Due: Sunday, Nov 18, 2012. 11:59pm (no extensions).

What to submit: A tarball with files `uthreads_semaphore.c`, `uthreads_mutex.c`, and `linux_lock_fairness.c`.

1. Cooperative User-Level Threading

User-level threading packages have become somewhat of a lost art. Their inherent disadvantages prevent them from being used in many contexts in which application-level concurrency is required. They do not support the use of multiple CPUs or cores, they lack the ability to preempt an uncooperative thread's access to the CPU, and they can make it difficult to prevent the OS from blocking an entire process if a blocking system call is made by a thread.

Their advantages include their low cost of context switching, the lack of need for any OS support, and full control over scheduling. Occasionally, they are used in high-performance server designs [1, 2] to increase scalability and push the boundaries of how many concurrent tasks can be managed on a single CPU or core.

In this exercise, you will be implementing mutexes and semaphores for a small user-level threading package. The need for such synchronization devices exists even in an environment that does not support preemption or allows the use of multiple CPUs, because context switches will still occur if threads block to wait for a resource or event.

The directory ˜cs3214/public_html/fall2012/exercises/uthreads contains an incomplete implementation of a user-level threading package called *uthreads*. Your task is to study its implementation and complete the methods associated with the types `uthreads_mutex_t` and `uthreads_sem_t`, as shown below.

```
/**
 * Support for mutexes.
 */
typedef struct uthreads_mutex {
    uthreads_t holder;          // if NULL, mutex is available;
                                // else denotes thread holding mutex
    struct list waiters;        // waiting threads, if any.
} *uthreads_mutex_t;

/* Initialize this mutex. */
void uthreads_mutex_init(uthreads_mutex_t m);

/* Acquire this mutex. */
void uthreads_mutex_lock(uthreads_mutex_t m);

/* Release this mutex. */
void uthreads_mutex_unlock(uthreads_mutex_t m);

/**
 * Support for semaphores.
 */
typedef struct uthreads_sem {
    int count;                  // value of semaphore.  Always non-negative.
    struct list waiters;        // waiting threads, if any.
} * uthreads_sem_t;

/* Initialize this semaphore. */
void uthreads_sem_init(uthreads_sem_t s, int initial);

/* Wait on this semaphore. */
void uthreads_sem_wait(uthreads_sem_t s);

/* Post ('signal') this semaphore. */
void uthreads_sem_post(uthreads_sem_t s);
```

You should fill in the skeletons `uthreads_mutex.c` and `uthreads_semaphore.c`. You will need to invoke the function `uthreads_block` and `uthreads_unblock` as appropriate to communicate with the scheduler. Since *uthreads* are not preemptive and do not support multiple cores, no use of atomic instructions is required. If your implementation is correct, running `make check` should print `Ok` 3 times. A correct solution can be implemented in 13 lines (uthreads_mutex.c) and 9 lines (uthreads_semaphore.c) of code.

2. Are Linux Mutexes Fair?

A mutex is said to be contended if it is held by another thread at the time when a thread attempts to acquire it. When a mutex is frequently contended, fairness becomes important. As different threads attempt to acquire the mutex, they should have the same chance of acquiring the mutex, and no thread should have to wait indefinitely for it.

In this exercise, you are asked to design and implement a test program that starts N threads that contend for the same mutex in a tight loop. Your program should count how often each thread was able to acquire the lock within a given timespan. To ensure that all threads have started when you start counting you may wish to reset the counters to zero some time (say 1 second) after the threads have started. The main thread should then sleep for some time (say 3 seconds) while the threads contend for the shared lock.

Make the number of threads a command line argument to your program and run it with 1, 2, 3, 4, 5, or more threads. You may wish to repeat each run a number of times.

1. How many times per second was a single thread able to acquire/release the mutex?
2. If you ran 2 threads, how many times per second was the mutex acquired/released?
3. In the case of 2 threads, were both threads able to acquire the lock roughly the same number of times?
4. In the case of 3 or more threads, were those threads able to acquire the lock roughly the same number of times?

5. The fairness of a lock implementation is also affected by whether the contending threads execute on different CPUs or cores or the same CPU. By default, when you start a multi-threaded process on Linux, Linux attempts to distribute the threads belonging to the process evenly across all available cores. The `taskset(1)` command can be used to restrict the set of cores on which a process's threads may be run.[1] For instance, if you ran `taskset -c 3 ./linux_fairness_test 4` then all threads created by the process started to execute the command `./linux_fairness_test` would be allowed to execute only on core 3. (On our rlogin cluster, each node contains 16 cores numbered 0-15).

   Repeat the previous cases (2 or more threads) while restricting the number of cores on which those threads can run to one core - choose one of the available 15. How does this restriction affect fairness?

