Design Document

Alex Miller

May 18, 2021

Modules, data structures, interfaces, invariants:

Our code base will consist of the following modules:

- The highest ranked modules containing C code will be two files – one called serial.c, the other parallel.c. These modules will, respectively, implement serial and parallel queue performance tests.

  serial.c

  - This file will implement the functionality of serial packet and checksum generator performance test, configurable to the following variables:
    - M - an integer representing the time in milliseconds that the experiment should run.
    - n - an integer representing the total number of sources
    - W - an integer representing the expected amount of work per packet
    - U - an char representing a flag specifying what sort of packet distribution to utilize in our tests. t indicates the use of Uniformly Distributed Packets , while f Exponentially Distributed Packets.
    - s - a non-negative integer, for seeding the packet generator. In practice this will correspond to trial numbers when testing is run.
  - Using user input, this file will create the specified packet generator. Call this generator *packet_gen*. It has n sources, a median work value of W, and utilizes the seed s.
  - It will also bind the specified method of extracting packets from *packet_gen* to a pointer desginated *packet_method* – the options for packet generators are Uniform and Exponential.
  - It will then pass *packet_gen*, *packet_method*, as well as any necessary variables to chksum_serial(); this method generates checksums from *packet_gen* from n sources using a average packet size of W and *packet_method* for M milliseconds. This method returns an integer, T – it first describes the number of packets that get fully processed in the allotted time. I will discuss the specifics of this method's implementation below.
  - serial.c will finally output T to the terminal, before cleaning up any leftover resources and exiting.

  parallel.c

  - This file will implement the functionality of parallel packet and checksum generator performance test, configurable to the following variables:
    - Like serial.c, this file is configurable to M, n, W, U, and s.
    - D - an integer representing the depths of the queues to be used in this test
    - L - a char representing the lock type to be used in our load balancing implementations. Setting this character to t, p, a, and m will direct worker threads to use either our Test and Set, the pthread Mutex, our Anderson's Array, or our MCS lock, respectively, in order to dequeue. The specifics of these lock implementations are discussed below.
    - S - a char representing a load balancing strategy. Setting this character to L, H, and A will direct our performance test to utilize either a LockFree, HomeQueue, or Awesome load balancing strategy, respectively. The specifics of these different strategies are discussed below.

- Using user input, this file will create the specified packet generator. Call this generator *packet_gen*. It has n sources, a median work value of W, and utilizes the seed s.
- It will also bind the specified method of extracting packets from *packet_gen* to a pointer desginated *packet_method*.
- Then, parallel.c will initialize a *lock_t* instance called *lock*. The *lock_t* struct describes an interface for accessing our locking algorithm implementations. This instance will be set to utilize the lock type specified by L. This interface is described in the file *lock.c* and is carried over from assignment 3a.
- serial.c will then pass *packet_gen*, *packet_method*, *lock*, S, as well as any necessary variables to chksum_parallel(); this method generates checksums from *packet_gen* from n sources using a average packet size of W and *packet_method* for M milliseconds. This method differs from serial_packet() in that it utilizes n threads in order to generate checksums, utilizing the load balancing technique specified by S. This method returns an integer, T – it first describes the number of packets that get fully processed in the allotted time. I will discuss the specifics of this method's implementation below.
- parallel.c will finally output T to the terminal, before cleaning up any leftover resources and exiting.

- The next level of modules will implement the functionalities of chksum_serial() and chksum_parallel(). As such it will be comprised of three files – chksum.c, queue.c, and lock.c

  chksum.c

  - This module will describe chksum_serial() and chksum_parallel, the methods that will bs used by serial.c and parallel.c.

    long chksum_serial()
    Input:
    * PackeSource_t packet_gen : a pointer to a packet source
    * volatile Packet_t * (*packet_method)(PacketSource_t *, int) : a pointer to a method for extracting packets from said source
    * int n : the number of sources that packets are being generated from
    * int M : the number of milliseconds this test should run for

    This simply loops through n sources within a non-terminating while loop, utilizing *packet_method* to extract new packets from *packet_gen*. Before this loop begins, a timer is started. At the top of the for-loop, if this timer exceeds M milliseconds, the main thread returns. Otherwise, the loop continues to generate checksums for source $i \in [n]$. Each time the method creates and processes a packet, a counter is incremented, who's final value is returned by this method once it returns from the loop.

    long chksum_parallel()
    Input:
    * Like chksum_serial, this method is configurable by *packet_gen*, *packet_method*, n, and M
    * int D : an integer representing the depths of the queues to be used in this test
    * L : a char representing the lock type to be used in our load balancing implementations.
    * S : a char representing a load balancing strategy.

