CMSC 23000 Project 3: Design and Test Document

Rachel Hwang

February 11, 2014

1 Design

My implementations will draw from the following modules:

- **locks.c/locks.h**
  A library containing all lock functions
- **serial_time_counter.c**
  An application taking a single time argument as run duration, spawning a single thread to increment a counter.
- **parallel_time_counter.c**
  The parallel version, spawning n threads which each increment under the supervison of a given lock type.
- **serial_work_counter.c**
  An application taking a single argument as a counter target value (BIG), spawning a single thread to increment up to that value.
- **parallel_work_counter.c**
  The parallel version, spawning n threads, each of which increment the counter (BIG)/n times.
- **serial_packets.c**
  Runs the firewall checksum application from project 2.
- **parallel_packets.c**
  The parallel version, again similar to project 2, but taking lock and queue picking stategy arguments.

Programs will also use a number of the provided utility modules, such as the stopwatch and packet gen.

1.1 Locks

Each of the five assigned locks is implemented in *locks.c*, each with a lock and unlock function. My implementations follow the textbook as exactly as possible, as illustrated below.

```
/* From "locks.h" */
typedef struct alock_t{
  volatile int *array;
  volatile int *tail;
  int max;
} alock_t;
```

```
typedef struct node_t{
  volatile int locked;
} node_t;


typedef struct clh_t{
  volatile node_t *me;
  volatile node_t *pred;
  volatile node_t **tail;
} clh_t;


/* TAS lock functions */
void tas_lock(volatile int *state)
{
  while (__sync_lock_test_and_set(state, 1)) {}
}

void tas_unlock(volatile int *state)
{
  __sync_lock_test_and_set(state, 0);
}


/* Exponential Backoff Lock functions */
void backoff_lock(volatile int *state, volatile int *backoff)
{
  int time;

  while(1) {
    while(*state) {};
    if (!__sync_lock_test_and_set(state, 1)) {
      return;
    }
    else {
      time = rand()*fmax(pow(2, (double)*backoff), MAX_DELAY);
      (*backoff)++;
      nanosleep((struct timespec[]){{0, 1000000*time}}, NULL);
    }
  }
}

void backoff_unlock(volatile int *state)
{
  __sync_lock_test_and_set(state, 0);
}


/* Mutex lock functions */
void mutex_lock(pthread_mutex_t *m)
{
  pthread_mutex_lock(m);
void mutex_unlock(pthread_mutex_t *m)
{
  pthread_mutex_unlock(m);
}


/* Anderson lock functions */
void anders_lock(volatile alock_t *a, volatile int *idx)
{
  *idx = __sync_fetch_and_add(a->tail, 4) % a->max;
  while(!(a->array)[*idx]) {}
```

```
}

void anders_unlock(volatile alock_t *a, volatile int *idx)
{
  (a->array)[*idx] = 0;
  (a->array)[(*idx + 4) % a->max] = 1;
}


/* CLH lock functions */
node_t *new_clh_node()
{
  return (node_t *)malloc(sizeof(node_t));
}

void clh_lock(volatile clh_t *lock)
{
  volatile node_t *curr = lock->me;
  curr->locked = 1;
  lock->pred = __sync_lock_test_and_set((lock->tail), curr);
  while ((lock->pred)->locked) {}
}

void clh_unlock(volatile clh_t *lock)
{
  volatile node_t *curr = lock->me;
  curr->locked = 0;
  lock->me = lock->pred;
}
```

Each of the lock functions takes arguments specific to it's type, eg. backoff takes a int storing current backoff value, which are passed in from the calling worker thread function.

1.2 Counter Programs

1.2.1 Serial

The serial counter programs consist of only a dispatch function and an increment function. The dispatch function is responsible for handling the program argument (either time or counter value), then spawning a single worker thread. In the case of $serial_{w}ork_{c}ounter$, passing the work arguments along to the thread. The thread runs a simple increment function, which loops, incrementing a counter at each iteration so long as the loop invariant is satisfied. The time an work versions of the thread function are as below.

```
/* From serial_time_counter.c */
int counter;
volatile int go;

/* Worker thread function: increments a counter */
void *increment()
{
  while(go) {
    counter++;
  }
  pthread_exit(NULL);
}
```

The time_counter programs operate using a simple flag. Each thread function uses a loop which checks *go* at every iteration. After spawning worker thread(s), the dispatch thread sleeps for the specified

number of time, then sets a global variable called *go* to 0, killing all threads.

```
/* From serial_work_counter.c */
int counter;

/* Worker thread function: increments a counter */
void *increment(void *arg)
{
  int n = *(int *)arg;

  while(counter != n) {
    counter++;
  }

  pthread_exit(NULL);
}
```

In the work_counter program, the worker is instead passed an argument which specifies the number of times a thread should increment. Once the value is reached, the thread exits.

