CS240 Assignment 4

Fall 2013, Prof. Hany Ramadan

Emaad Ahmed Manzoor (129110) November 28, 2013

The following describes the per-process and global state, and associated methods that manipulate these pieces of state.

Global State. Each `struct semaphore` is present in a `MAXSEMS` size array in the kernel, and maintains the following fields:

- `uint name`: Initially `0`, this records the name provided to the `sem_get` call.
- `int wakeups`: Initially `0`, this records the value of the semaphore, set by a `sem_get` call.
- `uint used`: Initially `0`, this records the number of processes using this semaphore. It is the total number of `sem_get`s minus the total number of `sem_delete`s on this semaphore.

Per-process State. An integer array `proc->sems` of `MAXSEMS` size is maintained, with each element either set to `0` or `1`. This is to prevent processes from calling semaphore functions on a handle without first calling `sem_get` to obtain the handle.

There is also a lock protecting the semaphore array in the kernel called `semlock`.

Functions. There are four functions that manipulate this state. All functions check if the `handle` is in the correct range, if the semaphore corresponding to the `handle` is `used`, and (except `sem_get`) if the calling process has its `proc->sems[handle]` flag set to 1.

`sem_get_proc(name, value, proc)` finds a new handle or returns an existing handle for a semaphore. If the `proc->sems[handle]` for the provided `proc` structure is `0`, it is set to `1` and the `used` count for the semaphore is incremented.

`sem_delete_proc(handle, proc)` decrements the `used` count for the semaphore and sets `proc->sems[handle]` to `0` for the provided `proc` structure. It then wakes up all processes waiting on this semaphore. Note that this ensures a process can call `sem_delete` only once on a semaphore obtaining via `sem_get.` Once all such processes have `exit`ed or called `sem_delete`, the `used` count becomes `0` and this semaphore can be used by another `sem_get` call.

`sem_wait(handle)` runs a loop checking the number of `wakeups` of the semaphore. If the number of wakeups is `0`, it `sleeps` with the address of the `handle`th element in the kernel semaphore array as the wait channel, and `semlock` as the lock. Once more than `0 wakeups` are observed, `wakeups` is decremented and the function breaks out of the loop.

`sem_signal(handle)` increments the number of wakeups for the semaphore and calls `wakeup` with the address of the `handle`th element in the kernel semaphore array as the wait channel.

Fork. On forking, `sem_get` is simulated on the child for every semaphore present in the parent's `proc->sems` array. This is done by calling `sem_get_proc` for every parent semaphore and providing the child's `proc` structure with the parent's semaphore's name.

Exit. On exit, `sem_delete` is simulated on every semaphore present in the parent's `proc->sems` array. This is done by calling `sem_delete_proc` and providing the parent's `proc` structure with the handles present in its `proc->sems` array.

Limits. `MAXSEMS` is the maximum number of semaphores in the system, defined in `params.h`.

Tests. Semaphores were implemented on top of the shared memory implemented in the previous assignment. The provided producer-consumer user program was modified to use shared memory.

Files Modified or Created.

`Makefile`	Entries for new source files.
`defs.h syscall.c syscall.h sysproc.c user.h usys.S`	Stub code and function prototypes for the system calls.
`semaphore.c semaphore.h proc.c proc.h semaphoretest.c param.h`	Semaphores implementation, parameters, process semaphore array and modified fork and exit functions.