CS416 Project 3: User Level Memory Management

Aditya Verma (175007700)
Joshua Siegel (169007779)

April 6, 2019

Abstract

We were required to write a single core user-level memory management system, and implement virtual address to physical address mapping. Such an implementation of the OS abstracting the physical memory to different virtual addresses is commonly known as **Virtual Memory**. This virtual memory was implemented on the concept of "pages". These pages were referenced using "page tables" which were pointed to by a "page directory". Finally, the concept of a "TLB" was implemented to enable quick memory access in certain cases.

1 Introduction

This implementation of a virtual memory system works in the following way:

1. The user requests some amount of memory by calling **a_malloc(memory_required)**.
2. The memory required is converted into its equivalent granularity in number of pages.
3. Error checking is carried out to determine the validity of the current state of physical memory, page tables, page directory, etc.

4. If successful, a virtual address pointing to the start of this page aligned memory region is generated and it's translation is stored in the page directory.

5. The virtual address is returned to the user.

6. The user can then use this memory space to **get_value()** or **put_value()**.

7. The allocated regions are freed by calling **a_free()** on a memory location and size.

8. Similar checks to that of **a_malloc()** are carried out in these functions as well.

9. Furthermore, a direct mapped TLB maintains memory locations accessed, and we handle conflicts according to tag calculated from the address.

The page-management paradigm here is a two-level page table system. The inner level page directory is indexed using the upper few bits of the virtual address. Then, we access the page table pointed to by the page directory entry. This is done by calculating the index using the middle few bits of the virtual address. Finally, the lower most bits in the logical address give us the offset within the page being pointed to by the page table entry. This process is explained more elaborately in the sections below.

To keep track of the status (allocated or free) of pages in physical memory, we maintain a bitmap. This bitmap is essentially an array of 1's and 0's. Each index represents 32 pages. Therefore, bitmap[0] refers to pages 0-31. This is done by exploiting the fact that integers are represented with 32 bits. Hence, each index has 32 individual bits, which are used to refer to 32 pages. Therefore page 75 will be the the 11th bit in bitmap[2]. Once the bitmap is created, all indices are initialized to 0, which represents the unallocated or 'free' state. When a page is allocated, it's corresponding bit in the bitmap is set to 1.

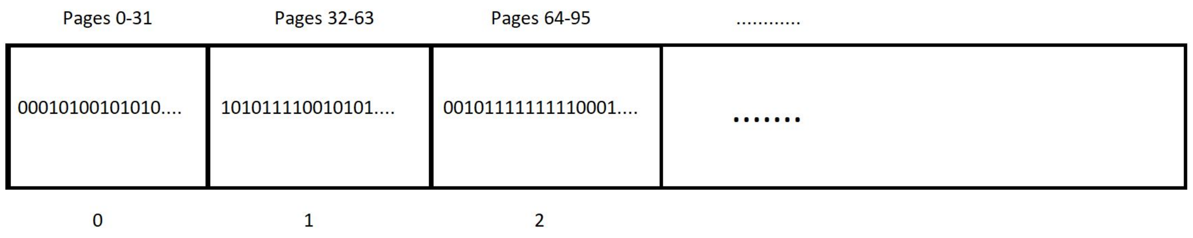

Figure 1: Pictorial representation of the bitmap implementation.

The page allocation algorithm is an optimized one. Given the current state of the bitmap, when a user requests some amount of pages, this algorithm returns the virtual address corresponding to the most optimal page location in physical memory. Since memory is page aligned in our implementation, when 'x' pages are malloc-ed, we search for the minimum contiguous sequence in the bitmap where at least 'x' consecutive pages are free. For example, if there is a block of 2 adjacent pages free, and a different block of 3 consecutive free pages (in physical memory), and the user wants to allocate 2 pages, the algorithm analyzes the bitmap, finds the 2-page block to be more optimal than the 3-page region, and allocates that block. In this way, we minimize fragmentation as best as we can.

