IWR, University of Heidelberg
Exercise Sheet 6

Winter term 2014/15
27. November 2014

Exercise for Course

Parallel High-Performance Computing

Dr. S. Lang

Return: 04. December 2014 at the beginning of the exercise or earlier

---

Task 12 Peterson Lock with ThreadTools (5 points)

Consider the following program segment, where two threads increment a common shared variable. Of course the processes are in general not executed sequential after each other, because of that one will not get the expected result 20 000 000 that had been computed by pure sequential execution:

```
parallel increment
{
  const int sections = 10000000;
  int count = 0;

  Process Π1                          Process Π2
  {                                   {
    for (int i=0; i<sections; i++)      for (int i=0; i<sections; i++)
    {                                   {
      count += 1;                         count += 1;
    }                                   }
  }                                   }
}
```

The critical section (critical section, CS) can for example be locked with the peterson algorithm presented in the lecture. This is shown with our abstract notation in the following listing:

```
parallel increment-peterson
{
  const int sections = 10000000;
  int in1 = 0, in2 = 0, last = 1;
  int count = 0;

  Process Π1                          Process Π2
  {                                   {
    for (int i=0; i<sections; i++)      for (int i=0; i<sections; i++)
    {                                   {
      in1 = 1;                            in2 = 1;
      // *                                // *
      last = 1;                           last = 2;
      // *                                // *
      while (in2 ∧ last == 1);            while (in1 ∧ last == 2);
        count += 1;                         count += 1;  // CS
      in1 = 0;                            in2 = 0;
    }                                   }
  }                                   }
}
```

Subtask (a)

On the lecture webpage the ThreadTools are provided for you in a `zip` compressed file `threadtools.zip`. These implement some wrapper classes, that simplify the functionality of `PThreads` by object-oriented realization. In the lecture these have been called *ActiveObjects*. Please consider the hints regarding the ThreadTools at the end of the sheet and on the webpage. You find with the ThreadTools an implementation of the above shown naive Peterson lock in the file `checkpeterson.cc`. You can compile the program with the command `make`. Test with at least 10 runs, whether the variable `count` stores the „correct“ result. Look next into the Makefile at lines 4 and 5:

```
4 CFLAGS     = -g -O0 -c -Wall
5 #CFLAGS     = -O3 -c
```

Comment out line 4 and in line 5, and compile new with `make clean` and then `make`. Hereby you generate the optimized code. Repeat the measurements. Do you obtain with the non-optimized as well as the optimized code the correct results? Can you explain, why the Peterson lock does also not work in the non-optimized case?

Subtask (b)

In the file `membarrier.hh` you can find a so-called *memory barrier*, that is realized by an assembly instruction. By means of this memory barrier the *out-of-order-execution* can be influenced manually resp. avoided. For repetition: By out-of-order execution machine instructions can be executed in advance, if the data, that is needed, is already available calculated in memory. The memory barrier can now instruct the sequence that is used to execute instructions or memory operations:

```
OP_1;
...
OP_n;
memBarrier();
OP_{n+1};
```

The memory barrier forces here, that the operations `OP_1` to `OP_n` have been finished completely, before `OP_{n+1}` may be executed. Insert now at the locations, that are signed with a star `// *`, a memory barrier, and test again several times with and without optimization (perform before a `make` unconditionally *always* a `make clean`!). How does your program behave? Discuss shortly your observations.

Subtask (c)

Now, remove the `memBarrier()` calls again. In C/C++ you can mark memory areas in such a way, that the sequence of read- and write operations onto these memory areas is preserved during compilation of the program. This is perfored with the keyword `volatile`. Read- and write operations for `volatile` variables always are exactly in programcode sequence inside the compiled program and may not be removed by optimization. Designate the four common variables `in[0]`, `in[1]`, `last` and `count` as `volatile`. Recompile again with and without optimization and repeat the experiments of (a) and (b) several times. What can you observe? Try to explain the effects, that you have observed!

Subtask (d)

Repeat Subtask (c) again, but now with the call of the memory barrier at the signed locations. Which effects can you observe now?

Task 13 ThreadTools: Semaphores (5 points)

In the lectures you have learned about the semaphore concept as well as the inactive waiting with condition variables. In the ThreadTools, that have been introduced in the last task, there is a class `Semphore`, that realizes the data type `Semaphore` by condition variables. Herefore it is derived from the class `Condition` and has the following code:

```
/** Implements a semaphore deriving from a condition variable.
 */
class Semaphore : private Condition<unsigned int>
{
public:

  //! \brief make a semaphore with initial value
  Semaphore (int init);

  //! \brief make a semaphore with initial value 0
  Semaphore ();

  //! \brief decrement value
  void P ();

```

```
    //! \brief release semaphore
    void V ();
};
```

It is your task to implement $P()$ and $V()$ methods of semaphores. For this you need the methods `acquire()`, `wait()`, `release()` and `signal()` of the base class `Condition`. The class `Condition` contains moreover a variable `value`, that is incremented or decremented by the semaphore. The provided ThreadTools contain a file `semaphore.hh` with the above header information as well as additionally in the file `semaphore.cc` the skeletal structure, whose methods you have to implement.

Task 14 ThreadTools: producer-consumer problem with ring buffer (10 points)

In the lecture you have discussed, how producer-consumer problems can be solved with semaphores. This problem shall now be implemented with the ThreadTools (in the lecture called `ActiveObjects`): A buffer is filled from a producer. Has the producer reached the end of the buffer, he rewrites the beginning of the buffer, older requests stored there shall not be overwritten. A consumer reads requests to be processed from buffer. For the program you need

- a buffer, that can store the desired data type,
- a semaphore, that locks free buffer locations,
- a semaphore, that locks occupied buffer locations,
- initialize the semaphore, that locks free buffer locations, at the beginning with the number of buffer locations, that are initially available.

Now write classes, that implement producer and consumer. These shall be derived from the class `BasicThread` and have to overload the virtual method `run()`, where producers and consumers perform their tasks. Use for the implementation the provided empty code skeleton `producerconsumer.cc`. This can be compiled with the command `make` without modification of the `Makefile`. For this task you need the solution of the prior Semaphore task. Who cannot solve it, may inform the instructor via email to get a solution.

Test your program with a buffer length of 5. The buffer may store arbitrary data types. The producer shall generate in a loop 20 pieces and write the current loop count as virtual good into the buffer. Let both threads print which buffer location they currently process and which good (loop iteration) they have actually stored or resp. retrieved.

Hints for ThreadTools (ActiveObjects)

The `ActiveObjects` introduced in the lecture are called in the exercises `ThreadTools`. A thread is realized by the base class `BasicThread`.

- On the lecture webpage you find a zip archive `threadtools.zip`, that provides the classes for the `BasicThreads`, general tools for thread-handling and several examples. It is also a Makefile added, compilation with Linux is done with `make`.
- You can use all methods from ThreadTools by inclusion of the file `tt.hh`. For the solution of Semaphore task you do not need to adapt the `Makefile`.
- For the solution of the Producer-Consumer-Problem please use the provided skeleton `producerconsumer.cc`. This is in the `Makefile` a target and is compiled when issueing a `make`.
- The skeleton `resources.cc` is needed for the next exercise sheet.
- Further read the detailed information on the webpage.