ECE250 Project 2 Design Document Matthew Li (m739li)

For this project, I designed four classes: HashTableOpen, HashTableOrdered, Memory, and Process. These classes are stored in their own separate header file in order to keep the code separated and organized.

HashTableOpen

In my HashTableOpen class, I have member variables memorySize (int), pageSize (int), hashSize (int), and table (Process**). The table variable is used to represent the hash table; the other variables are self-explanatory. I have 7 member functions in my HashTableOpen class including the constructor and destructor.

HashTableOpen(int memorySize, int pageSize): Each HashTableOpen consists of a memorySize, a pageSize, a hashSize, and an array of Process pointers named table. Two ints, memorySize and pageSize are passed as parameters to the constructor and will always have these parameters. These two parameters determine the value of the hashSize variable (memorySize/pageSize), which is also the size of the table. The table variable has all values initialized to nullptr, representing that no processes have ever been inserted into any of the indexes.

~HashTableOpen(): When the destructor is called, I start at index 0 of the "table" variable and check if the value is not nullptr. If this is satisfied, I delete the value stored at that index. This works because if a Process pointer was deleted at that index, a Process pointer with a PID value of 0 would remain at that index (not nullptr), and that would be deleted. If there was a Process pointer that was still active in the hash table, the value would not be nullptr and would be deleted as well. After looping through the entire table array, I delete the table array itself.

void insert(unsigned int PID, Memory *Memory)
int search(unsigned int PID)
void write(unsigned int PID, int ADDR, int x, Memory *Memory)
void read(unsigned int PID, int ADDR, Memory *Memory)
void deletePID(unsigned int PID, Memory *Memory)

HashTableOrdered

In my HashTableOrdered class, I have member variables memorySize (int), pageSize (int), hashSize (int), and table (std::vector<Process*>*). The table variable is used to represent the hash table; the other variables are self-explanatory. I have 8 member functions in my HashTableOrdered class including the constructor and destructor.

HashTableOrdered(int memorySize, int pageSize): Each HashTableOrdered consists of a memorySize, a pageSize, a hashSize, and an array of vectors of Process pointers. Two ints, memorySize and pageSize are passed as parameters to the constructor and will always have these parameters. These two parameters determine the value of the hashSize variable (memorySize/pageSize), which is also the number of vectors in the table. The vectors represent the chaining process, as each one would contain a Process pointer, and if a collision happened, it would simply insert the Process pointer into it's corresponding location.

~HashTableOrdered(): When the destructor is called, I have a double for loop that goes through each index in each vector in the array. Since the size of each vector represents the number of Process pointers in it, I simply delete all indexes of each vector. After looping through the entire table array, I delete the table array itself.

void insert(unsigned int PID, Memory *Memory)
int search(unsigned int PID)
void write(unsigned int PID, int ADDR, int x, Memory *Memory)
void read(unsigned int PID, int ADDR, Memory *Memory)
void deletePID(unsigned int PID, Memory *Memory)
void print(int m)

Memory

In my Memory class, I have member variables memoryNotFree (bool *) and memoryValues(int *). The memoryNotFree variable keeps track of allocated/unallocated pages, while memoryValues keeps track of the values stored in memory. I have 6 member functions in my Memory class including the constructor and destructor.

Memory(int memorySize, int pageSize): Each Memory consists of a memoryNotFree boolean array and a memoryValues array. Two ints, memorySize and pageSize are passed as parameters to the constructor and will always have these parameters. These two parameters determine the size of the two arrays; the array "memoryNotFree" has a size of memorySize/pageSize with all values initialized to false, while memoryValues has a size of memorySize with no specified initial values.

~Memory(): The destructor simply deletes the two arrays (memoryNotFree and memoryValues); since these arrays only hold booleans and ints, there is nothing else to deallocate.

bool getMemoryNotFree(int index): Returns a boolean representing whether the index parameter in memory is free or allocated.

int getMemoryValues(int startPageAddress, int index): Returns an integer representing the value stored in memory at the start page address and index parameters.

void setMemoryNotFree(int index, bool value): Sets the boolean value in memory representing whether the index in memory is free or allocated.

void setMemoryValues(int startPageAddress, int index, int value): Sets the integer value stored in memory at the start page address and index parameters

Process

In my Process class, I have member variables PID (unsigned int) and startPageAddress (int). The start page address is assigned the location in memory that has been allocated for the PID. I have 4 member functions in my Process class including the constructor and destructor.

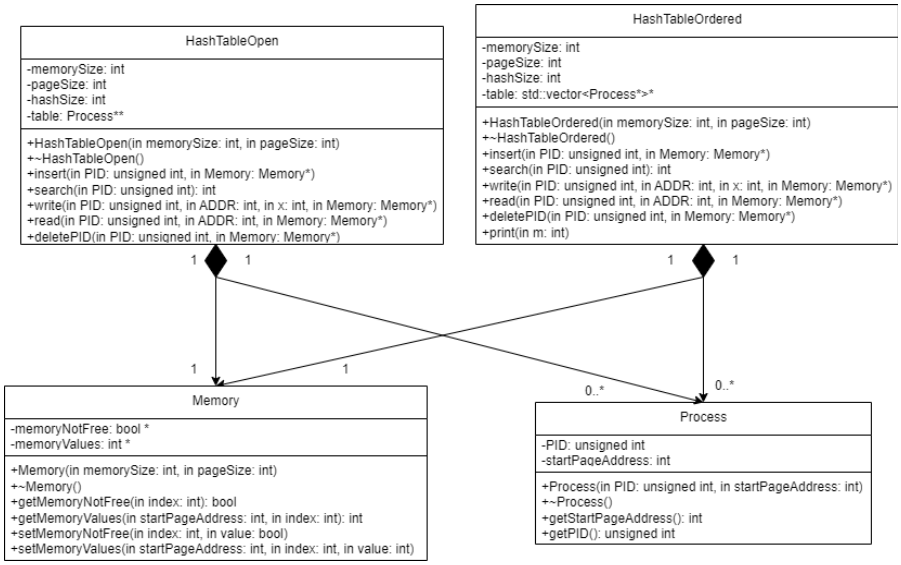
Process(unsigned int PID, int startPageAddress): Each Process consists of a PID and a startPageAddress. These variables are self explanatory and are passed as parameters to the constructor. Every process will always have these parameters, so I assign these values to their respective private member variables immediately.

