SFWRENG 3SH3: Operating Systems
Lab 4: Memory Mapped Files

1 Memory Mapped Files

A *memory-mapped file* is a segment of virtual memory which has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource. This resource is typically a file that is physically present on-disk, but can also be a device, shared memory object, or other resource that the operating system can reference through a file descriptor. Once present, this correlation between the file and the memory space permits applications to treat the mapped portion as if it were primary memory.

The memory mapping process is handled by the virtual memory manager, which is the same subsystem responsible for dealing with the page file. In UNIX-based operating systems, memory mapped files implement demand paging. Memory mapped files are loaded into memory one entire page at a time. The page size is selected by the operating system for maximum performance. Since page file management is one of the most critical elements of a virtual memory system, loading page sized sections of a file into physical memory is typically a very highly optimized system function.

Here's one resource for the related UNIX systems calls:
https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch04s03.html

2 Overall Objective

You are to write two C or C++ programs that manage a set of shared resources using UNIX memory mapped files. The assignment will deliver 3 files:

- The file `res.txt` contains the set of available resources. It can simply be of the form:

0	4
1	3
2	7

meaning that there are 4 units of resource type 0, 3 units of resource type 1, and 7 units of resource type 2. You may assume that the number of units of each resource is strictly less than 10.

- The file `alloc.cpp` implements a resource *allocator* program. The allocator opens `res.txt` and maps it to a memory region using the system call `mmap()`. You should make sure that the size of the region is not smaller than the file size. To this end, you can use the `fstat()` system call to get the file size. In a loop, it keeps asking how many units of a resource type is needed. Once entered, it subtracts the units from that resource type (if available) and then invokes system call `msync()` to synchronize the content of the mapped file with the physical file.

- The file `prov-rep.cpp` creates two processes:

  - The *parent* process opens the file `res.txt` and maps it to a memory region (before forking the child process). The parent process also acts as a *provider* of resources. In a loop, it keeps asking whether new resources need to be added. If yes, it receives from the input the resource type and the number of units and adds them to the memory region. Once added, using the system call `msync()`, it synchronize the content of the memory region with the physical file.
  - The *child* process is a *reporter* and reports three things every 10 seconds:
    * The page size of the system using the system call `getpagesize()`.
    * The current state of resources.
    * The current status of pages in the memory region using the system call `mincore()` (i.e., whether pages of the calling process's virtual memory are resident in core (RAM), and so will not cause a disk access (page fault) if referenced.)

Finally, notice that the memory mapped file itself is a shared resource and, hence, access to it must be mutually exclusive. Thus, you will use UNIX semaphores to enforce mutual exclusion. Remember that UNIX sempahores are different from POSIX sempahores that you used in Lab 3. Here, you will use system calls `semget()`, `semctl()`, and `semop()`.