Report on exercise #2

Matteo Corain S256654

Laboratory #6 – System and device programming – A.Y. 2018-19

The proposed solution for the implementation of semaphore primitives in the xv6 kernel makes use of a `sem` structure, defined in `sem.c` (a separate file, rather than reusing `file.c`, has been introduced for the purpose), which includes the following fields:

- An integer flag `alloc`, which is set to `1` when the semaphore is allocated and to `0` when the corresponding structure has to be considered free for usage;
- An integer `count`, which represents the value of the semaphore;
- An array of pointers to `proc` structures, called `queue`, with a maximum size equal to the maximum number of processes supported by the kernel (`NPROC`), representing the list of processes which are currently waiting on the semaphore;
- Two integers, `qstart` and `qend`, which indicate respectively the beginning and the end indices of the processes queue (circular buffer).

All the semaphores are stored in a global table called `semtable`, similar to the structure used for the implementation of the file table. In particular, this structure includes:

- An array of `sem` structures, with length `NSEMS` (defined in `params.h` to be equal to `10`);
- A spinlock `lock`, to protect all the accesses to the semaphore array and guarantee mutual exclusion.

Internal functions used to work on the semaphore table are defined in the `sem.c` file as well, and perform the following actions:

- `semtableinit()`: it initializes the spinlock of the semaphore table (necessary for its usage);
- `semalloc()`: it loops through the positions in the semaphore table searching for a free position (`alloc` is equal to `0`); as soon as one is found, it is marked as allocated and its index is returned (it will be used as a sort of "semaphore descriptor"); otherwise, `-1` is returned (all semaphores are currently being used);
- `semget()`: it returns a pointer to the `sem` structure relative to the passed semaphore descriptor; in case the descriptor is not valid (negative or higher than `NSEMS`, semaphore not allocated), it panics the kernel;
- `seminit()`: it initializes the given semaphore with the given initial value and the queue indices both to zero (queue is initially empty);
- `semdestroy()`: it marks the given semaphore as unallocated (`alloc` is set to zero), so that the relative entry in the semaphore table can be reused;
- `semwait()`: it decrements the semaphore count; if the semaphore value gets below zero (no more processes are allowed to enter the critical section), a pointer to the current process structure (stored in the variable `proc`, defined in `proc.h`) is stored in the process queue, the end terminal of the queue is circularly updated and the current process is put in a waiting state by calling `sleep()` (operation that also internally unlocks the spinlock of the semaphore table);
- `sempost()`: it increments the semaphore count; if the semaphore value is still less or equal than zero (at least a process is waiting for the semaphore to unlock), the first process in the queue is woken up by calling `wakeup()` (operation that reassigns also to the process the lock on the semaphore table) and the start terminal of the queue is circularly updated.

A call to the semtableinit() function has been placed in the main() of the kernel in order to initialize the semaphore table after the system has booted. Calls to the other internal functions are wrapped into the respective sys_ functions (defined in syssem.c), which are called by the syscall() function whenever one of the user-space semaphore primitives, defined in user.h, is used. In particular, those functions perform the following actions:

- sys_sem_alloc(): performed whenever the user calls sem_alloc(), it simply runs the internal semalloc() and returns the descriptor of the allocated semaphore;
- sys_sem_init(): performed whenever the user calls sem_init(), it retrieves the two integer arguments of the user-space function (the semaphore descriptor and its initial value) by calling the argint() function, then it calls the internal seminit() on the struct sem* returned by the semget() function to which the semaphore descriptor is passed;
- sys_sem_destroy(): performed whenever the user calls sem_destroy(), it retrieves the integer semaphore descriptor via argint() and then calls the internal semdestroy() function on the struct sem* returned by the semget() function to which the semaphore descriptor is passed;
- sys_sem_wait(): performed whenever the user calls sem_wait(), it retrieves the integer semaphore descriptor via argint() and then calls the internal semwait() function on the struct sem* returned by the semget() function to which the semaphore descriptor is passed;
- sys_sem_post(): performed whenever the user calls sem_post(), it retrieves the integer semaphore descriptor via argint() and then calls the internal sempost() function on the struct sem* returned by the semget() function to which the semaphore descriptor is passed.

In addition to the modifications and insertions presented above, in order to make the entire mechanism of calls working, the following files have also been updated:

- defs.h: the (incomplete) definition of the sem structure and the prototypes of the internal functions have been added;
- usys.S: a SYSCALL definition for each introduced system call has been added;
- syscall.h: a system call number for each introduced system call has been added;
- syscall.c: the prototypes of the sys_ functions have been added;
- Makefile: files sem.o and syssem.o have been added as kernel objects and _st has been introduced as a user program.