Exam in
EDAN25 Multicore Programming

May 26, 2011, 08.00 — 13.00

Inga hjälpmedel!
30 / 60 points are needed to pass the exam.

Examinator: Jonas Skeppstedt, tel 0767 888 124

1. (10p) What can `helgrind` help multicore programmers with?

   Answer *Three main areas of errors are detected:*

   (a) Invalid uses of the Pthreads API. Examples include unlocking a mutex not owner, and locking a non-recursive mutex multiple times.

   (b) Data races.

   (c) Inconsistent lock orderings.

2. (10p) Explain briefly the synchronization primitives mutex and condition variable. Why should the waiting for a condition be in a loop?

   Answer *A mutex is a lock with a sleep queue. A condition variable lets a thread wait for something to happen in the future, such as a change in the state of the data protected by a mutex. When a thread calls* `pthread_cond_wait` *with pointers to both a condition variable and an owned mutex, it atomically unlocks the mutex and starts waiting for the event. When it is signalled, by* `pthread_cond_signal` *(or* `pthread_cond_broadcast`*) it wakes up with the same mutex owned.*

   The reasons for waiting in a loop are:

   - *intercepted wakeups: a thread which owned the mutex and signals the condition variable can either unlock the mutex first or call signal first. If the mutex was unlocked first (which can reduce the number of context switches) a third thread can take the lock before the signalling occurs and this is called an intercepted wakeup. Since the third thread can change the state the first thread may need to wait again. Hence a loop is needed.*
   - *loose predicates: a predicate which indicates the woke up thread might have something to do and if it turns out it has not it should continue waiting.*
   - *spurious wakeups: a waiting thread can sometimes be woke up due to an unrelated event – such as a received UNIX signal. Again a loop is needed.*

3. (10p) What is false sharing and give examples of what programmers can do to reduce the risk of it?

   Answer *False sharing is the sharing of cache blocks without sharing of data which leads to useless cache misses since no new information is communicated at the cache miss. Distribute work in larger blocks. If suitable, allocate a complete cache block for certain data (but that can lead to other problems). Instead of data in an array indexed by thread id, put pointers to the objects in the array instead.*

4. (10p) Describe the architecture of the Cell processor and explain what the purpose is of using different tags in the DMA operations in an SPU program?

   Answer

   - *A Cell processor consists of:*
     - *One multithreaded 64-bit PowerPC processor (currently two threads)*

  - *Eight SIMD vector SPU processors with 128 registers and 256 KB RAM*
  - *One programmable memory interface controller*
  - *Two input/output interfaces*
- *The PowerPC processor is optimized for general purpose computing and SIMD vector processing*
- *The SPUs (synergistic processor units) are optimized for SIMD vector processing and concurrent data transfers.*
- *The SPUs have no caches and both data and instructions must be explicitly transferred from system memory by software using DMA.*
- *DMA requests are tagged with an integer in the range* 0..31 *and the SPU can wait for all pending requests with a certain tag value. This way, by using different tags for different loop iterations (for example) the programmer can achieve concurrency within an SPU since both computation and data transfers are happening concurrently.*

5. (10p) Consider the program fragment below with three threads. Explain how a sequentially consistent machine guarantees that if T_2 reads the value one from A, then if T_3 reads one from B, T_3 will also read one from A. Assume the instructions are executed by the different processors in the order shown.

```
int A = B = C = 0;

T1:             T2:             T3:
A = 1;
                if (A)
                    B = 1;
                                if (B)
                                    C = A;
```

Answer *Sequential consistency guarantees program order of memory accesses and write atomicity. Write atomicity guarantees that all processors see the order of writes to a particular memory location in the same order and that no processor may read the modified value before the write has completed (e.g. removed all copies). Since T_2 may not see the new value of A before T_3 is aware of the write (and removes its copy in case of a write-invalidate cache coherence protocol), it's certain that if T_3 reads* 1 *from B it will also read* 1 *from A.*

6. (10p) Compare release consistency with weak ordering.

Answer

- *Both WO and RC are relaxed memory models and distinguish between data accesses and synchronization accesses. Both allow reordering of data accesses.*
- *WO has one synchronization access, called sync, while RC has two: acquire and release.*
- *WO prevents reordering of sync and data accesses (and sync and sync) while RC allows the following reorderings:*
  - *data followed by acquire*
  - *release followed by data*
  - *release followed by acquire*