To split up the addresses into their respective parts, page directory index, page table index, and page offset, we implement bitmasks that are calculated using the number of bits needed to extract a specific element. We then do a bit-wise and operation on the virtual address and the bitmasks and then bit shift the result to get the final desired result.

1.0.1 Assumptions

For the main codebase mentioned above, we made the following assumptions:

1. The MEMSIZE that is used to create the physical memory will not exceed **MAX_MEMSIZE = 3*1024*1024*1024+1024*1024*500 bytes**. The physical memory will default to the value of MAX_MEMSIZE in such a scenario.
2. The user will not allocate more memory than the system has. We noticed that if the user allocates MAX_MEMSIZE and then tries to

create more than 60 threads, then the system will run out of memory and either won't allocate the physical memory or won't create the last threads.

3. PGSIZE is always a power of 2.
4. The user includes the math libraries in the Makefile by using the -lm flag.

2 Implementation

For this project, our primary goal was to create an efficient and correct memory abstraction system that would manage allocating physical memory and translating it to a virtual address space. All API's were included in one .c file as listed below.

There are 2 main pieces of code. These are:

1. **my_vm.c:** This file contains all the various API's called by the user when a memory space is to be allocated, freed, modified, etc. All the various memory abstraction as well as translation functions are also declared and written in this file. Our main goal in this library is to abstract the various data structures used to maintain the page tables and page directory, and allow the process to seamlessly interact with the virtual address space.
2. **my_vm.h:** This header file declares all the data stuctures, macros, functions, constants etc, that are used by my_vm.c. Note that since we use the math library, we require a -lm flag in the Makefile to link it.

2.1 Code and APIs

The final list of methods used by us can be found in the comprehensive list below:

1. **void set_physical_mem()**: This function is called when the memory space is being initialized for the calling process. Using the declared macro **MEMSIZE** as well as **MAX_MEMSIZE**, the appropriate amount of "physical memory" is set using the **mmap()** function. Then the various memory fields and bitmasks are initialized. Given the size of physical memory and the size of individual pages, the number of pages is calculated. The number of bits required to index the page directory and each individual page table is found. The fields required to access the TLB are also calculated. The data structures that are initialized here are: Page Directory, TLB, Bitmap. A brief overview of all these fields and their allocations is listed here:

   ```
   numTotalBits=(int)ceil(log2(mem_size));
   printf("total bits being used: %d\n",numTotalBits);
   numPages=(int)ceil((mem_size)/(PGSIZE));
   numPagesBits=(int)(numTotalBits-numOffsetBits);
   numOffsetBits = (int)ceil(log2(PGSIZE));
   numPageDirBits = numPagesBits/2; //Floor division
   numPageTableBits = numPagesBits - numPageDirBits;
   numTLBBits= (int)ceil(log2(TLB_SIZE));
   numDirEntries=pow(2,numPageDirBits);
   numTableEntries=pow(2,numPageTableBits);
   lower_bitmask= (int) pow(2,numOffsetBits)-1;
   middle_bitmask= (int) (pow(2, numPageTableBits)-1 ) <<
   numOffsetBits;
   upper_bitmask=(int) (pow(2, numPageDirBits)-1 ) <<
   (numOffsetBits+numPageTableBits);
   tlb_bitmask=(int) (pow(2,numTLBBits)-1)<<(numOffsetBits);
   ```

2. **pte_t* translate(pde_t *pgdir, void *va)**: This is called whenever a virtual address to physical address conversion is required. The function takes a virtual address and the pointer to a page directory, and

translates this logical address into a physical one. It calculates the page directory index, the page table index using the bits of the virtual address and the returns the physical address of that page. If the user tries to translate an address that isn't in our page directory or page table we return NULL indicating that it has no translation and likely isn't allocated.

