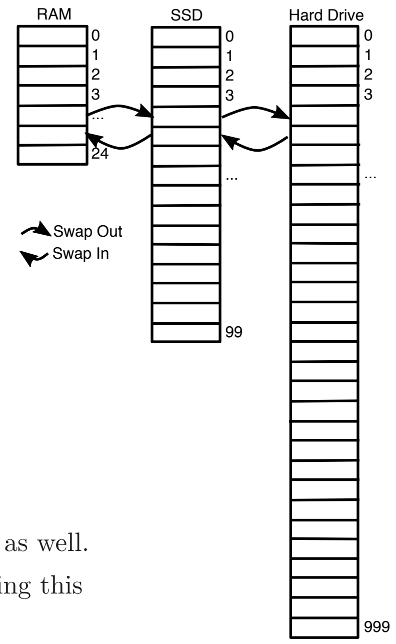
CS 3013 - Operating Systems

Project 4 (100 points)
Assigned: Tuesday, February 17, 2015
Checkpoint: Monday, February 23, 2015
Due: Friday, February 27, 2015

Project 4: Implementing Virtual Memory

Virtual memory is a powerful construction that allows programs to believe they have access to a larger amount of memory resources than is present in the physical RAM on a computer. In this project, we will emulate a virtual memory system for our users that uses RAM, SSD, and traditional spinning hard drives.



Storage Media

We will create three classes of storage: main memory (RAM), SSD, and magnetic hard disks. Each tier in the hierarchy will have different capacities and access times (or latencies). The RAM will be limited to 25 integers, the SSD will hold 100 integers, and the hard disk will have the capacity for 1000 integers. When a user accesses a memory region (henceforth, a "page") that is stored in RAM, the request will be serviced immediately. If the page is on SSD, the request will be delayed by 0.25 seconds using the `usleep()` function. If the page is on magnetic disk, the request will be delayed by 2.50 seconds (again using the `usleep()` function). The same delays occur when values are written to these media as well.

The system is limited to 1000 virtual memory pages. Requests exceeding this limit must be denied.

User API

While we could create our virtual memory system in the Linux kernel, we will perform our emulation in user space to make the project practical given the time constraints. However, in doing so, we will not be using standard memory libraries (e.g., `malloc` and `free`) and we will need to be more explicit about how the user will access memory. In particular, the user will make the use of the following functions:

- `vAddr allocateNewInt();` – This function reserves a new memory location, which is `sizeof(int)` in size. This memory block must be created in the emulated RAM, pushing other pages out of the emulated RAM into lower layers of the hierarchy, if needed. Returns `-1` if no memory is available.
- `int * accessIntPtr(vAddr address);` – This function obtains the indicated memory page from lower levels of the hierarchy, if needed, and returns an integer pointer to the location in emulated RAM. The page is locked in memory and is immediately considered "dirty." Returns NULL if the pointer cannot be provided (e.g., a page must be brought into RAM, but all of RAM is locked).
- `void unlockMemory(vAddr address);` – When the user is done using the indicated memory page for a while, they must unlock it, indicating that it can be swapped out to disk, if needed. Once a user calls `unlockMemory`, any previous pointers they had to the memory are considered invalid and must not be used.
- `void freeMemory(vAddr address);` – When the user is finally done with the memory page that has been allocated, the user can free it. This not only frees the page in memory, but it deletes any swapped out copies of the page as well.

The type `vAddr` is actually a typedef for an `signed short`. It is distinguished to make it clear that it represents a page number and not data.

As an example, consider the following function a user could write:

```
void memoryMaxer() {
   vAddr indexes[1000];
   for (int i = 0; i < 1000; ++i) {
      indexes[i] = allocateNewInt();
      int *value = accessIntPtr(indexes[i]);
      *value = (i * 3);
      unlockMemory(indexes[i]);
   }
   for (int i = 0; i < 1000; ++i) {
      freeMemory(indexes[i]);
   }
}
```

This function allows the user to create, access, update, and free memory. After the first 10 iterations of the loop, the memory system must swap old values to SSD. After the first 25 iterations, the SSD is full and SSD values must be swapped to the hard drive to make room for the memory to swap to the SSD. Accordingly, the first 10 iterations go quickly, the next 15 iterations are slower, and the final 75 iterations take FOREVER (`#define FOREVER "187.5 seconds"`).

