UNIVERSITY OF COPENHAGEN
Department of Computer Science
February 20, 2015

G Assignment 3 for "Operating Systems and Multiprogramming", 2015

Deadline: March 2, 2015 at 23:59

Handing in: via Absalon

Assignments in this Course

Your solutions to the G assignments are to be handed in by uploading a single archive file (in `zip` or `tar.gz` format) that contains the developed code (in C) and a brief report giving a high-level description of the solution (in `pdf` or `txt` format). The uploaded files should be named using the surnames of each group member and the assignment number as in:

`Jensen-Serup-G3.zip` or `Jensen-Serup-G3.tar.gz`.

Provide also a `Makefile` which will enable the teaching assistants to compile the code simply by invoking `make` in the extracted directory.

More information on the assignments is available at the beginning of G1 and in the course plan in Absalon.

G3: Problem Formulation

Topic

This assignment is on synchronization and it consists of two tasks. In the first task, you are asked to implement a thread-safe unbounded queue using fine-grained locking and use it in a multi-threaded producer/consumer test program. In the second task, you should implement userland semaphores as a synchronization mechanism for Buenos.

What to hand in

The uploaded archive file should contain:

1. A directory containing your C code for Task 1, including a makefile.
2. A source tree for Buenos which includes your work for Task 2 – and possibly programs you 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 (why you preferred one particular way out of several choices).

Assessment of the quality of your work will be based on the report. Therefore, it is important to *include there* the most *important parts of the C code* that you wrote or modified. Pay particular attention to documenting changes to existing Buenos code. You should also comment the C code itself (and 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` option in its `Makefile`, your code should also compile without warnings.

Task 1: A thread-safe unbounded queue

a) Implement a queue that can handle concurrent access by different threads in a safe manner. To accomplish this task, you should use `pthread` locks and condition variables. The queue should use fine-grained locking in order to allow concurrent enqueing and dequeing. Threads are not allowed to busy-wait/spin during operations on the queue.

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

b) Use your queue implementation in a multi-threaded `pthread` program. The program should be a classic multi-producer, multi-consumer program where multiple producers add (fictitious) products to the queue and consumers remove them. You should also implement a termination strategy such that the consumers terminate gracefully when the producers stop adding products to the queue. You might want to (carefully!) use `pthread_cancel` to terminate the consumers, but try to find a **better** solution.

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.

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

`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. Finally, 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.