

# CSIS0230A Principles of Operating Systems (Class A)

## Assignment 4

### Deadlock-free lock-like semaphores in PThread

Tutor: kmcheung@csis.hku.hk, Deadline Dec 10, 2003, 5:00pm.

Suppose you will soon write a program with multiple threads. The threads must coordinate to use a certain amount of resources. Each resource has a certain number of copies, which can be implemented using a semaphore initialized to a number  $n$ , the available number of copies of the resources. Whenever the program needs  $k$  copies of a resource, it will down the corresponding semaphore by  $k$ . Later the same process will up the semaphore to release the resource.

As in all synchronization problems, it is possible to have deadlocks. But in such an environment, using a fixed ordering scheme to implement deadlock prevention is not efficient (as the strict ordering scheme does not know about multiple resources, and hence will restrict access too often). You'd like to make use of the banker's algorithm to avoid deadlocks in a more efficient manner. In this assignment, you will write a C/C++ library<sup>1</sup> facilitating such "deadlock-safe resource reservation", and write programs to test your implementation.

#### 1. The scheme

To allow the banker's algorithm to be used, you will use your own implementation of semaphores (actually called "resource" in the description below), ignoring the semaphore implementation provided by the Pthread library. You will use Pthread library constructs (**mutexes** and **conditions**) to build your semaphores. Since Pthread semaphores are not univervally available, you decided to avoid using it. (You shouldn't need it anyway.)

The program that will call your library will follow a certain convention, so that your library gets the needed parameters for the banker's algorithm. The convention is described below. Here the functions with names starting with *safe\_res\_* will be provided by your library. They should all be callable from C: if you use C++, their definition should be prepended with **extern "C"**.

- All resources are created before any thread is created. To create a resource, the function

**int safe\_res\_create(unsigned int num\_copies)**

is called. The resource will then have *num\_copies* copies. If succeeded, the return value is a non-negative integer identifier that will be used later (in the *res\_id* argument of the functions below) whenever a resource operation is needed. The return value is  $-1$  if allocation fails. This can be due to two reasons: because the argument is larger than the  $INT\_MAX=2147483647$  limit<sup>2</sup>, or because some other resource operations has been invoked already. In such case, *errno* would be set to *EINVAL* (Invalid argument) and *EBUSY* (Device or resource busy) respectively.

- Upon creation, each thread will declare the maximum number of each resource that it needs. This is done by the following function.

**int safe\_res\_set\_max(int res\_id, unsigned int max\_needed)**

---

<sup>1</sup>Whether it is in form of a shared library, a static library or a set of object files is not significant in this assignment. All you need is to hand-in a *.c/.cc* file.

<sup>2</sup>This limit allows you to easily check overflows when increment and decrement the counts involving the number of copies of resources.

Here *res\_id* is the identifier of the resource, and *max\_needed* is the maximum number of copies of the resource that the thread will need. If this maximum is not set for a resource, the maximum is assumed to be 0. The function will check whether the change in maximum number of needed resources will leave the system in a deadlock unsafe state, and if so it refuses to set the new maximum number of resources. The return value is 0 if the maximum amount of resource is successfully set, and -1 if not. In the latter case *errno* is set to *EDEADLK* (Resource deadlock would occur) if the change would make the system deadlock-unsafe<sup>1</sup>; *EPERM* (Operation not permitted) if the more than *max\_needed* copies has already been allocated to the thread, or *EINVAL* (Invalid argument) if *res\_id* is not a resource identifier.

- Threads can then use

```
int safe_res_acquire(int res_id, unsigned int num_copies)
int safe_res_acquire_nonblock(int res_id, unsigned int num_copies)
```

to get and release *num\_copies* copies of the resource identified by *res\_id*. In case the resource cannot be obtained immediately without causing the system into a deadlock-unsafe state, the first “blocking” version will wait until the resource can be obtained, and the *\_nonblock* version will instead return an error immediately. If the function succeeds, it returns 0. If the function fails, it returns -1, with the *errno* variable set to *EPERM* (Operation not permitted) if the acquisition would make the thread exceeds the declared maximum; *EWOULDBLOCK* (Operation would block) if the acquisition would need waiting but *safe\_res\_acquire\_nonblock()* was called; or *EINVAL* (Invalid argument) if *res\_id* is not a resource identifier.

- After finish using the resource, the thread will call

```
int safe_res_release(int res_id, unsigned int num_copies)
```

to release the resource they have acquired. Here *res\_id* identifies the resource that is to be released, and *num\_copies* is the number of copies that is to be released. If successful the return value is 0, and if failed the return value is -1. The latter can happen if either *res\_id* is not a resource identifier, or the thread did not have acquired at least *num\_copies* of that resource. In either case the current allocation is not changed. The variable *errno* is then set to *EINVAL* (invalid argument) and *EPERM* (Operation not permitted) respectively for the above two cases.

