ECE250 Project 2: Hashing and Virtual Memory

Due Date: Friday, February 10, 2023 @11pm. Submit to the Dropbox on Learn

Overview

In this project, you will implement a hash table data structure to emulate virtual memory which will be further elaborated below.

You must create a C++ implementation for a hash table data structure. In this data structure, keys/values are mapped to a position in a table using a hash function. For this project, you will implement hash tables in which collisions resolve using two different techniques: (i) *open addressing* using double hashing and (ii) *separate chaining* where *the keys in each* chain are ordered*, descending,* by key*.* We will provide you with primary and secondary hash functions in the next section, and both functions use the division method.

You are permitted to use the vector class from the STL. No other STL classes may be used.

Hash Functions

Primary Hash Function

$h_1(k) = k \ mod \ m$, where *k* is the key and *m* is the size of the hash table.

Secondary Hash Function

For open addressing, you will implement the hash table using the double hashing technique to resolve collisions. The secondary hash function is $h_2(k) = \left\lfloor \frac{k}{m} \right\rfloor mod \ m$. Since $h_2(k)$ must be an odd number, you will add 1 to the resulting value if this value is even. Therefore, the hash values will be given by $h(k,i) = \big(h_1(k) + i * h_2(k)\big) \ mod \ m$, where i is an index variable from 0 to m-1.

Understanding Virtual Memory

Memory is an array of bytes in the computer. This linear data structure is broken up into smaller chunks known as “pages”. When a process runs on a modern operating system, it accesses memory in such a way that it appears to be the only process running on the computer. The operating system achieves this by mapping pointers that the process wants to access (the “virtual” pointers) to different memory addresses in physical memory (the “physical” pointers). As an example, virtual pointers may lie in the range (0x00 to 0x100) while the corresponding physical memory is in the range (0x2343 to 0x2443). Such a mapping process should be fast, otherwise the operating system spends most of its time looking up addresses. The operating system achieves this using hashing. Each mapping process is given a unique integer key known as its “Process ID” (PID). This is used as the hash key in this project. This key maps to the location of an object that contains the start of the memory page allocated to that process. The virtual address is then used as the offset from the start of that page. For example, a process whose physical pointers begin at 0x2343 attempting to access virtual address 0x50 would be given access to memory address 0x2343 + 0x50[1].

You will likely want to create a class that represents a process. This should contain, at a minimum, the PID and the start address for the physical memory page. Then, you use your hash function to store these objects in an appropriate data structure. The process for looking up physical memory is to use the PID to get the page address. Hashing the addresses directly via the PID is possible but harder (since you need to ensure that collisions do not overwrite memory), but you may do so if you wish.

In this project, memory will be represented by an array of integers of size N. Pages are contiguous blocks of memory within this array of size P. The size of the hash table then is $m = \frac{N}{P}$. For this project, N and P are chosen in such a way that m is an integer power of 2 and N is an integer multiple of P. You may assume that virtual pointers for all processes begin at virtual address 0 and end at P-1. When all m blocks are allocated, the hash table is said to be “full”.

[1] We are ignoring several complications that make this problem harder.

Program Design and Documentation

You must use proper Object-Oriented (OO) design principles to implement your solution. You will create a design using classes which have appropriately private/protected data members and appropriately public services (member functions). It is not acceptable to simply make all data members public. **Structs are not permitted.**

Write a short description of your design to be submitted along with your C++ solution files for marking according to the template posted to Learn.

Input/Output Requirements

Since you will be implementing two separate hashing implementations with *only one single driver program*, all test files will begin with either the string OPEN or ORDERED. These strings should be read but ignored.

We recommend writing *two separate driver functions* that are called from main to process the commands appropriately. Each test file will test only one of the two implementations.

As with previous projects, the driver program must read commands from standard input and write the output to standard output. The commands are described below. **All input files end with the string "END", which should end the program and produce no output.**

The Process IDs (PIDs), that are keys in this project, follow no specific format other than the fact that they will fit into a single unsigned integer.