3. **int page_map(pde_t *pgdir, void *va, void* pa)**: Called when trying to allocate a page for the user, this function creates a mapping from virtual address **va** to physical address **pa**, and stores it in the page directory→page table structure. If a mapping already exists, we do not overwrite it and instead do nothing.

4. **int page_unmap(pde_t *pgdir, void *va)**: Called when trying to unallocate a page for the user, this function destroys a mapping for virtual address **va**, and deletes/removes it from the page directory→page table structure by setting it to NULL. We also remove it from the TLB if it exists there.

5. **void *a_malloc(unsigned int num_bytes)**: This is called by the user similar to how **malloc()** would be called normally. This function attempts to allocate physical memory of size **num_bytes** bytes for the user. **getOptimalVacantPages()** is used to find the best region of memory to allocate, thereby reducing fragmentation as much as possible. Once a region has been found successfully, the bitmap is updated with the allocated pages, and the virtual address is generated. The virtual to physical address translations are stored in the page directory→page table structure. Finally, the new virtual address returned to the user.

6. **void a_free(void *va, int size)**: Called by the user when trying to free up space in physical memory. Given a virtual address, this function will free or unallocate all pages in physical memory that correspond to virtual addresses from **va** till **va+size**. Given that all pages corresponding to this region are found to be valid and allocated, they are then freed and their translations are removed/unmapped from the TLB and the page directory→page table structure. Finally, the bitmap is updated to reflect the new state of physical memory. We assume that size will always be positive. Since size is of type int, we assume the

user will not put in a value that will overflow a signed int. For example if you malloc all the memory, you cannot free all of it in one call since a signed int cannot hold more than $2\hat{3}1$. If size overflows, we return and do not free.

7. **void put_value(void *va, void *val, int size)**: Given a virtual address and a value, this function puts the given value in the physical memory location corresponding to the given virtual address. In case the size requested from the virtual address is unallocated or invalid, an error is thrown.

8. **void get_value(void *va, void *val, int size)**: Given a virtual address and a value holder, this function gets the value in the physical memory location corresponding to the given virtual address and returns it by storing it in the value holder. In case the size requested from the virtual address is unallocated or invalid, an error is thrown.

9. **getOptimalVacantPages(int pagesToAllocate)**: Given the current state of the bitmap, when a user requests some amount of pages, this function returns the virtual address corresponding to the most optimal page location in physical memory. Since memory is page aligned in our implementation, when x pages are malloc-ed, we search for the minimum contiguous sequence in the bitmap where at least x consecutive pages are free. It returns the index (in the bitmap) of the first page in this region.

10. **void mat_mult(void *mat1, void *mat2, int size, void *answer)**: Takes 2 square integer matrices of the same size as input and returns the result of the matrix multiplication of the 2 matrices in **answer**. The **get_value()** and **put_value()** functions are used to retrieve and store the matrix elements. We assume all entries in the matrices are 4 bytes.

11. **int log_2(int x)**: Returns the value of the logarithm to the base 2 of the argument **x**.

12. **unsigned int getPageOffset(void* va)**: Returns the offset within the page pointed to by the given virtual address. This offset is calculated using the lower_bitmask.

13. **unsigned int getTableIndex(void* va)**: Returns the Page Table index pointed to by the given virtual address. It uses the middle_bitmask to get the relevant middle bits from the virtual address used for indexing the Page Table that is pointed to by the corresponding Page Directory Entry.

14. **unsigned int getDirIndex(void* va)**: Returns the Page Directory index pointed to by the given virtual address. It uses the upper_bitmask to get the relevant upper bits from the virtual address used for indexing the Page Directory.

15. **unsigned int getTLBIndex(void* va)**: Returns the TLB index pointed to by the given virtual address. It uses the tlb_bitmask to get the relevant bits from the virtual address used for indexing the TLB.

16. **pte_t* searchTLB(pte_t* va)**: Checks the Translation Lookaside Buffer for a valid virtual to physical translation for the given virtual address and returns it. If no such valid entry is found, the Page Tables and Directory is searched and the translation is stored in the TLB using a direct-mapped policy. Finally, the physical address is returned.

