CMSC 23000 Assignment 2: Design and Test Document

Rachel Hwang

January 28, 2014

1 Design

1.1 Serial Queue

My design for this version of the program will be very similar to the serial implementation, with the obvious additions of *numSources* queues, and enqueue and dequeue modules. The specific functions and data structures are as follows:

```
/* DATA STRUCTURES */

// Store the queues
Seriallist_t **queues = create_queues(numSources);

// Store number of packets left to enqueue, per queue
int *count = (int *)malloc(sizeof(int)*numSources);
// Initialize countdown for all queues at 0
for (i = 0; i < numSources; i++) {
    count[i] = 0;
}

// Fingerprint destination
long *fingerprint = (long *)malloc(sizeof(long)*numSources);
// Initialize all fingerprints to 0
for (i = 0; i < numSources; i++) {
    fingerprint[i] = 0;
}

// Number of sources finished
int done = 0;

/* FUNCTIONS */

// Run the serial-queue procedure (serves as Dispatcher)
void serial_queue_firewall(int numPackets,
                    int numSources,
                    long mean,
                    int uniformFlag,
                    short experimentNumber);

// Create queues
SerialList_t **create_queues(int numSources);

// Enqueue packets while possible to one queue (called for each queue by Dispatcher)
int enqueue(int *count, Seriallist_t *q, int numPackets, int D, PacketSource_t
    *packetSource);

// Proccesses packets (serves as Worker)
void dequeue(Seriallist_t *q, int *fingerprint);
```

In short, this program will use an array of $Seriallist\_t*$ to store the queues, an int array to keep track of how many packets have been enqueued for each queue, a long array $fingerprint$ which like the example code serves as a place-holder destination, and a single int *done* which counts how many sources are finished.

In the abstract, the dispatch attempts to enqueue as many packets as possible for each source/queue pair. When it has tried every queue, the program next dequeues all packets in all queues. This process is looped until each queue has recieved the right number of packets.

$serial\_queue\_firewall()$ will serve as the dispatch, creating/initializing all necessary data structures, then in a while loop calls *enqueue*(), continuously attempting to enqueue packets in each queue until either the queue is full or the source has been finished. When all the packets from one source have been enqueued, *enqueue* returns 1, otherwise 0. Those results are stored in the int *done*. Each time a packet is successfully enqueued, the count for that queue is incremented. The program will then call *dequeue*(), essentially the worker thread, on each queue, which processes packets from a single queue until it is empty. This process repeats (while loop invariant: $(done < numSources)$?). When that loop finishes, all packets must be processed.

1.2 Parallel

The design for this implementation will be very similar to the Serial Queue version, using all the same data structures with the addition of an array of POSIX threads and a data structure to hold arguments passed to them.

```
/* DATA STRUCTURES (include those from Serial Queue) */

// Array for threads
thread_t *threads;

// Data type for threads
typedef struct thread_t {
  pthread_t p;
  int *fingerprint;
  int *count;
  Seriallist_t *queue;
  int id;
} thread_t;

/* FUNCTIONS */

// Run the parallel firewall procedure (serves as Dispatcher)
void parallel_firewall(int numPackets,
                       int numSources,
                       long mean,
                       int uniformFlag,
                       short experimentNumber);

// Create queues
SerialList_t **create_queues(int numSources);

// Enqueue packets while possible to one queue (called for each queue by Dispatcher)
int enqueue(int *count, Seriallist_t *q, int numPackets, int D, PacketSource_t
    *packetSource);

// Proccesses packets (Worker thread function)
void *dequeue(void *args);

// Create a new thread
int create_thread(Seriallist_t *queue,
           int *fingerprint,
```

```
            int *count);
```

The dispatcher *parallel_firewall*() operates precisely the same as the Serial Queue version except that it spawns threads before entering the enqueue process (a loop calling *enqueue*()), each of which are running *dequeue*(). In this version, the dequeue thread function loops continuously until its count reaches *numPackets*. Likewise, the enqueue loop executes until all threads are done, meaning that all packets have been enqueued. After exiting that loop, *parallel_firewall*() joins with all threads, at which point all packets have been processed.

2 Testing

For this project, I will employ unit testing. Each function in both my Serial Queue and Parallel code will be tested individually.

