CS 61
Lecture 24 Scribe Notes
26 November 2013
Benjy Levin & Tianyu Liu

(Demonstrates Assignment 6: Adversarial Network Pong)
Officially due last day of classes but class wide extension for a week until the end of reading period.
All other problem sets and extra credits are also due by the end of reading period.

Today: Handling threaded parallelism

Setting up a service server that handles multiple connections from multiple clients using multiple threads.

Recap:
Service server 5:
The goal of the service server program is to provide a network interface by which clients can query the mappings between services and port numbers.

There is a library function in the C library, but it's slow if called multiple times.
This server caches everything in memory and then responds to multiple requests.

It does this by:
1. Creating a listening socket
2. Accepts connections on that socket
3. For each connection to that socket, creates a new thread to handle requests

handle_connection - Reads line from input, removes white space, and calls

my_getservbyname (function that looks up the service name in the cache), print result out onto the same socket.

./serviceclient - works for one request at a time

./serviceblaster - opens multiple connections all at the same time - and the modified version now sends requests and gets responses (pretty much simultaneously)

Order that requests come back are arbitrary because the order that the threads are executed are arbitrary - potential race conditions

Tries 3 threads for 3 queries:./serviceblaster - n 3 discard http com-bardac-dw
The last two connection got back first with wrong results (Only #3 fd has the correct result)
The bug doesn't recur if rerun from same server.

Opens the connections in order, and Unix always uses the first available fd# – if you are opening and closing fd this rule doesn't apply but here the first connection gets fd #3 (0/1/2/ already used for stdin stdout stderr), open them in that order, write requests in that order, and then read in whatever order things come back.

Question: * Is this related to caching?
Answer: How would we figure this out?
Experiment: How about run more threads, and see what happens.
Exploratory phase of debugging

- 4 threads for a fresh service create the same bug.
- 100 threads also create the same bug.

How does the cache work?
servicecache.c defines the cache:
handle_connection -> my_getservbyname -> only entrance into the cache.
my_getservbyname calls to serivcecache.c which creates a cache upon first request (when the cache has not yet been created)

*** What happens when two threads simultaneously call my_getservbyname?**
We can use a **progress graph** to think about this.
(Progress graph is only sensible in two dimensions, but a true progress graph would have number of dimensions = number of threads)

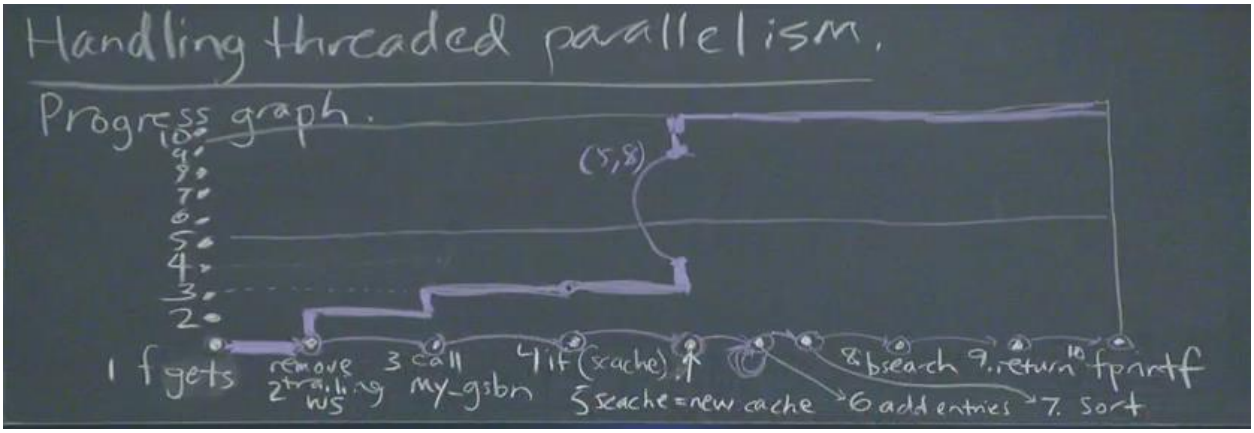
Each axis represents the program counter for exactly one thread
Therefore, a location in the progress graph tells you where every threads program counter is located.

