Threading + Proxy II

15/18-213
Recitation 13
4/15/2013

Outline

- **Proxy**
  - Due Thursday, April 25th
  - No late/penalty days
  - Absolute last time to turn in is April 25th, 11:59 PM
- **Threading**

Proxy - Functionality

- **Should work on vast majority of sites**
  - Reddit, Vimeo, CNN, YouTube, etc.
  - Some features of sites which require the POST operation (sending data to the website), will not work
    - Logging in to websites, sending Facebook messages
- **Cache previous requests**
  - Use LRU eviction policy
  - Must allow for concurrent reads
  - Details in write up

Proxy - Partner

- **Allowed to work with a partner**
  - Highly encouraged
  - No difference in grading vs. solo work
  - Sign-up on Autolab
- **Collaborating**
  - Splitting up work
    - Proxy and cache can be done independently...
  - Use Git for version control

Git

- **What is Git?**
  - Version control software
  - Easily collaborate/update shared project
    - Can roll back to previous version if needed
  - Already installed on Andrew machines
  - Set up a repo on GitHub, BitBucket, or AFS
    - Make sure only you and your partner can access it!
- **Using Git**
  - git pull
  - git add .
  - git commit -m “I changed something”
  - git push

Multi-threaded Cache

- **Why?**
  - Sequential cache would bottleneck parallel proxy
  - Multiple threads can read cached content safely
    - Search cache for the right data and return it
    - Two threads can read from the same cache block
  - But what about writing content?
    - Overwrite block while another thread reading?
    - Two threads writing to same cache block?

Read-Write Lock

- **Cache can be read in parallel safely**
- **If thread is writing, no other thread can read or write**
- **If thread is reading, no other thread can write**
- **Potential issues**
  - Writing starvation
    - If threads always reading, no thread can write
    - Fix: if a thread is waiting to write, it gets priority over any new threads trying to read
- **How can we lock out threads?**

Mutexes & Semaphores

- **Mutexes**
  - Allow only one thread to run code section at a time
  - If other threads are trying to run the code, they will wait
- **Semaphores**
  - Allows a fixed number of threads to run the code
  - Mutexes are a special case of semaphores, where the number of threads=1
    - Examples will be done with semaphores to illustrate

N^2

- **Let's write a program!**
  - Spawns N threads
    - Each thread stores the current value of a global variable, adds 1 to that value N times, then writes the result back into the global
    - After the threads have finished running, print the global
    - It should be N^2

N^2 – No Semaphores

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#define N 1000

static unsigned int global = 0;

//Have a thread add N to the global variable
void* threadFunc(void* vargp)
{
  unsigned int locGlob = global;
  int i = 0;
  for (i = 0; i < N; i++)
    locGlob = locGlob + 1;
  global = locGlob;
  return NULL;
}

int main()
{
  pthread_t tids[N];
  pthread_t tid;
  int i = 0;
  for (i = 0; i < N; i++) //Spawn n threads
    pthread_create(tids+i,NULL,threadFunc,NULL);
  for (i = 0; i < N; i++) //Wait for all to finish
    pthread_join(tids[i], NULL);
  printf("%u\n",global);
  return 0;
}
```

N^2 – No Semaphores - Output

```
twklein@catshark:~/private/15213$ gcc thread.c -pthread
twklein@catshark:~/private/15213$ ./a.out
987000
twklein@catshark:~/private/15213$ ./a.out
989000
twklein@catshark:~/private/15213$ ./a.out
993000
twklein@catshark:~/private/15213$
```

What went wrong?

- **Read-write racing!**
  - What should happen:
    - Thread 1: read global=0 into globLoc
    - Thread 1: add 1000 to globLoc
    - Thread 1: write global=globLoc=1000
    - Thread 2: read global=1000...
  - What actually happened:
    - Thread 1: read global=0 into globLoc
    - Thread 2: read global=0 into globLoc
    - ...

Fixing N^2 with Semaphores

- **Let's give each thread a read/write mutex to global**
  - Will ensure each thread reads/writes the correct value
  - Note: in this example, this will cause the code to essentially run sequentially, and thread overhead will actually give worse performance compared to a sequential solution

N^2 - Semaphores

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#define N 1000

static unsigned int global = 0;
sem_t mutex;
//Have a thread add N to the global variable
void* threadFunc(void* vargp)
{
  int i = 0;
  sem_wait(&mutex); //Start critical code
  unsigned int locGlob = global;
  for (i = 0; i < N; i++)
    locGlob = locGlob + 1;
  global = locGlob;
  sem_post(&mutex); //End critical code
  return NULL;
}

int main()
{
  pthread_t tids[N];
  pthread_t tid;
  sem_init(&mutex,0,1); //Initialize semaphore to allow only 1 thread
  int i = 0;
  for (i = 0; i < N; i++) //Spawn n threads
    pthread_create(tids+i,NULL,threadFunc,NULL);
  for (i = 0; i < N; i++) //Wait for all to finish
    pthread_join(tids[i], NULL);
  printf("%u\n",global);
  return 0;
}
```

N^2 – Semaphores - Output

```
twklein@catshark:~/private/15213$ gcc thread.c -pthread
twklein@catshark:~/private/15213$ ./a.out
1000000
twklein@catshark:~/private/15213$ ./a.out
1000000
twklein@catshark:~/private/15213$ ./a.out
1000000
```

Read-Write Locks Cont.

- **How would you make a read-write lock with semaphores?**
  - Luckily, you don't have to!
    - pthread_rwlock_* handles that for you
      - pthread_rwlock_t lock;
      - pthread_rwlock_init(&lock,NULL);
      - pthread_rwlock_rdlock(&lock);
      - pthread_rwlock_wrlock(&lock);
      - pthread_rwlock_unlock(&lock);

Proxy

- **Your proxy must be robust**
  - Cannot crash for any malformed/bad input
    - Assume the user is an idiot
    - Be wary of malformed web addresses, and in general, requests
  - Memory management
    - Free what you malloc
    - Webservers like proxy will run for a long time, and memory leaks will actually add up

Proxy

- **Test extensively!**
  - There is no autograded feedback for Proxy
  - Use your proxy with Firefox for visual feedback
  - Try everything you can think of to break your program
  - If you have questions about what should/shouldn't be working on your proxy, come talk to us
- **Start early**
  - Not as time-consuming as malloc
  - Collaborating can be difficult
  - *Test extensively!*