Lab 5: Synchronizing Threads

Master M1 MOSIG – Université Grenoble Alpes (Ensimag & UFR IM2AG)

2026

1 Instructions

This lab is not graded. The problems are to be solved on machine.

Before starting working on this lab, you should have finished solving all exercises from the practical section of Lab 4.

2 Debugging multi-threaded applications

We have introduced the debugging tool `gdb` during the first lab. `gdb` allows step-by-step execution during which the user can explore the state of the memory. `gdb` can be useful to debug multi-threaded applications.

The file `prod_cons_bug.c` is supposed to implement a producer-consumer synchronization between threads. Unfortunately this code is buggy.

Question 2.1: *Run the program described in `prod_cons_bug.c`. What is the problem that you observe?*

The goal of this exercise is to use `gdb` to identify the bug. You can find a description of how to use `gdb` with threads here: `https://sourceware.org/gdb/current/onlinedocs/gdb.html/Threads.html#Threads`. We summarize the main commands:

- `info threads`: Display a summary of all threads currently in your program.
- `thread threadno`: Make thread number `threadno` the current thread

Note also that you can interrupt an application running in `gdb` by sending it a `SIGINT` signal using `Ctrl-c`.

We also recall that for the current thread, the `backtrace` command provides a summary of how the thread reached its current state[1] and that you have to possibility to move in this stack of functions calls[2].

[1] `https://sourceware.org/gdb/current/onlinedocs/gdb.html/Backtrace.html#Backtrace`
[2] `https://sourceware.org/gdb/current/onlinedocs/gdb.html/Selection.html#Selection`

Question 2.2: *Fix the bug in the program using what you observe with* `gdb`*.*

3 Spinlock

In this exercise, you are asked to implement a spinlock to provide mutual exclusion for an *a priori* unknown number of threads.

Question 3.3: *What operations should be supported by the hardware to be able to solve the problem?*

Question 3.4: *Complete the provided file* `spinlock.c` *by implementing a spinlock using* `Test_And_Set()`*.*

Note: You can find the signature of the built-in atomic memory access operations supported by `gcc` on the documentation pages below. More precisely, in recent versions of GCC, there are two distinct APIs for this purpose:

- **the legacy API** (with function names having a **`__sync` prefix**): `https://gcc.gnu.org/onlinedocs/gcc-13.4.0/gcc/_005f_005fsync-Builtins.html`
- **the newer API for so called "memory model aware" atomic operations** (with function names having an **`__atomic` prefix**): `https://gcc.gnu.org/onlinedocs/gcc-13.4.0/gcc/_005f_005fatomic-Builtins.html`

For this exercise, we recommend using the legacy API (even though it will eventually be deprecated) because it is somewhat easier to apprehend than the newer API.[3]

Question 3.5: *Implement a new spinlock, using this time* `Compare_And_Swap()`*.*

The locks you have implemented until now do not ensure fairness. It is rather simple to implement a *fair* spinlock using `Fetch_And_Add()`. Such a lock is actually called a *ticket* lock.

Question 3.6: *Implement a fair spinlock using* `Fetch_And_Add()`*. By fair, we mean here that if thread A calls* `lock()` *before thread B, then thread A should be granted access to the critical section before thread B.*

Question 3.7: *Design a simple test to confirm that the new lock is fair and that the previous versions were not.*

[3] Note that, if necessary, it is possible to use the second API with in a way similar to the first one. For each `__sync` built-in function there is a corresponding `__atomic` built-in function. The arguments to these functions match (position and type), but the `__atomic` built-ins have an additional argument (memory model). Choosing the strongest memory model (`ATOMIC_SEQ_CST`) is a direct replacement for the `__sync` built-in functions. (This text is borrowed (with some simplifications), from the following page: `https://fedoraproject.org/wiki/Architectures/ARM/GCCBuiltInAtomicOperations`.)

Running the default test included in file `spinlock.c` with a adequate number of threads should show you that the lock based on `Fetch_And_Add()` is much less efficient than the *unfair* spinlocks.

Question 3.8: *What is the number of threads at which the performance of the ticket lock starts to really degrade?*

Question 3.9: *Try to explain this behavior.*