15-418/618 Spring 2020
Exercise 4

Assigned:	Mon., March 23
Due:	Mon., March 30, 11:00 pm

Overview

This exercise is designed to help you better understand the lecture material and be prepared for the style of questions you will get on the exams. The questions are designed to have simple answers. Any explanation you provide can be brief—at most 3 sentences. You should work on this on your own, since that's how things will be when you take an exam.

You will submit an electronic version of this assignment to Gradescope as a PDF file. Unlike previous exercises, we have prepared a separate document for you to provide your answers, available at:

http://www.cs.cmu.edu/~418/exercises/ex4-answer.pdf

For those of you familiar with the LaTeX text formatter, you can download the template and figure files at:

http://www.cs.cmu.edu/~418/exercises/ex4-answer.tex
http://www.cs.cmu.edu/~418/exercises/figs/fat-tree.pdf
http://www.cs.cmu.edu/~418/exercises/figs/fat-tree-links.pdf

Instructions for how to use this template are included as comments in the file. Otherwise, you can use the PDF document as your starting point. You can either: 1) electronically modify the PDF, or 2) print it out, write your answers by hand, and scan it. In any case, we expect your solution to follow the formatting of the answer document.

Problem 1: Lock Implementations

We saw in Lecture 18 (slide 7) that different synchronization schemes can have very different performance characteristics. In particular, we found that neither the mutex lock nor the spin lock implemented as library functions in Pthreads give very good performance, especially when increasing the number of threads contending for a single, shared global variable. In this problem, we will examine possible implementations of a lock based on atomic compare-and-swap (CAS) operations.

Figure 1A shows the declaration of data type `cas_lock_t`, consisting of a single `int` set to 1 when the lock is held and 0 when it is not. Figures 1B–C show that the lock is initialized and released by simply setting it to 0. Figure 2 then shows three different ways to implement the lock operation: All three make use of the atomic compare-and-swap (CAS) operation supported as a builtin operation by the GCC compiler (refer to slide 5).

A. Data structure

```
typedef struct {
    volatile int is_locked;
} cas_lock_t;
```

B. Initialization

```
void cas_lock_init(cas_lock_t *lock) {
    lock->is_locked = 0;
}
```

C. Release

```
void cas_lock_unlock(cas_lock_t *lock) {
    lock->is_locked = 0;
}
```

Figure 1: Data Structure and functions for CAS-based lock

A. Direct. The function loops until the CAS operation successfully sets the lock to 1.

B. Test. The function repeatedly tests the lock variable and attempts CAS only when it detects that the variable has been set to 0.

C. Backoff. Like test, but the program delays by a random amount of time for each test of the lock variable. The delays amounts are of the form $D \cdot 2^i$ where i is uniformly distributed between 0 and 20. That is, the delay amounts are exponentially distributed.

Figure 3 shows the performance of these three locking strategies when K different threads all repeately increment a single global variable, as a function of K. Times are measured in *nanoseconds per increment*, the average time (in nanoseconds) for each increment operation. These times are compared to two others approaches: using a standard Pthread mutex, and using an atomic fetch-and-add operation. We show for thread counts ranging from 1 to 12 when running on one of the 8-core GHC machines.

For each of the three lock implementations, give a brief explanation of why it achieves the performance it does, both in terms of its scaling with the number of threads and its performance relative to other schemes.

A. Direct.

B. Test.

C. Backoff.

Problem 2: Memory Consistency

Suppose you want to use the CAS-based lock from Problem 1 on a machine that provides no guarantees on the relative orderings of memory loads and stores. So, for example, the guarantees are even weaker than Intel's Total Store Ordering (TSO) guarantee. (You may wish to review the material from Lecture 14.)

