Performance Evaluation of Concurrency Primitives in Charcoal

Jacqueline Kay (jkay1) Joshua Gluck (jgluck2) Lisa Bao (lbao1)

5 October 2013

1 Abstract

At present there exist two major paradigms for handling asynchronous events: the loop/handler design pattern and multiple threads or processes run in parallel. Event-driven programming in the loop/handler pattern can simulate concurrency, but it cannot take advantage of modern multi-core processors. Multithreading allows the programmer to specify true simultaneous control flows, but correct and bug-free parallel programming is non-trivially more difficult [2]. The Charcoal programming language seeks to address these issues through a semi-cooperative threading framework called "activities" [4]. We propose to evaluate the comparative performance of different implementations of the yield concurrency primitive for activities in Charcoal.

2 Motivation

A fairly large proportion of modern computing must deal with asynchronous events, to one degree or another. Major examples include waiting for user input or receiving network packets. There have been two general methods of dealing with this problem: creating threads or processes to wait for these events and slice processor time between them, or utilizing event loop/handler patterns. Both of these solutions solve the problem of asynchronous event handling while also ensuring that processors do not spend a significant portion of their time waiting around doing nothing.

However, each of the existing primary solutions poses certain problems. Event loop/handler patterns create incredibly complicated control flow, since long-running tasks must be broken up into multiple event handling invocations. While multiple thread or process designs work well on a single CPU, the ability of threads and processes to run in parallel—combined with the advent of multiprocessor machines—has led to significantly more bug-prone thread-based event handling due to the inherently complex nature of parallel processing [2].

Therefore, creating simple, non-parallel control flow tools would greatly reduce the complexity of managing asynchronous events and increase software engineer efficiency by minimizing subtle bugs in the codes control flow.

3 Background

In event-driven programming, an event loop periodically checks for input messages or events. For instance, an event such as a key press may trigger a callback function which responds to this input by writing the a particular corresponding character to the screen. This technique is useful for reacting simply and appropriately to multiple sources of input which may occur asynchronously. However, it also presents a major problem: all computation must be completed in callback functions, while the event loop is reserved for waiting on the next event. This control flow can lead to messy, unconventional, and inefficient code as the programmer attempts to work around the enforced structure [3].

Event-driven programs often give the illusion of concurrency, but in fact they are not multithreaded [1]. Attempting to parallelize an event-driven program, perhaps by running event handlers on multiple processors, would seriously warp the intended control flow of the program. Multithreading addresses some of the problems of event-driven programming by allowing the programmer to express multiple simultaneous flows of control, rather than simulating concurrency with a series of callback functions. However, parallel programs are often more difficult to write because the programmer must handle race conditions and concurrent access to data.

In contrast, cooperative threads were designed with concurrent data access in mind. While conventional threads guarantee that each thread will eventually yield control to another thread, cooperative threads guarantee that a thread will never unexpectedly yield control to another thread [1]. Only one cooperative thread runs at a time, and a cooperative thread must explicitly yield before transferring control to another thread. Data structures that are shared between cooperative threads can only be accessed by other threads when the current thread gives up control of these resources.

Cooperative threads are equivalent to event-driven programs in their safety from race conditions. In fact, a program with cooperative threads can be converted into an equivalent event-driven program by replacing every block call with a return to the event loop. Thus, cooperative threads can be seen as a more expressive alternative to event-driven programming. However, a cooperating multithreading approach depends wholly on each thread yielding control at reasonable and appropriate periods of time.

4 Charcoal

We propose to compare various methods for implementing an "activity" in the Charcoal programming language [4]. Activities represent a middle ground in the choice between imperative thread asynchronous event handling and cooperative thread programming. They will be implemented in a manner similar to cooperative threads, but also influenced by the functionality of preemptive threads since the compiler includes a number of implicit yield invocations throughout the program.

We will focus our initial efforts on yield because of its importance to the Charcoal framework, specifically to the expected performance of activities frequently yielding control flow. If time permits, we will conduct a parallel performance evaluation for other concurrency primitives, such as mutex.

Goals

The milestone structure presented below affords us flexibility in case a milestone takes more time than expected. If we fall behind schedule and cannot feasibly implement more than one concurrency primitive, then we will focus on yield. Conversely, if implementation and testing one primitive takes less time than expected, we can redistribute some time in the schedule to work on other primitives.

4.1 Milestone 1

Complete at least two working implementations of yield which use different libraries or different algorithmic strategies. We plan to begin implementation work in a pair/trio programming session and then divide up the remaining work.

4.2 Milestone 2

Compile a variety of test cases and write a test benchmark for yield. Gather preliminary performance results comparing the two (or more) implementations. We plan for each team member to write a logical portion of the necessary test cases and gather experimental data for their benchmark.

4.3 Milestone 3

Implement and test other concurrency primitives, such as mutex. This work will be divided up similarly to the process of yield implementation, with any additional modifications based on our experience working as a team.

References

[1] Andreas Gustafsson. Threads without the pain. *Queue*, 3(9):34, 2005.

[2] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *ASPLOS*, pages 329–339, 2008.

[3] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). *Proceedings of HotOS IX*, 2003.

[4] Benjamin Ylvisaker. Charcoal: Easier concurrency for application developers. `http://charcoal-lang.org/`, 2013.