1.2.2 Parallel

The parallel counter programs are similar to their serial counterparts with the addition of a spawn function and a different worker function for each lock type. The dispatch function is responsible for intializing all thread arguments and passing them to the spawn function, which spawns n worker threads each running the appropriate increment function specified by the lock type. This is selected using a switch statement. Each increment function is essentially the same, differing only in that they call their specific lock with the correct arguments. Shown below is the tas thread function for the time program.

```
/* TSA Worker thread function  */
void *tas(void *arg) {

  thr_data_t *data = (thr_data_t *)arg;
  volatile int *counter = data->counter;

  while(go) {
    tas_lock(data->state);
    (*counter)++;
    tas_unlock(data->state);
    (data->my_count)++;
  }

  pthread_exit(NULL);
}
```

Each worker thread is passed a thread data struct containing all the the arguments necessary for any lock type. These arguments are intialized by the dispatch function and the struct is filled by the spawn function.

```
typedef struct thr_data_t{
  volatile int *state;
  volatile int *backoff;
  volatile int *counter;
  pthread_mutex_t *mutex;
  volatile alock_t *alock;
  volatile node_t **clh_tail;
  volatile int my_count;
} thr_data_t;
```

1.3 Packets

The serial implementation is virtually unchanged from project 2, except that this program runs for a certain duration, rather than until a set number of packets have been reached. As in the time counter programs, the threads a killed using a global *go* flag.

The parallel implementation is considerably more complex. Unlike in the previous assignment, the queue that each thread dequeues from is no longer predetermined. As such, each queue will have a lock on it that threads will attempt to acquire. As in the counter programs, the program takes a queue-selection argument which is passed to the spawn threads function. Based on that strategy, the spawn function then creates n threads, each of which are passed a thread data struct containing a function pointer to the specified lock type functions. The worker thread function then selects a queue using the specified strategy, and calls the dequeue function on it, locking and unlocking using the given lock function pointers. The worker thread continues until the *go* flag is set to 0.

The try lock functions have yet to be implemented, but are essentially very slight modifications of each lock function, simply returning the lock state rather than spinning.

1.3.1 Awesome?

Although my locks are implemented, I have not yet completed the packets programs, and thus have not yet devised an improved queue selection strategy. By comparing the performance of the given queue selection strategies, I will be able to determine what the largest sources of contention are, and design a strategy accordingly, attempting to circumvent those points of contention. The two major factors in queue selection strategy performance will be eveness of packet distribution, and overhead. In each run, I can observe the number of packets that each queue has processed and see which strategy has the most even distribution. A comparision of the runtimes of LockFree with HomeQueue with each lock type will measure overhead. Creating an optimal strategy will involve balancing these two factors based on the perfomance of the given strategies, borrowing from the strategy which distributes work most evenly, but trying to reduce overhead.

2 Testing

Conveniently, tests for the locks are pretty much implemented in the counter programs. Lock testing will consist of calling each lock type which a slightly modified version of the parallel_work program with a range of arguments. Each lock type will be called with $n = \{1, 2, 4, 8, 16\}$. If the counter values (both the global and the counters private to each thread) after each run are as they should be, we can be reasonably confident that locks, as well as our counter programs are working correctly.

As for the packets program, each queue-strategy function is somewhat difficult to test individually. The best method of testing would be, very similar to tests for the counter programs, to keep a counter for each queue which is incremented whenever the queue is dequeued from. Test functions will check that after each run, the counter values are as expected.

Since each of the component functions are tricky to test in isolation, we can take advantage of the fact that the counter programs are essentially integration tests for themselves and simply check counter values over multiple runs. In this case, correct values over many runs and input combinations are a reasonable guarentee of program correctness; the output will be incorrect if any components do not

function as expected. A similar strategy may be used to check the packets program, modifying the thread functions slightly.

