# ECE492/592 – Operating Systems Design: Project #3

**Due date: October 25, 2020**

## Objectives

- To understand Xinu's implementation of semaphores.
- To understand deadlocks and their detection.
- To understand the priority inversion problem. **[Required only for students taking the course at the graduate level – ECE592]**
- To implement different synchronization mechanisms in Xinu, verify their correct operation and evaluate their performance on representative test cases.

## Overview

This programming assignment focuses on synchronization. Your goal is to implement locks in Xinu based on the `test_and_set` hardware instruction. You will start with a basic implementation and progressively refine it.

## Programming assignment

### General requirements:

- You will use the same header file (`include/lock.h`) for all lock variants, but implement each lock variant in a different `.c` file (as indicated below).
- Your implementation *cannot* rely on semaphores or on interrupts disabling (although you don't need to eliminate interrupts disabling from Xinu's code).
- Your code should assume a fixed number of locks.
- You are not required to implement functions to delete locks. However, your lock initialization functions should return `SYSERR` if the maximum number of locks of a particular type has been reached.
- The unlock system calls should return `SYSERR` if a process tries to release a lock that it hasn't previously acquired (i.e., if that process is not the current owner of the lock).

1. Provide an assembly implementation of the atomic `test_and_set` function. The function should have the following declaration:

   ```
   uint32 test_and_set(uint32 *ptr, uint32 new_value);
   ```

   and it should conceptually implement the following code atomically:

   ```
   uint32 test_and_set(uint32 *ptr, uint32 new_value) {
      uint32 old_value = *ptr;
      *ptr = new_value;
      return old_value;
   }
   ```

Your implementation should be based on the `XCHG` x86 instruction (see Intel Architecture Software Developer's Manual, Volume 2) and be written entirely in assembly. Note that the GNU assembler (*as*) used in the development-system machine uses the AT&T System V/386 assembler syntax, which is slightly different from the Intel x86 syntax described in the manual. For example, the `ctxsw.S` file is written using the AT&T syntax.

You can find information on the differences between the Intel and AT&T syntax here:
http://www.cs.cmu.edu/afs/cs/academic/class/15213-f01/docs/gas-notes.txt
You can find more information (including how to instruct *as* to support the Intel syntax) here:

http://web.mit.edu/rhel-doc/3/rhel-as-en-3/i386-syntax.html

Name the file containing the `test_and_set` implementation `testandset.S` and place it in the `system` folder. Comment each line of this assembly file.

2. Implement a spinlock based on your `test_and_set` function. The spinlock should have an initialization, a lock and an unlock functions declared as specified below. You are free to define the `sl_lock_t` data type as you wish.

```
syscall sl_initlock(sl_lock_t *l);
syscall sl_lock(sl_lock_t *l);
syscall sl_unlock(sl_lock_t *l);
```

The spinlock implementation should be in a `system/spinlock.c` file. Include in `lock.h` constant `NSPINLOCKS` defining the maximum number of spinlocks that can be used, and initialize this constant to 20.

3. Implement a lock that limits busy waiting by putting the current process to sleep. In particular, your implementation should follow Section 28.14 of the textbook (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf). Note that you need to implement your own `park`, `unpark` and `setpark` primitives (you can, whenever possible, reuse existing Xinu's system calls). The lock should be based on the `test_and_set` function that you have implemented, and should have an initialization, a lock and an unlock functions declared as specified below. You are free to define the `lock_t` data type as you wish.

```
syscall initlock(lock_t *l);
syscall lock(lock_t *l);
syscall unlock(lock_t *l);
```

The lock implementation should be in a `system/lock.c` file. Include in `lock.h` constant `NLOCKS` defining the maximum number locks of this kind that can be used, and initialize this constant to 20.

**Note**:

- While you cannot use interrupt disabling directly in the lock/unlock primitives, you can use interrupt disabling in the `park`, `unpark` and `setpark` system calls (*only when strictly necessary for correct operation*). If you disable interrupts at the beginning of these system calls, you must enable interrupts before exiting them.

- The textbook implementation above can incur a deadlock even in the presence of a single lock when the running threads have different priorities. Make sure that your implementation is not subject to this problem.
- The lock's queue should be handled in FIFO fashion (even in the presence of processes with different priorities).

4. Modify your lock implementation (3) so to automatically detect the presence of deadlocks due to circular dependencies (i.e., deadlocks involving multiple locks). Your implementation should notify of the presence of the deadlock (without performing any corrective actions) as soon as the deadlock is originated. Note that, since locks originate when a cyclic dependency is created, the lock detection code should be part of the `al_lock` system call.

   The initialization, lock and unlock functions should now be declared as follows.

```
syscall al_initlock(al_lock_t *l);
syscall al_lock(al_lock_t *l);
syscall al_unlock(al_lock_t *l);
bool8   al_trylock(al_lock_t *l);
```

   Note that this lock variant has the additional `al_trylock` function (similar to POSIX threads `pthread_mutex_trylock`). `al_trylock` tries to obtain a lock and it returns immediately to the caller if the lock is already held. The function returns `true` if it has obtained the lock, and `false` if it hasn't. The lock implementation should be in a `system/active_lock.c` file. Again, you are free to define the `al_lock_t` data type as you wish. Include in `lock.h` constant `NALOCKS` defining the maximum number of "active" locks that can be used, and initialize this constant to 20.

