CS 15-618 Project Checkpoint 1
Synchrony (ID 24)

Patricio Chilano (pchilano)
Omar Serrano (oserrano)

April 27, 2018

Summary

We are right on track with our original schedule, but we are making some updates to our plans in response to the feedback we received on our proposal, which includes:

- Reframing our approach to the problem to be more purposeful.
- Substitute the Java concurrent package with a C/C++ multi-threading library when making comparisons with external libraries.
- Change our *nice-to-have* to include parallelization of a problem or algorithm using our concurrent data structures.

All of these are in a fluid state because we've just received the feedback, but these pivots shouldn't affect us negatively, because we are still working on tasks that are not affected by them.

Work Done

We created our git repo along with a couple of scripts to setup our working environment: one script to setup git to use git-clang-format so that uniform formatting is maintained among teammates, and another one to install and setup the cmake environment used for the building process.

We implemented a basic single-threaded doubly-linked list and a coarse-grained multi-threaded doubly-linked list. We also created a HashMap class that can be changed to use any specific doubly-linked list for implementing the buckets. Since each bucket can be managed independently from the others, once we have a doubly-linked list implemented with some specific synchronization mechanism, implementing the equivalent for the HashMap should be straighforward. Figure **??** and Figure **??** show the interface for our list and HashMap respectively.

Finally we used GoogleTest to create unit tests for the lists and the HashMap. As of now we just created tests to check for correctness, not benchmarks to measure speedup.

```
/**
 * A simple doubly linked list with basic operations:
 */
template <typename T> struct DlList {
  DlNode<T> *head{nullptr};
  unsigned size{0};

  ~DlList();
  DlNode<T> *Insert(T value);
  bool Remove(T value);
  bool Contains(T value) const noexcept;
  T *Find(T value) const noexcept;
  unsigned Size() const noexcept;
  bool Empty() const noexcept;
};
```

Figure 1: This is the single threaded list, but other lists share the same interface.

```
/**
 * A simple HashMap with basic operations:
*/
template <typename K, typename V> class HashMap {
  class Element {
    K key;
    V value;
  };
  DlList<Element> map[MAX_BUCKETS];
  unsigned size{0};

  void Insert(K key, V value);
  void Remove(K key);
  bool Has(K key);
  unsigned Size() const noexcept;
  // using subscripts to read or write to the map
  V &operator[](K key);
};
```

Figure 2: This is the HashMap class. It can be easily changed to use any particular list for the buckets

Week	Assigned To	Deliverable
4/16	O	Doubly-linked list with fine-grain locks
	O,P	Find third party C/C++ multi-threading library
	P	Lock-free doubly-linked list
4/23	O	Hash map with fine-grain locks
	P	Lock-free HashMap
	O,P	**Check point 2**
4/30	O	Benchmarking harness for linked-lists
	P	Benchmarking harness for HashMap
	P,O	Benchmarks for third party C/C++ multi-threading library
	P,O	Compare results with hypothesis. Identify improvement spots
	O,P	Final report
	*	*Parallelize algorithm/problem with our data structures

Table 1: Weekly schedule. Check points include the items preceding them. The * on the last means it is a *nice-to-have*, but we are not commiting to it. The *Assigned To* column indicates with an O for Omar or a P for Patricio who will work on a given task.

Goals

We are still planning to deliver the following:

- Implement a concurrent doubly linked list using different synchronization methods: course-grain locks, fine-grain locks and lock-free techniques.
- Compare the performance of each implementation of the doubly linked list against each other and against the serial version with respect to the number of threads.
- Implement a concurrent HashMap using different synchronization methods: course-grain locks, fine-grain locks and lock-free techniques.
- Compare the performance of each implementation of the HashMap against each other and against the serial version with respect to the number of threads.
- Use some external C/C++ library that implements concurrent lists and HashMaps to compare it against our versions.

Instead of trying another data structure as a *nice-to-have*, we are now looking forward to the opportunity of applying our concurrent data structures to an algorithm or problem. We don't have concrete ideas right now, but have floated the idea of doing something like a search or graph algorithm, like A^*, for example.

Schedule

Table **??** contains an updated version of the schedule. The new schedule is the same as the original, minus completed tasks, and three modifications:

- Remove Java-related tasks, since we won't be using Java's concurrent package.
- Add a task for this week to find a C/C++ multi-threading library to use for benchmarking.
- Change *nice-to-have* to one where we apply our concurrent data structures to an algorithm or a problem.

The table contains a column to indicate how tasks are assigned. We are currently right on track with our original schedule.

Pending Issues

This is a summary of the pending issues we have to resolve:

- Define the concrete techniques we are going to use to implement fine-grain locking and lock-free mechanisms.
- Find a C/C++ library to use for benchmarking.
- Think of a problem or algorithm that could be parallelized using our data structures.