3 Experiments

3.1 Counters

1. **Idle Lock Overhead 1**
   Hypothesis : The time counter programs will be assessed by the final value of the counter, hence best performance in this case will mean a larger resulting couter value. The less lock overhead, the more of the alloted time the program spends on incrementing. Since the serial verision is lock-free, the parallel implementation will always have a speedup of less than one. As for relative lock perfomance:
   - TAS will perform the best, since there will never be any contention for the lock in this experiment and it has the least ammount of over head. It will do strictly better than Backoff, since Backoff does the same work, but with the added delay of a small backoff and calculating backoff time.
   - Mutex performance is a mystery, as I am unfamiliar with its implementation. Based on what I have observed, it tends to be slower than the TAS locks.
   - ALOCK will perform worse than either of the TAS locks. This is due to the fact that lock state is not simply stored in one location, but in an array. I suspect that the lookup time will be more costly than in the TAS locks.
   - CLH Lock will perform even worse than ALOCK. It does essentially the same work as ALOCK, but with the added expense of updating list tail and queue pred at each lock/unlock. The cost of managing the linked list will create noticable overhead.
2. **Idle Lock Overhead 2**
   Hypothesis : Results for this experiments should be very similar to the previous experiment, but with the role of time and counter value reversed. Since this is just a different method of measuring the same overhead, I expect all the lock performance relationships to be the same as describes above, although best performance in this case will mean lowest time.
3. **Lock Scaling 1**
   Hypothesis : This experiment in essence measures how often threads compete for the same lock and the delay between successful lock grabs. By optimizing backoff time, we are reducing the probability that a thread will try to grab a locked lock. I expect lock realtive performance to be as follows:
   - TAS lock performance will be the worst. There is no management whatsoever in that all threads will constantly be trying to acquire the lock. Cache Coherance Traffic will be an issue.
   - Backoff will outperform TAS, as the small random backoff time will increase the likelihood that a lock will be free when a thread tries to acquire it. This reduces the number of fruitless lock checks that each thread performs. Less values will be trashed in and out of the cache, improving on the cache coherance issue.
   - Again, Mutex is something of a black box. Mutex will linkely do worse than either ALOCK or CLH because it, like the TAS lock, does not attempt to order requests in any way.
   - ALOCK will do better than any of the previous locks since each thread will be checking a location specific to them rather than a central location. This eliminates cache coherancy problems with a comparatively small increase in overhead and increased space requirements. Unlike Backoff, it also minimizes the delay between successful lock grabs.
   - CLH's main improvement on ALOCK is the reduced space requirements. It should therefore perform very similarly to ALOCK, perhaps slightly worse due to the overhead of managing the list.

4. **Lock Scaling 2**
   As with experiments 1 and 2, experiments 3 and 4 measure the same aspects of performance using only slightly different metrics. Thus, the lock performance relationships should be the same as in the previous experiment.

5. **Fairness**
   Hypothesis : Lock fairness will be as follows, using standard deviation as a measure of fairness (small deviation = fair)
   - TAS will be very unfair, as it offers no guarentees whatsoever about ordering. Each thread's number of contributions will appear random.
   - Backoff will be better than TAS, but still unfair. The increasing backoff time makes it more likely that the first thread to try a lock will be the first thread to acquire it, but this is not guarenteed. - Mutex also makes no guarentee of order, I believe, so it may be comparable to TAS in fairness.
   - ALOCK & CLH locks will both be very fair, since they impose a first come, first serve ordering for lock acquisition. Since a thread cannot be in queue more than once, each worker should acquire the lock about the same number of times.

3.2 Packets

1. **Idle Lock Overhead**
   Hypothesis : HomeQueue will perform worse than LockFree since the programs will be identical save for lock overhead. The overhead observed should be virtually identical to that observed in the Lock Overhead experiments run previously; the ordering of lock overhead will be the same.

2. **Speedup with Uniform Load**
   Hypothesis : There are a large number of argument combinations here, so I will discuss queue selection strategies then make some general predictions.
   - LockFree: Because the packets will be uniform, I expect the lock free version to outperform the lock versions since the load inbalance will be small. The load balancing benefits of the other strategies will not compensate for the added overhead of using a lock.
   - RandomQueue: I expect will perform better than LastQueue for small mean work per packet, since contention for locks will be frequent if each thread attempts to acquire locks in a set pattern. TAS may outperform Backoff, since the queue selected is already random. Inserting a delay will provide no benefits.
   - LastQueue will significantly reduce the number of fruitless lock grabs, as it will prevent a thread from trying the same lock consecutively. Performance will increase with mean packet work, since larger W values means that a lock is more likely to be held for a longer period and thus consecutive checks are unlikely to help. This stratgey will render Backoff lock unhelpful. It's difficult to predict precisely how locks with interact with each of these queue strategies, except that locks which impose a delay before a thread tries a lock again maybe worse in the randoma nd last strategies, since the worker tries a different lock after failing to secure a first.

   As in project two, I think that performance will peak at $n = 8$ and $W = 8000$.

3. **Speedup with Exponential Load**
   Hypthesis Results should be similar to the previous experiment, except that load balancing is far more important in this scheme, so locked strategies will outperform the unlocked. In the exponetial scheme, the load will not be balanced by default. The gap between the locked and unlocked strategies should increases as W increases, since load imbalance with large mean load will lead to an even more unbalanced load.

4. **Speedup with Awesome**
   Hypothesis : I cannot speculate about this, as it will depend entirely on my design.