Assignment 3 – Memory Management Library

Operating Systems (CS416)

Rutgers University

Name: **Jishan Desai** (jpd222), **Malav Doshi** (md1378)

The project implements the working of a memory management unit using multi-level page table which supports the multiple 4K page sizes and all the group members have contributed equally.

Knowing the following variables may help to understand the documentation faster:

- **int offset**: stores the offset bits.
- **int external_bits:** stores the external bits.
- **int internal_bits:** stores the internal bits.
- **int init:** 1 if it is the first-time library is called. 0 otherwise.
- **int tlb_count:** number of mappings stored inside the TLB.
- **int hit:** stores the number of hits in the TLB
- **double miss:** stores the number of misses in the TLB.
- **double translations:** Stores the number of times the library checks the TLB for mapping.
- **pde_t page_dir:** Structure for page directory.
- **void* PhyMem:** Physical memory.
- **unsigned long vp:** Total number of virtual pages.
- **unsigned long pp:** Total number of physical pages.
- **pthread_mutex_t Pbitmap:** Mutex lock for physical bitmap.
- **pthread_mutex_t vbitmap:** Mutex lock for virtual bitmap.
- **pthread_mutex_t PageDir:** Mutex lock for page table directory.
- **pthread_mutex_t SetMemLock:** Mutex lock for physical memory.
- **pthread_mutex_t tlbLock:** Mutex lock for TLB access.

The detail of each function and how it works is given below:

----------Library Functions [Memory] ----------

1**. void SetPhysicalMem():** This function is called only when the library is first called using ***myalloc().*** This function allocates physical memory. It also calculates ***vp*** and ***pp*** and also the ***offset.*** After that it calculates ***external_bits*** and ***internal_bits*** and also initializes the physical and virtual bitmaps and TLB.

2. **void* Translate(pdet_t *pgdir, void* va):**

- This function translates the given ***va*** to physical address using the page table and page directory entries and TLB.
- First it checks for cases if virtual address was ever allocated. If not return null.
- Otherwise, it checks if the ***va*** mapping is present in the TLB. If present, return that physical address.
- If not then it gets the physical address from the Page table by indexing into page directory and page table using external and internal bits. Then it adds the physical address to the TLB and then returns the address.

3. **int PageMap(pdet_t* pgdir, void* va, void* pa):**

- Given a virtual address (va), using the external bits and internal bits to the function locate page directory index and page table index and writes the corresponding physical address.
- The function is called in ***myalloc() to*** create a new page table entry for the given mapping of ***va*** to ***pa.***

4. **int *get_next_avail(int num_pages):**

- **num_pages** = number of pages to allocate in virtual memory.
- The function returns array of available contiguous virtual pages (index) and updates the virtual bitmap values accordingly.
- So, let's say index 2,3 and 4 are free then the function will return an array containing index 2,3 and 4 and change the values of those index to 1 (indicating that these virtual pages are taken).

5. **int* get_next_availPhy(int num_pages):**

- **num_pages** = number of pages to allocate in physical memory.
- The function returns array of available physical pages (index) and updates the physical bitmap values accordingly.
- So, let's say index 2 and 4 are free then the function will return an array containing index 2 and 4 and change the values of those index to 1 (indicating that these physical pages are taken)

6. **void* myalloc(unsigned int num_bytes):**

- **num_bytes** = number of bytes to allocate.

- If function is called the first time, then set up memory, pages, bitmaps and locks in **SetPhyicalMem().** If (variable) init = 1, then we are calling myalloc for first time otherwise not.
- Regardless of myalloc been called first time or not, function will do following things for every myalloc request.
    1. Get next available virtual pages (using **get_next_avail**).
    2. Get physical pages to back those virtual pages (using **get_next_availPhy()**)
    3. Save the mapping of each virtual page and physical page in page directory (using **PageMap()**)
    4. Return the address of where virtual page starts. If we get virtual pages of index 1,2 we return 0x2000 (index 1 is converted to an address using **AddressToIndex()**)

7. **int myfree(void* va, int size):**

- **va = address to free.**
- **size = amount of bytes to free.**
- We find the start index of the virtual page where va maps to (using **AddressToIndex()**)
- Then we find number of pages to free
- Then we find the index of last page (virtual) to free.
- Then we free every page physical page backing those virtual pages (using freePhyPages)
- Then we change the all virtual pages to be free to 0 (from 1)
- If the above process was successful function returns 0 otherwise –1.

8. **void PutVal(void* va, void* val, int size):**

- **va** = virtual address to insert val into
- **val** = val to insert.
- **size** = number of bytes to insert.

This function gets the physical address of the virtual address using ***Translate()*** and copies the ***val*** into the physical address. If the ***size*** is greater than ***PGSIZE,*** then the multiple pages are allocated using multiple Translate and keeping track of how many more bytes are still to be written.

9. **void GetVal(void* va, void* val, int size):**

- **va** = virtual address to get val from
- **val** = location where retrieved information is to be stored.
- **size** = number of bytes to insert.

This function maps the ***va*** to physical address and copies the values in it to ***val.*** If the values are stored at different locations , i.e some bytes are stored in page 0x1000 and others are stored in 0x2000. Then using similar idea as PutVal (I.e using multiple translates). Bytes are copied to val from the different pages.

