G3

OSM 2016 Course Team – Version 1, February 29, 2016

Table of Contents

This is the third in a series of 5 G-assignments ("G" for "Godkendelse" and/or "Gruppeopgave") which you must pass in order to be admitted to the exam in OSM (http://www.webcitation.org/6eoBjRWvD). We encourage pair programming (https://en.wikipedia.org/wiki/Pair_programming), so please form groups of 2-3 students.

This assignment is about synchronization. In the first task, we ask you to extend KUDOS with userland semaphores. In the second task, we ask you to make the handed out priority queue thread-safe (outside of KUDOS).

1. Before you start

Along with this assignment, we also hand out a fresh KUDOS source and a working G1 priority queue, both of which you need to extend.

The KUDOS part of the handout contains a solution to G1 and silly implementations of `syscall_spawn` and `syscall_exit`, which is not a valid solution to G2.

2. What to submit

Throughout this course, you should always submit entire, working source code tree in a single `tar.gz` archive. The uploaded archive should contain:

a. A source code tree which includes your work for both tasks, and programs you used for testing.
b. A short report where you document your code, discuss different ways to solve the tasks, explain the design decisions (why you preferred one particular way out of several choices), and assess your solution (especially through testing).

We assess the quality of your work be based on your report. Therefore, it is important to mention the most *important parts of the C code* that you wrote or modified. Pay particular attention to documenting changes to existing KUDOS code. You should also use comments in the C code itself (but use the report for long explanations).

The KUDOS code you hand in should compile without errors with the original setup. As KUDOS uses `-Werror` option in its `Makefile`, your code should also compile without warnings.

3. Task 1: Userland semaphores for KUDOS

The basic KUDOS system supports kernel sempahores as defined in the file `kudos/kernel/semaphore.h` with the type `semaphore_t`, but these cannot be used to synchronize userland processes — yet! Your task is to use kernel semaphores to implement userland semaphores.

The system call interface in `userland/lib.h` and `userland/lib.c` already provides wrappers for these calls, so you should not modify those files.

A kernel semaphore is available to any kernel thread that knows its address; for userland semaphores we instead identify a semaphore by a text string.

In the kernel, create the files `kudos/proc/usr_sem.h` and `kudos/proc/usr_sem.c`, and implement the following types and functions, using kernel semaphores for all underlying operations:

```C
typedef struct {
  ...
} usr_sem_t;
```

The `usr_sem_t` type is at the center of your solution. Define enough attributes for the functions below to work correctly.

You will also need to define a static array of `usr_sem_t` entries and initialize them, like with `process_table` in G2 (and with `thread_table` in `kudos/kernel/thread.c`).

```C
usr_sem_t* usr_sem_open(const char* name, int value);
```

Return a handle to a userland semaphore identified by the string `name`, which can then be used in the future to refer to this semaphore.

If the argument `value` is zero or positive, a fresh semaphore of the given name will be created with `value`, unless a semaphore of that name already exists, in which case `NULL` should be returned.

If `value` is negative, the call to `usr_sem_open` returns an existing semaphore with the given name, unless such a semaphore does not exist, in which case `NULL` is returned.

```C
int usr_sem_destroy(usr_sem_t* sem);
```

Try to remove the userland semaphore `sem`, making its space available for future semaphores. Fail if there are threads currently blocked on the semaphore. **Be aware** that there is a potential race condition if a thread blocks on a semaphore while it is being destroyed. You should not handle this race condition, but you should write about it. Return 0 if nothing went wrong, or a negative number in case of an error.

```c
int usr_sem_procure(usr_sem_t* sem);
```

Procure (execute the P operation on) the userland semaphore `sem`. Return 0 if nothing went wrong, or a negative number in case of an error.

```c
int usr_sem_vacate(usr_sem_t* sem);
```

Vacate (execute the V operation on) the userland semaphore `sem`. Return 0 if nothing went wrong, or a negative number in case of an error.

Then call these user semaphore functions from `kudos/proc/syscall.c`. Use the matching system call numbers defined in `kudos/proc/syscall.h`:

```c
...
#define SYSCALL_USR_SEM_OPEN 0x301
#define SYSCALL_USR_SEM_DESTROY 0x302
#define SYSCALL_USR_SEM_PROCURE 0x303
#define SYSCALL_USR_SEM_VACATE 0x304
...
```

Finally, add a userland program to test that your userland semaphores work correctly. Use `syscall_spawn` to spawn several processes, and make them do work with the same semaphore(s).

Note that, while the KUDOS handout code **does** contain a basic spawn-capable `syscall_spawn`, there is no proper support for `syscall_join` or `syscall_exit` (since they are not critical to being able to test userland semaphores). If you have a working solution to G2, you are welcome to copy over that solution and use that instead.

The userland semaphore syscall functions have the same names and type names as in your `kudos/proc/usr_sem.h`, except that function names have a `syscall_` prefix.

Please consult the chapter on advanced synchronization (https://kudos.readthedocs.org/en/latest/advanced-synchronization.html) in the KUDOS documentation for background knowledge.

Hints:

- Specify a maximum length for the semaphore name in order to have equally-sized struct elements in your semaphore table.
- To tell KUDOS' `make` system that there's a new file in town, modify `module.mk` — each subdirectory has one of these.
- Even though the `usr_sem_t` type is called the same in both the kernel and in userland, it does not actally have to **be** the same type. The only important part is that every `usr_sem_t*` value in userland means something in the kernel, as userland programs should never need to dereference it.

4. Task 2: A thread-safe priority queue

In the code handout, you will find a working G1 priority queue. Your task is to use the `pthread` library to make the priority queue thread-safe. For each API function, decide which `pthread` mechanisms are necessary. Recall the priority queue interface:

```
int queue_init(struct queue *queue);

int queue_push(struct queue *queue, int pri);

int queue_pop(struct queue *queue, int *pri_ptr);

int queue_destroy(struct queue *queue);
```

C

However, we change the API from G1 slightly. In G1, `queue_pop` returns a non-zero value in error when the queue is empty. You must change this behaviour. Instead, `queue_pop` should now **block** if the queue is empty, and stop blocking when a new priority is pushed.

Write a test program that uses `pthread` to create a several threads, each of which should push and pop in some order.

Hints:

- Look up `pthread` condition variables and mutexes. They can come in useful (although they are not the only way to solve the assignment).
- Does your `pthread` code refuse to finish compiling? You must tell `gcc` that it should link with the `pthread` library. You do this with the `-pthread` command-line argument.

§ 5. Challenge of the Week

"Challenge of the Week" is a completely optional, typically tough, but also enlightening part of an assignment. Good solutions will be called up during the following Friday lectures, and handed a student-friendly prize.

This week's challenge is to extend the thread-safe priority queue with a new function:

```c
int queue_pop_many(struct queue *queue, int n, int *pri_ptrs);
```

This function pops not just one priority off the queue, but `n` priorities!

The popped priorities are to be stored in `pri_ptrs`. It is the responsibility of the caller to ensure that `pri_ptrs` points to sufficient allocated memory for `n` priorities to be stored.

If there were no further requirements, this would not be much of a challenge. Why? To make it more challenging, we add an extra requirement: The popping must occur in FIFO order. That is, if there is more than one `queue_pop_many` call currently blocking, then all pushed priorities must go to the thread which started blocking first, until that returns, and then go on to the thread which started blocking second, until **that** returns, and so on.

Version 1
Last updated 2016-02-28 15:37:59 CET