~Process(): A default destructor is used for this class because each individual process is deleted through the destructor of HashTableOpen or HashTableOrdered.

int getStartPageAddress(): Returns an integer representing the start page address of the current Process.

unsigned int getPID(): Returns an unsigned integer representing the PID of the current Process.

UML Diagram



Asymptotic Upper Bound Runtime

HashTableOpen

Insert: First, I find the index of the hash table that the Process would go into by calculating h1(k), h2(k), and using these values to calculate h(k, i). There will always be a spot in the hash table to insert into due to the memorySize constraint. I then increment i in a loop until it reaches hashSize - 1, on each iteration checking if the table at index h(k, i) is nullptr, the Process' PID stored there is 0, or if the Process' PID stored there is the same as the one I'm inserting. Nullptr means that there was never a Process at that index, so that index would be the index to put the new Process and I break out of the loop. If the Process' PID stored there is 0, that means there used to be a Process there but it was deleted, so I keep track of this index as the index to insert. For future iterations, I stop checking if the Process' PID is 0 because I already know where the new Process will be inserted. If the Process' PID is the same as the one I'm inserting, I output "failure" and return to main. This takes O(1) time because we assume uniform hashing, meaning there are no collisions and the loop is guaranteed to only iterate once.

Search: First, I calculate h1(k), h2(k), and use these values to calculate h(k, i). I then increment i in a loop until it reaches hashSize - 1, on each iteration checking if the table at index h(k, i) is nullptr or the Process' PID stored there is the same as the one I'm inserting. Nullptr means that I reached the end of the chain and the PID was not found, so I return -1. If the Process' PID is the same as the one I'm searching, I return the index where it was found. If the loop finishes, I return -1 since that means I iterated through the entire table and did not find the PID. My main function outputs different strings depending on the return value, and the reason I don't output it inside of search() is because my write(), read(), and deletePID() functions use search(); as an example, I don't want the output of search() to appear when I call write(). This takes O(1) time because when assuming uniform hashing, we will either find or not find the PID on the first search and will not have to search further.

Write: First, I check if ADDR is outside of the virtual address space, in which case I output "failure" and return to main. Otherwise, I call search() with my PID, and if it returns -1 (meaning it did not find the PID), I output "failure" and return to main. Otherwise, I output "success", set my memoryValues array's value at the corresponding address to x, and return to main. This takes O(1) time because search() takes O(1) time and writing a value to memoryValues array is also O(1).

Read: First, I check if ADDR is outside of the virtual address space, in which case I output "failure" and return to main. Otherwise, I call search() with my PID, and if it returns -1 (meaning it did not find the PID), I output "failure" and return to main. Otherwise, I output the corresponding memory location as well as value stored there. This takes O(1) time because search() takes O(1) time and outputting a string also takes O(1) time.

Delete: First, I call search(), and if it returns -1 (meaning the PID was not found), I output "failure" and return to main. Otherwise, I output success, set my memoryNotFree array's corresponding index to false, delete the Process, and replace it with a new Process with a PID of 0. This acts as a flag for a deleted Process, so when inserting and searching, I know when my actual chain ends and won't stop too early/too late. This takes O(1) time because search() takes O(1) time, and all other operations are O(1) as well.

HashTableOrdered

Insert: First, I loop through the corresponding vector that my new Process should go into. During each iteration, I check if the PID of the vector's index is less than the new PID and if both PID's match. Since this vector should be sorted descending, once the PID of the vector's index is less than the new PID, it is no longer possible for there to be a duplicate further down the chain. If this is satisfied, I output "success", set my memoryNotFree array's corresponding index to true, use the vector insert function to insert the new process and return to main. If a matching PID is ever found, I output "failure" and return to main. This takes O(1) time because when assuming uniform hashing, we will either find or not find the PID on the first search and will not have to search further.

Search: Mostly the same as HashTableOpen, except I don't calculate h(k, i) and just use h1(k) and iterate through the vector at that index. Given uniform hashing, there will only be one Process in that vector, so it would take O(1) time.

Write: Mostly the same as HashTableOpen, except the index in the table is calculated differently ([PID % hashSize][index] rather than [index]). This doesn't change the runtime.

Read: Mostly the same as HashTableOpen, except the index in the table is calculated differently ([PID % hashSize][index] rather than [index]). This doesn't change the runtime.

Delete: Mostly the same as HashTableOpen, except the index in the table is calculated differently ([PID % hashSize][index] rather than [index]). Also, I simply use the vector erase() function rather than replacing it with a new Process with a PID of 0. This doesn't change the runtime because assuming uniform hashing, we will only have one element in each vector; erasing it would take O(1) time.