Advanced Computer Architecture 2016
Lab 2 — Multiprocessors and Memory Ordering

1 Introduction

The purpose of this lab assignment is to show the need for synchronisation and the effects of consistency in multiprocessors. To demonstrate this, we will use a very simple algorithm (Algorithm 1) where two threads operate on a shared variable in parallel.

A working implementation of Algorithm 1 needs to perform the increment and decrement of the shared data atomically, without interference from the other thread.

In this assignment we will use some x86 assembly code. You will not write any of the actual assembler instructions yourself, but you might still find the Intel Architecture Manuals[1] handy as a reference. Check the Lab Preparation Video 2 for an overview of all the assembler instructions that you might need.

It is highly recommended to solve this assignment in groups of two students. Talk to the teaching assistant if you, for some reason, want to work in some other configuration. This lab assignment is examined in the computer lab. During the examination, you will be asked to demonstrate and explain your solutions.

1.1 Atomic instructions

For simple tasks, such as incrementing or decrementing a counter, atomic instructions are the right tools for the job. The x86 ISA specifies a `lock` prefix that can be used with *some* instructions to force them to execute atomically. Some other architectures take a different approach. For example, the Alpha and PPC use a special load and

[1] `http://www.intel.com/products/processor/manuals/`

Algorithm 1 Test code for two threads sharing data. n is the number of iterations to execute and t is the thread number. We would intuitively expect $shared\_data$ to be 0 after both threads have executed, which is only the case if the implementation is properly synchronized.

Require: $shared\_data = 0$
  for $i = 1$ to n **do**
    if $t = 0$ **then**
      $shared\_data \Leftarrow shared\_data + 1$
    else
      $shared\_data \Leftarrow shared\_data - 1$
    end if
  end for

a conditional store instruction that does not modify memory if the data the store depends on has changed. Such conditional stores can be used to implement other atomic instructions, such as compare and swap.

Atomic instructions are usually used to implement concurrent algorithms, such as hash maps and linked lists. They are also used to implement various mutual exclusion algorithms and barriers.

1.2 Memory ordering

The x86 architecture, like all modern processors, aggressively optimizes memory accesses using various caching and buffering mechanisms. These optimizations affect the order of memory accesses on the memory bus. However, the hardware is designed so that single-threaded user space applications can not detect such reordering.

When multiple threads are involved, the complexity increases significantly, and some of the reordering can be detected. For example, a thread executes a store followed by a load, due to buffering of the store, the load may complete before the store is globally visible. To force memory accesses to be ordered the x86 architecture provides a set of *fence instructions*.

We will only use the `mfence` instruction that prevents both loads *and* stores from being reordered over the fence. In addition to this instruction, the x86 ISA specifies separate load and store fences that only prevent reordering of one of the access types.

For simplicity and compatibility reasons, Intel requires that memory accesses are never reordered past atomic instructions. This allows most synchronization algorithms that use atomic instructions to work correctly without memory fences.

1.3 What are critical sections, and who is this Dekker guy?

The simplest way to update shared data structures is to define critical sections, where only one thread is allowed to execute at a time. A mutual exclusion (*mutex*) mechanism ensures that only one thread can execute in the critical section at any given time. In addition to ensuring mutual exclusion, mutex implementations also ensure that memory accesses can not be reordered to ocur outside the critical section.

Several algorithms have been developed to solve the critical section problem. We will use one called *Dekker's algorithm*.

Dekker's algorithm was attributed to the Dutch mathematician *Theodorus J. Dekker* in a manuscript from 1965 by *Edsger W. Dijkstra*. See Algorithm 2 for a pseudo-code description of the algorithm.

1.4 The Lab Assignment

All the files related to this part of the assignment can be downloaded from the course forum. Download them and extract them in a suitable working directory in your home directory. The provided skeleton code consists of a handful files described below.

lab2.h Common data structures and declarations. No need to edit this file.

lab2.c Common code for running the experiments. You do not need to make any changes in this file.

