Recitation 12: Synchronization

Monday April 17
Your TA(s)

Outline

- Logistics
- Proxylab
- Makefiles
- Threading
- Threads and Synchronization

So you wanna TA for 213

- What qualifications are we looking for?
  - Decent class performance, but also critical thinking skills
  - Like computer systems + want to help others like systems!
  - Have a reasonable ability to gauge your schedule + responsibilities
  - Leadership potential! Take initiative, we love to see it 😌
  - Ability to tell students:
    - “Did you write your heap checker”
    - “Run backtrace for me”
    - rinse and repeat, it’s mouthwash baby

Apply at https://www.ugrad.cs.cmu.edu/ta/F23/

ProxyLab

- **ProxyLab is due next Thursday (April 27). Checkpoint is due FRIDAY (April 21).**
  - One grace day for each
  - Proxy Final may NOT be submitted after the last day of classes per University policy
  - 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
- **Note on CSAPP**
  - Most CSAPP functions have been removed
  - Error check all system calls and exit only on critical failure

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

- **Multi-threaded cache design**
  - Be careful how you use mutexes. Do not hold locks over network / file operations (read, write, etc)
  - Using semaphores is not permitted
  - Be careful how you maintain your object age

Join / Detach

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

```
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?**

```
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);
}
```

Join / Detach cont.

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

```
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);
}
```

sleep will not help solve race conditions!!!

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?

```
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);
    for (int i = 0; i < 2; i++)
        pthread_join(tid[i]);
    printf(“%d\n”, count);
    return 0;
}
```

Synchronization

- **Is not cheap**
  - 100s of cycles just to acquire without waiting
- **Is also not that expensive**
  - Recall your malloc target of 15000kops => ~100 cycles
- **May be necessary**
  - Correctness is always more important than performance

Semaphore Review

- **Semaphores are non-negative global integers for synchronization**

- **P(s) -- "wait until it's my turn"**
  - while(s == 0) { wait(); } s--;
- **V(s) -- "I'm done"**
  - s++;

- **P/V are implemented to run *atomically***

Other Synchronization

- **Mutexes -- similar to semaphores**
  - Only two states
  - ~2 times faster than semaphores
- **Reader-Writer Locks**
  - Allows multiple threads to read at the same time, but only one if it needs to write
- **These will be discussed in more detail in lecture**

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

For proxy it’s sufficient to just use mutexes!

(using semaphores is forbidden)

Threads Revisited

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

```
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);
    for (int i = 0; i < 2; i++)
        pthread_join(tid[i]);
    printf(“%d\n”, count);
    return 0;
}
```

Associating locks with data

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

```
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_mutex_lock(klock);
match = search(k);
pthread_mutex_unlock(klock);

if (match != -1)
{
  pthread_mutex_lock(vlock);
  printf(“%zd\n”, space[match]);
  pthread_mutex_unlock(vlock);
}
```

Locks gone wrong

1. **RWLocks are particularly susceptible to which issue:**

[✓ a. Starvation]   b. Livelock   c. Deadlock

1. **If some code acquires semaphores: S1 then S2, while other readers go S2 then S1. What, if any, order can a writer acquire both S1 and S2?**

No order is possible without a potential deadlock.

Proxylab Reminders

- **Plan out your implementation**
  - "Weeks of programming can save you hours of planning"
    – Anonymous
  - Arbitrarily using mutexes will not fix race conditions
- **Read the writeup**
- **Submit your code (days) early**
  - Test that the submission will build and run on Autolab
- **Final exam is only a few weeks away!**

Appendix

- Calling exit() will terminate all threads

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

Client-to-Client Communication

- **Clients don’t have to fetch content from servers**
  - Clients can communicate with each other
  - In a chat system, a server acts as a facilitator between clients
  - Clients could also send messages directly to each other, but this is more complicated (peer-to-peer networking)
- **Running the chat server**
  - `./chatserver <port>`
- **Running the client**
  - `telnet <hostname> <port>`

- **What race conditions could arise from having communication between multiple clients?**

Appendix: Makefiles

- Makefile: tells program how to compile and link files

```
# List of all header files (for fake cache.c file)

DEPS = csapp.h transpose.h

# Rules for building cache

cache: cache.o transpose.o csapp.o

transpose.o: transpose.c $(DEPS)

cache.o: cache.c $(DEPS)

csapp.o: csapp.c csapp.h
```