10. **void MatMul(void* mat1, void* mat2, int size, void* answer):**

- mat1 = first matrix to be multiplied
- mat2 = second matrix to be multiplied
- size = size of the matrices (we assume both mat size will be same).
- answer = matrix which stores results of the mat1 and mat2 matrix multiplication

Multiplies matrices mat1 and mat2 by using GetVal to get the values of matrix and PutVal to insert values into the answer matrix after performing the calculations.

----------Translation Lookaside Buffer ----------

Our TLB designed using Linked Lists. Where every entry is a node, and every node stores the timestamp, virtual address and physical address that corresponds to the virtual address. Also the TLB uses LRU eviction policy.

Functions to handle TLB are:

11. **int add_TLB(void* va, void* pa):** This function is called when there is a miss in TLB. It adds the mapping to the TLB. If the TLB is full, then it replaces the mapping which is least recently used (LRU).

NOTE: add_TLB() is same as put_in() function mentioned in Assignment description.

12. **void* check_TLB(void* va):** This check if the mapping of the given ***va*** is present in the TLB.

13. **void print_TLB_missrate():** Calculates and prints the TLB miss rate. It divides the number of issues by the total number of translations to calculate the miss rate.

----------Helper Functions----------

14. **int getExternalBits(void* va):** Calculates the external bits using bit shifting.

15. **int getInternalBits(void* va):):** Calculates the internal bits using bit shifting.

16. **int getOffset(void* va):):** Calculates the offset bits using bit shifting.

17. **void* indextToAddress(int index):** Converts given index to virtual address.

18. **int AddressToIndex(void* va):** Given the virtual address it finds the index in the virtual bitmap.

19. **void* indextToPhyAddress(int index):** Converts given index to physical address, and returns that converted address.

20. . **int PhyAddressToIndex(void* pa):** Given the virtual address it finds the index in the physical bitmap and returns that.

21. **int getBytesToCpy(void* currA):**

- **currA:** Current Address passed in.
- Function takes the currA and finds how many bytes are left till the end of the current page. Let's say currA is 0x1500 and page size is 4096 then, the function will return how many bytes are there between 0x2000 and 0x1500.
- Returns these number of bytes calculated.
- The function is used in **PutVal()** and **GetVal()**.

22**. int power(int x, int y):** Gives the value of ***x*** raised to the ***y***.

23. **int freePhyPages(void* va):**

- **va =** virtual address which is currently been freed
- With the help of external and internal bits the function indexes the Page Table entry end erases the mapping between va and physical address.
- Then the function sets the corresponding entries in the physical bitmap to 0 (so not in use, hence freed)
- Function returns 0 if above steps were done successfully. Otherwise return –1.

24. **void add_TLB_node(struct tlb_node* node):** This function adds the given ***node*** to the TLB.

25. **void replace(void* va, void* pa):** This function replaces the node which was least recently used using the virtual address given.

----------THREAD SAFE ----------

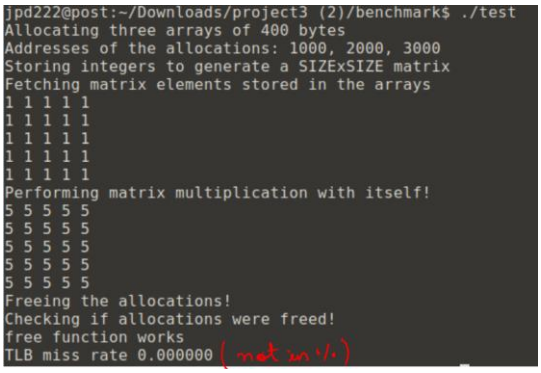
To ensure that our implementation is thread safe, instead of taking naïve approach of locking functions, we lock the data structures before using them. So, every data structure has its own lock associated with it. So, tlb has its own lock, bitmaps have their own individual locks and so on.

a. We lock the setPhyMem just to ensure that not all threads create their own memory and tlbs and bitmaps.
b. We lock virtual and physical bitmaps in myfree() myalloc() get_next_avail() and get_next_availPhy(). To ensure, no two threads get allocated same virtual address when myalloc is called, and so on.
c. Moreover, we lock page directory before accessing. In functions freePhyPages(), Translate () , and PageMap().
d. Finally, we lock the TLB before using or reading from the TLB's linked list in TLB's functions.

----------RESULTS----------

For Single Threaded function (benchmark):

For **SIZE** of Matrix is 5 and TLB size = 120 and page size = 4096



General observation was as the page size of increased, miss rate decreased.

We thought that miss rate was already minimum. So, one way we could make it more optimal is:

- We are adding entry into our cache when we are doing page mapping (every time myalloc is called). The issue with that is it is not guaranteed that because something is myalloced it will be always used. Rather we should wait till we miss and then at that point we should make an entry into the cache.

----------**CHALLENGING PARTS**----------

The only challenging part to us was taking care of part in PutVal and GetVal where we had to put half of the bytes were on one page and some bytes on other pages and similarly grab the bytes from different pages into same val pointer in GetVal implementation.