17. **int checkAllocated(void *va, int size)**: This function is used to check the series of pages starting from the virtual address *va. All pages corresponding from the one pointed to by **va** up until the page pointed to by **va+size** are checked. If an unallocated page is found in this range, a -1 is returned. Otherwise, a 0 is returned.

18. **void setBit(int page_num)**: Page number **page_num** in the bitmap array is set to a '1'. This is done when we are allocating the aforementioned page.

19. **void clearBit(int page_num)**: Page number **page_num** in the bitmap array is set to a '0'. This is done when we are freeing up the aforementioned page.

20. **int testBit(int page_num)**: The current value of the page number **page_num** in the bitmap array is returned.

21. **void printTLB()**: Prints the current state (tags and translations), of the TLB.

3 Testing and Analysis

Note: Testing was done on the following machines:

1. ls.cs.rutgers.edu
2. kill.cs.rutgers.edu

We tested our program in a variety of different ways including:

- Matrix Multiplication of two 40x40 matrices of integers from 1-1600
- Storing the numbers from 1-250,000,000 in array A and then copying them to array B using get_value and put_value
- Allocating all the memory and then trying to allocate more, then freeing all the memory and trying to allocate more
- Allocating all the memory and then freeing one block and trying to allocate one block, two block, etc.
- Using multiple threads to store the numbers 1-1024 in malloced memory and then reading them.

We tested our implementation with Page Sizes of:

- 4096 Bytes
- 16*4096 Bytes
- 256*4096 Bytes
- 1024*4096 Bytes

3.1 BONUS: TLB Hit Rates

As part of the Bonus assignment, we were able to implement a direct-mapped Translation Lookaside Buffer (TLB). The functioning of this TLB is similar to the way a hashtable is accessed. When an access to a certain page is made, the TLB is checked first, for a quick lookup, as opposed to traversing the Page Directory and Page Tables. If the page isn't found in the TLB, then we fetch it using the latter structure, and store it in the TLB.

We analyzed our implementation of the TLB using random accesses as well as sequential ones. As expected, the sequential memory accesses whad a very high TLB hit rate, and this hit rate only increased, as more sequential accesses were made. An example of this increasing hit rate was shown using **mat_mult()**:

- Matrix size: 1×1
  - TLB Hit Rate: 0.6666
  - TLB Miss Rate: 0.3333
- Matrix size: 2×2
  - TLB Hit Rate: 0.9318
  - TLB Miss Rate: 0.0682
- Matrix size: 4×4
  - TLB Hit Rate: 0.9875
  - TLB Miss Rate: 0.0125
- Matrix size: 16×16
  - TLB Hit Rate: 0.9997
  - TLB Miss Rate: 0.0003
- Matrix size: 32×32
  - TLB Hit Rate: 1.0000
  - TLB Miss Rate: 0.0000

List: TLB Rates with increasing size of matrices

Another test was done where we stored and immediately read a variable in both a random memory location and sequential memory locations. Our results were consistently:

- Random Set and Read:

  - TLB Hit Rate: 0.5001
  - TLB Miss Rate: 0.4999

- Sequential Set and Read:
  - TLB Hit Rate: 0.9999
  - TLB Miss Rate: 0.0001

This makes sense since after the first random access, the translation is already in the TLB for the read, and that is our only TLB hit. Therefore we get 1 miss and 1 hit per access for random access. For sequential the translation for the whole page is already in the TLB so everything but the first access is a hit. Note that all the TLB tests were done with Page Size=4096 and TLB size=16.

4 Conclusion

We successfully implemented our own virtual memory management system with a fully functional direct mapped TLB. We noticed the performance of the TLB improves drastically during sequential accesses instead of random accesses. We also got to explore the limits of the iLab memory systems. Overall this project taught us a lot about virtual memory systems and we had a lot of fun doing it.