Transactional Memory

For this assignment, we will be using the transactional memory support that is available in GCC $\geq$ 4.7. Start by reading `http://gcc.gnu.org/wiki/TransactionalMemory` for a brief overview.

Transactional memory support in GCC is still an experimental feature. If you happen to find any bugs (including particularly poor performance), please report them!

Exercise 1: Warm-Up

1. Download `transactions.cpp` from the Student Portal. Find out how to compile the file with GCC. (Hint: `man gcc`.) Run the resulting executable. What value is reported for `diff`? **(1 pt)**

2. Read the description of `__transaction_atomic` near the start of Section 4 in the *Draft Specification of Transactional Language Constructs for C++*. In data-race-free programs, do atomic transactions in GCC provide *strong* or *weak isolation*? **(1 pt)**

3. Edit `transactions.cpp` to turn all transactions into ordinary (i.e., non-transactional) statements (by removing all occurrences of the `__transaction_atomic` keyword). Re-compile and re-run the program. What output do you observe? **(1 pt)**

4. Discuss the program considered in Question 3. Does it contain race conditions and/or data races? What does this imply for its behavior? (See `http://www.hboehm.info/c++mm/why_undef.html` for some background information on the semantics of data races in C++.) **(2 pts)**

Exercise 2: Double-Ended Queue

Recall the implementation of a double-ended queue based on a doubly-linked list with a sentinel node at each end (see Figure 1) that was discussed in class.

Use this approach to implement a C++ class `DQueue` with the following public methods:

```
// constructs an empty deque
DQueue()
// pushes the given value to the left end of the deque
void PushLeft(int val)
// pushes the given value to the right end of the deque
void PushRight(int val)
// pops the leftmost value from the deque (-1 if empty)
int PopLeft()
// pops the rightmost value from the deque (-1 if empty)
int PopRight()
```

Do not use container classes from the C++ Standard Library (such as `std::deque` or `std::list`) for this exercise.

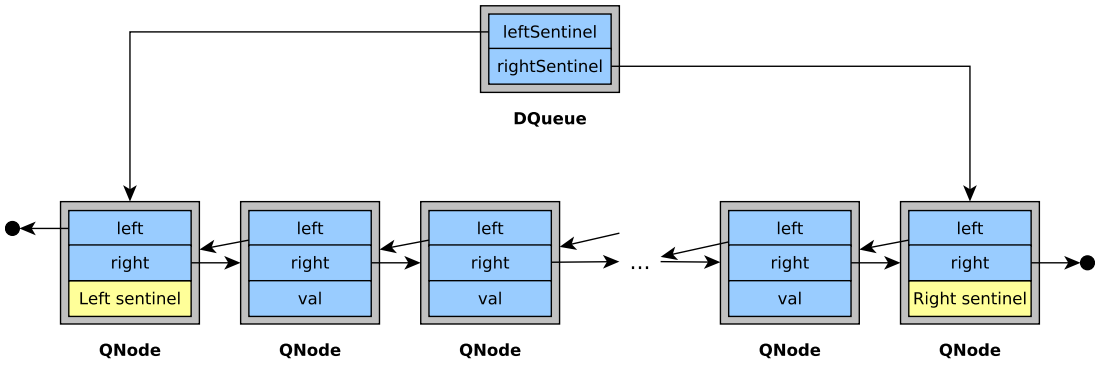

Figure 1: A double-ended queue (aka *deque*), implemented as a doubly-linked list with a sentinel node at each end.

Use `__transaction_atomic { ... }` where appropriate to implement the four push/pop methods in a thread-safe manner. Do not use other forms of synchronization (such as locks or C++ atomic types).

You can start by extending the skeleton code in `deque.cpp`, which is available on Studium. **(5 pts)**

Exercise 3: Performance Measurements

Write a driver program that exercises your `DQueue` class to determine its performance under different workloads. For each of the following measurements, use 1, 2, ..., N concurrent threads, where N is the number of logical CPUs available. Try to run your measurements on a machine with at least 8 logical CPUs.

As before, you can use `lscpu` to print the number of CPUs and other information on the CPU architecture.

Monitor what else is going on on the machine, especially if you conduct these measurements during lab hours!

1. Each thread repeatedly calls `PushLeft`. Determine the total number of push operations that are executed per second. (Make sure your measurements are not affected by thrashing.)

2. Each thread repeatedly calls `PushLeft` followed by `PopLeft`. Determine the total number of push/pop pairs that are executed per second.

3. Each thread repeatedly calls `PushLeft` followed by `PopRight`. Determine the total number of push/pop pairs that are executed per second.

For each measurement, visualize your findings in a plot with operations/s on the Y axis and number of threads on the X axis. **(3 pts)**

Write a 10-15 line summary that explains the performance plots. **(2 pts)**

Remarks: `transactions.cpp` demonstrates a way to execute code for a specified amount of time. If your driver program uses shared data (e.g., a counter for operations executed), be careful to avoid data races.

