UNIVERSITY OF COPENHAGEN
Department of Computer Science
Robert Glück glueck@diku.dk
Jost Berthold berthold@diku.dk

February 24, 2012

G-Exercise 3 for "Operating System and Multiprogramming", 2012

Hand in your solution by uploading files in Absalon before Mar.6

Exercise Structure for the Course

Please see the introduction on exercise 1 for information about the exercise structure.
Solutions to G-exercises are in general handed in by uploading an archive file (zip or tar.gz) to Absalon. When you upload your solution to Absalon, please use the last names of each group member in the file name and the exercise number as the file name, like so:
`lastname1_lastname2_G3.zip` or `lastname1_lastname2_G3.tar.gz`.
Use simple txt format or pdf for portability when handing in text, and comment your code.

About this Exercise

Topics

This exercise recapitulates thread/process synchronisation methods, needed in various parts of a multithreaded operating system. In the first task, you are requested to implement locks and condition variables for kernel threads in Buenos. The second task is about the CREW (concurrent read - exclusive write) problem. You are asked to implement different CREW variants using Pthreads and compare their behaviour using measurements.

What to hand in

Please hand in your solution in a single archive file (zip or tar.gz format) which contains:

1. A source tree for Buenos which includes your work on task 1 (as documented in the report).
2. A directory containing C code and a Makefile for task 2, as well as your measurement results of the different variants.
3. A report in pdf or txt format.
   Your report should discuss the design decisions you took for your lock and condition implementation (in addition to commenting the C code), and explain in detail how you measured the the behaviour of the CREW simulations for task 2 and what you observed.

Tasks

1. **Locks and condition variables for kernel threads in Buenos**

   The Buenos kernel provides an implementation of spinlocks (using assembly instructions, load-link and store-conditional, `kernel/_spinlock.S`), but holding spinlocks for a long duration is wasteful and unsafe to use with interrupts enabled. In order for easier kernel programming, we implement mutual exclusion locks that block threads when they cannot obtain the lock, and condition variables to synchronise on. Chapter 5 of the Buenos roadmap describes the pre-implemented mechanisms for synchronisation in Buenos: spinlocks and the sleep queue. Buenos also provides an implementation of semaphores, but *semaphores may not be used to solve this exercise.*

(a) Implement the following functions and corresponding types for handling locks in kernel threads:

- `int lock_reset(lock_t *lock);`
  Initialize an *already allocated* `lock_t` structure such that it can be acquired and released afterwards. The function should return 0 on success and a negative number on failure.
- `void lock_acquire(lock_t *lock);`
  Acquire the lock. A simple solution could use *busy-waiting*, but this is inefficient. In your solution, you should use the sleep queue to let kernel threads wait.
- `void lock_release(lock_t *lock);`
  Releases the lock.

Your task includes defining the type `lock_t` and fully specifying the operations.

(b) **Condition variables**

Implement the following functions and corresponding types for handling condition variables in Buenos. The implementation should use *Signal and continue* (described in the book [Silberschatz,p. 246], sometimes called MESA-semantics). This means that a thread calling `condition_signal` (S) continues executing and the waiting threads aren't started until (S) is finished or waits itself. This way the waiting threads can't know if the condition is still satisfied, but only that it once was.

- `void condition_init(cond_t *cond);`
- `void condition_wait(cond_t *cond);`
- `void condition_signal(cond_t *cond);`
- `void condition_broadcast(cond_t *cond);`

Your task includes defining the type `cond_t` and specifying the operations.

Your functions and types should be in `kernel/lock_cond.c` and `kernel/lock_cond.h`. In order to compile your files with the rest of Buenos you have to add `lock_cond.c` to the list FILES in `kernel/module.mk`.

2. **Pthreads-based Solutions for the CREW Problem**

The book and the lecture have presented basic solutions for Concurrent Read - Exclusive Write (CREW), a classical synchronisation problem in systems and database programming (see [Silberschatz,p.241f] and the slides). The CREW problem is to protect data from concurrent access when it is being modified. While read access may be granted to several threads concurrently, write access needs to be exclusive. The Pthreads API includes special read-write locks (`rwlock_t`) for this purpose, an example program is available on Absalon.

(a) Implement a basic CREW solution (as demonstrated in the lecture) with Phtreads locks and condition variables, as a test program which simulates a number of concurrent readers and writers. Explain in your report how you realise mutual exclusion for writers and concurrent read access.

(b) Basic solutions often lead to writer starvation: readers can access the protected data more easily and effectively block writers. Create a second version of your CREW simulation program, adding a mechanism which gives writers priority.

(c) Design a proper method to measure the behaviour of your CREW solutions.

For example, using timestamps for each read or write operation, you could create a histogram of reads and writes per time interval during the program run. Be careful not to incluence your results by code introduced for measuring.

Explain what you observe in your measurements. Can you observe the writer starvation in 2a, and how does your version from 2b improve the situation? How do the ready-made rwlocks behave?