Project 2: Memory Management (16%)

CS371 Fall 2019

# Due: Tuesday, November 26 at 11:59:59pm

# Overview

You have **3** tasks in this project. Please complete all of them. Task 1 and Task 2 are to create parts of the operating system that simulate the behaviors of the real operating system: a simple virtual memory simulator and a dynamic memory allocator. Task 3 is to learn to use git branch.

Note: this document is the regular option of project 2. Students who seek honors option should check out the [honors option of project2](https://docs.google.com/document/d/1o6K4tvTS7IpnYFZZs1GyutwmVtatL8ngyQLtDbarlf0/edit#heading=h.kd5morerdaft).

[Intro](#_x9nixnwu9ndg)

[Task 1:Implement a VM simulator (50%)](#_ld95zqv0vy7j)

[Task 2: Implement a dynamic memory allocator (50%)](#_zcz0ale01py2)

[Task 3: Learn to use git branch (10%)](#_4vo927hw4l5u)

[Submission Instructions](#_7rhy0nwy55zn)

# Learning Outcomes

Covering memory virtualization, this project is meant to help you:

* thoroughly understand how on-demand paging works
* understand the concept of fragmentation and the trade-off between varied-size vs. fixed-size small granularity through implementing a dynamic allocator
* learn to use git branch

It may require review on your part, such as some data structures from your prerequisite cs241, which is an appropriate action for the class.

# Group

You can work in a group of maximum 3 students. Only one submission is required from a group. If there is any change to your group, please refill [the grouping form](https://forms.gle/ssB2wspeY4Zz6FVM7) before the deadline of this project.

# Platform

This project must be performed on a POSIX system (Ubuntu, Mac OSX, ...). If you are a windows user, you will need a Linux box. If you do not have one, you can install it on a virtual machine. Below is a link to a tutorial for installing Ubuntu on VirtualBox. Please choose Ubuntu 15.\* or above.

[Installing Ubuntu on VirtualBox](https://brb.nci.nih.gov/seqtools/installUbuntu.html)

# Intro

Although virtual memory and memory allocation are normally implemented in the operating system kernel, it can also be implemented at the user level. This is exactly the technique used by modern virtual machines and cloud service providers (such as, docker containers). So you will be learning a very useful indirection technique without having the headache of writing kernel-level code.

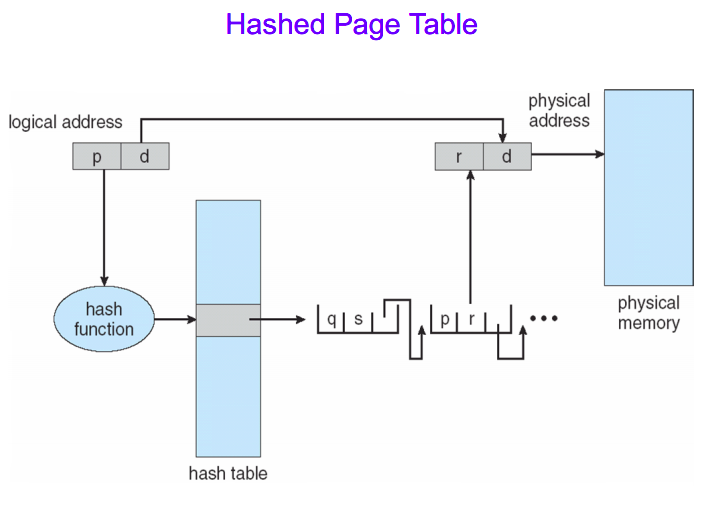
# Task 1:Implement a VM simulator (50%)

Your task 1 is to implement a VM simulator for 16 bit address. It is similar to 32 bit architecture where each page is usually 4KB (that is, 12 bits). Our training 16 bit architecture is divided into ration 10/6 bits, which means the page size is 64 bytes and there are 1024 virtual page numbers. In theory we can use an array of 1024 PTEs to store all the mappings, that is, each index/key is a VPN and each VPN has a pre-allocated slot in the array although it may never be mapped. However, this is not the most efficient data structure to store PTEs. Imagining we have only very small fractions of virtual pages mapped to physical frames, majority of PTEs are simply invalid, wasting the space of the array. In real system, it is often that page table is too big to completely fit into memory and it is necessary to use more hierarchical or more compact data structure than an array.

In this task, we will implement the page table as an inverted page table. At this point of time, let us simplify the problem and assume there is only one single process in task 1. Without loss of generality, in the future we could always expand and include PID as part of VPN.

The simplest form of an inverted page table contains one entry per physical page in a linear array. Since physical pages are now mapped to virtual, each entry contains a virtual page number instead of a physical. The physical page number is not stored, since the index in the table corresponds to it. As usual, information bits are kept around for protection and accounting purposes. In order to translate a virtual address, the virtual page number is compared against each entry, traversing the array sequentially. When a match is found, the index of the match replaces the virtual page number in the address to obtain a physical address. If no match is found, a page fault occurs. While the table size is small, the lookup time for a simple inverted page table can be very large. Finding a match may require searching the entire table. This is far too inefficient, so in order to reduce the amount of required searching, hashing is used.

A hashed inverted page table maps virtual page numbers to page table entries. Since collisions may occur, the page table must do chaining. Since each member in the chain must map to a physical page and therefore must have a corresponding page table entry, the chain can be represented as a sequence of page table entries, with each entry pointing to the next entry in the chain. This requires an extra 4 bytes per entry. Now in order to translate a virtual address, virtual page number are hashed to get an entry in the hash table. The entry’s pointer to a page table entry is followed, and the virtual page number are compared against that stored there. If they don’t match, the page table entry’s next pointer is followed to get another page table entry, and the process is repeated until a match is found.



