Operating System Project 3 Report

Author: Jianxin Deng (jd1216), Xuanang Wang (xw272)

Important Note: Please compile our code with -lpthread -lm, we included <math.h> and <pthread.h> in our code.

We uploaded the makefile for test.c

Part 1, Logic of each Function:

Global variables:

```
char* starting_address_physical_mem;
int num_pages;
int num_bits_vpn;
int num_bits_offset;
int num_bits_outer;
int num_bits_inner;
int num_outer_entries;
int num_inner_entries;
pde_t* starting_address_pde;
pte_t* starting_address_pte;
tlb* tlb_store;
double num_lookUp;
double num_misses;
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

/* bitmap usage variables */
typedef unsigned char* bitmap_t;
bitmap_t physical_bitmap;
bitmap_t virtual_bitmap;
int number_bitmapArray_entries;
```

TLB struct in .h file:

```
32  //Structure to represents TLB
33  typedef struct tlb{
34      /*Assume your TLB is a direct mapped TLB with number of entries as TLB_ENTRIES
35      * Think about the size of each TLB entry that performs virtual to physical
36      * address translation.
37      */
38      int stored_virtual_num;
39      unsigned long stored_physical_addr;
40  }tlb;
```

The global variable tlb* tlb_store would be an array of type tlb. Stored_virtual_num represents the unique virtual page index which would be used to see whether the desired virtual to physical transaction is already existed. Stored_physical_addr represents the corresponding starting address of a physical page.

set_physical_mem:

This function is responsible for calculating/initializing most of the global variable which would be shared for every function. For example calculating the bits needed for inner/outer/offset, initializing virtual/physical bitmap and TLB, etc.

add_tlb:

This function is responsible for adding in a virtual to physical mapping into the global TLB array. The parameter tag represents the unique index of a virtual page. We would first calculate which entry this mapping should be stored by tag % TLB_ENTRIES, then, we simply put in the physical address that was stored in the corresponding inner page table entry into the desired entry in the TLB array.

check_TLB:

This function checks whether the desired virtual to physical transaction has

already existed in the global TLB array. It is done by checking whether the desired TLB array entry (same calculation method as add_tlb) stores the tag of the current virtual page. If so, this function returns the stored physical address. If not, it returns zero and increment miss by one. Both ways, number of loop up would be incremented by one.

print_TLB_missrate:

Simply divide the global variable num_misses by global variable num_loopup (which just mentioned in check_TLB). Uncomment the bottom two print functions in this function to see number of misses/number of loop up.

Translate:

First take out the vpn, outer page and inner page bits of virtual address using get_mid_bits. Then calculate the index of this page in the starting_address_pte, where we store virtual page to physical page. For example, starting_address_pte[1] means a page in the physical memory. With this index, we can check the TLB to see if there is already a translation, if yes, we add the offset bit to the address stored in TLB and return output address to user. If there is no such entry in TLB, then first we add a TLB entry from virtual address to physical address (without offset). Then we check the bitmap of this page, if the bit is 0, it means there was something

wrong (If user use functions properly, this should not happen). After check bitmap, we get the physical address using the index we calculate before (starting_address_pte[index]), then add offset to this physical address and return the final physical address.

Page_map:

First translate input va to pa(physical address). First check if a map already exists, if so, return immediately. Otherwise, similar to translate, get the vpn, outer page and inner pages from va, and calculate index of va (no offset). Then we check if given pa is correct, if not, return -1. Otherwise, map va to pa and set both bitmap and return 0.

get_next_avail:

This function is responsible for finding the continuous virtual page that a user wants. First, we would check whether the 0th entry of page directory is 0x0. If this is true, the whole virtual page table has not been used. Therefore we malloc just one inner page table and store its starting address into the 0th entry of outer page table, set the corresponding virtual bitmap, store the index for outer/inner page table and return. If not, this would be the case that we need to loop through each existing (allocated) virtual page's bitmap and see whether there are continuous slots acquired by the user. If yes, we set the corresponding bitmap, store the index for outer/inner

page and return. If not, we would go out of the loop and see whether all the outer virtual page is being allocated. If yes, it means there would be no more space to store more data and print out error. If not, we would realloc the inner page table array by size (original total inner page table size + number of inner table entries) and store the corresponding “starting” index into the desired outer page table entry, set corresponding virtual bitmap, store the index for outer/inner page table and return.

a_malloc:

