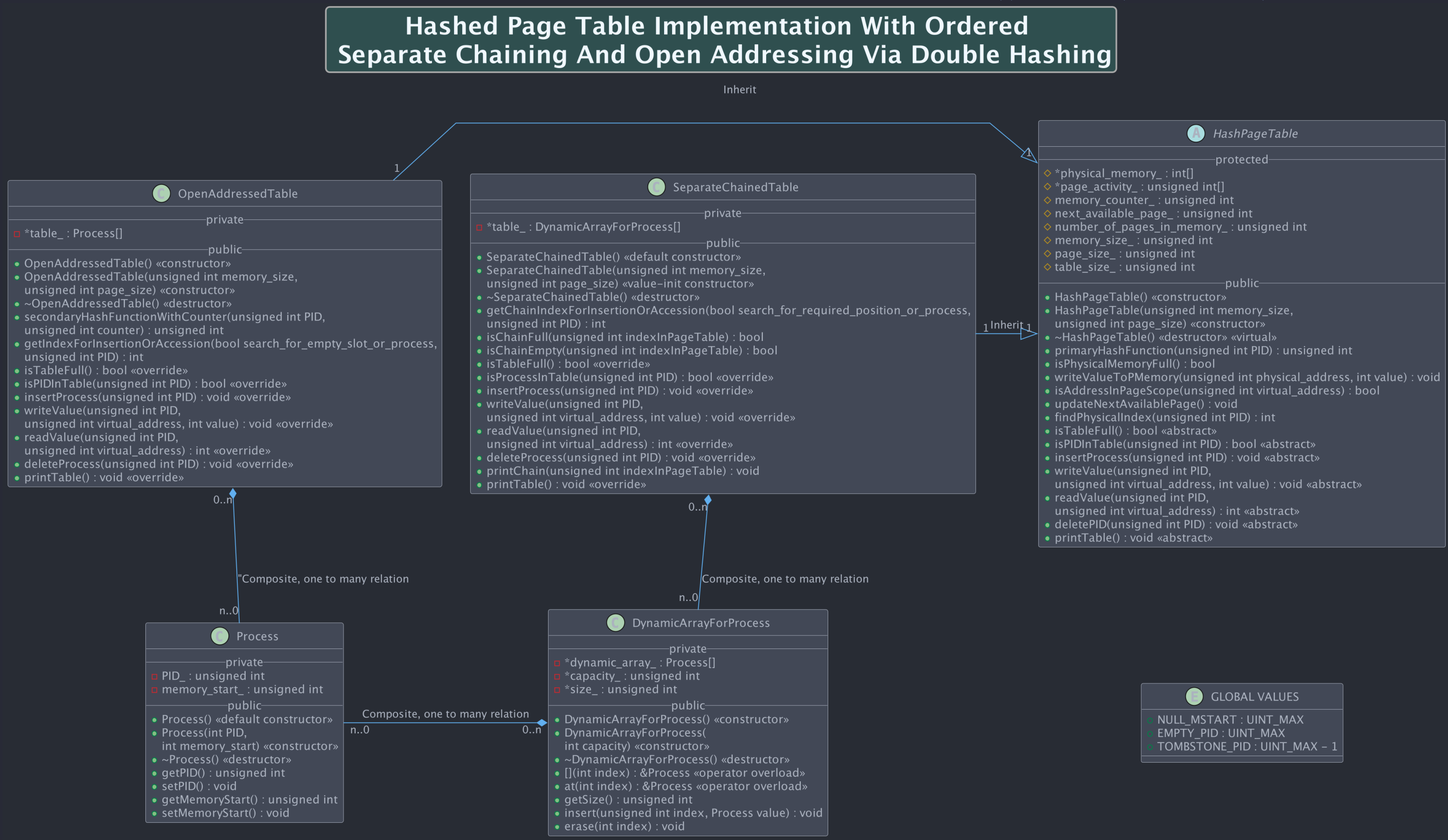
Abstract

This is the design document of Chaitanya Sharma for Project 2 for ECE 250's Winter 2023 offering.

1 Introduction

My program consisted of a base abstract class HashPageTable, and two derived classes ,namely OpenAddressedTable and SeparateChainedTable with dynamic array class for Chaining called DynamicArrayForProcess. Also, according to the first word in the test case, the main function calls either of OpenAddressedTableDriver or ChainedTableDriver for which I follow the parsing designs given by the ECE 250 staff. I have three GLOBAL VALUES defined, which act as placeholders for empty and deleted values, there values are EMPTY_PID= 0, NULL_MSTART= $UINT\_MAX$ = 4294967295 and TOMBSTONE_PID= $UINT\_MAX$ = 4294967295.
I've tested all edge test cases possible and also used the test cases provided by the jekelautograder on https://github.com/JZJisawesome/ece250-testcases.



