Project 4: Deadlock Prevention[1]

This project deals with deadlocks where you will design a smart locking library that prevents deadlocks.

Marking

The project will be graded based on correctness (90%) and correct memory management (10%). For correctness, you can assume that there will be more than 45 test cases for each project, and every test failure will cause one point deduction. For correct memory management, you will lose 5% for memory leak and 5% for memory error (such as incorrect memory access). The correct management will be checked by valgrind.

Overview

In this project you will implement a smart locking library that prevents deadlocks. Deadlocks will be prevented by ensuring that circular wait never occurs. The library will internally rely on pthread mutex locks.

Design

The smart lock structure will be defined as follows:

```
typedef struct {
    pthread_mutex_t mutex;
    /* Add other variables */
} SmartLock;

void init_lock(SmartLock* lock) {
    pthread_mutex_init(&(lock->mutex), NULL);
}

int lock(SmartLock* lock) {
    pthread_mutex_lock(&(lock->mutex));
    return 1;
}

void unlock(SmartLock* lock) {
    pthread_mutex_unlock(&(lock->mutex));
}

/*
 * Cleanup any dynamic allocated memory for SmartLock to avoid memory leak
 * You can assume that cleanup will always be the last function call
 * in main function of the test cases.
 */
void cleanup() {

}
```

As shown above, `SmartLock` will internally hold a `pthread_mutex_t` instance for correct locking and unlocking mechanism. You need to update `init_lock()`, `lock()`, `unlock()` and `cleanup()`

functions to incorporate the deadlock prevention logic in `SmartLock`. While APIs for `init_lock()` and `unlock()` appear straightforward, the `lock()` function returns an int value. This value should either be `0` or `1` where `1` indicates that the lock got acquired and `0` indicates that the lock was not acquired (in order to prevent deadlocks). This means, even if the program calls lock, it is not always guaranteed to acquire the lock and hence, it must first check the return value before proceeding, as shown below:

```
while(lock(&mySmartLock) == 0);
```

In the above code, `mySmartLock` is guaranteed to be acquired when the while loop ends.

The `cleanup()` function will release any dynamically allocated memory of `SmartLock` (such as internal resource allocation graph described in the next paragraph) to avoid memory leaks. You can assume that `cleanup()` will always be the last function call in main function of the test cases, and valgrind will be used to detect memory leaks and memory errors during grading.

Deadlock prevention will be performed by ensuring that `lock()` does not result in a circular wait. As shown in Figure 4, when `lock()` is called, a dotted edge is represented in the resource allocation graph (RAG). If the allocation cannot be satisfied, the calling process is blocked (i.e., the solid request edge is maintained as shown in top case) or `0` is returned back to the calling process (i.e., request edge is not maintained as shown in the bottom case) so that the process does not block.

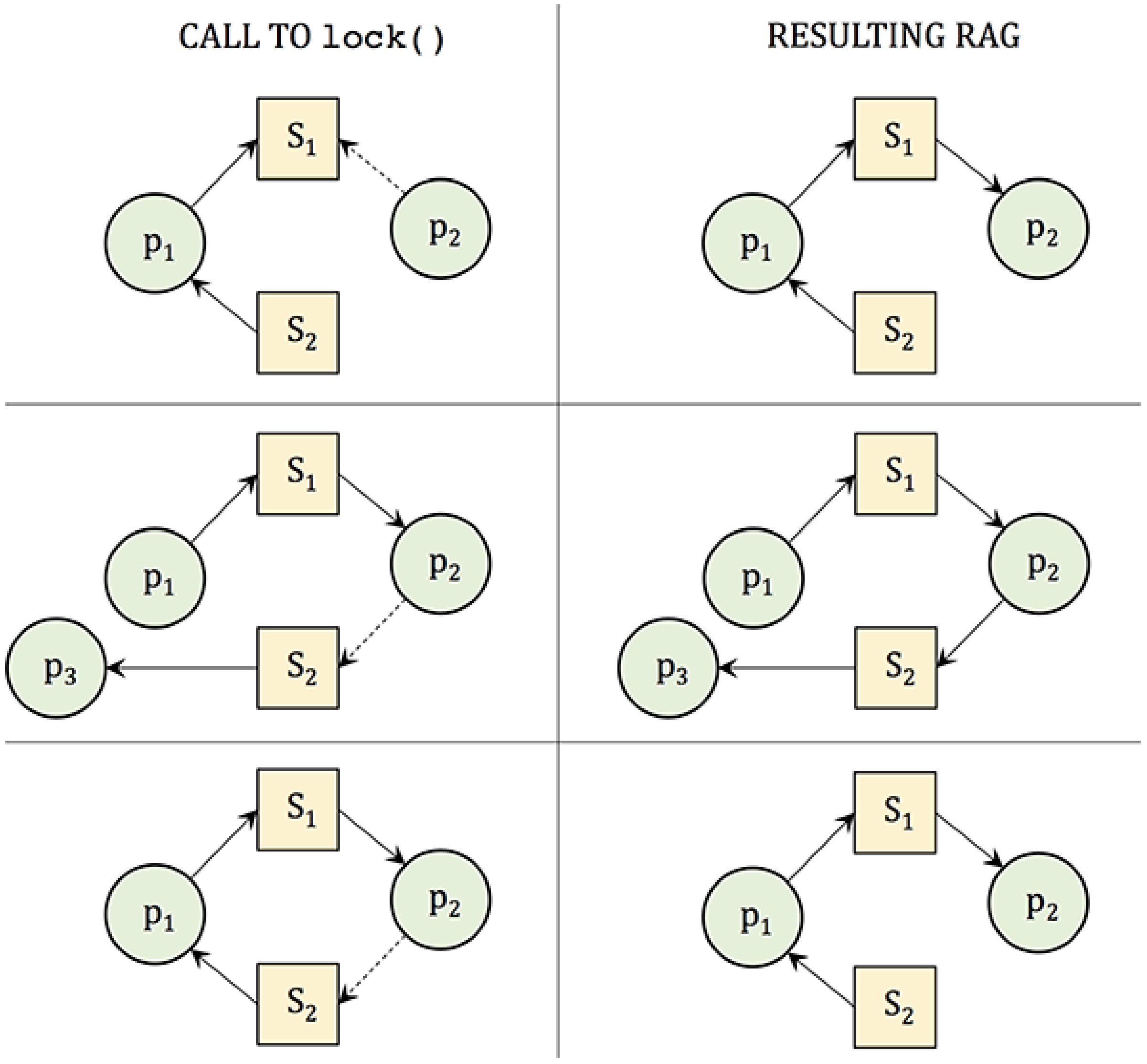

Figure 1: Behavior of SmartLock

Note that in order to perform cycle detection, you will need to represent each thread and each lock with a unique id. This can be achieved by using `pthread_self()` for threads and by assigning unique integer id for each `SmartLock` variable.

Submission

Submit an archive `deadlocks.tar.gz` of your code (including `main.c`) and make file (Sample Codes) on CourSys. We will build your code using your Makefile, and then run it using the command: `./locking`. You may use more than one .c/.h file in your solution, and your Makefile must correctly build your project. Please remember that all submissions will automatically be compared for unexplainable similarities.

1. Created by Keval Vora. top