— Part 2 —

Exercise 4: Semantics: Exceptions

GCC allows to throw an exception from `__transaction_atomic { ... }` transactions.

1. Write a short (sequential) program that throws an exception from an atomic transaction, and produces different output depending on whether the transaction is committed or aborted. **(1 pt)**
2. Run your program. Is the transaction committed or aborted? **(1 pt)**
3. Read Section 9 of the *Draft Specification of Transactional Language Constructs for C++*). What is the semantics of the cancel-and-throw statement (without the `outer` attribute)?[1] **(1 pt)**

Exercise 5: Semantics: Nesting

GCC supports nesting of `__transaction_atomic` blocks, and a `__transaction_cancel` statement to abort atomic transactions (see Section 4.4 and Section 8, respectively, of the *Draft Specification of Transactional Language Constructs for C++*).

Do *not* consider the `outer` attribute (Section 8.1) for this exercise.

1. Write a short (sequential) program that produces different output, depending on whether the semantics of nesting is *flattened*, *closed* or *open*. **(2 pts)**
2. Run your program. What is the semantics of nesting in GCC? **(1 pt)**

Exercise 6: Relaxed Transactions

1. Download `relaxed.cpp` from Studium. Try to compile the file with GCC, and explain why this fails. Relate your explanation to Section 4 of the *Draft Specification of Transactional Language Constructs for C++*). **(1 pt)**
2. In line 6 of the file, replace `__transaction_atomic` with `__transaction_relaxed`. Compile and run the program. What output do you observe? **(1 pt)**
3. What (other) outputs would be possible? Does `__transaction_relaxed` provide SLA and/or TSC (for race-free programs)? Relate your answer to Section 3 of the *Draft Specification of Transactional Language Constructs for C++*. **(2 pts)**

[1] You don't have to write code to try this out. Not every version of GCC actually supports the cancel-and-throw statement.

— Part 3 —

Exercise 7: Work Stealing

Consider a system in which a user regularly creates new *jobs*. Each job is modeled by a non-negative integer that indicates its duration in seconds.

Jobs are processed by N *processors*. Each processor maintains its own *job queue*. New jobs arrive at the right end of the queue. A processor that is idle removes the leftmost job from its queue and executes it. (That is, for the duration of the job, the processor cannot do anything else. You may simply let the processor `sleep` for the specified duration to model this.)

When its own job queue is empty, an idle processor checks the job queues of all other processors. If the *rightmost* job in some other queue has a duration > 0, the idle processor steals the job. Stealing means that the idle processor removes the job from the other processor's queue and executes it (without further queuing).

Extend the code in `work-stealing.cpp`, which is available on Studium, to implement processors with the described behavior. Use your `DQueue` class from Exercise 2 to implement job queues. Your program should produce output similar to the following:

```
0: scheduling a job on processor 1 (duration: 3 s).
  Processor 3: stealing job (duration: 3 s) from processor 1.
1: scheduling a job on processor 0 (duration: 6 s).
  Processor 1: stealing job (duration: 6 s) from processor 0.
2: scheduling a job on processor 3 (duration: 7 s).
  Processor 2: stealing job (duration: 7 s) from processor 3.
3: scheduling a job on processor 2 (duration: 4 s).
  Processor 0: stealing job (duration: 4 s) from processor 2.
4: scheduling a job on processor 3 (duration: 5 s).
  Processor 3: executing job (duration: 5 s).
5: scheduling a job on processor 0 (duration: 1 s).
6: scheduling a job on processor 0 (duration: 4 s).
  Processor 1: stealing job (duration: 4 s) from processor 0.
  Processor 0: executing job (duration: 1 s).
...
```

Remarks: Note that jobs with a duration of 0 should never be stolen. Since the `DQueue` class from Exercise 2 does not provide a method to peek at the rightmost job without removing it from the queue, idle processors will have to use `PopRight` to obtain the rightmost job, followed by `PushRight` if the job's duration was 0. The intermediate state, in which the job is erroneously removed, should not be visible to other processors.

Use `__transaction_atomic { ... }` or `__transaction_relaxed { ... }` where appropriate. Do not use other forms of synchronization (such as locks or C++ atomic types). **(10 pts)**

Grading

Your total exercise points (0-35) will be *divided by 3.5 and rounded (half up)* to arrive at the scale of 0–10 points for this assignment.

Submission Instructions

Please submit all textual answers (including graphs and figures) in one single PDF file.

Explanations may be brief, and should rarely exceed 10-15 lines per question. However, you should include enough detail to convince the reader that you understand the topic and cared to take the question seriously.

Please submit all programs that you wrote in separate `.cpp` files. The file names should clearly indicate to which exercise each program belongs. (It is *not* necessary to submit your modified programs for Exercises 1.3 and 6.2.)

Please use the submission portal on Studium to submit your files.

Submission deadline is **Friday, 2021-11-26 at 17:00**.