3. Using the Amazon Elastic Compute Cloud (EC2)

In this exercise, you will install a virtual machine in the Amazon Elastic Compute Cloud (EC2) to prepare for project 5, in which you will develop and run a web service in the cloud. Amazon's EC2 service is an example of emerging infrastructure as a service (IaaS) products that provide users with access to virtual machines.

Thanks to a teaching grant from Amazon, every student is provided with free credits that can be used towards Amazon's web services. In your grades directory, you'll find a file 'amazon-coupon' that contains your personal coupon code.

The goal of this exercise is to familiarize yourselves with how to use the Amazon EC2 cloud. To this end, you'll have to set up an account, and set up and launch a virtual machine or 'instance.' The following steps will be necessary.

- Sign up for an Amazon Web Services (AWS) account at www.amazon.com/aws. Redeem your coupon.
- Go to https://aws.amazon.com/ec2/ and sign up for Amazon EC2. You will need to provide a credit card here, which however will only be charged should you exceed the credits provided as part of Amazon's grant ($100).
- Using the AWS Management Console, launch an instance. Choose an AMI (Amazon Machine Image) (the basic Amazon Linux AMI 1.0 is a suitable choice, as may be others). Launch the instance. Create a key pair. Store the .pem file created, you'll need it to log on to the instance. Make sure the .pem file is readable and writable only by you (chmod 600 yourkeyfile.pem will do).
- Create a new security group. A security group includes the required firewall settings to ensure your instance's service will be reachable. By default, only port 22

[1]Another way to accomplish the same is using the `pthread_setaffinity_np()` call.

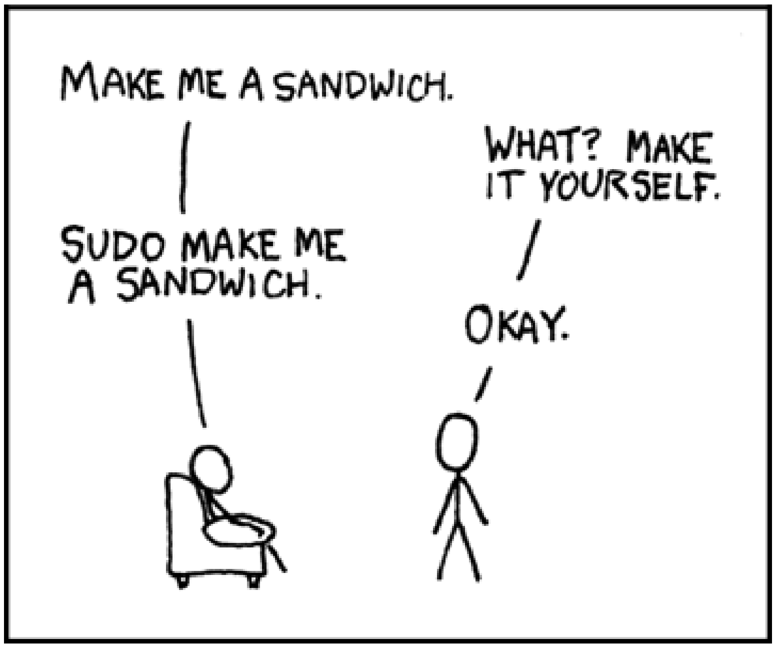

Figure 1: Source: xkcd.com

(SSH) is open. If you wish to run Apache's httpd, add port 80 (HTTP). Launch the instance.

- Under My Instances, select your running instance and find its Public DNS name (for instance, ec2-184-73-27-136.compute-1.amazonaws.com). Log on to your instance using ssh like so:

  ```
  ssh -i yourkeyfile.pem ec2-user@ec2-184-73-27-136.compute-1.amazonaws.com
  ```

  where yourkeyfile.pem is where you stored your key pair. You should now have shell access to your instance.

- To be able to work on your virtual machine, you may need to install additional tools, such as gcc and svn. If you've used Amazon's Linux AMI 1.0 image, you should be able to use `sudo yum install gcc subversion` to that effect. The sudo command allows authorized users to perform commands that require administrative privileges. (See Figure 1 and sudoers(5).)

- I recommend you try to build projects 3 and 5 on your virtual machine. Check out your SVN, and make and test those projects.

- Don't forget to terminate or stop your Amazon instance. Please note that while your instance is running, charges of currently $0.085 per hour (or $2.04 per day) incur until you shut down the instance.

- The instructions above created an ephemeral instance with no persistent storage. If you terminate it, all data will be lost and you will have to repeat all steps, but all charges will stop. If you stop it (rather than terminating it, see AWS console), the data on the boot partition (which includes all files you've installed and/or up-loaded) will be retained; a smaller charge of $0.10 per GB/month applies to keep the data of the boot partition of a stopped instance until it is restarted or terminated.

- Feel free to explore. For instance, you could try out Amazon's Elastic Block Store to create persistent volumes that appear as hard drives in your instances.
- No submission is required for this (Amazon-related) part of the exercise.

Bibliography

[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[2] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 268–281, New York, NY, USA, 2003. ACM.