   When a lock is detected, the code should generate the following output:

```
lock_detected=<list of processes involved separated by hyphen, sorted by
increasing process ID>
```

   e.g.

```
lock_detected=P1-P2-P4
```

   Your implementation should support situations where multiple deadlocks are present (and detect them all).

5. **[Only ECE592 students]** Modify your lock implementation (3) so to as to avoid priority inversion. Use the mechanism seen in class – if you need a reference, you see the "Basic Priority Inheritance Protocol" described in the following paper:

   *L. Sha, R. Rajkumar and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," in IEEE Transactions on Computers, vol. 39, no. 9, pp. 1175-1185, Sep 1990.*

   The initialization, lock and unlock functions should now be declared as follows.

```
syscall pi_initlock(pi_lock_t *l);
syscall pi_lock(pi_lock_t *l);
syscall pi_unlock(pi_lock_t *l);
```

This lock's implementation should be in `system/pi_lock.c` file. Again, you are free to define the `pi_lock_t` data type as you wish. Include in `lock.h` constant `NPILOCKS` defining the maximum number locks of this kind that can be used, and initialize this constant to 20. As in part (3), the lock's queue should be handled in FIFO fashion even in the presence of processes with different priorities.

Your code should output all changes in priority (when they occur), including the process ID, the old and new priority as follows:

```
priority_change=P<pid>::<old priority>-<new priority>
```
e.g.
```
priority_change=P2::3-5
```

6. Write different *representative* test cases as follows:

   **Test case #1** `(main-deadlock.c)`
   `main-deadlock.c` should be used to verify your active lock (4) implementation. This test case should include two parts.
   - Part 1 should trigger a deadlock and verify that the deadlock detection code works properly.
   - Part 2 should be a corrected version of the code of Part 1 that avoids the deadlock by making use of the `trylock` function.

   **Test case #2** `(main-pi.c)` **[Only ECE592 students]**
   `main-pi.c` should be used to verify your implementation (5). Make sure that your test case also checks the transitivity of priority inheritance.

**Help**: As a reference, we are providing some test cases that you can use to test your spinlock and lock implementation. Make sure you have a look at the `README` file before you use those test cases.

**Additional requirements:** For some of the test cases, including timing information helps. For example, in order to model a critical section with a given duration, you can include a function that runs for a specified time and invoke it between a *lock* and an *unlock* operation. As an example, have a look at the `run_for_ms` function in file `test_timing.c`.

To this end, as in project #2:
- add a `runtime` field to the process control block (in `process.h`) indicating the current running time of a process (i.e., the amount of time the process has been in `PR_CURR` state).
- update variable "`ctr1000`" to record the number of milliseconds elapsed since boot.

**Recommendation:** Xinu's *send* function has a problem. As it is now, if a process tries to send a message and the receiver has already a pending message, *send* returns an error (`SYSERR`). Unfortunately, the implementation of the *kill* function does not handle this case. This can be problematic in situations where multiple threads try to send a message to the same receiver at the same time (note that the `test_spinlock` and `test_lock` test cases spawn up to 50 concurrent threads). If a thread misses a message, on a *receive* it will end up waiting forever. To avoid this problem, you can perform a minor modification to the *send* primitive. Specifically: if a process tries to send a message to the parent, and the parent has a pending message, the scheduler will be called again (to give the

parent the opportunity to read the pending message before proceeding). This might not handle all possible cases, but should be enough for the kind of experiments you will want to perform in this project.

## Report

Include in the report:

- A brief description of your implementation approach for (4) and (5). You don't need to describe the implementation of points (1)-(3).
- A description of your test cases, indicating whether their outcome is as you expected.

Your report should not exceed:

- ECE492 students: 2 pages using Times New Romain, 11-point font.
- ECE592 students: 3 pages using Times New Romain, 11-point font.

## Submissions instructions

1. **Important:** We will test your implementations using different test cases. Therefore, do not implement any essential functionality (other than your test cases) in the `main.c` file. Also, turn off debugging output before submitting your code.
2. Go to the `xinu/compile` directory and invoke `make clean`.
3. As for the previous project, create a `xinu/tmp` folder and **copy** all the files you have modified/created into it (the `tmp` folder should have the same directory structure as the `xinu` folder).
4. Go to the parent folder of the `xinu` folder. Compress the whole `xinu` directory into a *tgz* file.

```
tar czf xinu_project3.tgz xinu
```

5. Submit your assignment – including *tgz* file and report – through Moodle.

## Grading rubric

| | ECE492 | ECE592 |
|---|---|---|
| *Report* | 10 | 10 |
| *Part 1 – test&set* | 10 | 5 |
| *Part 2 – spinlock* | 10 | 5 |
| *Part 3 – lock w/ guard* | 30 (15 for good effort) | 20 |
| *Part 4 – deadlock detection* | 30 (15 for good effort) | 25 (10 for good effort) |
| *Part 5 – priority inversion* | - | 25 (10 for good effort) |
| *Testcase code* | 10 | 10 |