COMS 352: Intro to Operating Systems
Project 2: Memory Mapped Files

100 Points

Due : Friday, April 19, 2019, 11:59 pm

- 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.

- Overall Objective

You are to write **two** C or C++ programs that manage a set of shared resources using UNIX memory mapped files.

<u>You will submit 3 files for this project:</u>

- 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 a **Child** process (so there are 2 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 user input, the resource type and the number of units, and
  - adds them to the memory region.
- Once added, using system call **msync()**, it synchronizes the content of the memory region with the physical file.

    - The ***child*** process is a ***reporter*** and reports **<u>three things</u>** 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, **<u>access to it must be mutually exclusive</u>**.

Thus, you will use **UNIX semaphores** to enforce mutual exclusion.

Remember that UNIX semaphores are **different** from POSIX semaphores that you used in Project 1.

Here, you will use system calls **semget(), semctl()**, and **semop().**

- Guidelines:
- You should use C/C++ to develop the code.
- **You should work on this project individually.**
- You need to turn in electronically by submitting a zip file named: Firstname_Lastname_Project2.zip.
- Source code must include proper documentation to receive full credit (you will lose 10% of your score, if the code is not well documented as follows).
- File Comments: Every .h and .c file should have a high-level comment at the top describing the file's contents and should include your name(s) and the date.
- Function Comments: Every function (in both the .h and the .c files) should have a comment describing:
    - what function does;
    - what its parameter values are
    - what values it returns (if a function returns one type of value usually, and another value to indicate an error, your comment should describe both types of return values).
    - o In-line Comments: Any complicated, tricky code sequences in the function body should contain in-line comments describing what it does.
- All projects require the use of a **make file** or a certain script file (accompanying with a readme file to specify how to use the script/make file to compile), such that the grader will be able to compile/build your executable by simply typing "make" or some simple command that you specify in your readme file.
- **Source code must compile and run correctly on the department machine "pyrite", which will be used by the TA for grading. If your program compiles, but does not run correctly on pyrite, you will lose 15% of your score. If your program doesn't compile at all on pyrite, you will lose 50% of your score (only for this issue/error, points will be deducted separately for other errors, if any).**
- You are responsible for thoroughly testing and debugging your code. The TA may try to break your code by subjecting it to bizarre test cases.
- You can have multiple submissions, but the TA will grade only the last one.