Project Report

Green Thread Library

Ningyu Zhang

Apr. 26, 2016

VUNetID: zhangn1

**Introduction**

Green threads can be viewed as “user-mode” threads that do not have the one-to-one relationship to a kernel thread (unlike POSIX threads). Thus, green threads can be scheduled by user libraries other than the operating system. Thus, an essence of a green thread library is the schedulers it supports. This project presents an attempt to implement a simple green thread library that is extended from an open source green thread library written in C (MPU, 2013, retrieved from <https://github.com/mpu>).

This project implements a series of key changes to allow a three simple schedulers (FIFO, RR, and OTHER) decide which thread in a green thread table is going to execute next based on 1) the entry/reentry time of the threads (FIFO), 2) the quantum or time slice that the threads have consumed (RR), and 3 the priority of the threads (OTHER). The method section will introduce these changes in detail.

**Method**

This project is built upon an open source C green thread library (MPU, 2013, retrieved from <https://github.com/mpu>). What the built-in scheduling policy in the previous library is to find the next green thread on the thread table that is in a “Ready” state and then yield the computational resources (unit64\_t registers) to the succeeding green thread. This default scheduling behavior is similar to an RR fashion except that there does not exist a concept such as quantum to limit the amount of time for the computational resource being available to the current green thread.

In order to enable the three different scheduling policies in the existing library, a few changes have been applied to the existing library. The following table summarizes four major types of changes.

|  |  |
| --- | --- |
| Change | Description |
| 1 | Declared an enum indicating the scheduling policy “FIFO”, “RR”, and “OTHER” and changed the signature of “gtinit” to specify the desired scheduling policy. It is also possible to switch the scheduling policy on-the-fly by changing a global variable “current\_policy.” |
| 2 | Added a global monitor “entrance\_counter” for a thread table to count how many green threads have entered or reentered the thread table. On the other hand, a new field “thread\_queue\_entrance” is added to the gt struct to track the index (time) of a thread being added to the table. The thread with the smallest index (earliest entry time) is guaranteed to be scheduled next. |
| 3 | Added a global constant variable “TIME\_SLICE” to indicate the quantum of a thread. Accordingly, a clock\_t variable “start\_time” is initialized at the beginning of the function “f” evoked by a green thread. Conceptually, the point after the printf statement could be viewed as a check point. Thus, the clock() function is called again to get the number of CPU clocks has been spend on the current thread. Once the CPU time exceeds the “TIME\_SLICE” (checked by comparing TIME\_SLICE to the CPU clocks divided by (double)CLOCKS\_PER\_SEC), the function f will yield the resources to another thread. |
| 4 | Added a field “priority” of type “unsigned short” to the “gt” struct to indicate the priority (ranged from 1 to 99) and changed the signature of “gtgo” accordingly. The scheduler (in “gtyield”) scans through the table and select the thread with the highest priority as the succeeding thread. |

The changes made in this project did not involve complicated algorithms or data structures. Nor it significantly improves the performance or usability of the current library. They mainly serve to illustrate some fundamental pieces of knowledge in scheduling.

**Result**

To illustrate how different scheduling policies would change the behaviors of a series of operations, I designed 4 test runs with different scheduling policies and test conditions. All test cases follow the same 3 instructions “gtgo(f,80); gtgo(f,10); gtgo(f,1); ” where “f” is a function pointer to the function being executed in the thread, and the integer value indicates the priority level of that thread.

To begin with, the RR policy with a quantum of 0 (a thread would always be preempted out because the time spent is nonzero) resulted in the same behavior as the default scheduling policy would create: all threads are perfectly alternated. Due to the simple implementation of the f function, the RR policy with a small quantum of 100 ms would still be enough for a green thread to finish. Thus the sequence looks as if no concurrency happened: the 3 instructions was instructed one by one for there was sufficient quantum for each individual thread to finish.

The “OTHER” scheduling policy (top priority first) worked in an interesting manner: because the first and second threads have higher priorities than the third thread, it happed that they alternated before both finished, and only after that moment the third thread with the lowest priority got a chance to kick in.

Finally, the FIFO policy had the same result as the default policy (because the “gtgo” function always finds the next available spot on the thread table, thus forming a natural topology of how the threads is placed on the table) and RR with 0 time-slice.

