Proxy: Concurrency & Caches

Elie Krevat
(with wisdom from past terms)
Outline

- Proxy Lab logistics
- Concurrency and multi-threading
- Synchronization with semaphores
- Caching objects in your proxy
Proxy Lab logistics

- Late days = min(3, partner1, partner2)
- Both partners should hand in code
- Test your proxy well!
  - You may share testing ideas, not code
- No autograde, so schedule a time slot to demonstrate your proxy with a TA
Concurrency options

1. Processes
   ✦ Fork a child process for every incoming client connection
   ✦ Difficult to share data among child processes

2. Threads
   ✦ Create a thread to handle every incoming client connection
   ✦ Our focus today

3. I/O multiplexing with Unix select()
   ✦ Use select() to notice pending socket activity
   ✦ Manually interleave the processing of open connections
   ✦ More complex!
Processes vs. threads

- Processes and threads are similar:
  - Have logical control flow
  - Run concurrently with others
  - Context switched

- Processes are more expensive
  - Don’t share code and data like threads

- Threads enable sharing (except stack?)
Why we love threads

- Divide and conquer
  - Split a large array among many threads
- Wait and dispatch
  - Wait on new req, spawn thread, wait again
- Interleave tasks in parallel
  - Copy file while updating progress bar
- Threads are fast because share memory
Why we hate threads

- Coordinating threads can be difficult
- Unintended memory sharing
- Debugging becomes much harder
  - Many threads with different interleavings
- Race conditions and more!
Multiple threads

- Multiple threads can be associated with a process
- Each thread has its own logical control flow (instruction flow)
- Each thread shares the same code, data, and kernel context
- Each thread has its own thread ID (TID)

Thread 1 (main thread)
- stack 1
- Thread 1 context:
  - Data registers
  - Condition codes
  - SP1
  - PC1

Shared code and data
- shared libraries
- run-time heap
- read/write data
- read-only code/data
  - 0

Thread 2 (peer thread)
- stack 2
- Thread 2 context:
  - Data registers
  - Condition codes
  - SP2
  - PC2

Kernel context:
- VM structures
- Descriptor table
- brk pointer
Posix threads (pthreads)

- Creating and reaping threads
  - pthread_create
  - pthread_join
  - pthread_detach
- Determining your thread ID
  - pthread_self
- Terminating threads
  - pthread_cancel
  - pthread_exit
  - exit [terminates all threads]
  - return [terminates current thread]
Creating threads

```c
/* in main() */
while (1) {
    clientlen = sizeof(clientaddr);
    connfdp = Malloc(sizeof(int));
    *connfdp = Accept(listenfd,(SA *)&clientaddr,&clientlen);
    Pthread_create(&tid, NULL, thread, connfdp);
}

/* thread routine */
void *thread(void *vargp) {
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo_r(connfd); /* thread-safe version of echo() */
    Close(connfd);
    return NULL;
}
```
Issues with threads

- Joinable vs. detached threads
  - Rogue threads can conflict with others
  - But detached threads can’t be reaped/killed by others (automatic)
    - Joinable by default, call pthread_detach()
- Unintended memory sharing
  - Shared resources need synchronization
- Threads must use thread-safe functions
Semaphores and mutexes

- What if two threads read/free cache at same time?
- Want to announce “I’m changing cache, stay away!”
- Use semaphores and mutexes
  - Semaphores are counters for shared resources, used with atomic test & set operations
  - Think of a mutex as a binary semaphore
    ```
    Lock_mutex(cache_mutex)
    //fiddle with the cache...
    Unlock_mutex(cache_mutex)
    ```
  - Code between lock/unlock is atomic
  - Still must protect right code areas, use consistent locking
Thread-safe system calls

- All functions in the Standard C Library are thread-safe
  - Examples: `malloc`, `free`, `printf`, `scanf`
- Most Unix system calls are thread-safe, some exceptions

<table>
<thead>
<tr>
<th>Thread-unsafe function</th>
<th>Reentrant version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asctime</code></td>
<td><code>asctime_r</code></td>
</tr>
<tr>
<td><code>ctime</code></td>
<td><code>ctime_r</code></td>
</tr>
<tr>
<td><code>gethostbyaddr</code></td>
<td><code>gethostbyaddr_r</code></td>
</tr>
<tr>
<td><code>gethostbyname</code></td>
<td><code>gethostbyname_r</code></td>
</tr>
<tr>
<td><code>inet_ntoa</code></td>
<td>(none)</td>
</tr>
<tr>
<td><code>localtime</code></td>
<td><code>localtime_r</code></td>
</tr>
<tr>
<td><code>rand</code></td>
<td><code>rand_r</code></td>
</tr>
</tbody>
</table>
gethostbyname() thread-safe?

- Resolving DNS hostnames is not thread-safe
- Fix 1: Pass pointer to struct in function
- Fix 2: Lock & copy (caller frees mem)

```c
struct hostent
*gethostbyname(char name) {
    static struct hostent h;
    <contact DNS and fill in h>
    return &h;
}

hostp = Malloc(...));
gethostbyname_r(name, hostp, ...);

struct hostent
*gethostbyname_ts(char *p) {
    struct hostent *q = Malloc(..);
    P(&mutex); /* lock */
    p = gethostbyname(name);
    *q = *p;   /* copy */
    V(&mutex);
    return q;
}
```
Avoid common lock hazards

- Don’t hold a lock while making sys call
  - May be killed before unlocking
- Don’t protect large complicated code blocks with mutex
  - Only the important areas
  - Reduces contention and improves perf
- Correct locking: only unlock mutexes you already own, lock ones you don’t
The Proxy Cache

- Proxy should cache complete HTTP response
  - Include headers, bucketize by same URL request
- Don’t need to parse response
  - Real proxies might
- Don’t use str funcs on binaries (Lec13 sld 48)
- Don’t cache if size(resp) > MAX_OBJECT_SIZE
- Use LRU eviction policy (accessed earliest)
- Evict when run out of cache space
  - size(cache) + size(new_entry) > MAX_CACHE_SIZE
Cache synchronization

- One cache shared by all proxy threads
  - Must carefully control access
- What operations should be locked?
  - add_cache_entry
  - remove_cache_entry
  - lookup_cache_entry
- Many readers can peacefully co-exist, but if writer arrives it must synchronize
Threading is an efficient way to gain concurrency, but be careful!

Need to synchronize multiple threads when accessing shared mem structs
- Use semaphores/mutexes
- Write and call thread-safe code

Symptoms of concurrency problems
- If proxy hangs, you may be forgetting to unlock
- If cache corrupted, you may be forgetting to lock