Paging Infrastructure

To make all this happen, you must create the following memory components:

- **Page Fault Handler:** When the requested page is not in RAM, you must retrieve it from the appropriate backing store. If it is on the hard disk, you must put it onto SSD. You must then transfer it from the SSD into main memory. The page fault handler must evict pages to make room when swapping in. The page fault handler is responsible for inserting the appropriate delays based on the memory type.
- **Page Eviction Algorithms:** When a page fault occurs, but there is no place to store the faulted page, you must evict a page to make room. However, you cannot evict a locked page. But, which of the unlocked pages do you evict? You must create two different algorithms to evict pages, at least one of which must use some notion of history.
- **Page Table:** How can you tell if a page is locked or not? Is the page allocated? Is the page in RAM, on disk, or in memory? How can you tell when the page was last accessed? All of this information must be stored in the page table, which is an array of size 1000 of structs. Each struct contains information about the page.

Multiple users may wish to access memory at the same time. Select a synchronization primitive that will ensure mutual exclusion for the relevant memory regions. Importantly, note that you may not lock all of the memory in RAM, on SSD, or on disk while performing a page fault. Instead, you must lock the minimal entries possible while avoiding deadlock.

Stress Testing

Once you have implemented your virtual memory system, write a few different user functions, like the `memoryMaxer` function above, that make use of memory. Then, create a threaded program that uses these functions simultaneously. Your program and functions must thoroughly test each of the requirements of the system. If your stress tests pass, you should be assured that you have implemented the project correctly. You should also include some timing metrics, like those you created in Project 1, to confirm the delays across the different storage media are implemented correctly.

As part of the submission process, provide a text file `testing_methodology.txt`, that describes your testing functions and why you selected them. Indicate how you ensured your tests are thorough and why a user should have confidence in your system. If you have statistical results, those should be presented as well.

Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. However, as an example, code that is comparable to a basic page fault handler with a simple eviction algorithm will be considered sufficient for a checkpoint. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

Deliverables and Grading

When submitting your project, please include the following:

- Your source code files for the test functions and the virtual memory routines,
- a text file `testing_methology.txt` (as described previously),
- a Makefile that compiles your code
- Output from your tests.
- A document called README.txt explaining your project and anything that you feel the instructor should know when grading the project. Only plaintext write-ups are accepted.

Please compress all the files together as a single .zip archive for submission. As with all projects, please only standard zip files for compression; **.rar, .7z, and other custom file formats will not be accepted**.

The project programming is only a portion of the project. Students should use the following checklist in turning in their projects to avoid forgetting any deliverables:

1. Sign up for a project partner or have one assigned (URL: `https://cerebro.cs.wpi.edu/cs3013/request_teammate.php`),
2. Submit the project code and documentation via InstructAssist (URL: `https://cerebro.cs.wpi.edu/cs3013/files.php`),
3. Complete your Partner Evaluation (URL: `https://cerebro.cs.wpi.edu/cs3013/evals.php`), and
4. Schedule your Project Demonstration (URL: `https://cerebro.cs.wpi.edu/cs3013/demos.php`), which may be posted slightly after the submission deadline.

A grading rubric has been provided at the end of this specification to give you a guide for how the project will be graded. No points can be earned for a task that has a prerequisite unless that prerequisite is working well enough to support the dependent task. Students will receive a scanned markup of this rubric as part of their project grading feedback.

Groups **must** schedule an appointment to demonstrate their project to the teaching assistants. Groups that fail to demonstrate their project will not receive credit for the project. If a group member fails to attend his or her scheduled demonstration time slot, he or she will receive a 10 point reduction on his or her project grade.

During the demonstrations, the TAs will be evaluating the contributions of group members. We will use this evaluation, along with partner evaluations, to determine contributions. If contributions are not equal, under-contributing students may be penalized.