List out relevant program counter states for the serviceserver:
1) fgets (read from the buffer)
2) remove trailing space
3) call my_getservbyname
4) if(scache) available, (6-10) create a search entry and call binary search to lookup the entry and return the result
5) scache = new_cache
6) add entries
7) sort
8) binary search to look up the entry
9) return result
10) fprintf in the server

In this server, every thread is running this logical flow, so it is possible that all threads are at a different position in the progress graph

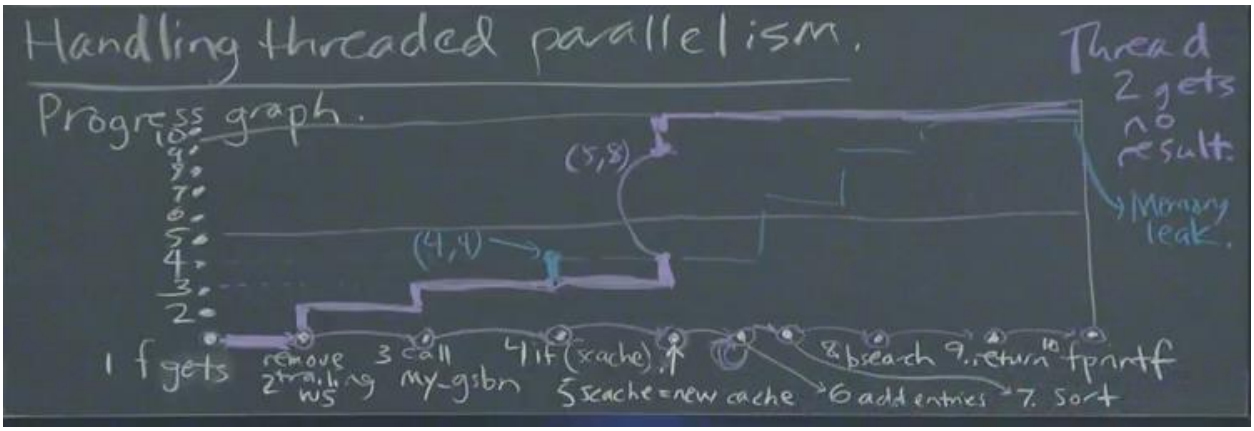
What is interesting about this progress graph Is that we can now see how two threads interact with each other and their possible interleavings.

Each graph presents one possible outcome.
With a race condition, the threads can run in any order.



In the case of the example above, while thread #1 is filling up the cache, thread #2 sees that scache has a non-null value, and then searches the empty cache (that thread #1 has not yet populated), and returns 0. Afterwards, thread #1 runs and returns the right answer.

***What other bug could this progress graph introduce?**
If the two threads found that scache is NULL at the exact same time, two caches would be created, and there will be a memory leak since there is only one global variable, and therefore only one of the caches will be stored in this variable and available to other threads. This bug would not be exposed by serviceblaster as both threads would return correct results.
This occurs at point (4,4). (see below)



***Question: Is there a way to get two threads confused with what they are looking for?**
Answer: What is logically local to a thread? Threads have different stacks – so they have different local variables and different program counters- the buffer is a local variable, if the buffer were made a global – now there is only 1 copy, and would result in some very “blehg” behavior.

***Question: Are we assuming that all the individual steps are all atomic?**
Answer: That is logically what the progress graph is assuming. The progress graph is an abstraction, but this is not true in reality. What is atomic? System calls are atomic. Many instructions, but not all, have atomic effect - we will talk more about this next time.

On the other hand, there are parts of this graph where the atomicity does not matter. In particular, anything that a thread does to its stack, to its registers and its local variables, it doesn't matter if it is atomic, because no other threads are effecting that memory.

