Design Document
Jonathan Lam

I designed a Hashtable class for the hash table and a Process class for each process within the hashtable. The hash table implementation for this design uses double hashing or separate chaining for handling collisions.
p2.hpp is the header file containing the class declarations.
classdef.cpp contains the class definitions and printing success(), failure() functions.

The member variables for the Process class are:

- *unsigned int pid;* Stores the process ID as the hash key in the process.
- *unsigned int address;* Stores the start address for the physical memory page.

Member variables are private to avoid unintentional assignment to variables within main(). Both are unsigned int as they cannot be negative values.

The member functions for the Process class are:

- *Process(unsigned int pid, unsigned int address);* The constructor takes in the pid and the start address of the process that it needs to store as required in the project.
- *~Process();* Default destructor. No deallocation needed in Process.
- *unsigned int getpid();* Returns pid in the process. Has a runtime of O(1).
- *unsigned int getaddress();* Returns the start address in the process. Has a runtime of O(1).

The member variables for the Hashtable class are:

- *std::vector<std::vector<Process>> table;* A vector of vector of processes for the implementations of both double hashing and chaining.
- *std::vector<int> orderedmem;* A vector of integers that keeps track of the memory that has been used by processes.
- *int *mem;* A dynamically allocated array of integers that represents memory.
- *int size;* Stores the size of the hash table.
- *int occupied;* Keeps track of the number of processes loaded into the hash table.
- *int pagesize;* Stores the size of each page.
- *bool open;* Boolean value for either open addressing with double hashing or separate chaining.

Member variables are private to avoid unintentional assignment to variables within main().

The member functions for Hashtable class are:

- *Hashtable(unsigned int size, unsigned int pagesize, bool open);* The constructor takes in an unsigned int size, pagesize, and the boolean open that indicates the maximum size of the hash table, the size of each page, and whether it uses double hashing or chaining respectively. The above 3 parameters are assigned to their corresponding member variables. mem is initialized as a dynamically allocated int array with the size of [size * pagesize]. Vector orderedmem is assigned with the start address of each page.
- *~Hashtable();* The destructor deallocates the memory int array. Vectors do not need to be deallocated in the destructor as they will be deallocated automatically.
- *int hashprimary(unsigned int pid);* The primary hash function $h1(k)$ that takes in pid to mod with size of the hash table. Has a runtime of O(1).
- *int hashsecondary(unsigned int pid);* The secondary hash function $h2(k)$ that takes in pid to divide then mod with size of the hash table. Has a runtime of O(1).
- *int search(unsigned int pid, bool helper, bool insert);* For open addressing: loops continuously with $h(k, i) = (h1(k) + i * h2(k)) \ mod \ m$ where m is the size of the hash table, k is the pid, and i is the iterating variable for double hashing. If the table at the index obtained is not empty, return the index if the pid in the parameter matches the one at the index. If the table at the index is empty and if the insert boolean is true, which indicates the search function is used for insert function, then the index with the empty spot is returned. Otherwise, -1 is returned to indicate the pid does not exist in the hash table. For chaining: if the table at the index with hashprimary(pid) is empty, return index if it is for the insert function, otherwise return -1 to indicate pid not found. If the table at the index is not empty, loop continuously through the size of the vector at the index of the table to check for if pid in the parameter matches with the pid at the index of the vector of the index of the table. If there's a match, return the index of the vector if the search function is used as a helper function for other functions, otherwise, return the index. Return -1 if nothing matches. Assuming uniform hashing, the search function has a runtime of O(1) since the hash function is only used once.
- *void insert(unsigned int pid);* Insert function works for both double hashing and chaining. Assign temp with the result of the search function to check if the pid exists already and calls failure() if that is true. If pid does not exist and the hash table is not full, assign index with the result of the search function with insert boolean true to get the index for inserting the process. Assign address with the last element of the orderedmem vector to get the next available start address for the memory and update the vector. Use push_back function for vectors on the table at the index obtained to add a process with pid and address to the index in the table. Increment occupied and print success(). Assuming uniform hashing, the insert function has a runtime of O(1) since the search function with the same assumption has a runtime of O(1).
- *void write(unsigned int pid, int addr, signed int x);* Initialize k with the result of the search function as a helper function. If k returns -1 or addr parameter is out of range (larger than size of page), then print failure() and return. Otherwise, for open addressing: write parameter x to mem int array at the address of the process' stored start address at table[k][0] + addr parameter and print success(). For separate chaining: initialize index with the result of the primary hash function of pid and write parameter x to mem int array at the address of the process' stored start address at table[index][k] + addr parameter and print success(). x parameter can be negative and is signed