2 Class Structure

2.1 class Process

This is a very simple class which only holds two pieces of information, the PID_ and the memory_start_, and I've provided the appropriate constructors(), a default destructor() and getters and setters for the two variables. The memory_start_ is like a virtual pointer to an index(or address) in the physical memory. And in the abstract design, two processes should neither ever have the same PID_ or the same memory_start_, except when they're deleted or initialized to by the default constructor which would give them the value of TOMBSTONE_PID or EMPTY_PID respectively and a memory start of NULL_MSTART.

2.2 class HashPageTable

In this class most of the functions are actually pure virtual functions, except the constructor(), destructor(), status boolean helper functions like isPhysicalMemoryFull, the helper functions which deal with the physical memory like findPhysicalIndex and obviously the primaryHashFunction(). I've allocated all types of data structures which I would need, but aren't unique to the derived classes, in this class like the physical_memory_[], page_activity_[] and more. I've used the page_activity_[] to keep track of the statuses of each page in physical memory, and used a virtual pointer called next_available_page_ which I update using updateNextAvailablePage(). I'll explain all other logic since they're unique to the derived classes. The runtime of all functions defined in this class are O(1) since they all just use the primaryHashFunction() to their operations, and the primaryHashFunction() is O(1), and the updateNextAvailablePage() is dependant on the hash function, so due to our assumption of even spread, it is also O(1).

For the destructor, I delete all appropriate dynamically allocated objects and equal them to nullptr.

2.3 class DynamicArrayForProcess

Instead of using the bloated std::vector class, I've created my own dynamic array class which is dynamically allocated process array, and for it's design, I've taken inspiration from the actual std::vector class, and I've used the same logic for the DynamicArrayForProcess class. Since I would only be using the functions insert(), erase(), size() and the appropriate operators overloads operator[] and at, I've only implemented those functions. When referring to std::vector class, as "bloated", I mean that for the purpose of a Page Table, most of its functionalities would be useless, and hence I've created my own class. Since even when using std::vector, we were assuming the insertion, deletion and copy operations(for increasing size), we were assuming the runtime of the functonalities to be insignifigant, inside the concept of Hashing. Hence, the runtime of all the functions in this class are O(n) but considered O(1) for the abstraction purpose of this project. This class will produce an aggregation object of the Process class's objects.

For the destructor, I delete all appropriate dynamically allocated objects.

2.4 class SeparateChainedTable

The main component of this class is a pointer which when initialized properly is an a dynamic array object of type

DynamicArrayForProcess with the size $\frac{N(memory\ size)}{P(page\ size)}$. Hence, it essentially acts as a 2D array. Since the DynamicArrayForProcess object itself initializes without an element and resizes according to need, I do not initialize the internal index DynamicArrayForProcess objects with a specified size. In this class, I have an empty constructor which just calls the empty constructor() of the HashPageTable class and initializes table_ to nullptr , a value-init constructor(), which calls the value-init constructor(), but at the same time, initializes the table_ with the value $\frac{N}{P}$.

For the destructor, I delete all appropriate dynamically allocated objects as well as call the destructor of the HashPageTable class.

Then there are chain status boolean functions like isChainEmpty() and isChainFull() whose names are self-descriptive, and functions like isTableFull() and isProcessInTable() which returns boolean values.

2.4.1 int getChainIndexForInsertionOrAccession(bool search_for_required_position_or_process, unsigned int PID) ★ a special feature of my code ★

I call this function a double option finder function, as function serves two purposes, one for insertion and one for accession, decided by the bool parameter. I've created this in such a way so that this is the single most iterative function in this whole class, and such that it first gets the table_'s index using the primaryHashFunction() and then then iterates through the DynamicArrayForProcess object at that index, and returns the index of the DynamicArrayForProcess object which is either empty or has the same PID as the one passed in the parameter according to the bool parameter. The amortized runtime analysis of this function is O(1) which means that the runtime is O(1) on average, with a worst-case runtime of O(n). The bool parameter is used with the value true for insertion and false for accession. All my other major functions look like one-liners due to the ease of use of this function. Hence, I find this function to be the most important function in this class.