Algorithm 2 Code for thread i to run a critical section, thread j is the second thread that competes for the critical section. The $turn$ variable should be initialized to 0 and both $flag$ variables should be initialized to $False$ prior to executing the algorithm.

$flag_i \Leftarrow True$
while $flag_j$ **do**
  if $turn \neq i$ **then**
    $flag_i \Leftarrow False$
    while $turn \neq i$ **do**
      Do nothing or sleep
    end while
    $flag_i \Leftarrow True$
  end if
end while

Do critical work

$turn \Leftarrow j$
$flag_i \Leftarrow False$

lab2_asm.h Inline-assembler implementations for all of the atomic instructions used in this assignment. No need to edit this file, but it is a good idea to read through this file to see what the atomic instructions look like.

cs_pthread.c Reference implementation of the synchronisation code. You do not need to edit this file.

cs_dekker.c Implement Dekker's algorithm here.

test_critical.c Implement Algorithm 1 using using critical sections here.

test_incdec.c Implement Algorithm 1 using atomic increments and decrements here.

test_cmpxchg.c Implement Algorithm 1 using atomic compare and exchange instructions here.

Makefile Automates the compilation. You can simply type **`make`** to compile both the pthreads version and the version using your own synchronisation primitives. You may use **`make clean`** to remove automatically generated files from the working directory.

Despite you should not modify the `Makefile`, it might be useful to have a look inside. The file contains rules to build the `lab2` binary. You compile the application by executing the **`make`** command in the source directory. There are multiple targets in the `Makefile`, e.g. the *clean* target. To execute the *clean* target, which cleans up the working directory, you can simply run **`make clean`**.

The test type and critical section implementation can be selected using command line options. Run **`./lab2 -h`** for information about available options.

You should be able to test whether you have placed your critical sections correctly by using the *pthreads* critical sections implementations. Once the critical section has been placed correctly, you may start working on your implementation of Dekker's algorithm.

1.5 Tasks

Perform the following tasks on a multicore x86 machine. You may use the department's UNIX machines. If you use your own laptop, make sure it has a high-end, (i.e. not Atom based) multicore processor.

1. Run the *critical section* tests with the *pthreads* critical sections implementation (**`./lab2 -t critical -c pthreads`**). Does the counter return to its initial value? Why?/Why not?

2. Insert calls to `enter_critical` and `exit_critical` into `test_critical.c` to enter and exit critical sections to allow for correct parallel execution of the test. Use the *pthreads* version to test this. Does the counter return to the initial value now?

3. Implement Dekker's algorithm for synchronisation in `cs_dekker.c`.

   (a) Why do the *flag* and *turn* variables have to be volatile?

   (b) Why doesn't the straightforward implementation work?

4. Add suitable memory barriers to the code to make the synchronisation work correctly. Use the `MFENCE` macro.

5. Implement Algorithm 1 in `test_incdec.c` using atomic and non-atomic *inc* and *dec* instructions. You may (read: should) use the functions defined in `lab2_asm.h`. What happens when you use the non-atomic instructions? Why? What happens when you use atomic instructions?

6. Implement Algorithm 1 in `test_cmpxchg.c` using both atomic and non-atomic compare and exchange instructions. You may use the functions defined in `lab2_asm.h`. What happens when you use the non-atomic instructions? Why? What happens when you use atomic instructions?

7. Compare the runtime performance when using critical sections and atomic increments/decrements. Which one is faster and why?

8. Compare the runtime performance when using atomic and non-atomic increments/decrements. Is there any difference in performance? Why?

9. **Bonus:** Implement queue locks in `cs_queue.c` using atomic instructions. See the lecture notes for details about the algorithm. Show your solution to the course assistant.

2 Revisions

2011 — Andreas Sandberg
2012 — Jonas Flodin
2013 — Andreas Sembrant, Mahdad Davari
2015 — Germán Ceballos, Moncef Mechri
2016 — Germán Ceballos