Recitation 14: Proxy Lab Part 2

Instructor: TA(s)

Outline

- Proxylab
- Threading
- Threads and Synchronization
- PXYDRIVE Demo

ProxyLab

- **Checkpoint is worth 1%, due Thursday, Nov. 29th**
- **Final is worth 7%, due Thursday, Dec 6th**
- **You may use at most one grace / late day for each phase**
  - Last day to submit checkpoint: Friday, Nov. 30th
  - Last day to submit final: Friday, Dec 7th
  - There will be no extensions!

- **You are submitting an entire project**
  - Modify the makefile
  - Split source file into separate pieces

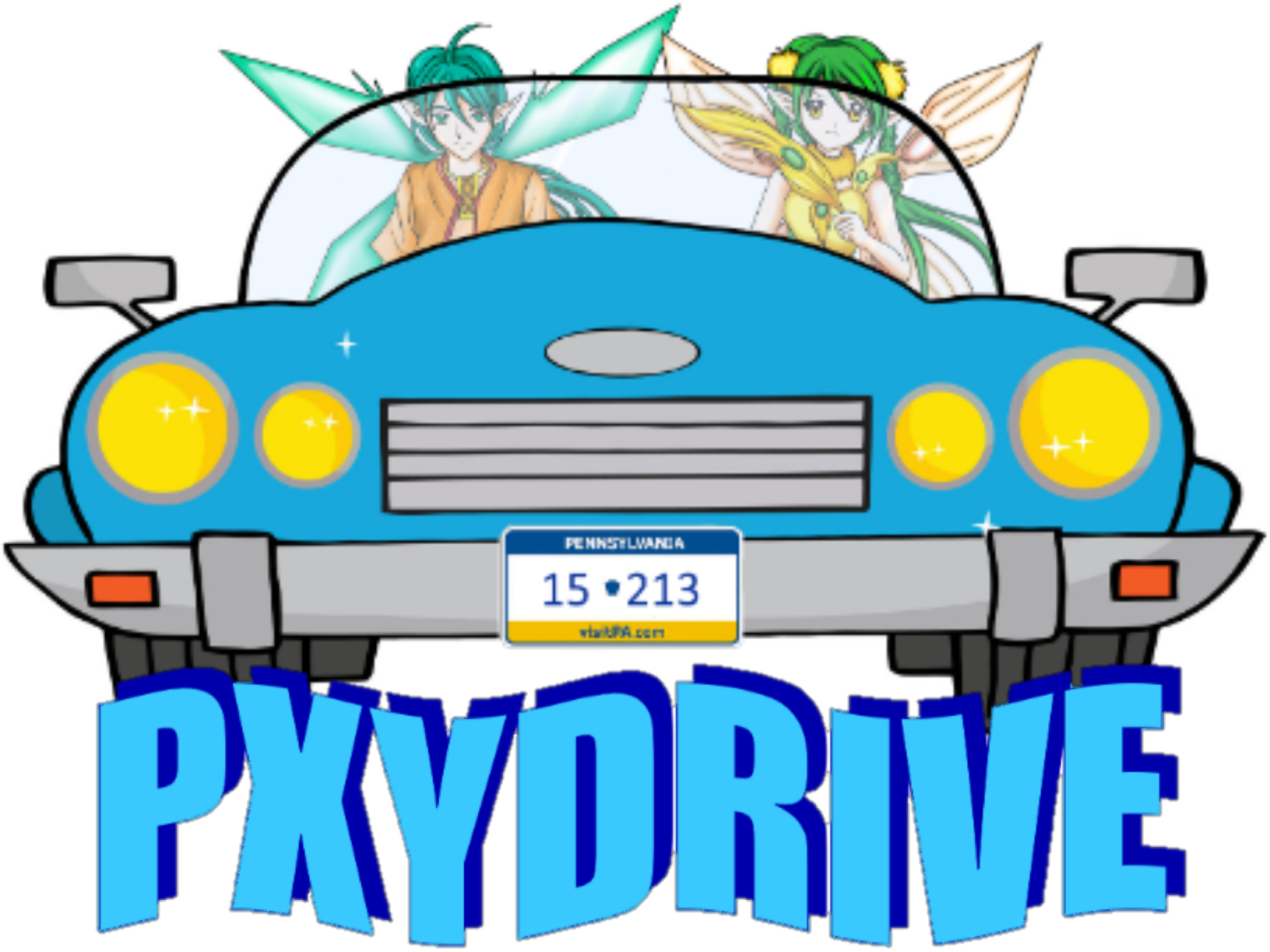
- **Submit regularly to verify proxy builds on Autolab**
- **A proxy is a server process**
  - It is expected to be long-lived
  - To not leak resources
  - To be robust against user input