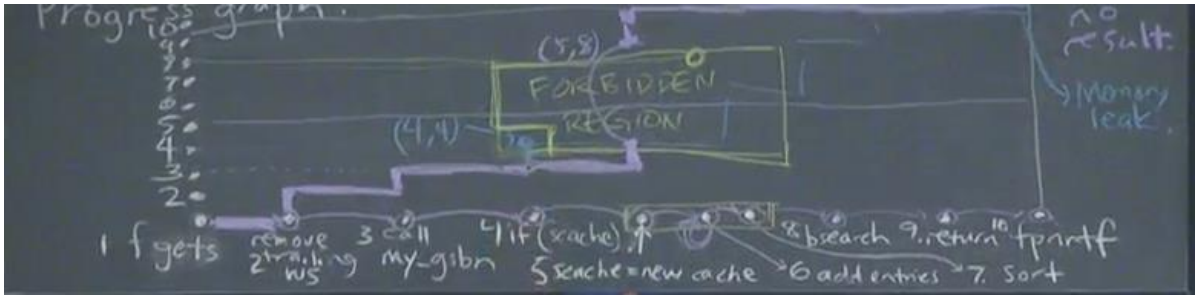
What makes this graph problematic? What is the basis for a race condition?
Memory Access Conflict – occurs when two or more threads access the same data, and at least one of those accesses is a **write**. If all threads are reading it doesn't affect the global variable itself, but if there is a write, that will induce nasal demons (undefined behaviour).

We need to use **synchronization objects** to enforce a specific order, to ensure that the system never does the wrong thing.

Returning to an earlier question: Why is it happening every time?
Answer: The server file is enormous, opening the file and copying that file into memory takes a lot of time. During this initial window of time, the thread that's reading the file blocks, and the other threads quickly execute and don't do any blocking operations at all, because they don't try open the file and read from it.

Forbidden region on the graph is a place where if the program counters of the two threads are there at the same time, we will have a bug.

Data point that may cause crashes:
1) Binary search while adding entries (bad point – might be accessing memory that has just been freed)
2) Involve modifying the shared memory cache (a global variable). One thread in 5 through 7 , other thread in 4 through 7. (without (4,4))



We need to figure out a way to force the system to never pass through this forbidden region.

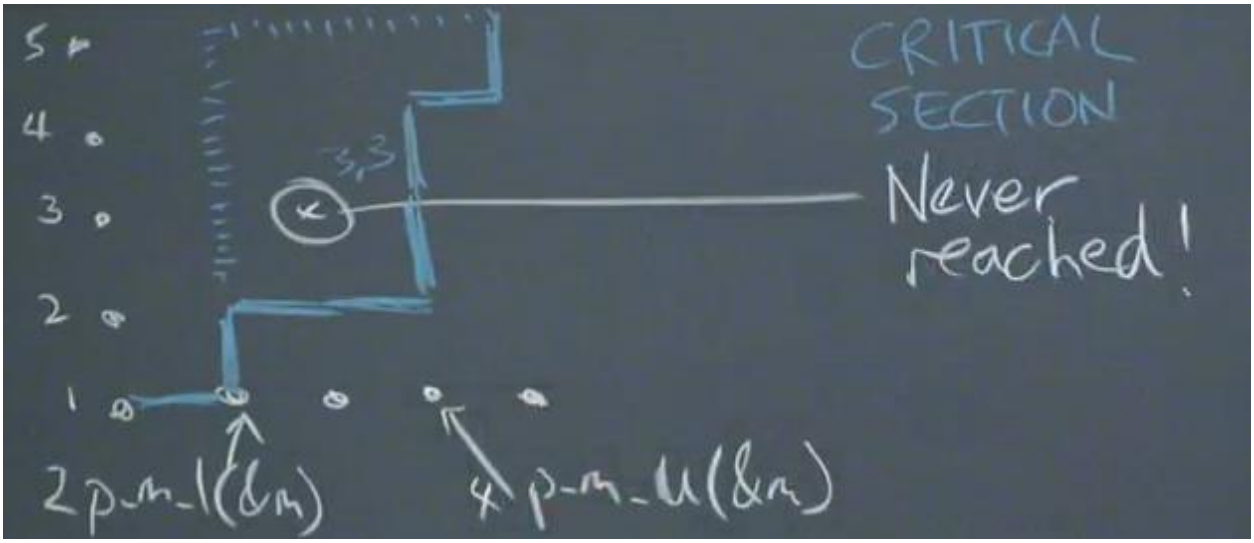
Quick fix: Call my_getservbyname before accepting new connections – prepopulating cache before accepting any connections. But this is specific to our program because once the cache is loaded into memory; it is completely stable - we never modify it. Next we will examine things that we can modify.

