Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

System Construction Course 2015,

Assignment 8

Felix Friedrich, ETH Zürich, 10.11.2015

Implementing a Mutex on Lock-Free A2

Goal of this exercise is to implement a scheduled lock on a lock-free operating system, thereby taking into account lock-free implicit cooperative scheduling.

Lessons to Learn

- Understand mechanisms of lock-free scheduling and cooperative scheduling
- Investigate performance differences of spin-locks vs. scheduled locks

Preparation

1. **Update your repository**. Copy new and modified files / directories from the `Work` folder located in (repo)/a2/ to your work directory.
2. **Recompile the compiler**: Within A2, open file `Fox/Fox.Tool` and execute the contained compile command. Restart A2 before using the compiler.
3. Connect the Raspberry Pi to your computer using the Usb-To-Serial TTL cable as we did in Assignment 2. Connection to the RPI can be done via the WMV24Component in A2 but you are also free to use any telnet client you prefer.

The tool for this exercise contains scripts for x86 based computers and for Raspberry Pi. The former can be used in otder to run the code in a virtualization environment such as bochs, qemu, VirtualBox or VMWare or on bare hardware. The latter will be used in order to run the tests on our Raspberry Pi computer and is recommended for this exercise.

Implement a Mutex

A mutex is a data structure with an algorithm to enforce mutual exclusion of several threads on a set of code regions. For a given mutex `m`, the code regions consist of intervals designated by pair of statements `Mutexes.Acquire(m)` and `Mutex.Release(m)`.

Prepared for this exercise are files `Mutexes.Mod` containing a `CAS`-based spin-lock implementation of Mutexes and `TestMutexes.Mod` to test the correctness and performance of the implementation.

Tasks

1. Build the kernel using the script in the assignment tool. Copy the kernel to the SD card of the RPI, open a terminal connecting via serial port to the RPI and check that you see the output of the test module.
2. The stub module `Mutexes.Mod` does not implement a mutex yet. Implement a mutex as a CAS-based Spin-lock first. Check that it works and observe its performance.
3. Change the implementation of the mutexes in `Mutexes.Mod` such that they employ a scheduled lock instead of a spin-lock. Of course, you are free to experiment later and implement a hybrid scheme but for now you ”just” implement the mutex such that when the mutex is currently taken by a different process, then the process goes sleeping on some waiting queue. Test correctness and performance of your implementation.

Hints

- The expression `CAS(v, oldvalue, newvalue)` returns the observed value of the variable `v` with the corresponding type of `v`.
- `Activities.GetCurrentActivity()` returns the currently running activity (process).
- Module `Queues.Mod` contains a lock-free queue with precautions against the ABA problem using Hazard-Pointers as discussed in the lecture. While you are encouraged to browse and understand the implementation, you can consider it a given and don't have to worry about lock-free queues in this assignment. `Queues.Enqueue(act,q)` can be used in order to enqueue an activity `act` to queue `q`. `Queues.Dequeue(it,q)` can be used in order to dequeue an item `it` from queue `q`. It returns `TRUE` upon success. Recall that type guards of the form `it(Activities.Activity)` can be used in order to interprete `it` as of type `Activities.Activity` in a type-safe way.
- The most intricate part of this assignment is about scheduling. Recall from the lecture that it is often required to execute a switch finalizer on behalf of the new scheduled thread as indicated with the following lines of code:

```
IF Activities.Select (nextAct, Activities.IdlePriority) THEN
  Activities.SwitchTo (nextAct, SwitchFinalizer, ADDRESS OF Data);
  Activities.FinalizeSwitch;
END;
```

  `Activities.Select` fetches a new process from the ready queues with at least idle priority. `Activities.SwitchTo` performs a synchronous context switch to `nextAct`. This resumed activity continues its execution by first calling the specified finalizer procedure with the given argument. As a rule of this lock-free kernel, each invocation of `Activities.SwitchTo` must be followed by a call to the `Activires.FinalizeSwitch`. This finalizes the task switch performed by calling the task switch finalizer of the previously suspended activity.

  For an example use of this mechanism, refer to procedure `Activities.Switch`.
- Be aware that in lock-free programming at any times something can happen. Consider all possible interleavings.

Documents

- System Construction Lecture 8 and Lecture 9 slides from the course-homepage `http://lec.inf.ethz.ch/syscon`
- F. Negele, Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System, Dissertation, ETH Zürich (2014)