KING ABDULLAH UNIVERSITY OF SCIENCE AND TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

CS240 - Assignment 4

“We have shared memory ... now all we need are Semaphores.”

Maruan Al-Shedivat (ID: 129096)

November 27, 2013

1 SEMAPHORE IMPLEMENTATION

For this assignment, semaphores were added to xv6. As it was proposed in the assignment description, the four system calls were integrated: `sem_get`, `sem_delete`, `sem_signal`, `sem_wait`. The processes that want to use a semaphore to communicate with each other should, first, agree on a name (a positive `uint`), then get handles for the same semaphore using `sem_get` system call, and then synchronize themselves using `sem_signal` and `sem_wait`.

The implementation of the semaphores is the following. In kernel, we create a static array called `sem_table` that contains structures `sem_t` with the information on all the semaphores available in the system.

```
struct sem_t {
  uint name;          // Semaphore name
  int value;          // Semaphore value
  uint deltime;       // Semaphore delete time
};
struct sem_t sem_table[MAXSEMNUM];
```

So, obviously, in this design, the number of available semaphores is fixed and equal to `MAXSEMNUM` – the constant which was defined in *param.h*. Initially, all the semaphore structures in the array are zero-structures. To allow a concurrent access to the array, we also define a system-wide lock, called `semlock`.

When a process gets a semaphore by name and size, the system checks the provided parameters and returns a handle (or an error code) to the process. In the current design, the handle is nothing but an index of a semaphore info structure in the `sem_table` (starting from 1), and sem_get guarantees that the returned handle is the smallest available. While getting a handle, we also record the time of the get procedure. This time stamp along with aforementioned deltime provide us a layer of security. To be able to execute sem_delete, sem_signal or sem_wait, a process should not only provide a proper handle, but also be eligible: its get time for the requested handle should be greater than the delete time for the semaphore with this handle. This additional check protects us from a situation in which a malware process might delete or signal semaphores just using a handle it didn't get via regular sem_get.

While sem_signal has a trivial implementation – after a few checks it increments the value of the semaphore and wakes up all the processes waiting for the current semaphore – sem_wait and sem_delete have a couple of interesting moments. First, in sem_wait we perform sleeping in a while loop, and also check every loop iteration whether the semaphore is present (nobody deleted it) and whether the time stamps are coherent[1]. Second, in sem_delete we not only clean the sem_t structure, but also generate a wakeup in case if the value of the given semaphore was 0 (i.e. probably, somebody slept on that semaphore, and we need to make it know that the semaphore was deleted).

2 Semaphore Testing and Discussion

The described above implementation was tested on a producer/consumer model. Shared memory buffer[2] was created and then the test process forked. The child process played a role of a consumer (i.e. it "took" the resources from the buffer) and the parent process played the role of a producer (i.e. it "put" new resources into the buffer). The test user function code was based on the example code provided in the assignment handout (with a couple of bugs fixed). The only difference is that the number of producers and consumers was increased up 4 and 4, respectively. This made the concurrency behaviour richer and the whole experiment more interesting.

All the rest details are minor, and they can be found in the source code itself which is well commented.

[1] It might happen that somebody deleted the semaphore a process was sleeping on, then some other process created a new semaphore, and it happened that the handle it got coincides with the handle of the just deleted one. In this case, the time stamps step will rescue the situation.

[2] The shared buffer was based on the KSM implementation from the previous assignment.