The following table presents the tests discussed above.

|  |  |  |  |
| --- | --- | --- | --- |
| **RR** | | **FIFO** | **OTHER** |
| **Time slice = 0 (ms)** | **Time slice = 100 (ms)** | **gtgo(f, 80);**  **gtgo(f, 10);**  **gtgo(f, 1);** | **gtgo(f, 80);**  **gtgo(f, 10);**  **gtgo(f, 1);** |
| **$ ./gttest**  **1 0**  **2 0**  **3 0**  **1 1**  **2 1**  **3 1**  **1 2**  **2 2**  **3 2**  **1 3**  **2 3**  **3 3**  **1 4**  **2 4**  **3 4**  **1 5**  **2 5**  **3 5**  **1 6**  **2 6**  **3 6**  **1 7**  **2 7**  **3 7**  **1 8**  **2 8**  **3 8**  **1 9**  **2 9**  **3 9** | **$ ./gttest**  **1 0**  **1 1**  **1 2**  **1 3**  **1 4**  **1 5**  **1 6**  **1 7**  **1 8**  **1 9**  **2 0**  **2 1**  **2 2**  **2 3**  **2 4**  **2 5**  **2 6**  **2 7**  **2 8**  **2 9**  **3 0**  **3 1**  **3 2**  **3 3**  **3 4**  **3 5**  **3 6**  **3 7**  **3 8**  **3 9** | **$ ./gttest**  **1 0**  **2 0**  **3 0**  **1 1**  **2 1**  **3 1**  **1 2**  **2 2**  **3 2**  **1 3**  **2 3**  **3 3**  **1 4**  **2 4**  **3 4**  **1 5**  **2 5**  **3 5**  **1 6**  **2 6**  **3 6**  **1 7**  **2 7**  **3 7**  **1 8**  **2 8**  **3 8**  **1 9**  **2 9**  **3 9** | **$ ./gttest**  **1 0**  **2 0**  **1 1**  **2 1**  **1 2**  **2 2**  **1 3**  **2 3**  **1 4**  **2 4**  **1 5**  **2 5**  **1 6**  **2 6**  **1 7**  **2 7**  **1 8**  **2 8**  **1 9**  **2 9**  **3 0**  **3 1**  **3 2**  **3 3**  **3 4**  **3 5**  **3 6**  **3 7**  **3 8**  **3 9** |

**Discussion**

Green threads implement a many(user threads)-to-one (kernel thread) threading model that allow the application to create a large number of threads that can execute concurrently (Cunningham & Cunningham,2009).

Many features of green threads are double-edged swords: they have major advantages and weaknesses at the same time. For example, switching between green threads incurs relatively small overhead comparing to kernel threads because of the avoidance of frequent user-kernel type switching. Therefore, a significant advantage of using green threads is the low cost. However, some people (e.g. Cunningham & Cunningham, 2009) also indicated that green threads could not take full advantage of multiprocessors because of the same reason -- they have access to only one kernel thread.

Another example is their performance. Green threads’ multithreading abilities on operating systems that do not provide native thread support will provide a gain in performance. But It is not always a better solution on OSs that support threads. Plus, writing sophisticated schedulers for green threads might not result in better solutions than libraries that have matured and been tested over time.

It seems that the strength and limitations of green threads can be best summarized in this anecdote. Historically, the Java runtime system used green threads in its runtime thread and system support layer (Oracle, 2010). Problems such as not able to run Java threads in parallel on multiprocessors arose and the green threads library was replaced with native Solaris threads (Oracle, 2010).

**Conclusion**

This project report introduced an attempt implementing a simple green thread library that support different scheduling policies. It presented a few key implementation issues of the library and the result running a small set of test cases. Finally it briefly discussed the advantages and disadvantages of green threads on the empirical level.

**Reference**

Cunningham & Cunningham. (2009) *Green vs. Native Theads.* Retrieved on April 22, 2015 from <http://c2.com/cgi/wiki?GreenVsNativeThreads>

MPU (2013) *Green Threads*. retrieved on April 22, 2015 from <https://github.com/mpu>

Oracle (2010) JDK 1.1 for Solaris Developer's Guide retrieved on April 22, 2015 from <https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqe/>