A. Direct

```
void cas_direct_lock(cas_lock_t *lock) {
    while (!__sync_bool_compare_and_swap(&lock->is_locked, 0, 1))
        /* Keep trying */
            ;
}
```

B. Test and loop

```
void cas_test_lock(cas_lock_t *lock) {
    while (1) {
        while (lock->is_locked)
        /* Keep trying */
            ;
        if (__sync_bool_compare_and_swap(&lock->is_locked, 0, 1))
            break;
    }
}
```

C. Test and loop with backoff

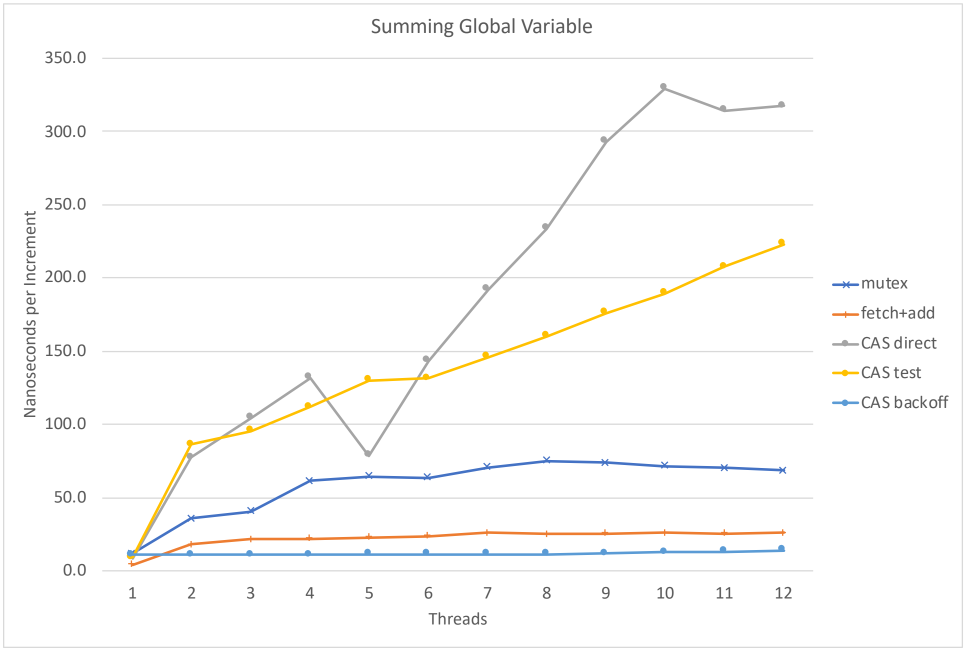
```
void cas_backoff_lock(cas_lock_t *lock) {
    while (1) {
        while (lock->is_locked)
            /* Back off by random amount */
            random_delay();

        if (__sync_bool_compare_and_swap(&lock->is_locked, 0, 1))
            break;
    }

}
```

Figure 2: Different implementations of lock acquisition

Threads	1	2	3	4	5	6	7	8	9	10	11	12
mutex	11.57	35.62	40.40	61.34	64.62	63.51	70.71	75.06	73.85	71.60	70.12	68.32
fetch+add	3.95	18.10	21.32	21.99	22.96	23.82	25.78	24.86	25.33	25.68	25.61	25.78
CAS direct	8.77	77.18	104.17	131.58	78.58	143.13	192.11	233.60	293.26	329.50	314.32	317.34
CAS test	9.00	86.10	95.38	111.76	129.96	131.16	146.13	160.31	176.31	189.60	207.36	223.28
CAS backoff	10.70	10.73	10.93	10.99	11.15	11.13	11.13	11.16	11.63	12.47	12.89	13.74



Threads	1	2	3	4	5	6	7	8
global	3.99	16.13	17.58	17.34	18.46	18.60	18.02	17.68
local	5.56	2.78	1.89	1.42	1.16	0.97	0.83	0.73

Figure 3: Performance of different synchronization schemes when summing single global variable (in nanoseconds per increment)

A. Operating on global variable

```
int global_fence_val = 0;

static void fence_global() {
    __sync_fetch_and_add(&global_fence_val, 0);
}
```

B. Operating on local variable

```
static void fence_local() {
    int fence_val = 0;
    __sync_fetch_and_add(&fence_val, 0);
}
```

Figure 4: Possible Implementations of a fence operation

.