Please create a class that implements abstract class [Memory](https://drive.google.com/file/d/1M63norEsAvbThGW4nUsBGG6IVDfHGKUb/view?usp=sharing). This class implements paging and virtual addressing using a page table and a swap file. You must implement page table as a hashed inverted page table as described above. You can treat the swap file as a virtual disk with unlimited size.

* Constructor creates an instance of Memory with specified size and prepare swap file with specified file name. Please note Memory instance must be created with specified size limit and data that are not in memory should always be stored in the swap file. The structure of swap file is arbitrary.
* Function void write(int addr, byte value)writes specified value into memory at specified position (address). The simplest way to implement it is to write-through: write into the memory and write back to the swap file. It must solve the case when the requested page is not in memory.
* Function byte read(int addr) returns value that was stored at address addr. Similar to write function this function must solve situation if the page is not in the memory. The simplest way to implement it is to load the page from the swap file into memory first.
* Please note for the above two functions, you are not allowed to read/write directly from/to the swap file without going through memory. Your implementation receives zero if it by-passes memory because it totally defeats the purpose of the task.
* You can use any page eviction algorithm such as FIFO when under memory pressure.
* **You must implement your own hash table. You can only use primitive types in Java.**
* The class must pass the test class [TestMemory](https://drive.google.com/file/d/10won7GbDtEQvs6Tdn6mbLDwxu496mACJ/view?usp=sharing).

# Task 2: Implement a dynamic memory allocator (50%)

Task 1 is to simulate memory management from the perspective of kernel. As a compliment, task 2 is to focus on the memory allocation of virtual address space (mainly heap) to the user. This task will help us gain an in-depth understanding about issues that complicate the allocation, such as external fragmentation, which arises when there are many small gaps between allocated memory blocks.

Please create a class that implements abstract class [MemoryAllocation](https://drive.google.com/file/d/1zOWUTvBNieSx5P2kReaMrGtD3OM5YjCL/view?usp=sharing). This class implements dynamic allocation of memory on heap. Your task is to implement this class using first-fit free List.

What is a first fit? A linked list of free blocks (here blocks are varied-size, contiguous memory chunks) are usually used to track available memory by an allocator. Each block in the list is represented by starting address and size of the block. first-fit searches the list and finds the first free block, that is as big as the request or larger, and allocates this memory.

More specifically,

* Constructor creates an instance of MemoryAllocation with memory of specified size. A List of free blocks should be a member of this class. At the beginning all memory is free for allocation.
* Function int alloc(int size) allocates memory with specified size. If the memory is available the function returns pointer (offset) of the beginning of allocated memory. Otherwise it returns 0.
* Function void free(int addr) release allocated memory. The memory is referenced by its pointer (offset). The function must detect if it is a valid address, that is, the function must detect if the memory was previously allocated.
* Function int size() returns the total size of available memory, it is sum of all available blocks of memory.
* Function int max\_size() returns the size of the biggest available block of memory. It is the biggest size that can be allocated.
* **You must implement your own linked list of free blocks.** You can only use primitive types in Java.
* The class must pass the test class [TestMemoryAllocation](https://drive.google.com/file/d/1lbr7f3nZy_D24WOkTqBxSRl2dIcEyiDQ/view?usp=sharing).

# Task 3: Learn to use git branch (10%)

(1) Create a new branch for project 2 and push it to the server.

git checkout -b project\_2

git push origin project\_2

Now, you can download the starter files and continue to work on this branch for your second project.

(2) Check in your code often. When submitting your code.

git commit -a

git push

are two basic commands you should master to check in your code.

The Git site has an excellent reference manual online. You must submit the git repo URL, the branch name and the commit ID through Blackboard, or you lose significant points for the project if I can not find the correct commit. The commit id is a long hash value that works as an identifier, and you can look it up in your commit history by using git log.

(3) After you finish task 1 and task 2, create a new branch for project 2 and push it to the server.

git checkout -b project\_2\_concurrent //now we are on this new branch

git push origin project\_2\_concurrent

In project 3, we will turn the non-concurrent linked list used in task 2 into a concurrent linked list because multi-threads may invoke the allocator at the same time and thus some synchronization must be enforced. This support of concurrency should be considered as a “feature” and in the real world a new feature is often developed on a “branch”, and only merged to mainstream after being thoroughly tested. We will follow this routine and prepare the branch for the concurrency feature. If you have implemented the LinkedList in task2 as an inner class of MemoryAllocation, please move it out of MemoryAllocation class and make it an independent class. Please push your changes to github server.

We will realize we now take a divergence from branch project\_2, and the change we make on project\_2\_concurrent is oblivious to branch project\_2.

# Submission Instructions

* You must submit all your source code to **github**:
  + All your work for this project should be placed on either project\_2 branch or project\_2\_concurrent branch.
  + All your work for this project should be placed under directory project\_2, which should be immediately under CS371, the same repo you created for project 1.
  + The source code for task 1 should be in a folder called vm immediately under project\_2;
  + The source code for task 2 should be in a folder called allocator immediately under project\_2;
  + LinkedList class should be under the folder allocator on project\_2\_concurrent.
  + The git commit history must show all the members of the group have contributed and committed at least one procedure, or the entire group receives 20% penalty.
* You must submit the completed grading sheet *CS371\_project2\_grading\_sheet.doc* through **Blackboard**.
* A submission is only considered on time when both code and the grading sheet are submitted on time.

Credits: Thanks to Prof. Petr Stephan for task ideas and materials, and google for hash table picture. The tasks are edited and modified to suit the needs of CS371 fall 2019 at Pace by Dr. Jun Yuan, who does not claim any intellectual property rights of these materials.