Recitation 14: Proxy Lab Part 2

Instructor: TA(s)
Outline

- Proxylab
- Threading
- Threads and Synchronization
ProxyLab

- **ProxyLab is due in 1 week.**
  - No grace days
  - Make sure to submit well in advance of the deadline in case there are errors in your submission.
  - Build errors are a common source of failure

- **A proxy is a server process**
  - It is expected to be long-lived
  - To not leak resources
  - To be robust against user input
Proxies and Threads

- Network connections can be handled concurrently
  - Three approaches were discussed in lecture for doing so
  - Your proxy should (eventually) use threads

- Threaded echo server is a good example of how to do this
Join / Detach

Does the following code terminate? Why or why not?

```c
int main(int argc, char** argv)
{
    ...
    pthread_create(&tid, NULL, work, NULL);
    if (pthread_join(tid, NULL) != 0) printf(“Done.\n”);
    ...
    void* work(void* a)
    {
        pthread_detach(pthread_self());
        while(1);
    }
```
Join / Detach cont.

- Does the following code terminate now? Why or why not?

```c
int main(int argc, char** argv)
{
    ...
    pthread_create(&tid, NULL, work, NULL); sleep(1);
    if (pthread_join(tid, NULL) != 0) printf("Done.\n");
    ...
    void* work(void* a)
    {
        pthread_detach(pthread_self());
        while(1);
    }
```
When should threads detach?

- In general, pthreads will wait to be reaped via pthread_join.

- When should this behavior be overridden?
When should threads detach?

■ In general, pthreads will wait to be reaped via pthread_join.

■ When should this behavior be overridden?

■ When termination status does not matter.
  ▪ pthread_join provides a return value

■ When result of thread is not needed.
  ▪ When other threads do not depend on this thread having completed
Threads

■ What is the range of value(s) that main will print?
■ A programmer proposes removing j from thread and just directly accessing count. Does the answer change?

```c
volatile int count = 0;

int main(int argc, char** argv)
{
    pthread_t tid[2];
    for(int i = 0; i < 2; i++)
        pthread_create(&tid[i], NULL, thread, NULL);
    for (int i = 0; i < 2; i++)
        pthread_join(tid[i]);
    printf("%d\n", count);
    return 0;
}

void* thread(void* v)
{
    int j = count;
    j = j + 1;
    count = j;
}
```
Synchronization

- **Is not cheap**
  - 100s of cycles just to acquire without waiting

- **That is also not expensive**
  - Recall your malloc target of 15000kops => ~100 cycles

- **May be necessary**
  - Correctness is always more important than performance
Which synchronization should I use?

- Counting a shared resource, such as shared buffers
  - 
- Exclusive access to one or more variables
  - 
- Most operations are reading, rarely writing / modifying
  - 
Which synchronization should I use?

- Counting a shared resource, such as shared buffers
  - Semaphore

- Exclusive access to one or more variables
  - Mutex

- Most operations are reading, rarely writing / modifying
  - RWLock
Threads Revisited

- Which lock type should be used?
- Where should it be acquired / released?

```c
volatile int count = 0;

void* thread(void* v)
{
    int j = count;
    j = j + 1;
    count = j;
}

int main(int argc, char** argv)
{
    pthread_t tid[2];
    for (int i = 0; i < 2; i++)
    {
        pthread_create(&tid[i], NULL, thread, NULL);
        printf("%d\n", count);
    }
    for (int i = 0; i < 2; i++)
    {
        pthread_join(tid[i]);
    }
    return 0;
}
```
Associating locks with data

- Given the following key-value store
  - Key and value have separate RWLocks: klock and vlock
  - When an entry is replaced, both locks are acquired.
- Describe why the printf may not be accurate.

```c
typedef struct _data_t {
    int key;
    size_t value;
} data_t;

#define SIZE 10
data_t space[SIZE];
int search(int k) {
    for(int j = 0; j < SIZE; j++)
        if (space[j].key == k) return j;
    return -1;
}

...  
pthread_rwlock_rdlock(klock);
match = search(k);
pthread_rwlock_unlock(klock);

if (match != -1) {
    pthread_rwlock_rdlock(vlock);
    printf("%zd\n", space[match]);
pthread_rwlock_unlock(vlock);
}
```
Locks gone wrong

1. RWLocks are particularly susceptible to which issue:
   a. Starvation    b. Livelock    c. Deadlock

2. If some code acquires rwlocks as readers: LockA then LockB, while other readers go LockB then LockA. What, if any, order can a writer acquire both LockA and LockB?

3. Design an approach to acquiring two semaphores that avoids deadlock and livelock, while allowing progress to other threads needing only one semaphore.
Locks gone wrong

1. **RWLocks** are particularly susceptible to which issue:
   a. Starvation  
   b. Livelock  
   c. Deadlock

2. If some code acquires rwlocks as readers: LockA then LockB, while other readers go LockB then LockA. What, if any, order can a writer acquire both LockA and LockB?
   
   No order is possible without a potential deadlock.

3. Design an approach to acquiring two semaphores that avoids deadlock and livelock, while allowing progress to other threads needing only one semaphore.
Proxylab Reminders

- Read the writeup

- Submit your code (days) early
  - Test that the submission will build and run on Autolab
Appendix

- Calling exit() will terminate all threads

- Calling pthread_join on a detached thread is technically undefined behavior. Was defined as returning an error.