Introduction to Lab 2

Andreas Sandberg <andreas.sandberg@it.uu.se>

Division of Computer Systems
Dept. of Information Technology
Uppsala University

2010-09-28

What is lab 2?

. . . or what is consistency, and who cares anyway?

The purpose of this assignment is to give insights into:

1. how to program muli-processors
2. why synchronization is needed
3. how synchronization may be implemented
4. how memory consistency affects program behavior
5. how heavy-weight synchronization can be avoided with atomic instructions

What is a process?

A process contains the following:

- A set of memory mappings (heap, code, etc)
- Environment variables
- Signal handlers
- A list of open file descriptors (files, devices, network connections, etc)
- UID/GID/PID and some more TLAs[1]
- One or more *threads*.

[1] Three Letter Abbreviations

What is a thread?

A thread is an independent flow of control within a process

What is a thread?

A thread contains:

- A set of registers. Including:
  - Program Counter
  - Stack Pointer
- A scheduling priority

Why do we need synchronization?

```
if (balance > amount)
        balance = balance − amount;
```

What happens if multiple threads execute the code above at the same time?

How do we update shared state correctly?

Bringing order to chaos

Two common approaches:

- Use critical sections
  - Heavy-weight approach.
  - Operating systems usually provide an API to do this.
- Atomic instructions
  - Relatively light-weight compared to above method.
  - Serializes memory accesses on the system.
  - May need to write assembler or use compiler pragmas/intrinsics.

x86 memory ordering

- Defined in Volume 3A (System Programming guide) of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*.
- Memory ordering depends on access type:

  Processor Ordering for “normal” memory operations. Very similar to *Total Store Order*.

  Total Lock Order for instructions with the `lock` prefix. Atomic instructions behave as if the system implemented *Sequential Consistency*.

What is Processor Ordering?

An incomplete description

In an individual processor:

- Writes are not reordered with other writes.
- Reads may be reordered with older writes to different locations.

In a multi-processor system:

- Writes by a single processor are observed in the same order by all processors.
- Writes from an individual processor are *not* ordered with respects to writes from other processors.
- Memory ordering obeys causality.
- Any two stores are seen in a consistent order by processors other than those performing the store.

Forcing memory order

It is possible to force memory ordering using memory fences.

Assembler:

mfence

GCC intrinsics:

```
__builtin_ia32_mfence();
```


What is an atomic instruction?

- Atomic instructions perform their action as *one* unit without exposing intermediate state
- Naturally aligned loads and stores (up to 64 bits) are generally atomic, i.e. it's impossible to read a half-updated word.[2]
- Most instructions accessing memory can be turned into atomic instructions by adding a `lock` prefix.

[2] They still adhere to *Processor Ordering* and *not Total Lock Order*.

Simple examples

Incrementing a number:

```
lock inc 0x0(%eax)
```

Decrementing a number:

```
lock dec 0x0(%eax)
```

Exchange

```
xchg %eax, 0x0(%ebx)
```

- Exchanges the value in memory location `0x0(%ebx)` with the value in `%eax`
- Always atomic, the `lock` prefix is optional

Compare and exchange

```
lock cmpxchg %ebx, 0x0(%ecx)
```

- Uses `%eax` as an implicit operand
- Is `%eax` is equal to `0x0(%ecx)`?
  - true Write `%ebx` into `0x0(%ecx)`
  - false Write `0x0(%ecx)` into `%eax`

Background

. . . or who is this Dekker guy anyway?

- Dekker's algorithm solves the critical section problem for 2 threads without fancy hardware support.
- Attributed to the Dutch mathematician *Theodorus J. Dekker* in a manuscript from 1965 by *Edsger W. Dijkstra*.

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

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

Limitations

- Only works for two threads.
  - . . . but we don't care.
- Does *not* work with weak consistency models.
  - Requires memory barriers to force the processor to order accesses.

In the lab

What you'll be doing (hopefully)

- You will:
  - Implement Dekker's algorithm and use memory barriers to make it run correctly on x86.
  - Implement a simple algorithm using different types of atomic instructions instead of critical sections
  - Do performance studies for different types of implementation strategies

Bonus Implement queue locks using atomic instructions

- Complete lab manual on the course homepage[3]

[3] http://www.it.uu.se/edu/course/homepage/avdark/ht10

Important dates

- Groups:

Prep. Room 1549, *now*–17:00
A 2010-09-30, Room 1549, 08:15–12:00
B 2010-09-30, Room 1549, 13:15–17:00
C 2010-10-01, Room 1549, 08:15–12:00

- Deadline: See course homepage

Summary

And remember. . .

Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.[4]

[4] http://www.lysator.liu.se/c/ten-commandments.html