Jacob Saltzman
CS 4414: Operating Systems
Spring 2017

Project 3: Designing a Virtual Memory Manager

Purpose

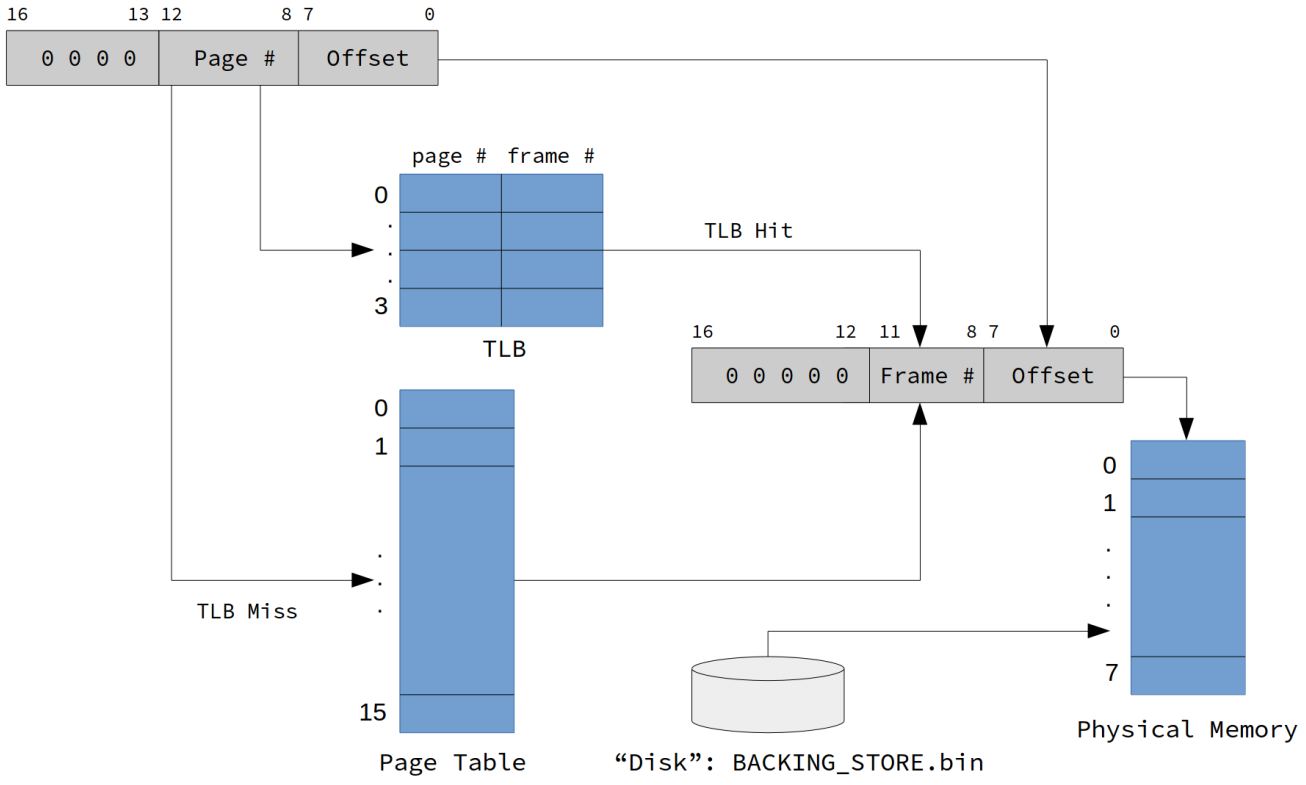
The purpose of this project was to design a virtual memory manager in simulation. While not useful in and of itself, it would not be hard to implement translate this technique to a real operating system using real memory.

