SDP 2022 – OS161 Laboratory – 3

In order to complete this lab, you the following pre-requisites are important:

- ***completing (and understanding) lab 1. Lab 2, though important, has no direct implications on this task, that does not require user processes and system calls.***
- ***watching lessons os161-synchro, and understanding the semantics of semaphores, locks and condition variables, as well as the behaviour of spinlocks and wait channels in OS1616.***
- ***using the debugger.***

Note on using options:

as said at the end of the demo "os161-synchro-demo", consider the fact that semaphores, and more importantly locks, are already used in the OS161 kernel, in particular in the boot phase, as well as (semaphores) in the I/O instructions involving the console (locks do not work properly, but this does not affect the behaviour of the kernel, which is not particularly critical, in the basic version, in relation to multi-threading). However, if you change the implementation of the semaphores, and/or implement locks, consider that the kernel could "crash" immediately without completing the boot. For this reason ***it is important to be able to easily isolate*** *the various parts of added code, possibly through* ***options and conditional compilation****. It is also STRONGLY recommended to proceed step-by-step, if possible, with EASY/SIMPLE implementations first, then passing to the more critical ones only once the first ones have been developed and verified (possibly with debugging). Keep in mind that, in an execution with GDB, the combination of keys Ctrl-G (control-G) in the execution window (the console) allows to interrupt the execution, passing the control to the debugger: it can be useful in some cases (not all, it depends on the extent of the problem) in which a program is in infinite loops or other errors.*

Implementing locks and condition variables (kernel-side)

Please refer to the previous OS161 labs, where you were asked to perform kernel thread execution tests. Therefore, the problem of user processes and/or threads and their synchronization is NOT addressed.

Synchronization test and locks implementation

Try the `sy1` and `sy2` tests and trace their execution using the debugger. `sy1` performs a synchronization test by means of *semaphores*, `sy2` by means of *locks*. *Semaphores* are implemented, while *locks* are NOT.

HINT: *semaphores are implemented in "NON busy waiting" mode. To implement them, "wait channels" are used. Wait channels are a type of condition variable made in OS161, ASSOCIATED with a spinlock. Since spinlocks are locks with "busy waiting", they are suitable for cases where (within the kernel) there is a limited wait. Read (and trace with the debugger) the creation of a semaphore: you will observe that the spinlock protects (by mutual exclusion) the access to the counter inside the semaphore, while the wait channel is used to implement waiting and signaling (i.e. a thread waits to be awakened, by reporting, from another thread).*

You are asked to implement *locks* in 2 different ways (use the *conf.kern* options, or manually defined constants, and conditional compilation to differentiate the two implementations):

1. using *semaphores*: a *lock* is indeed equivalent to a *binary semaphore* (counter with maximum value 1). You need to change/complete `struct lock` in `kern/include/synch.h`, and complete functions `lock_create()`, `lock_destroy()`, `lock_acquire()`, `lock_release()`, `lock_do_i_hold()` in `kern/thread/synch.c`.

2. using directly *wchan* and *spinlock*. This is certainly the preferred version, for which it is advisable to refer, adapting it appropriately, to the implementation of *semaphores* (it is, ALMOST, a copy, PAY ATTENTION TO THE DIFFERENCES).

WARNING: *a lock is not just a binary semaphore. Unlike the semaphore, the lock has the concept of "ownership", i.e.* `lock_release()` *can only be called by the thread that obtained the lock through* `lock_acquire()` *and currently owns it. This implies that the* `struct lock` *must have a pointer to the thread owner: remember that the* `curthread` *symbol is available (it is not a global variable but a* `#define` *made in* `current.h`*, usable as a global variable). The attempt to* `lock_release()` *by a non-owner thread can (optionally) be treated as a fatal error (*KASSERT *or* panic*) or simply be ignored, not releasing the lock.* KASSERT *is the recommended choice because, since it is a kernel thread calling it, an incorrect* `lock_release()` *attempt is a real error in the kernel. Also note that* the `lock_do_i_hold()` *function must be implemented. The function should return to the calling program whether or not the current thread is the owner of a lock received as a parameter. In all probability, it is a matter of reading a pointer stored in the* `struct lock`*: reading should therefore be done with this pointer in mutual exclusion (or at least it is necessary to consider whether multiple access from multiple threads is critical or not). Attention, if a spinlock is used to implement* `lock_do_i_hold()` *as a critical section, to the fact that OS161 does not allow a thread to wait on* `wchan_sleep()` *while owning more that one spinlock (i.e. the only spinlock owned must be the one to be released and later acquired again): so beware when calling* `spinlock_acquire ()` *on multiple spinlocks before calling* `wchan_sleep()`.

Synchronization test and implementation of condition variables

Similar to *locks*, also *condition variables* must also be implemented. Try `sy3` and `sy4` tests and trace their execution using the debugger. The tests should indicate possible problems in the implementation of *condition variables*, however errors that are not reported by these tests are still possible (in particular *sy3*, simpler test program).

For completeness, *condition variables* are described below. For a more complete discussion, you can refer to https://en.wikipedia.org/wiki/Monitor_(synchronization) and to the "os161-synchro" lesson.
A *condition variable* is essentially a synchronization primitive that allows you to wait for a condition (possibly false at the present time) to become true.
A *condition variable* is always accompanied by a *lock* (ATTENTION, A *LOCK*, while the *wait_channel* is associated with a *spinlock*), passed as a parameter to the functions, which guarantees its protected access in mutual exclusion).

The functions provided by OS161 condition variables are:

- `cv_wait():` releases the *lock* received as a parameter (which must have been previously acquired by the calling thread), and put the calling thread on hold for a `cv_signal()` or `cv_broadcast()`.

- `cv_signal()` and `cv_broadcast()` perform the same task, but differ in the number of threads woken up, the first wakes up only one (among those waiting) the second all.

A *condition variable* could be implemented by using *semaphores*: the wait (`cv_wait()`) could be implemented through a `P()`, while `cv_signal()` through `V()` on the *semaphore*. Implementing `cv_broadcast()` with semaphores would be a bit more tricky, as it would require a counter of the *waits* in progress. Furthermore, there would be a semantic problem: with *semaphores*, one should manage the fact that a `V()` would be possibly "remembered" (in the case of no thread waiting) and could unlock a future `P()`: this behavior is incorrect with *condition variables* (`cv_signal()` and `cv_broadcast()` only unlock threads waiting now, not at a future time). SAID IN OTHER TERMS: the *condition variable* created with *semaphores* would behave slightly differently from HOW IT SHOULD.

It is therefore recommended to create *condition variables* directly via *wait_channel* (and *spinlock*). Since *wait_channel* has a semantic substantially similar to *condition variables* (but has a *spinlock* instead of a *lock*), a *wait_channel* can be used for wait/signal operations, but the *lock* must be managed (correctly). The test programs for *condition variables* are `sy3` and `sy4`, which call the functions `cvtest()` and `cvtest2()`.

The files containing definitions and functions about *condition variables* are `kern/include/synch.h` and `kern/thread/synch.c`. The file containing `cvtest()` and `cvtest2()` is `kern/test/synchtest.c`.

ATTENTION: the lock *release and the thread wait operations (in the* `cv_wait()`*) must be done atomically, i.e. avoiding that another thread acquires the* lock *before the thread goes on hold (on the* wait_channel*). For this purpose, it is recommended to take advantage of the* spinlock *associated to the* wait_channel.