    This method is responsible for initializing the data structures needed for the functioning of n worker threads calculating checksums of packets generated by *packet_gen*, extracted by *packet_method*, and placed in FIFO Lamport queues.
    Therefore it will allocate space for n instances of the *thread_args_t* data type, which I will describe below. Call this array T. Following that, it will call create_queue_pool(n, D) in order to allocate space for *packet_queue_t* array of size n, each with depth D. Call this array Q. Q represents a pool of Lamport queues.
    Finally it will allocate an array of n *lock_t* instances, all initialized by the lock type specified by

L. Call this array L
Following the initialization of these data structures, this method binds the pointer:

* void * (*worker_method)(void *)

to one of three methods describing the following functionalities depending on the specification of S. If S is:

* L, then threads should utilize a Lock-Free strategy. This strategy dictates 1 - 1 correspondence between queues and threads, as specified in previous assignment. While a thread's queue has not signaled that it is done queueing new packets, the thread waits until there is a packet in the queue, before dequeueing a packet and processing its checksum.
* H, then threads should utilizes the Home-Queue strategy. This is like the Lock-Free strategy, with the main difference being that, while the Lock-Free strategy does not utilize locking, the Home-Queue threads utilize an array of n locks (initialized in the dispatcher thread), which are in likewise associated in a 1 - 1 correspondence with the n queues in Q. Each thread using this strategy locks their queue's lock before calling its dequeue() method, and unlocks it after completing said call.
* A, then the threads should utilize my Awesome strategy(!)

These functionalities will be implemented in $void *L\_worker(void *args)$, $void *H\_worker(void *args)$, $void *A\_worker(void *args)$, respectively. These are the methods that can be bound to $worker\_method$.
Each of these functions will take in $thread\_args\_t$ struct with the following attributes:

* int i : an index into the array, Q
* packet_t *Q : a pointer to the Lamport queue in Q at index i
* lock_t *L : a pointer to a instance of a locking algorithm.
* int count : an integer to record a threads throughput. Initially equal to 0, this value should be incremented each time its worker thread generates a checksum.

The thread this method is called in is the dispatcher thread – this thread starts n worker threads, which will be responsible for calculating checksums of packets generated by $packet\_gen$, extracted by $packet\_method$, and placed in Q. Each thread will generate checksums in the manner proscribed by $worker\_method$. In order to do this, the dispatcher thread, $\forall i \in [T]$, sets:

* $T[i].i \leftarrow i$
* $T[i].Q \leftarrow \&Q[i]$
* $T[i].i \leftarrow \$\&L[i]$

and spawns a thread to call $worker\_method$ with $\&T[i]$ as an argument.

After chksum_parallel() has initialized and spawned its worker threads and data structures, it carries out a performance test. After starting a timer, it cycles through the n queues in Q, each associated with one of n packet sources. One queue at a time, the dispatcher thread will wait until the queue contains space for another packet, write to the queue, and then move on to the next queue. It will repeat this process so long as the timer has been running for less than M milliseconds. Once the timer is up, the dispatcher threads signals all threads to stops, joins them, and adds up the counters stored in the $thread\_args\_t$ instances in T. Call this value C. This value represents the combined total throughput of all threads over the course of the test.
This method terminates by cleaning up the resources initialized above and returning the value stored in C.

