responsible of saving the VPN and pa in the tlb_entries array to minimize the memory lookup and efficient result. This function will be called when the tlb first misses then it will add the entry in tlb array. We are adding the entry as index, vpn

% TLB_ENTRIES, which will be easier to find the entries fast and will replace the entry if the tlb is full.

pte_t * check_TLB(unsigned long vpn)

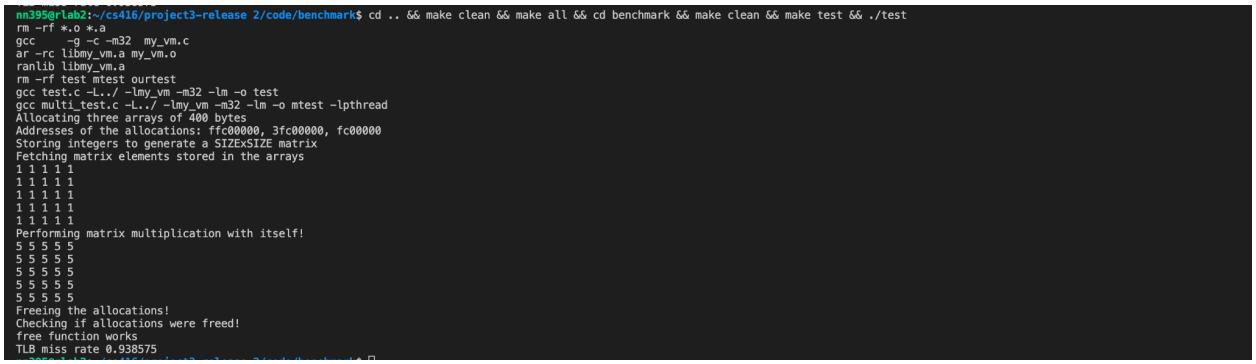
This function is responsible for checking if the entry is already stored in array and if it is then it returns the pa of that directly. If the entry is not available then return NULL. again it checks for entry availability as vpn % TLB_ENTRIES and then compares the vpn if same then return the pa.

void print_TLB_missrate()

To find the miss_rate, we need the tlb_lookup and tlb_miss which we created as global variable and incremented in translate and pagemap. First, increment tlb lookup to check if it is present in tlb entry by calling check_TLB() then if not successful then increment tlb miss. After getting the tlb lookup and tlb miss counts, we can divide the miss/lookup and the answer should be in double to get the precise miss rate.

2 - Benchmark Results

Test.c works fine



Page size = 4096
Normal page size, TLB miss rate 0.938
*Page size * 2 , TLB miss rate 0.937965*
*Page size * 3, TLB miss rate 0.937965*
*Page size * 10, TLB miss rate 0.937965*

Possible Issues

There are some synchronization issues that may cause segmentation fault

Multi-test.c

```
nn395@rlab2:~/cs416/project3-release 2/code/benchmark$ cd .. && make clean && make all && cd benchmark && make clean && make test && ./mtest
rm -rf *.o *.a
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
rm -rf test mtest ourtest
gcc test.c -L../ -lmy_vm -m32 -lm -o test
gcc multi_test.c -L../ -lmy_vm -m32 -lm -o mtest -lpthread
Allocated Pointers:
fc000000 3d030000 e000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f00000
0 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f00000
0 f000000 f000000 f000000 f000000 f000000 f000000 f000000 f000000
initializing some of the memory by in multiple threads
Randomly checking a thread allocation to see if everything worked correctly!
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 multiplications in multiple threads threads!
Segmentation fault
```