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

- **Multi-threaded cache design**
  - Need to have multiple readers or one writer
  - Be careful how you use mutexes – you do not want to serialize your readers
  - Be careful how you maintain your object age

- **Tools**
  - Use **PXYDRIVE !**


PENNSYLVANIA
15 • 213
visitPA.com
PXYDRIVE

Some practice

- Get the tarball

- `$ wget http://www.cs.cmu.edu/~213/activities/pxydrive-tutorial2.tar`
- `$ tar -xvf pxydrive-tutorial.tar`
- `$ cd pxydrive-tutorial`

PXYDRIVE Tutorial 1

- **What happens when you haven't implemented a concurrent proxy and are expected to handle multiple requests?**
- Open **`basic-concurrency.cmd`**

PXYDRIVE Tutorial 1

- **`>generate random-text1.txt 2K`**
  - Generates a 2K text file called *random-text1.txt*
- **`>generate random-text2.txt 4K`**
  - Generates a 4K text file called *random-text2.txt*
- **`>serve s1`**
  - Launches a server called *s1*
- **`>request r1 random-text1.txt s1`**
  - Requests r1 from s1
- **`>request r2 random-text2.txt s1`**
  - Requests r2 from s1
- **`>wait *`**
  - Waits for all transactions to finish
  - Needed in the trace, not in the command-line

PXYDRIVE Tutorial 1

- **`>respond r2`**
  - Respond to client with r2 **(Out of order)**
- **`>respond r1`**
  - Respond to client with r1
- **`>trace r1`**
  - Traces the transaction *r1*
- **`>check r1`**
  - Checks the transaction *r1*
- **`>trace r2`**
  - Traces the transaction *r2*
- **`>check r2`**
  - Checks the transaction *r2*

PXYDRIVE Tutorial 1

- Run trace with –f option:

- ```
  $ ./pxy/pxydrive -p ./serial-proxy
    -f basic-concurrency.cmd
  ```
  ```

What went wrong?

```
>request r1 random-text1.txt s1
Client: Requesting '/random-text1.txt' from angelshark.ics.cs.cmu.edu:27329
>request r2 random-text2.txt s1
Client: Requesting '/random-text2.txt' from angelshark.ics.cs.cmu.edu:27329
>wait *
ERROR: Warning.  1 events still pending after timeout of 3000 milliseconds
  Event[Request r2 requesting TIME=1.064 SERVER = s1  URI = /random-text2.txt ]
>respond r2
ERROR: Invalid request ID 'r2'
>respond r1
Server responded to request r1 with status ok
```

```
--------------------------------------------------------------------------------
Reponse NOT sent by server
--------------------------------------------------------------------------------
Response NOT received by client
--------------------------------------------------------------------------------
```

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_detatch(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);
}
```

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;
}
```

PXYDRIVE Tutorial 2

- What happens when we pass a statically allocated connected descriptor to the peer thread in our proxy?

```
connfd = Accept(listenfd, (SA *)&clientaddr,
                &clientlen);
Pthread_create(&tid, NULL, thread, &connfd);
```

- $ ./pxy/pxydrive -f mixed-concurrency.cmd
  -p ./static-concurrent-proxy

What went wrong?

```
ERROR: Request r5 generated status 'error'.  Expecting 'ok' (Got empty response for URL request http://angelshark.ics.cs.cmu.edu:23346/random-text5.txt)
>check r6
ERROR: Request r6 generated status 'error'.  Expecting 'ok' (Source file is 12000 bytes long.  Response file is 12637 bytes long)
>respond r4
Server responded to request r4 with status ok
>respond r2
```

```
*** Error in `./static-concurrent-proxy': double free or corruption (out): 0x00007ffdde90fe00 ***
*** Error in `./static-concurrent-proxy': double free or corruption (out): 0x00007ffdde90fe00 ***
```

- This can happen due to a race condition between the assignment statement in the peer thread and the `accept` statement in the main thread!
- This can result if the local `connfd` variable in the peer thread gets the descriptor number of the next connection
- Make sure to **dynamically** allocate memory for each connected descriptor returned by `accept`!

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

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?