- The thread can release all resources that it is holding by calling the following function.

```
void safe_res_release_all(void)
```

This releases all resources that had been acquired by the thread. You may assume that all threads will release all its resources before terminating.

- The banker’s algorithm is not very efficient, so 10% of the score will be allocated to minimizing the number of times that the algorithm is invoked.
- There is also a fairness problem: if you examine all sleeping threads in a fixed ordering, those later in the ordering might always be delayed because the threads earlier in the ordering acquires the resources available for them. An alternative scheme is to check the process

---

<sup>1</sup>This should happen only if either (1) the thread tries to change the maximum amount of resource when it is holding some resource, or (2) the thread tries to set the maximum amount to be larger than the number of copies of the resources as created by *safe\_res\_create()*.

in the order in which they start to sleep. 5% of the score will be allocated for this scheme.

## 2. Hints

- See the man pages for `pthread_create()` and `pthread_join()` to see how to manage threads. Also, don't forget to use `-lpthread` when compiling your program. If you use C++ when implementing the library, you will also need to use `-lstdc++` to link with the standard C++ library.
- Debugging a multi-threaded program can be very difficult. In general, try to print out a lot of messages until you are sure the part you write is correct. It may also help to define some signal handlers that will print out messages concerning the current state of the threads.
- You need to use mutexes and conditions to implement your scheme. Synchronization using mutexes and conditions feel quite different from synchronization using semaphores. When using semaphores, you view it as a data structure holding a concrete value, representing some real-world numbers directly. For example, in the consumer-producer problem of the textbook, the two semaphores directly represent the number of free and filled buffers.

Neither mutex nor conditions have any value associated. A mutex always allows at most one thread to locking it at the same time, and requires that the thread locking it must be the same as the thread unlocking it subsequently. See `pthread_mutex_init(3)`, `pthread_mutex_lock(3)` and `pthread_mutex_unlock(3)` for details.

Conditions correspond to *events* in real-world. A thread can **wait** for a condition (i.e., to wait for an event to occur). Any thread that makes this condition to occur should **signal** the condition. This wakes up one or all the threads waiting for the condition. To avoid the race condition that the condition is signalled just before another process wait for it, a condition should always be associated with a mutex. You should ensure that the mutex is locked whenever you wait and signal the condition. Waiting a condition will atomically unlock that mutex at the same time, and when the condition is signalled the mutex will be reacquired before signal completes. See `pthread_cond_init(3)`, `pthread_cond_wait(3)`, `pthread_cond_signal(3)` and `pthread_cond_broadcast(3)` for details.

- You need to keep a significant amount of information to implement the banker's algorithm, and these information are shared and needed by all the threads. You need to protect this shared data structure from concurrent access.
- The most important idea is that if a resource cannot be safely acquired immediately, the thread will go to sleep, waiting for somebody to wake it up. So each thread has a condition, and the `safe_res_release()` and `safe_res_set_max()` function signals these conditions to wake threads up. The easy implementation is to wake up all sleeping threads so that they all try to check whether the condition is safe enough for resource allocation. But you should think more carefully to avoid having to wake up those threads that is still unsafe, and avoid invoking the banker's algorithm at all if you already know that it is not safe yet. (More hint: if an allocation of thread 1 is found to be unsafe because the banker's algorithm cannot remove thread 1, 2 and 3 although all other threads are removed, what operation of which thread might make the allocation safe? What about some threads reducing the maximum need of a resource?)

### 3. Provided code

In the web page, you can find two files. The header file `safe_resource.h` should be included from both the library implementation file and the actual program that use the library. The test program `test_resource.c` test your library in one specific request sequence. It's primary intention is to show you how the functions are to be used, although it can be one of the tests for your program as well. You should develop more tests to try different request sequences, although you don't need to submit those.

### 4. Handin

Send 2 files for this assignment:

- The implementation of the library in `safe_resource.c` or `safe_resource.cc`, depending on the language that you choose to use.
- A file called `readme.txt` telling the names and IDs of the members (at most 2) in your group, and the known problems in your library. You should also write in this file the optimizations you have used to reduce the number of times the banker algorithm is invoked, and give reasons why you think it does not cause threads the miss their chance to obtain the requested resources.

### 5. References

- Textbook chapter 8, banker's algorithm.
- PThread thread management: man pages and info pages for functions described in Section 2.