ECE592 – Operating Systems Design: Homework #3

Due date: October 31, 2017

Objectives

- To understand Xinu's implementation of semaphores.
- To understand deadlocks and their detection.
- To understand the priority inversion problem.
- 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

Important: 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.

1. Provide an assembly implementation of the atomic `test_and_set` function. The function should have the following declaration and 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` datatype as you wish.

   ```
   void sl_init (sl_lock_t *l);
   void sl_lock(sl_lock_t *l);
   void sl_unlock(sl_lock_t *l);
   ```

   The spinlock implementation should be in a `system/spinlock.c` file.

3. Implement a lock that does not lead to busy waiting. The lock should again 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` datatype as you wish.

   ```
   void init (lock_t *l);
   void lock(lock_t *l);
   void unlock(lock_t *l);
   ```

   The lock implementation should be in a `system/lock.c` file.

4. Modify your lock implementation (3) so to automatically detect the presence of a deadlock. Your implementation should notify of the presence of a deadlock (without performing any corrective actions). The initialization, lock and unlock functions should now be declared as follows.

   ```
   void al_init(al_lock_t *l);
   void al_lock(al_lock_t *l);
   void al_unlock(al_lock_t *l);
   bool 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` datatype as you wish.

5. Modify your lock implementation (3) so to avoid priority inversion. Use 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.

```
void pi_init(pi_lock_t *l);
void pi_lock(pi_lock_t *l);
void 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` datatype as you wish.

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

Test case #1 `(main-basic.c)`

`main-basic.c` is meant to evaluate the correctness of your lock implementation by allowing multiple threads to perform the summation of a large array of integer numbers. <u>You can use this test case just on the lock variant of question (3)</u>. Write three versions of the global summation code:

```
/* performs the summation serially (one thread) */
uint32 serial_summation(uint32 *array, uint32 n);

/* performs the summation in parallel without using locks */
uint32 naive_parallel_summation
           (uint32 *array, uint32 n, uint32 num_threads);

/* performs the summation in parallel using locks */
uint32 sync_parallel_summation
           (uint32 *array, uint32 n, uint32 num_threads);
```

where

- `array` is the array of numbers to be summed;
- `n` is the size of the array;
- `num_threads` is the number of threads performing the summation.

`naive_parallel_summation` should provide incorrect results using Xinu scheduler (especially on large arrays). Run your test case using different array sizes and numbers of threads, and verify that the lock-based summation always provides correct results. Indicate in the report if your test case confirms that your implementation works as you expected (briefly explain).

Test case #2 `(main-perf.c)`

`main-perf.c` should be used to compare the performance of the spinlock (2) and lock (3).

Test case #3 `(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 situation 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 #4 `(main-pi.c)`

`main-pi.c` should be used to both verify and evaluate your priority inversion-free implementation. In other words, you will use `main-pi.c` to compare the performance and behavior of your implementations (3) and (5) – the former without and the latter with priority inheritance. This file should contain two test cases: one where priority inversion allows efficient execution, and one where the effectiveness of priority inversion is limited by the formation of a chain of blocking.

Important – include in the report:

- A description of your implementation approach.
- A description of your test cases, indicating whether its outcome is as you expected.

Please be clear and succinct.

Submissions instructions

1. **<u>Important</u>:** 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 homework assignments, 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_homework3.tgz xinu
```

5. Submit your assignment through Moodle. Please only upload only one *tgz* file.
6. Print your report and bring it to class on the due date.