2.1 Serial Queue

Dispatcher and Worker functionality have been split into the enqueue and dequeue functions. Each should perform their objectives individually, then they can be tested in tandem. If both enqueue and dequeue perform as expected in both the standard and 'end condition' cases, I can be quite confident that my queues all behave as expected. To ensure that load is distributed properly, the *count* for each queue should be the same upon completion. To ensure correctness, results are compared to results from the given serial program.

enqueue():

- Attempt to enqueue 1 packet (numPackets = 1), assert that resulting queue state is as expected and return value is 1. Do the same for a larger value.
- Call on a full queue. Assert that queue is unchanged, length $< D$ and that the return value is 0.

dequeue():

- Attempt to dequeue 1 packet (q->size = 1), assert that resulting queue is empty and that fingerprint value has been updated. Do the same for a larger value.
- Attempt to dequeue an empty list. Assert that nothing happens.
- Enqueue on a single queue, then dequeue on it. Assert that the resulting list is empty.

create_queues():

- Create x queues. Assert that the correct number of queues are returned.

serial_queue_firewall():

- Run with a single queue. Assert that queue is empty after completion and that $fingerprint$ value has been changed. Try with larger *numPackets* values.
- Assert that all *count*[i] values are equal to *numPackets*+1 upon completion to assure proper load distribution and coverage of all packets.
- compare $fingerprint$ results from this function and the original serial code. Assert that they are the same.

2.2 Parallel

Tests for Serial Queue, unless overwritten here, also apply. *dequeue*():

- Perform the same tests as in Serial Queue, but passing in arguments in the defined thread struct.

create_thread():

- Create a new thread and assert that its values (in thread struct) are filled as expected.
- Create multiple threads and assert that their values are correct and that their queues are empty upon completion.

parallel_firewall():

- Run with a single thread. Assert that queue is empty after completion and that $fingerprint$ value has been changed. Try with larger *numPackets* values.
- Assert that all *count*[i] values are equal to *numPackets*+1 upon completion to assure proper load distribution and coverage of all packets.
- compare $fingerprint$ results from this function and the original serial code. Assert that they are the same.

3 Experiments

1. Parallel Overhead
   Since the Serial Queue adds significant overhead without the benenfit of any parallelization, I expect the Queue version to perform significantly worse than the original serial. Every single packet must now be enqueued and dequeued. In order to stay synchronized, the dispatch module must also continuously check *count* and *done* values. All of these costs, apart from the allocation and intialization of the necessary data structures, grow proportional to the number of packets that must be processed overall. I therefore hypothesize that the speedup will remain relatively fixed (but a slight decrease) as n increases. On the otherhand, as W increases, the ammount of time that the program spends on queue-related overhead becomes a smaller and smaller percentage of total time. I hypothesize that speedup will improve as W increases. My best estimate is serial/serial-queue ≈ 0.7 when $W = 25$, and for any values of n.
2. Dispatcher Rate
   This experiment directly measures the benefit of increased parallelization. Since increasing the number of workers does not significantly increase the ammount of overhead that the dispatcher does (for instance, I expect about the same number of calls to the *enqueue*() to be made at different W values), I hypothesize that as W increases, the speedup will increase sharply before slowing down at some threshold point, creating a speedup graph like a logarithmic function. This point will depend on what percentage of the total times work runs in parallel. The greater the proportion, the longer and steeper the increase before the threshold point. I optimistically estimate (having no idea how long the actual checksum operations take) that little, $< 50\%$ of the total time is spent on parallelizable work, especially with a minimized W value, and so that threshold value, where speedup improvements decrease or halt, will come around $n = 8$.
3. Speedup with Uniform Load
   As discussed above, by Amdahl's law, greater proportions of parallelizable work give way to greater speedups. I hypothesize therefore that at greater W values, the speedups increase with n more sharply. However, I still believe that n increases will improve speedup only until a certain threshold point, althouh that point is in part controlled by the W value.

4. Speedup with Exponentially Distributed Load
   Because of the variable single packet load, work will be less evenly distributed, and hence average execution time will increase since some queues must recieve a disproportionate ammount of work. I thus hypothesize that the speedup graph will be shaped similarly to the that of the unifrom load experiment, but will be strictly less overall. I cannot predict whether the threshold point (n value at which speedups cease to increase) will come earlier, although I think that it may be likely given that load increases exponentially.