Specification

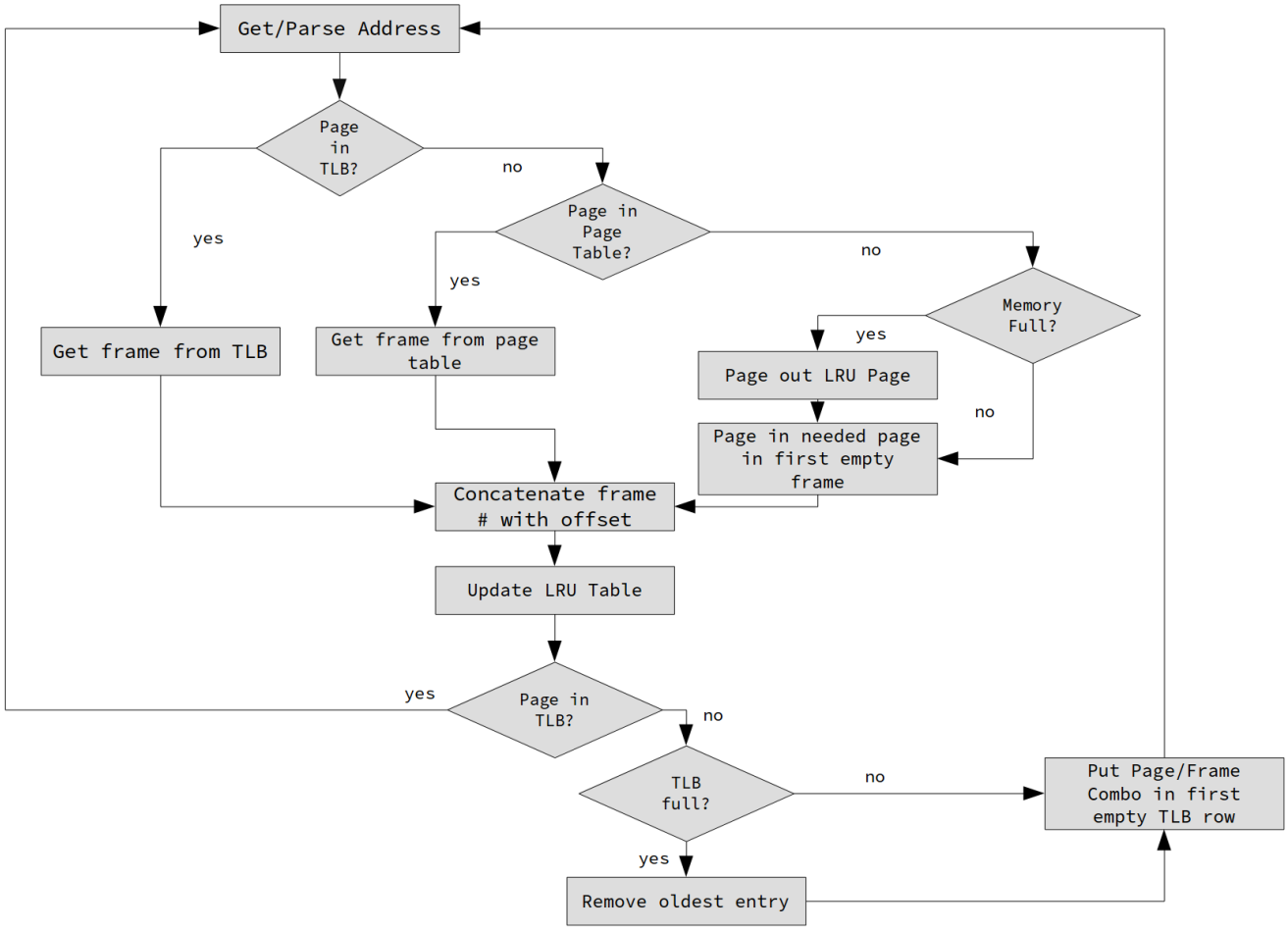
The system was specified: A byte-addressable memory system consisting of a TLB, page table, and physical memory. The virtual address space was to contain addresses on the closed interval [0, 4095], consisting of 16, 256-byte pages. The physical memory was to contain 8 frames of 256-byte pages. The TLB was to have four entries. Both the physical memory and TLB utilized the first-fit algorithm for page placement. The TLB was to use FIFO replacement, while the page table was to use the LRU page replacement strategy. Results were to be logged to stdout using a clear and logical format.