Command	Parameters	Description	Output
M	N P	Create a new hash table with size of $m = \frac{N}{P}$. N: memory size P: page size You may assume that this command appears exactly once, as the first command in the test file after the type of test, and N, P > 0 always.	**success**
INSERT	PID	Insert a PID (a key). If it is possible, a new page of memory is allocated to PID. Note that regardless of the type of hashing used, if all pages are used insertion is not possible and the table is considered to be full. Your program must keep track of which page is allocated to which process or the commands below will not work. Assume PID > 0; this means that 0 can be used as a sentinel value.	**success** if the insertion was successful **failure** if the insertion was unable to complete since the table was full or the key was already there
SEARCH	PID	Search for the key PID in the table.	**found** PID **in p** if the desired key was found in the position **p** of the hash table **not found** if the desired key was not found
WRITE	PID ADDR x	Write the integer (x) to the memory address. Please take into account that both PID and ADDR (virtual address) are given to determine the physical address as discussed earlier in this document.	**success** if the PID was found in the table and ADDR is within the virtual address space allocated to the process **failure** if the PID was not found or the address is outside of the virtual address space

Command	Parameters	Description	Output
READ	PID ADDR	Read the integer stored in the memory address. Please take into account that both PID and ADDR are given to determine the physical address as discussed above.	ADDR x This command prints the value of the integer (x) stored in that memory location (ADDR) followed by a newline if the PID is found in the table and ADDR is within the virtual address space allocated to the process **failure** if the PID was not found or the address is outside of the virtual address space
DELETE	PID	Delete the key PID from the hash table. **NOTE:** This de-allocates the memory page associated with that PID, but you do not need to do anything with the memory stored there.	**success** if the deletion was successful **failure** if the deletion was unable to complete since the key was not found in the hash table
PRINT	m	Print the chain of stored keys in position m of the hash table. Keys should be printed in **descending order**. Keys in the chain are separated by one space. The output indicates that the chain is empty if that is the case. **NOTE:** This command is only for separate chaining. It will not be used to test double hashing.	**Example #1: PRINT 10** **If position 10 has 3 keys, the output will be:** 3423422 2070203 2062023 **Example #2: PRINT 20,** **If position 10 has an empty chain, the output will be:** chain is empty
END		All input files finish with END command.	This command does not print any output. Once it is encountered your program should end

You must analyze the average runtime of your algorithms in your design document. Assume uniform hashing, the average runtime for each search, write, read, insert, or delete operation *into the hash table only* is constant. Ignore the problem of allocating/deallocating the pages themselves.

Test Files

Learn contains some examples input files with the corresponding output. The files are named test01.in, test02.in and so on with their corresponding output files named test01.out etc.

Valgrind and Memory Leaks

Five percent of the grade (5%) will be allocated to memory leaks. We will be using the Valgrind utility to do this check. The expected behaviour of Valgrind is to indicate 0 errors and 0 leaks possible, with all allocated bytes freed. To test your code with Valgrind, presuming you are using an input file test01.in, you would use the following command:

```
valgrind ./a.out < test01.in
```

Submitting your Program

Once you have completed your solution and tested it comprehensively on your own computer or the lab computers, you should transfer your files to the eceUbuntu server and test there. We perform automated testing on this platform, so if your code works on your own computer but not on eceUbuntu it will be considered incorrect. **A makefile is required for this project since the exact source structure you use will be unique to you.**

Once you are done your testing you must create a compressed file in the tar.gz format, that contains:

- A typed document, **maximum of two pages**, describing your design. Submit this document in PDF format. The name of this file should be xxxxxxxx_design_pn.pdf where xxxxxxxx is your maximum 8-character UW user ID and n is the project number (e.g., mstachow_design_p2.pdf).
- Your test program, containing your main function.
- Required header files that you created.
- Any additional support *.cpp files that you created.
- A makefile, named Makefile, with commands to compile your solution. Your executable must be named a.out.

The name of your compressed file should be **xxxxxxxx_p2.tar.gz**, where xxxxxxxx is your UW ID as above.