```
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 RWLocks: 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_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

1. **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.**

PXYDRIVE Tutorial 3

- Debugging a proxy that suffers race conditions
- Remember that one of the shared resource for all the proxy threads is the cache
- `$ ./pxy/pxydrive -f caching.cmd -p ./race-proxy`

- `Do take some time understanding the trace file for this tutorial`

What went wrong?

```
# Make sure initial requests have not been evicted
>request r01n random-text01.txt s1
Client: Requesting '/random-text01.txt' from angelshark.ics.cs.cmu.edu:10401
>request r02n random-text02.txt s1
Client: Requesting '/random-text02.txt' from angelshark.ics.cs.cmu.edu:10401
>request r03n random-text03.txt s1
Client: Requesting '/random-text03.txt' from angelshark.ics.cs.cmu.edu:10401
>wait *
>trace r01n
== Trace of request r01n ============================================================
Initial request by client had header:
GET http://angelshark.ics.cs.cmu.edu:10401/random-text01.txt HTTP/1.0\r\n
Host: angelshark.ics.cs.cmu.edu:10401\r\n
Request-ID: r01n\r\n
Response: Deferred\r\n
Connection: close\r\n
Proxy-Connection: close \r\n
User-Agent: CMU/1.0 Iguana/20180704 PxyDrive/0.0.1\r\n
\r\n
------------------------------------------------------------------------------
Message received by server had header:
GET /random-text01.txt HTTP/1.0\r\n
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:45.0) Gecko/20100101 Firefox/45.0\r\n
Connection: close\r\n
Proxy-Connection: close\r\n
Host: angelshark.ics.cs.cmu.edu:10401\r\n
Request-ID: r01n\r\n
Response: Deferred\r\n
\r\n
------------------------------------------------------------------------------
Reponse NOT sent by server
------------------------------------------------------------------------------
Response NOT received by client
------------------------------------------------------------------------------
Response status: ok
  Source file in ./source_files/random/random-text01.txt
Request status:  requesting
>check r01n
ERROR: Request r01n generated status 'requesting'.  Expecting 'ok
```

What went wrong?

- We realize that resources r01n (may be different in your case), was expected to be cached
- Let's understand this scenario with two threads running concurrently on the proxy.
- T1: At the time of check, T1 sees that it has cached the requested object.
- Another thread, say T2: Is trying to add a new object to the cache and is performing an eviction. This thread could possibly delete the object from the cache after the time of check but before T1 sends the cached object to the requesting client (time-of-use)
- This is an example of the Time-of-check Time-of-use race condition

PXYDRIVE Tutorial 4

- Debugging a proxy that suffers a deadlock
- Run the same trace but with another faulty proxy
- `$ ./pxy/pxydrive -f caching.cmd -p ./deadlocked-proxy`

What went wrong?

```
>#  Check that have initial requests in cache (and mark them as used)
>request r01c random-text01.txt s1
Client: Requesting '/random-text01.txt' from angelshark.ics.cs.cmu.edu:7249
>request r02c random-text02.txt s1
Client: Requesting '/random-text02.txt' from angelshark.ics.cs.cmu.edu:7249
>request r03c random-text03.txt s1
Client: Requesting '/random-text03.txt' from angelshark.ics.cs.cmu.edu:7249
>wait *
ERROR: Warning.  2 events still pending after timeout of 3000 milliseconds
  Event[Request r03c requesting TIME=6.355 SERVER = s1  URI = /random-text03.txt ]
  Event[Request r02c requesting TIME=6.355 SERVER = s1  URI = /random-text02.txt ]
```

What went wrong?

- We can notice a few timeout events and also some threads were waiting for the event that caused timeout.
- Let's consider two proxy threads T1 and T2 as usual.
- Suppose T1 holds a lock on a shared resource (could be the cache in our case) and never releases it. (you might have missed to perform pthread_unlock ! Or might have messed with the order of locking and unlocking)
- Another thread, say T2: Is trying to hold a lock on the same resource. (worse condition could be that it is already holding a lock on another shared resource that T2 needs). T2 waits for T1 to release the lock on the first resource and T1 in turn waits for T2 to release lock on that another resource.

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.