Deadlock Demolition

Entire Assignment due **2019-03-06 23:59**
Graded files:

- libdrm.c

Content

Learning Objectives

The learning objectives for Deadlock Demolition are:

- Synchronization Primitives
- Deadlock Detection
- Resource Allocation Graph

Overview

This week, you will be building a library for deadlock-resistant mutex (drm) locks. A drm should behave like a pthread_mutex_t but not allow itself to be locked by a thread if the attempt to lock it would result in deadlock. We will use binary semaphore notation for your lock's functions: i.e. post is equivalent to unlocking and wait is equivalent to locking.

You will be writing four functions:

- `drm_t *drm_init();`
- `int drm_post(drm_t *drm, pthread_t *thread_id);`
- `int drm_wait(drm_t *drm, pthread_t *thread_id);`
- `void drm_destroy(drm_t *drm);`

To detect deadlock, you will need to maintain a Resource Allocation Graph and be able perform cycle detection on it. See this page (http://cs241.cs.illinois.edu/coursebook/Deadlock#resource-allocation-graphs) for more information about Resource Allocation Graphs.

The general flow of the algorithm for lock and destroy are self-explanatory, they just need to clean up resources. The fun happens in wait and post (lock and unlock respectively). When a thread posts

- Check to see if the vertex is in the graph. If it is not, return without unlocking the mutex in the `drm_t` .
- Otherwise if an edge from the drm to the thread exists, remove the edge and unlock the `drm_t`

For drm wait

- Add the thread_id value, or pointer you can assume the pointer is unique to the thread, to the graph if not already present
- If locking the mutex would cause a deadlock, then fail otherwise lock and return success
  - One case that would deadlock is a thread trying to lock a mutex it already owns. Think about how to factor this in
  - Another case is adding that edge would cause a cycle in the resource allocation graph. You can do this by adding the edge to the resource allocation graph, checking for deadlock and if no deadlock occurs, then

locking and returning.

Since your Resource Allocation Graph will need to represent both drm locks and threads as vertices, use a shallow graph (see graph.h). You will need to lazy initialize a global graph in the `drm_init` function. Here is an example of lazy initialization with an integer variable:

```
int *g
  void init(){
    if(g == NULL)
      g = malloc(sizeof(int))
  }
```

NOTE: the provided graph data structure is not thread-safe.

Good luck!

Testing

Testing is ungraded, but highly recommended

Please test this on your own in a variety of ways. Be careful of race conditions! They can be hard to find! We've given you a `libdrm_tester.c` file to write tests in.

Testing Tips

See man pages for pthread_self(3) for passing a pthread_t pointer to drm_post and drm_wait.

You may want to simulate situations where deadlock would occur using a standard mutex lock.

Consider logging important events inside of your functions.

Thread Sanitizer

We have another target executed by typing `make tsan` . This compiles your code with Thread Sanitizer.

ThreadSantizer is a race condition detection tool. See this page (http://cs241.cs.illinois.edu/coursebook/Background#tsan) for more information

We will be using ThreadSanitizer to grade your code, but we will ONLY test for data race warnings, NOT any other warning type.

Helpful Hints and Notes

- Make sure you thoroughly test your code! Race conditions can be hard to spot!
- Attempting to visualize your code or diagram it in certain cases can sometimes be a huge aid and is highly recommended!

ANYTHING not specified in these docs is considered undefined behavior and we will not test it For example, calling drm_destroy on a locked drm. Doing the same thing on a pthread_mutex_t is is undefined behavior, and is for drm's as well. *We will not test undefined behavior*.