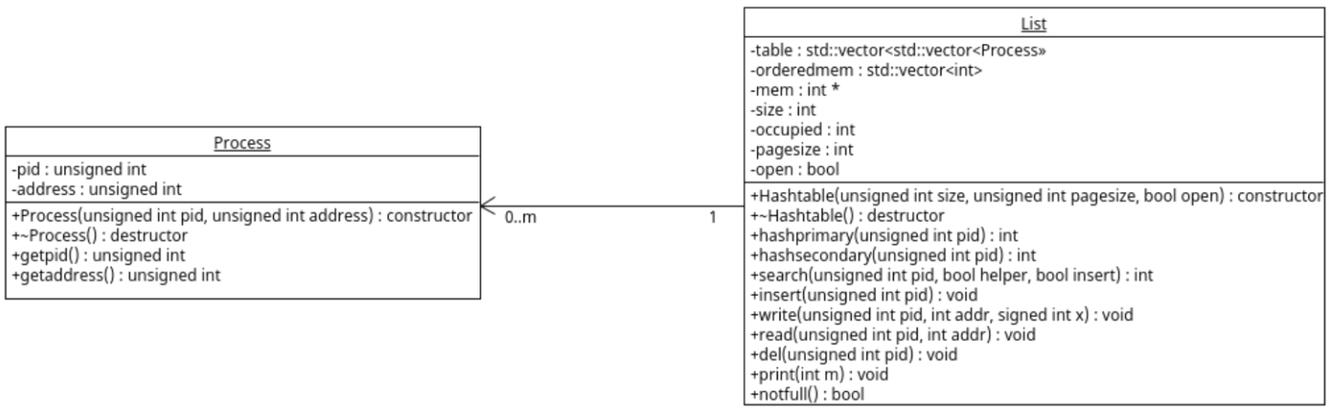
for exp Assuming uniform hashing, the write function has a runtime of O(1) since the search function with the same assumption has a runtime of O(1).

- *void read(unsigned int pid, int addr);* Initialize k with the result of the search function as a helper function. If k returns -1 or addr parameter is out of range (larger than size of page), then print failure() and return. Otherwise, for open addressing: read mem int array at the address of the process' stored start address at table[k][0] + addr parameter and print success(). For separate chaining: initialize index with the result of the primary hash function of pid and read mem int array at the address of the process' stored start address at table[index][k] + addr parameter and print success(). Assuming uniform hashing, the read function has a runtime of O(1) since the search function with the same assumption has a runtime of O(1).
- *void del(unsigned int pid);* Initialize k with the result of the search function as a helper function. If k returns -1, pid is not found, print failure() and return. Otherwise, push_back the start address stored at the process to be deleted into the vector orderedmem to store the now freed up memory space. For open addressing: erase() the begin() of the vector at table[k]. Decrement occupied and print success(). For separate chaining: Initialize index with hashprimary(pid) and erase() the begin() + k (the index of the vector at the table[index] that has the process to be deleted) at the appropriate index of the table. Decrement occupied and print success(). Assuming uniform hashing, the del function has a runtime of O(1) since the search function with the same assumption has a runtime of O(1).
- *void print(int m);* The print function is only for separate chaining. Print "chain is empty" if the vector is empty at the index m of the table. Otherwise, initialize temparray with the size of the size of the vector that will be printed. It gets the pid of each process at table[m] and sorts in descending order into the temparray. It then prints the temp array to print the chain in descending order. Assuming uniform hashing, the print function has a runtime of O(1).
- *bool notfull();* A boolean helper function that returns true if the hash table is not full and false otherwise by checking hash table size > occupied. Has a runtime of O(1).

In main():

- M command; Initialize hash table with hash table size, page size, and boolean for determining open or ordered with the given parameters.
- INSERT command; Takes in PID parameter and calls insert function with PID.
- SEARCH command; Takes in PID parameter and initializes int temp with the result of the search function. If temp returns -1, print "not found", otherwise print "found PID in p" with corresponding parameters.
- WRITE command; Takes in PID, ADDR, and x parameters and calls the write function with the parameters.
- READ command; Takes in PID and ADDR parameters and calls the read function with the parameters.
- DELETE command; Takes in PID parameter and calls the del function with PID.
- PRINT command; Takes in m parameter and calls the print function with m

UML Diagram:



Test Cases:

Tested normal operations of each command with either OPEN or ORDERED. Tested inserting keys that already exist or inserting when the table is full. Tested search for both open and ordered for processes that have been placed in different indices to test it's functionality. Tested write and read together for correct printing and ability to write and read x from the appropriate index of the int mem array. Tested delete and then search to check if the delete function is working properly for both open and ordered. Tried deleting a process that doesn't exist, deleting the first, middle, and last element of the chain in chaining, deleting the first inserted element with previous collision in open addressing. Tested print for empty chains and for correct printing in descending order of the chain. Tested deleting a process and inserting a process in the same spot with an appropriate pid. Tested out using up memory available for the hashtable.