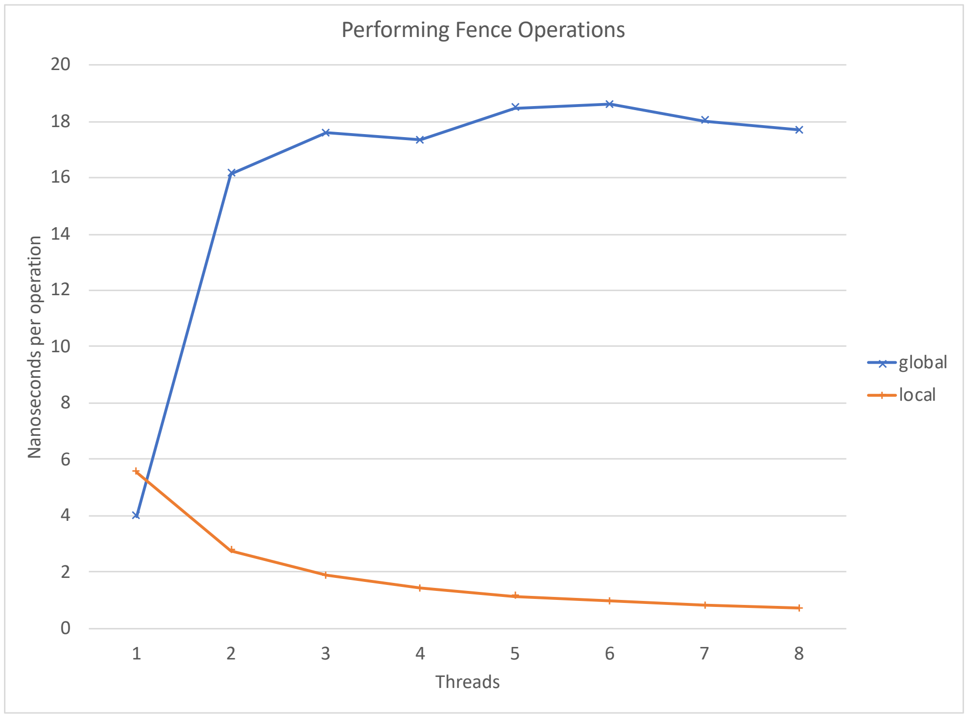


Figure 5: Performance of different implementations of a fence operation (in nanoseconds per fence)

The functions in the GCC library of builtins provide the following guarantee:

> . . . these builtins are considered a *full barrier*. That is, no memory operand will be moved across the operation, either forward or backward. Further, instructions will be issued as necessary to prevent the processor from speculating loads across the operation and from queuing stores after the operation.

With that in mind, we can use the fetch-and-add operation to implement a fence operation, as is shown in Figure 4. This figure shows two implementations according to whether the fetch-and-add is performed on a global variable (A) or on a local variable (B). Figure 5 shows the performance of a program that spawns K threads, each of which make N calls to one of the two fence functions. The performance is expressed in terms of the average number of nanoseconds for each fence operation, computed as $T \cdot 10^9/N \cdot k$, where T is the total runtime (measured in seconds) for the program.

A. Explain why these two fence operations have such different relative performance.

B. Explain why the local version has performance that improves with the number of threads.

C. We are concerned about whether the different implementations of a CAS-based lock would operate correctly under this weak memory ordering. We can assume the following about how locks get used:

   - Locks are initialized by the main thread before it spawns other threads that may use the locks.
   - A thread uses a lock to protect its access to global, shared state. It will acquire the lock before reading or writing shared state, and it will release the lock when it has completed any updates to the shared state.