In general, we need a method that will actively prevent code from entering the forbidden region.

Primitive that threading packages give us access to:
Mutual Exclusion lock, aka **mutex lock**: Synchronization object with two operations:
lock and *unlock*. Designed to work correctly even for when access conflicts occur. Designed to work around the problems of concurrent programming.

Pthread = POSIX thread - threading package we are using.
3 important functions
pthread_mutex_t:
pthread_mutex_init(&m,NULL).
pthread_mutex_lock(&m);
pthread_mutext_unlock(&m);

Progress graph:



(1)(2 -lock)(3)(4 - unlock)
Symantics of the mutex enforce a forbidden region that will never be reached (3,3).
Thread two cannot execute until thread 1 unlocks the mutex
Multiple valid paths around the forbidden region
This is why they it is called a mutex lock - there is a portion of the code that threads are mutually excluded from! Also known as **critical section**.

How would we fix this serviceserver using mutex locks?
Where do we store the mutex lock?
Whole point is to coordinate between threads
So needs to be placed in global storage so that all threads have the same lock (otherwise defeat the purpose of mutex locks and coordination)

```
pthread_mutex_lock(&mutex);
If (!the_scache)  //4
    scache_create(); // 5-7
pthread_mutex_unlock(&mutex);
```

Put mutex lock around the if condition and turn it into a critical section.
If we put the lock after step 8 (binary search that does not contain any writes), we will limit concurrency, which will greatly slow down other threads, and thus the overall speed.

Serviceserver-06:
Like serviceserver-05 but it is attempting to enforce a maximum number of 100 threads at any one time.
Increments a global variable (n_connection_threads) when the thread starts, decrements it when the thread ends, and in the loop that accepts new connections, have a while loop that checks if there are too many threads, and waits until one exits.

Access conflicts:
++/ -- to n_connection_threads
but the machine essentially implements it as:
n_connection_threads = n_connection_threads + 1
Which involves a load, modification of registers and a store, and can therefore have interleavings because it is not atomic.
So we need…… MUTEX!

Lock before the access conflict, and unlock after access conflict - before and after the incrementing and decrementing of n_connection_threads.

But we also have an access conflict in the while loop that is checking for the number of connections.

But we cannot just lock before the while loop and unlock after the while loop, because after it enters the while loop holding the mutex, and one of the active threads ends and wants to decrement the number of connections, that thread cannot acquire the mutex to do so, because the while loop still has it! **DEADLOCK**
Therefore the program will be dead after we reach the max number of threads, because it will never be able to create a new connection and will wait forever.

Deadlock: Circular wait on mutual-exclusion locks. (In our case it blocks every thread)
We can find a circle in the thread ID's where each thread is waiting for another in the circle
main thread is waiting for connection thread to exit but connection thread is waiting to acquire mutex lock that main thread holds.

Need a way to avoid this situation.

Side note – if both threads hit acquire lock at same time – computer will make a random choice as to which will get the lock, but will only choose one.

Another **synchronization object**: **conditional variable**: It is a **condition detection object** (not a mutual exclusion object) that has 2 operations:
wait (for a condition to become true) and **signal** (that a condition has become true).

pthread_cond_wait(&cond, &mutex);

The mutex starts out in the locked state.
Atomically releases the mutex and checks the condition, blocks until the condition is true, and then re-acquires the mutex.
The condition represents the case when some thread has died:

pthread_cond_signal(&cond);

The "coolest" thing about condition variables is that they **block**, so the serviceserver is **not** in a **polling** state (not repeatedly checking, using all of the CPU).

Condition variables lets you wait for something without race conditions.