First, we calculate how many pages a user would need and declare the starting index of virtual inner/outer page table that would be passed into get_next_avail as parameters. Then, we would check whether the starting address of the physical memory array is 0x0. If it is, we know the a_malloc is being called the first time. In this situation, we would call set_physical_mem to set up the global variables we need and go ahead. Then, we would call get_next_avail to find the desired starting virtual page’s outer/inner index. Since the virtual page is continuous, we can simply loop before the looping variable reaches the number of page user wants. In the loop, we would check one bit at a time in the physical bitmap to see whether there is a free page. If there is, we would set the corresponding physical bitmap, store its starting address into the corresponding inner page table’s entry and increment the index for one (go

to the next inner page entry to store another physical address). The loop would automatically being terminated or break once it found all the physical pages needed. Then, we would call the helper function va_generator to get a self-made virtual address, add those pages' translation into the TLB and return the virtual address to the user.

A_free:

First check the size to see how many pages need to be free, let us call it "num_free_pages" (based on PGSIZE). Then calculate of index of virtual page using vpn. Set the virtual bit to 0 for virtual page with index, index + 1…index + num_free_pages. Then we need to set physical bit as well. First we get the physical page address using the index we calculate before (starting_address_pte[index]). Then to get the index of this physical page, we traverse the starting_address_physical_mem, where we store physical page addresses, compare until starting_address_pte[index] == starting_address_physical_mem [temp_index]. Now we know that this physical page is at temp_index. Using this index we can unset the physical bit.

Put_value:

First using vpn to calculate the starting physical address of va (no offset), then let starting physical address + PGSIZE to get the tail physical address.

Using translate function to get physical address of va (with offset). After these we can get the actual space left which we can store the data, by letting tail physical address minus physical address. If this space is greater than input size, just use memcpy to put input val at this physical address. If not, then it means current physical page can't store that many bytes. Then we put the number of bytes that current page can store and go the next page and continue memcpy the bytes left, repeat this process until the input size of bytes are stored.

Get_value:

Very similar to put_value. When using memcpy, the destination parameter is input val and source parameter is translated physical address (from va).

Part 2:

PGSIZE 4096, TLB entry size 512

```
[jd1216@ls all done(clean)]$ ls
benchmark  Makefile  my_vm.c  my_vm.h  test.c
[jd1216@ls all done(clean)]$ make
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
[jd1216@ls all done(clean)]$ cd benchmark
[jd1216@ls benchmark]$ ls
Makefile  test.c
[jd1216@ls benchmark]$ make
gcc test.c -L../ -lmy_vm -m32 -o test -lm -lpthread
[jd1216@ls benchmark]$ ./test
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
TLB miss rate 0.000000
```

PGSIZE 4096, TLB entry size 2 (To check miss rate)

```
[jd1216@ls all done(clean)]$ ls
benchmark  Makefile  my_vm.c  my_vm.h  test.c
[jd1216@ls all done(clean)]$ make
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
[jd1216@ls all done(clean)]$ cd benchmark
[jd1216@ls benchmark]$ ls
Makefile  test.c
[jd1216@ls benchmark]$ make
gcc test.c -L../ -lmy_vm -m32 -o test -lm -lpthread
[jd1216@ls benchmark]$ ./test
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
TLB miss rate 0.125000
[jd1216@ls benchmark]$
```

PGSIZE 8192, TLB entry size 512 (To check bigger PGSIZE)

```
[jd1216@ls all done(clean)]$ ls
benchmark  Makefile  my_vm.c  my_vm.h  test.c
[jd1216@ls all done(clean)]$ make
gcc     -g -c -m32  my_vm.c
ar -rc libmy_vm.a my_vm.o
ranlib libmy_vm.a
[jd1216@ls all done(clean)]$ cd benchmark
[jd1216@ls benchmark]$ ls
Makefile  test.c
[jd1216@ls benchmark]$ make
gcc test.c -L../ -lmy_vm -m32 -o test -lm -lpthread
[jd1216@ls benchmark]$ ./test
Allocating three arrays of 400 bytes
Addresses of the allocations: 2000, 4000, 6000
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
TLB miss rate 0.000000
[jd1216@ls benchmark]$
```

Part 3:

As the screen shots showed above, our code does support PGSIZE greater than 4096 (8192, 12228…).

Part 4, Possible Problems with our code:

We assume the user would not ask for space that is more than one inner page table per a_malloc call. i.e. if inner page table has 1024 entries, the user would not ask for more than 1024 * PGSIZE bytes per a_malloc call.

A_free: After call a_free, the physical memory is not cleaned. In other

words, initially all data in the fresh physical page should be 0x0, if user put some data in this physical page and called free. The data is still there.