   With these principles in mind, we must make sure that the locks perform their intended role. In particular:

   - The initialization of a lock must complete before any thread attempts to acquire it.
   - A thread must not cause any changes to global state before it completes the acquisition of a lock.
   - A thread must complete any updating of global state before it causes any activity that other threads could interpret as the releasing of a lock.

   For each of the five functions in Figures 1 and 2, determine where fence operations must be inserted to guarantee these ordering properties. You should insert only a minimum set of fences. Justify why these particular fences are required and why no others are.

Problem 3: Interconnection Networks

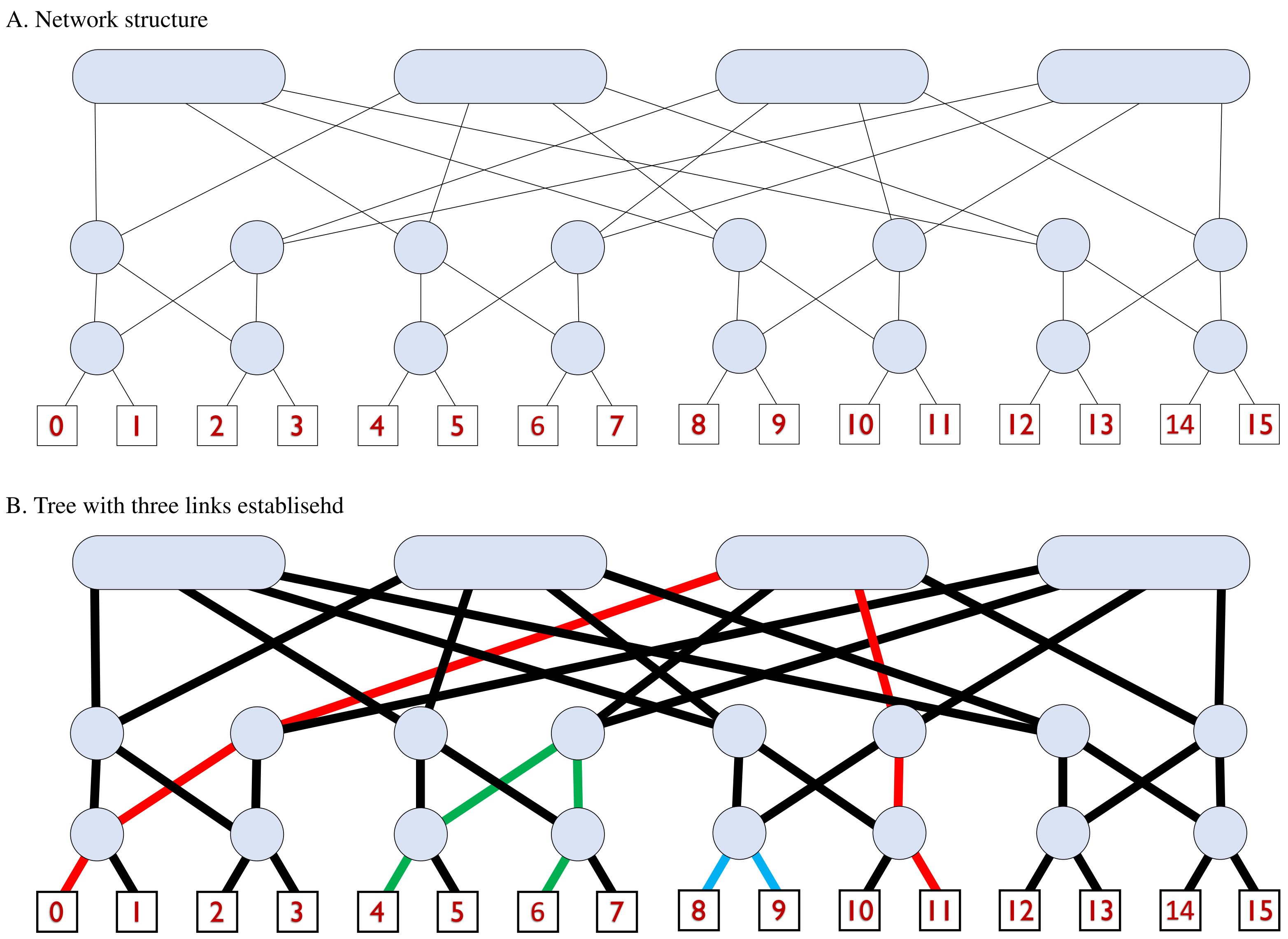


Figure 6: Fat-Tree Network

This problem concerns *fat-tree networks*, as discussed in Lecture 15:

http://www.cs.cmu.edu/~418/lectures/15_interconnects.pdf.

These networks were originally proposed in 1985 by Charles Leiserson, a CMU PhD alumnus. They are a form of *indirect network*, meaning that the network is constructed separately from the computing elements.

Figure 6A illustrates a fat-tree network constructed using a set 4-port switches. It is identical to the one shown on Slide 36 of the lecture. (Although some switches are shown as circles and some as ovals, they are all identical.) We will characterize these networks by two parameters: the *switch connectivity* k, where each switch has k ports ($k = 4$ for the network of Figure 6A, and the number of *levels*, l. We will use the notation $N(k, l)$ to denote a *maximal* network with switch connectivity k and l levels. By maximal, we mean that it is the largest network that can be constructed with those switches and that many levels. We use the notation $P(k, l)$ to indicate the number of *external ports* provided by a network of type $N(k, l)$. These ports can either be connected to computing elements or to other switches in order to form larger networks.

These networks can be defined recursively:

- Network $N(k, 1)$ consists of a single switch, with all k ports being external.
