**UNIVERSITY OF COPENHAGEN**

Department of Computer Science

Jost Berthold Jyrki Katajainen

`berthold@di.ku.dk` `jyrki@di.ku.dk`

February 21, 2014

**G-Assignment 3 for "Operating Systems and Multiprogramming", 2014**

**Deadline: March 3 at 23:59**

**Handing in: via Absalon**

# G3: Problem Formulation

## Tasks

This assignment is on synchronization and it consists of two tasks. In the first task, you are asked to implement a thread-safe stack and use it in a multi-threaded matrix-multiplication program. In the second task, you should implement userland semaphores as a synchronization mechanism for Buenos.

## What to hand in?

Please hand in a package that includes the following:

1. A directory containing your C code for the first task, including a makefile.
2. A source tree for Buenos which includes your work for the second task—and possibly programs used for testing.
3. A short report where you document your code, discuss different possibilities to solve the tasks, and explain the design decisions made.

Otherwise, the same general rules apply as for the previous G-assignments.

## Task 1: A Thread-Safe Stack

**a)** Implement a stack that can handle concurrent access by different threads in a safe manner. To accomplish this task, you can use `PThread` locks and non-blocking condition variables (also known as Mesa-style condition variables or signal-and-continue condition variables).

The interface of your stack should be similar to that given in `stack.h`, but the actual implementation should not have the same restrictions as the naive implementation provided in `stack.c` (both files are available on Absalon). Also, avoid any undefined behaviour. In particular, think what `pop` should do if the stack is empty, and which methods should be protected against concurrent access and which methods work fine without protection.

**b)** Use your stack implementation in a multi-threaded `PThreads` program that multiplies two matrices row by row. As a starting point, use the multi-threaded matrix-multiplication program (provided on Absalon) that was used to demonstrate the `PThreads` API (but it does not use any synchronization apart from `pthread_join`). In contrast to the example program, your program should use a fixed number of worker threads which synchronize their work via a stack implemented by you. The worker threads should see the production of the rows of the result matrix as individual *tasks*. These tasks are pushed onto the stack by the main thread, and popped off and solved by the worker threads.

Think especially about how the system should terminate. You might want to (carefully!) use `pthread_cancel` to terminate the worker threads, or find a better solution for termination.

## Task 2: Userland Semaphores for Buenos

The basic Buenos system supports kernel semaphores, see file `kernel/semaphore.h`, but these cannot be used to synchronize userland processes. Chapter 5 of the Buenos roadmap describes the preimplemented mechanisms for synchronization in Buenos: spinlocks, sleep queue, and semaphores. Use kernel semaphores to implement userland semaphores.

More specifically, implement the following system calls[1] with the behaviour outlined below. You should also add wrappers for these calls to the system-call interface in `tests/lib.c`. Use the system call numbers shown here in `proc/syscall.h`.

`proc/syscall.h`

```
...
#define SYSCALL_SEM_OPEN      0x300
#define SYSCALL_SEM_PROCURE   0x301
#define SYSCALL_SEM_VACATE    0x302
...
```

`usr_sem_t* syscall_sem_open(char const* name, int value)`

A call returns a handle to a userland semaphore identified by the string specified by `name`, which can then be used by the calling userland process.

If the argument `value` is zero or positive, a fresh semaphore of the given name will be *created* with `value`. On the other hand, if a semaphore with that name already exists, `NULL` is returned to indicate an error.

If `value` is negative, the call to `syscall_sem_open` returns an existing semaphore in the system with the given name. If no semaphore with that name exists, `NULL` is returned to indicate an error.

`int syscall_sem_p(usr_sem_t* handle)`

A call `syscall_sem_p(h)` procures (executes the P operation on) the userland semaphore referred to by `h`. As usual, if the value of the underlying semaphore is zero, the executing process should block on the semaphore; otherwise the call should return immediately.

`int syscall_sem_v(usr_sem_t* handle)`

A call `syscall_sem_v(h)` vacates (executes the V operation on) the userland semaphore referred to by `h`, unblocks a blocked process, if one exists, and increments the semaphore value otherwise.

The return value of the two above-mentioned system calls is 0 if the respective operation succeeds, and a negative number if an error occurred.

`int syscall_sem_destroy(usr_sem_t* handle)`

A call `syscall_sem_destroy(h)` invalidates the userland semaphore referred to by `h`. This operation should fail if there are threads blocked on the semaphore. Be aware that there might be race conditions when a thread is about to block on a semaphore which is about to be destroyed.

The functionality for userland semaphores should match the functionality implemented in Buenos for kernel semaphores. Your implementation should use the kernel semaphores for all operations. Naturally, a user process should not get access to kernel memory. Be aware that

- `usr_sem_t` should be defined as `void` in the library, but the handle which is returned by `syscall_sem_open` cannot be used as a "real" memory address;
- you should specify a maximum length for the semaphore name.

[1]The API described here is a variation of the POSIX semaphore API that provides `sem_open`, `sem_wait`, and `sem_post`.