Design

The high level design of the system was specified in the textbook, with some small modifications as specified by the instructor. The system level diagram is shown below:



The basic methodology for translating virtual addresses to physical addresses is shown in the following flow chart:



The functionality for the different parts of the system were implemented using classes. The class structure for the project is as follows:

```
class PhysicalMemory {
public:
    PhysicalMemory();
    int FindFirstFrame();
    char GetMemoryContents(int frame, int offset);
    bool isFull();
    void PageIn(int frame, char pagein[FRAME_SIZE]);
```

```
    void PageOut(int frame);
private:
    static const int n_frames = N_FRAMES;
    static const int frame_size = FRAME_SIZE;
    char memory[n_frames][frame_size];
    char occupied[n_frames];
}

class PageTable {
public:
    PageTable();
    int LookupPage(int pagenum);
    int LookupPage_no_LRU(int pagenum);
    void SetPageToFrame(int pagenum, int framenum);
    bool PageIsValid(int pagenum);
    void PrintPageTable();
    void PrintInversePageTable();
    int GetLRUPage();
    void UpdateLRUList(int last_used);
    void PageOut_table(int pagenum);
private:
    static const int pgtable_entries = PAGE_TABLE_ENTRIES;
    int pgtable[PAGE_TABLE_ENTRIES];
    int valid[PAGE_TABLE_ENTRIES];
    std::vector<int> LRU_list;
};

class TranslationLookasideBuffer {
public:
    TranslationLookasideBuffer();
    bool isFull();
    TLBReturnData_t LookupTLBFrame(int pagenum);
    int UpdateTLB(int pagenum, int framenum);
    void PrintTLB();
private:
    int pagecol[TLB_ENTRIES];
    int framecol[TLB_ENTRIES];
    int occupied[TLB_ENTRIES];
    std::queue<int> FIFO_tlb;
};

class MemoryManager {
public:
    MemoryManager();
    char ReadMemory(int addr);
    int TranslateAddress(int addr);
    void PrintPageTable();
    void PrintTLB();
    void PrintInversePageTable();
    void PrintAll();
    void PrintStats();
private:
    void FileSeek(int fpage, char* dest);
```

```
    char* backend_store_filename;
    PageTable page_table;
    PhysicalMemory physical_memory;
    TranslationLookasideBuffer tlb;
    uint32_t total_accesses;
    uint32_t page_faults;
    uint32_t tlb_hitrate;
};
```

The PhysicalMemory emulates a physical memory. At its core, it is a 2D array of the format memory[frame#][offset]. It contains methods to page data in and out. It also keeps track internally of whether a frame is occupied by a page or not. It supports PageIn() and PageOut(), and has first-fit functionality via a method which returns the position of the first available frame. Like a real memory would be, it is totally unaware of the workings of the TLB and the page table.

The PageTable consists mainly of two arrays: pgtable[16] and valid[16]. These arrays hold the frame corresponding to the page, and whether the page is valid (e.g. in memory), respectively. It has accessor and mutator methods for both. It also contains a vector called LRU_list used for page replacement. The LRU algorithm works as follows: First, when a page is paged into the memory, its numer popped into the list at the tail. Every time the page is accessed, the LRU vector is scanned and the corresponding page # is moved to the tail of the list. When a page replacement is required, the page # at the head of the list is selected for replacement. LRU is in general a slow technique since it requires scanning the entire list every memory access.

The TranslationLookasideBuffer contains two main arrays: page[4] and frame[4]. On every memory access, page[] is scanned for the page currently being dereferenced. If a match is found at index j, frame[j] is returned and used to craft the physical address. It uses first-fit while being filled, and FIFO for replacement. FIFO uses a queue aptly named FIFO_tlb. When an entry is added to the TLB, its index is enqueued. When a page is dereferenced that is not in the TLB, the dequeued index is replaced. In practice, due to the fact we are using first-fit, the replacement repeats (0,1,2,3,0,1,2,3,0,1...).

Lastly, the MemoryManager manages all three pieces. It contains an instance of each and orchestrates the communication between them. It also contains the functionality to read from the BACKING_STORE.bin file. The member function MemoryManager::ReadMemory() essentially performs all the functionality shown in the flow chart shown earlier. It also tracks statistics for total accesses, page faults, and the TLB hitrate.

The main() function for the program is very simple. It instantiates a MemoryManager instance, and feeds it addresses from the addresses.txt file. It then reports the necessary statistics and shows the final page table, inverse page table (frame -> page pairs) and TLB table. Its pseudocode is shown below:

```
int main() {
     MemoryManager mmu;
     int address_list[] = ReadAddressFile();
     for addr in address_list:
          char data = mmu.ReadMemory(addr);
          print("Data at %d is %d", addr, data);
     mmu.PrintTables();
     mmu.PrintStatistics();
}
```

The code was documented in Doxygen; a pdf of the code documentation is available in the .tar file.