Memory Management

In this experiment, you will emulate a simple memory management system in the OS. Your memory manager will handle memory allocation, access, and release from user applications. It will also support page replacement policies so that applications have access to a much larger array list than the available memory. You will need to consider thread safety as your memory manager is shared among many threads.

In a nutshell, the memory manager emulation works like the following (see the next section for a detailed discussion of the sample code):

1. On initialization, the MemoryManager allocates a fixed number of PageFrames. The number will not change throughout the entire process. The MemoryManager will manage these pages, including tracking empty pages, and allocations, etc.

2. The MemoryManager allocates memory to applications through the creation of ArrayList objects. All subsequent access to the memory from the MemoryManager should go through the ArrayList interfaces. That is, you should not access the underlying memory arrays directly. Each ArrayList has an ID assigned by the MemoryManager. You can choose to use or not to use that ID, but you should not remove it.

3. The `mma` should support `PageIn`s and `PageOut`s once its PageFrames runs low. To do so, the MemoryManager can choose to track memory accesses. For emulation, you can track all memory accesses in software, but please keep the tracking overhead as low as you can.

4. The MemoryManager and all single accesses to the memory should be thread-safe.

Codebase Introduction

The test cases in `mma_test.cc` serve as documentation for the MemoryManager interface usage.

We encapsulate such a memory segment in the class of `ArrayList` (see `lib/array_list.h`). `ArrayList` provides basic `Read` and `Write` functions for workloads to access memory. Note that `ArrayList` only registers the memory segment in the `memory_manager` (`mma` for short) with a unique identification `array_id` instead of holding the memory directly. You can allocate ArrayLists using the `mma`.

We provide interface definition of `mma` in `lib/memory_manager.h`. See the file for places you can modify and places you cannot modify.

Specifically, you should implement the following functionalities in the `mma` :

- `PageFrame`: `mma` organizes its memory space as an array of pages. The page size is 4KB. In this experiment, we implement page data structure as `PageFrame`. `PageFrame`should support random access and serialization/deserialization (i.e., save the content of pages in disks using `WriteDisk()` method and recover a page from disk files using `ReadDisk()` method)

- `PageInfo`: `mma` records necessary information for each page (e.g., the current virtual page it holds on). You can put additional states of each page in this structure for other functionalities if you see necessary.

- `ReadPage/WritePage`: when applications read/write a new value to the ArrayList (using the ArrayList interface), the value should eventually find its correct memory location in the pre-allocated memory in MemoryManager. It is your call whether to do the address translation in the ArrayList object or in the `mma`. Note that ArrayList accesses 4 bytes of memory at a time in this experiment.

- `PageReplacement`: when the number of page frames allocated in ArrayLists exceeds the available number of page frames in the `mma`, `mma` should page out some pages to disk files. Implement your `mma`'s page replacement algorithms with `PageOut` method to store a page into a file and `PageIn` method to load a page from the file. You should use a reasonable algorithm to minimize the number of page-ins and outs. For simplicity, you can use `array_id` and `virtual_page_id` to identify disk files of each page.

- `Allocate/Release`: when `mma` allocates an `ArrayList`, it should assign a unique ID to the ArrayList, and allocate PageFrames to the ArrayList. Note that you are no need to allocate physical pages to some `ArrayList` as soon as it is created. When as soon as an `ArrayList` is destroyed, `mma` should reclaim the PageFrames (both in memory and on disk), synchronously. Note that a PageFrame is the smallest allocation unit in this experiment, so allocating slightly larger memory than required to applications is acceptable.

- If your replacement policy requires, you can track the memory accesses (read or write or both) in the `mma`.

`mma_test.cc` provides basic tests for memory manager validation and observation experiments, the first three tasks for single-thread scenarios, and the fourth task for multi-thread scenarios.

You can utilize basic tools in `utils.h` to write tests for your `mma` functions.

TODO

Q1

Implement your `mma` and `ArrayList` to support single-thread scenarios. The page replacement algorithm should be FIFO. Observe and record the time it costs to pass the first three tests.

Q2

Implement a clock algorithm (approximate LRU) for page replacement instead of your FIFO in Q1. Observe and record the time it costs to pass the three tests. Compare two parts of experiment results, analyze the difference.

Q3

Change the `mma` memory allocation from 1 to 10 and re-run test 2 for both algorithms. Observe and record the time it costs to pass the tests. Analyze the reason for the result variation and the differences between the two algorithms.

Q4

Implement your `mma` to support multi-thread scenarios while guaranteeing thread safety. The page replacement algorithm should be the clock algorithm. Vary the thread number from 10 to 20 to pass the 4th test. What can you observe? Try to analyze the results.

Grading

We will use extra workloads to test your `mma`, so your implementation should not rely on `mma_test.cc`. We grade with the following three requirements:

1. Correctness: pass all tests in `mma_test.cc` and extra tests;
2. Performance: we will consider the time it takes to pass the tests. For concurrent tests, it should be significantly faster than running all tasks sequentially - so you should not let `mma` process requests one by one.
3. Observation and analysis: submit a PDF to display your experiment results in Q1 - Q4 and try to explain the rationales behind the results.

Submit the diff file that can be correctly applied to `ec3155c3`.