- Network $N(k, l)$ is constructed by assembling $k/2$ *subnetworks*, each of type $N(k, l-1)$ along the top. Along the bottom, the network has an additional $P(k, l-1)$ switches. For each switch along the bottom, $k/2$ of its ports connect to the external ports of the subnetworks (one port per subnetwork), while the other $k/2$ ports serve as external ports for the new network.

A. The network of Figure 6A shows a network of type $N(4, 3)$, and slide 36 from Lecture 15 shows one of type $N(6, 3)$. However, it's a bit difficult to see the recursive structure in these illustrations. Identify the $k/2$ subnetworks of type $N(k, 2)$ for $k = 4$ in Figure 6. You can do this by modifying the diagram in Figure 6A. Use different colors for the switches to indicate the different subnetworks and the additional switches. For your convenience, the original diagram is available as:

   http://www.cs.cmu.edu/~418/exercises/figs/fat-tree.pptx

B. Derive a closed-form formula for $P(k, l)$.

C. Routing in a fat tree involves setting up a *link* between two leaf nodes. We assume that each switch can simultaneously maintain connections between any pairs of its ports, as long as there is no conflict.

   If we number the ports along the bottom from 0 to $N - 1$, where $N = P(k, l)$, then routing from port i to port j involves choosing any available upward path from i until reaching a switch that has a downward path to j, and then following this path downward. This follows the scheme of *tree routing* described in Slide 32, except that there can be have multiple upward paths from which to choose. Figure 6B illustrates the network with three of these links established.

   A maximal fat-tree network has the property that it *nonblocking*, meaning that for any permutation π over the set $\{0, \ldots, N-1\}$, it can simultaneously maintain the complete set of links between nodes i and $\pi(i)$ for $0 \leq i < N$. For the network of Figure 6A, show that you could set up the eight links forming a *mirror permutation*, mapping port i to port $N - i - 1$ for $0 \leq i < N/2$. You can do this by modifying the diagram in Figure 6B. Use different colors to illustrate the different links.

You may find it helpful to work from the following Powerpoint document:

http://www.cs.cmu.edu/~418/exercises/figs/fat-tree-links.pptx