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

February 22, 2013

G-Exercise 3 for "Operating System and Multiprogramming", 2013

Hand in your solution by uploading files in Absalon before March 4

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_G1.zip` or `lastname1_lastname2_G1.tar.gz`.
Use simple txt format or pdf for portability when handing in text, and explain your code with comments. When you write C code, it is required to also write a Makefile which enables the teaching assistants to compile your code by a simple "make" invocation.
Please also read the document *Krav til G-opgaver* (in Danish) provided on Absalon. It explains the idea of the exercises and general requirements for code that you hand in.

About this Exercise

Topics

The topic of this exercise is synchronisation. In the first task, you will implement locks and condition variables as a synchronisation mechanism for Buenos (kernel) threads. The concept of locks and condition variables are known from the PThreads API as `mutex` and `cond`, and should be implemented in the same spirit.
In the second task, you are asked to analyse and solve a synchronisation problem, which includes avoiding starvation.

What to hand in

Please hand in your solution by uploading a single archive file (zip or tar.gz format) with:

1. A directory containing your C-code for task 2 (one or more files), and a respective Makefile.
2. A source tree for Buenos which includes your work on task 1 (as documented in the report) – and possibly programs you have used for testing.
3. A short report in pdf or txt format.

   Your report should discuss possibilities to solve the tasks, and explain the design decisions you took for your solution (why you preferred one particular way out of several choices).

Your report is the basis for evaluation of the exercise. Therefore, it is important to *include in the report* the most *important parts of the C-code* that you yourself wrote or modified. Pay particular attention to documenting changes to existing Buenos code. You should also comment the C code itself (refer to the report for longer explanations).
The Buenos code you hand in should compile without errors with the original setup. As Buenos uses `-Werror` in its Makefile, this implies that your code is also warning-free.

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.*

1. 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.

2. **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.

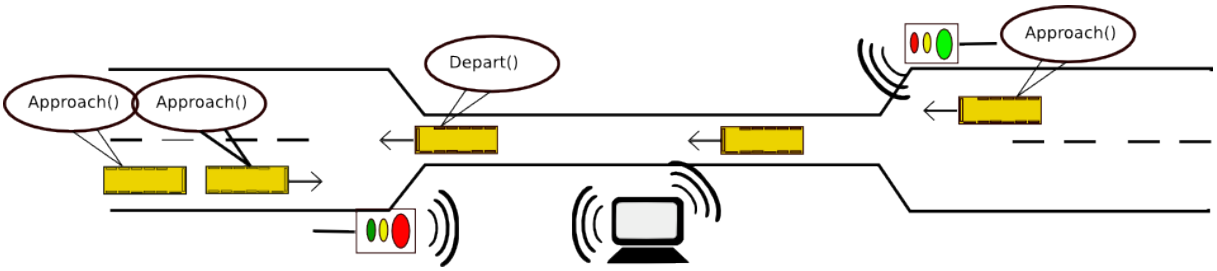
   - **void** condition_init (cond_t *cond);
   - **void** condition_wait (cond_t *cond, lock_t *lock);
   - **void** condition_signal (cond_t *cond, lock_t *lock);
   - **void** condition_broadcast (cond_t *cond, lock_t *lock);

   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. One-Lane Synchronisation

The Danish government has decided to renovate Nørreport station in Copenhagen. During the renovation, there is only one lane for buses to pass the station in north-south direction.
A bus can safely pass Nørreport if and only if there are no oncoming buses (in the opposite direction). However, several buses which travel in the same direction can (and must be allowed to) pass the narrow section at the same time.
To prevent accidents, sensors are installed in the two streets where buses approach. Whenever a bus approaches the narrow section or departs from it (in either direction), the sensors notify a controller computer which controls signal lights at either end of the narrow section to stop buses or make them pass. The controller program should synchronise traffic under all conditions, including rush hour, during which most buses approach the narrow section from the same direction. (Translation: your solution should be free from starvation.)



1. Describe a possible mechanism to synchronise traffic in the narrow pass, which realises mutual exclusion and avoids starvation under all conditions.

2. Implement a multithreaded simulation of the synchronisation mechanism in PThreads, using mutexes and condition variables.

   Instead of sensor input, assume that a bus is represented by a thread that calls Approach() and Depart() functions in the controller, passing an argument indicating the direction of travel (north/south). If necessary, Approach() will then (safely) stop the arriving bus by changing the correct signal light to red and blocking the calling thread.

3. Analyse your solution and explain how a rush hour situation is handled.

---

3. Optional Bonus Task:

Port your solution of Task 2.2 to Buenos kernel threads using the locks and condition variables from Task 1. As the code will need to execute in kernel-mode, you should start your simulation by giving a kernel boot argument, in the same way as `testconsole` works (see init_startup_fallback in `init/main.c`).