Report on exercise #2

Matteo Corain S256654

Laboratory #3 – System and device programming – A.Y. 2018-19

The proposed solution makes use of a global buffer, called `buffer`, with size `BUF_LEN` (set to 20), and two global semaphores (of type `sema_t`, defined in `cond_sem.h`) to manage the access to that buffer using a single producer and single consumer protocol (`empty`, `full`). The type `sema_t` is in fact a structure used to implement a synchronization mechanism based on conditions, and includes:

- A `pthread_mutex`, called `lock`, and a `pthread_cond`, called `cond`, used to implement a synchronization mechanism based on conditions;
- An integer value, called `value`, which represents the current value of the semaphore.

The different primitives used to manipulate semaphores of type `sema_t` are implemented in the `cond_sem.c` library as follows:

- `sema_init(sema_t *sema, int initial_val)`: the initialization function copies the initial value of the semaphore and initializes the lock and the condition (via `pthread_mutex_init()` and `pthread_cond_init()`);
- `sema_wait(sema_t *sema)`: the wait function locks the mutex (via `pthread_mutex_lock()`), checks if the value of the semaphore is zero and, in case, waits on the condition variable (via `pthread_cond_wait()`), decrements the value of the semaphore and finally unlocks the mutex (via `pthread_mutex_unlock()`); externally, the wait function behaves as a normal wait on a semaphore, blocking the execution of the thread if the current value of the semaphore is zero;
- `sema_post(sema_t *sema)`: the post function locks the mutex (via `pthread_mutex_lock()`), increments the value of the semaphore, checks if the value of the semaphore is one and, in case, performs a signal on the condition variable (via `pthread_mutex_signal()` instead of `pthread_mutex_broadcast()` so that it is guaranteed that at most a single thread will be unlocked for each call) and finally unlocks the mutex (via `pthread_mutex_unlock()`); externally, the post function behaves as a normal post on a semaphore, unblocking the execution of a thread waiting on the semaphore if the new value of the semaphore is one;
- `sema_destroy(sema_t *sema)`: the destroy function destroys the mutex and the condition (via `pthread_mutex_destroy()` and `pthread_cond_destroy()`).

The main thread performs the following operations:

- It initializes the random seed (via the `srand()` function);
- It allocates (via `malloc()`) and initializes (via `sem_init()`) the two semaphores (to `BUF_LEN` for the `empty` semaphore and to `0` for the `full` semaphore);
- It creates (via `pthread_create()`) and joins (via `pthread_join()`) the two threads for the producer and the consumer;
- It finally destroys (via `sem_destroy()`) and releases (via `free()`) the semaphores.

The producer and consumer threads implement a standard producer and consumer algorithm, using the proposed versions of the semaphore primitives instead of the library ones.