As such, this method describes neutral interface with which to test the throughput of a dispatcher distributing work among n worker threads, which can be configured to utilize varying and programmable locking algorithms and load balancing strategies.

queue.c

– This module will describe the construction and functionality of $packet_queue_t$ struct. This struct implements FIFO Lamport queue. It's declared with the following struct members and methods:

* $head, tail$: Volatile integers specifying the locations of the head and tail of the queue

    - *packets* : a Lamport queue with depth D
    - *done* : a Volatile boolean designating whether or not we should keep reading from Q. This is initially set to $false$
  - Methods:
    - int enq(packet_queue_t *Q, volatile Packet_t x) : queue a $Packet\_t$ instance in the Queue Q. Return 1 if Q is full, 0 if otherwise and x was enqueued successfully.
    - volatile Packet_t *deq(packet_queue_t *Q) : dequeue the next $Packet\_t$ from Q. Return a NULL value if the queue is empty
    - packet_queue_t *create_queue_pool(int num_q, int D) : allocate an array of num_q packet_queue_t structs, each of depth D
    - void destroy_queue_pool(packet_queue_t *Q) : Destroy an array of packet_queue_t structs allocated as a block queues

lock.c

- The locks mentioned above, will be implemented in a file called lock.c. The file will describe the implementation of a Test and Set lock, an Anderson's Array Lock, and an MCS lock, as specified in the textbook and translated into C. The Mutex lock will be called from the pthread library, but wrapped so as to work with our $lock\_t$ interface. The interface has the following attributes associated with it:
  - char type : a reference to a lock type, as specified by input
  - void *l : a pointer to a lock object; this pointer is passed to the initialization, lock, and unlock functions.
  - void (*init_thread)(void *) : This function initializes any thread specific structures required by a lock. It should be called before a thread acquires the lock the first time.
  - void (*try_lock)(void *) : a pointer to the lock's try_lock function; it throws an exception to the calling thread if the lock object is busy. Otherwise, the calling thread acquires the lock.
  - void (*lock)(void *) : a pointer to the lock's lock function; it should be called on the void *l (which points to the initialized lock) in order to acquire the lock.
  - void (*unlock)(void *) : a pointer to the lock's unlock function; it should be called on the void *l (which points to the initialized lock) in order to release the lock.
- Each locking algorithm type will have the following methods associated with it:
  - init(int n): initialize a lock to synchronize n threads; n is only used in initializing locks that allocate space for locks prior to use, such as Anderson's lock. Otherwise, n is discarded.
  - destroy(void *lock): destroy the initialized *lock* and free any memory allocated to support it
  - init_thread(void *lock): in the case of the MCS lock, this method needs to be called by each thread utilizing the lock before it acquires the lock for the first time.
  - try_lock(void *lock) : try_lock *lock*
  - lock(void *lock): lock *lock*
  - unlock(void *lock): unlock *lock*
- This interface should be able to be initialized and freed either one at a time or as a block. As such, this module will also contain the following methods:
  - lock_t *new_lock(char type, int n) : initialize a single lock of the kind specified by *type* which will manage n threads.
  - lock_t *new_lock_pool(int size, char type, int n) : initialize a lock pool with *size* locks of the kind specified by *type*, all of which will each manage n threads. This pool is allocated as a block of contiguous memory
  - int destroy_lock(lock_t *L) : destroy a single lock_t instance
  - int destroy_lock_pool(int size, lock_t *L) : destroy a pool of contiguously allocated locks of size *size*.
- This interface

- Our last module consists of testing and analysis protocols, test_script.sh and analysis.py
  - test_script.sh will consist of a shell script that utilizes serial.c, and parallel.c. This test script will execute these files and record their output, generating experimental data in an automated fashion and storing that data in csv files.
  - analysis.py will be responsible for analyzing data, making graphs, and outputting quantitative data to compare the performance of locks.
- A note on Invariants... what are they???

Test plan:

Need to try and come up with issues youre going to run into as you develop... Do you expect perfroamcen to increase in all test cases?? – ID which one you suspect Need to justify how we are delimting time to begin and end, as well as talk about how we are measuring through put with shared timer should mention that this construction lacks any satisfying way of generating proofs of correctness – but as the last use of our lamport queue was correct enough – perhaps the implementations could stand to be tested for definite number of packets, before moving on to throughput tests...

Expected Performance (Performance Hypotheses):

Exp