2.4.2 void insertProcess(unsigned int PID)

This function takes in the PID of the process to be inserted, and then calls the getChainIndexForInsertionOrAccession function with the bool parameter as true and then inserts the process at the index returned by the function using the insert function of my DynamicArrayForProcess class. It also prompts the virtual pointer to move to the next empty spot using the updateNextAvailablePage() on the page_activity_ array and then it decrements the number_of_pages_in_memory_ by one. The amortized runtime analysis of this function is O(1) since it works by directly using the getChainIndexForInsertionOrAccession() function.

2.4.3 void writeValue(unsigned int PID, unsigned int virtual_address, unsigned int value)

This function modifies the value of physical_memory_ array at the index calculated by using getMemoryStarty() getChainIndexForInsertionOrAccession() on table_ array's index decided by the primaryHashFunction() and adding an offset of virtual_address to it. The amortized runtime analysis of this function is O(1) similar to the insertProcess() function.

2.4.4 int readValue(unsigned int PID, unsigned int virtual_address)

This function operates in a similar fashion to the writeValue() function, but instead of modifying the value, it returns that value. The amortized runtime analysis of this function is O(1) similar to the insertProcess() function.

2.4.5 void deleteProcess(unsigned int PID)

It uses the erase() function of my DynamicArrayForProcess class to delete the process from the table_ array's index decided by the primaryHashFunction(). Then it decrements the number_of_pages_in_memory_ by the number of pages the process occupied, changes the status of the page in the page_activity_ array to 0 and then prompts the virtual pointer to move to the next empty spot using the updateNextAvailablePage() function. The amortized runtime analysis of this function is O(1) similar to the insertProcess() function.

2.4.6 void printChain()

This function initiates a for loop which iterates through the table_'s index decided by the primaryHashFunction() and then prints the PID of the processes. The amortized runtime analysis of this function is O(1) since it only operates on the DynamicArrayForProcess object at the index decided by the primaryHashFunction() with the assumption of uniform spread of the PID values.

2.4.7 void printTable() CONVENIENCE FUNCTION: beautifully prints the page table

2.5 class OpenAddressedTable

This table acts as a hash table structure, but instead of resolving collisions using chaining, it uses open addressed via double hashing. A core feature of this class is that it uses Lazy Deletion methodology, which I've declared globally using the #define directive. The structures of the constructors and destructor are similar to the ChainedTable class, but the value-init constructor creates a table_ array of Process objects, and the destructor deletes the table_ array and other dynamically allocated objects. Similar to the ChainedTable class, this class also has a table_ array of Process objects, and there is also a double option finder function. For the destructor, I delete all appropriate dynamically allocated objects as well as call the destructor of the HashPageTable class. I have helper boolean functions like isTableFull() and isProcessInTable() which are used to check the status of the table.

2.5.1 unsigned int secondaryHashFunctionWithCounter(unsigned int PID, unsigned int counter)

This function is the secondary hash function used in conjunction with the primary hash function. It uses the secondaryHashFunction() function to calculate the index, and then it adds the counter to it. This function is used most importantly in the double option finder function.

2.5.2 int getIndexForInsertionOrAccession(bool search_for_empty_slot_or_process, unsigned int PID)

This function is the double option finder function. It takes in the PID of the process to be inserted or accessed, and then it uses the primaryHashFunction() function to calculate the index. I've implemented a lambda function called index() inside this function to calculate the result of the secondaryHashFunctionWithCounter() function in conjunction with the primaryHashFunction() function. I implemented a Hash Path concept in this function along with Lazy Deletion methodology. I consider the hash path to be the sequence of indexes that the index() function returns on incrementing the counter variable.

I've implemented Lazy Deletion by deciding to have values which have been pre-used empty to have the UINT_MAX and ever-empty slots to have the 0 value since we're given that all PIDs > 0. This way, I decide to ignore the re-used empty when searching a process, and stop when I see an empty slot, and in while searching for insertion i.e. an empty slot, I stop when I see a slot which has been pre-used empty or an ever-empty slot.

Along with this, I also implemented a non-empty counter inside this function, which is incremented every time I see a non-empty slot, and keep comparing this value to the total number of active processes in the page table. This lets me stop much before if I keep seeing pre-used empty slots, while searching for a process if I've already seen the total number of active processes in the page table, since I. All of this makes my function run on an amortized O(1) time complexity which is the best.

2.5.3 void other functions(unsigned int PID)

All other corresponding functions are similar to the ChainedTable class and overriden from the base HashPageTable class, but they operated on the 1D array table_ instead of the 2D array table_. And since all of them are directly dependant on the getIndexForInsertionOrAccession() they all run on an amortized O(1) time complexity.

2.5.4 void printTable() CONVENIENCE FUNCTION: beautifully prints the page table