Project title: Persistent memory emulation to implement simplified NV-Heaps.

Members: Abhik Bose (Roll no - 184050002), Akash Trehan (Roll no - 150050031)

Problem description and scope: Using persistent memory to store user-defined object is beneficial for several applications like a database. Recent developments of faster non-volatile memory technology can possibly achieve comparable data rate and latency similar to volatile DRAM based primary memory. However, handling data inconsistency is more challenging with non-volatile memory. For example, consider the non-volatile memory has a pointer holding an address of a volatile object. During a power cycle, those pointers will become invalid and accessing them may lead to disastrous result. This context demands the development of a specialised data structure to store the user-defined objects in non-volatile memory.

NV-heap is a robust volatile heap like non-volatile abstract data structure and comes with a custom memory allocator, automatic garbage collection, concurrency control and safeguard against system failure etc. Our aim of this project is to lean and build a simplified version of NV-heap.

Sub-components of the project: We'll implement a simplified version of NV-heap as a C library for GNU/Linux based systems, similar to the original work. We've planned to follow the given modular approach.

1. **Ordinary non-volatile heap:** We're starting from an ordinary non-volatile heap data structure without any sophisticated feature for consistency control. Notable Linux system calls for this context are mmap() and msync().
2. **Modifying the memory allocator:** We'll proceed to modify the memory allocator for specialised features like automatic garbage collection, atomicity, durability, recovery etc.
3. **Controlling pointers:** NV to V and NV to another NV pointers need special handling in a non-volatile heap. We'll implement them as much time permits.
4. **Implementing Transactions and logging:** This is similar to transections control in the database. This define policy to decide when a transaction is permanent. A transection can be reverted back until permanent. Transactions control is one of the well-known approaches for failure recovery. We may implement a simplified version of Transaction manager as time permits.
5. **Thread Safety:** After the NV-Heap basic implementation we'll try to extend it to a thread-safe NV-Heap data structure.
6. **Performance analysis of NV-Heap implementation:** We'll analyze access times etc for all operations provided by the NV-Heap. We're yet to decide the performance analysis tools and procedure.

Intended deliverables:

1. A C/C++ library implementing NV-Heap functionality.
2. C/C++ programs demonstrating the use of the NV-Heap library. Exact details will be planned according to the progress of the library implementation.
3. Demonstration of the NV-Heap library and performance analysis using the programs.

Design questions: The design of NV-Heap is actually a demonstration and learning vehicle of a robust non-volatile heap data structure implementation.

1. **Choice of Heap:** Heap and trees are important data structures. This NV-Heap data structure implementation is quite similar to a tree data structure including several pointer operations, with several possibilities of catastrophic consistency violationn. So, implementation of this can possibly be extended to any other non-volatile user-defined abstract data class.
2. **Choice of C/C++:** Performace is critical as persistent storages are the bottleneck of most systems. So, we've chosen C/C++ as a fast and secure programing language.
3. **Physical Media:** To avoid complexity we'll use NVME flash-based SSD as persistent storage.
4. **NV-Heap feature selections:** Selecting the most important features of NV-Heap is quite tricky considering limited timespan. We considered the custom memory allocator, automatic garbage collection as the most important feature.

Implementation details: We're in a process of deciding all implementation details. Following is some of the implementation details.

1. **Accessing persistant storage:** we've used Linux mmap() system call to get persisteant storage mapped virtaul memory in user program's address space. We used msync() system call to immediately flash written data to disk.
2. **Atomicity and durability:** Fixed sized, non-volatile entry will be maintained as redo-log to provide atomicity.
3. **Concurrency:** Concurrency will be implemented by bookkeeping count variable and generational lock.
4. **Allocation and Deallocation:** Allocation and deallocation will be protected with proper lock and manipulation of pointers and count variable.

Evaluation plan --- correctness, performance, demonstration of features

1. **Correctness:** Correctness will be checked by several allocation and deallocation over multiple power cycle. Expected result may be calculated theoretically or from a volatile heap implementation. Thread safety, concurrency etc will also be checked for correctness.

2. **Performance:** We’ll measure the access time with variable stress for all implemented operations. The data structure access time to storage access time will signify the data structure’s performance. Exact benchmark tools are not yet decided.
3. **Demonstration of features:** We’ll demonstrate the implemented features of the C/C++ NV-Heap library using custom written user programs.