CSE 451
Problem Set #2

Due: 12:30pm, Tuesday, October 23, 2012

For both parts of this assignment, use Mesa-style locks and condition variables, and follow best practices in terms of ensuring correctness. List any constraints that hold for your data structures when no locks are held.

We have provided a test harness for each problem based on sthreads (described in the book); sthreads are very similar to Pintos threads, but they run at user-level so you can debug and test your code on any linux machine. The test routines, and stub implementations of the assignments, are in the files nettest.c (part 1) and cachetest.c (part 2).

1. Implement a multi-threaded bounded network queue that stores *k* packets and provides fair allocation among connections that share the queue. Threads put packets into the queue (for connection c) with *void putpkt(c, pkt)*; the network driver pulls packets off the queue with *pkt = getpkt()*. By fair allocation, we mean that getpkt() should return packets in round robin order among connections.

Our test harness creates 10 connections, but your implementation should not assume it knows how many connections there will be. Your solution should work when the buffer size is smaller than the number of connections.

2. Implement a highly concurrent, multi-threaded file buffer cache. A buffer cache stores recently used (that is, likely to be used soon) disk blocks in memory for improved latency and throughput. Disk blocks have unique numbers and are fixed size. The cache implements two routines *readblock(char *x, int blocknum)* and *writeblock(char *x, int blocknum)* that read/write complete, block-aligned blocks. *readblock* reads a block of data into x; *writeblock* (eventually) writes the data in x to disk.

On a read, if the requested data is in the cache, the buffer will return it. Otherwise, the buffer must fetch the data from disk, making room in the cache by evicting a block as necessary. If the evicted block is modified, the cache must first write the modified data back to disk. On a write, if the block is not already in the buffer, it must make room for the new block. Modified data is stored in the cache and written back later to disk when the block is evicted.

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.