CSE 551
Problem Set #1

Due: 4:30pm, Thursday, October 21, 2010

1. Write pseudo-code in Java, Python, C++ or C, for a (very) simple UNIX shell capable of executing a command through a pipe. The focus of this question is on the concurrency, so you do not need to support command line arguments, set environment variables or the like. Feel free to assume any convenient string parsing package. For example, if the user types *ls | wc*, your program should fork off the two programs, which together will calculate the # of files in the directory. For this, you will need some (but not necessarily all) of the following UNIX system calls: *fork, execve, open, close, pipe, dup, and wait*. It is important to note that these system calls have subtly different semantics than Mesa (particularly *fork* and *wait*), so you'll need to read the man pages and/or the UNIX paper carefully. *pipe(int fdes[2])* returns a pipe such that bytes written on *fdes[1]* can be read from *fdes[0]*. One tricky step concerns how to replace standard in and out (file descriptors 0 and 1); this is the role of *dup*. Please note: The code does not need to compile or run; we are interested in the algorithm, not the syntax. However, it is cool to see it run, as it shows the power the UNIX model gives the developer. Please add enough comments so that the TA can understand your intent.

2. Write pseudo-code for a highly concurrent, multithreaded file buffer cache, using Mesa-style locks and condition variables, and following best practices in terms of ensuring correctness. A buffer cache stores recently used (that is, likely to be used soon) disk blocks in memory for improved latency and throughput. These blocks can be file data blocks, indirect blocks, inodes, and directories – that is, any persistent storage in the file system. Assume that the file system is to run on a system with dozens of CPUs, and so must be able to do many file operations in parallel.

   You are to implement two routines: *bool readblock(char *x, int blocknum)* and *bool writeblock(char *x, int blocknum)*. For simplicity, you may assume that calls to these routines are in terms of complete, block-aligned block reads and writes. *readblock* reads a block of data into the buffer passed as input; *writeblock* (eventually) writes the data in the buffer to the disk (on flush). The return value is true on success; false on failure. Blocks can be in cache or on disk, and cache misses may cause blocks to be evicted from the cache, and potentially to be written back to disk. Multiple threads can call *readblock* and *writeblock* concurrently, and to the maximum degree possible, those operations should be allowed to complete in parallel. You should assume the disk driver has been implemented; it provides the same interface as the file buffer cache: *dblockread(char *x, int blocknum)* and *dblockwrite(char *x, int blocknum)*. The disk driver routines are synchronous – the calling thread blocks until the disk operation completes – and re-entrant – while one thread is blocked, other threads can call into the driver to queue requests. Likewise, you should assume threads, locks and condition variables are provided by the language runtime system or standard library.