Helium 2 Scheduler

Neil Klingensmith

28 June 2008

# Introduction

The Helium 2 scheduler is a real-time scheduler that provides a multiprogramming environment, allowing multiple threads to share control of the system’s CPU. All threads running in a Helium environment are assigned a priority level between 0x0 (highest priority) and 0xffff (lowest priority), and the Helium scheduler gives control of the CPU to the highest priority thread that is ready to run. Threads in Helium do not need to have unique priorities—multiple threads can share a common priority.

Helium 2 uses an Earliest Deadline First algorithm to schedule threads. This means that the thread whose deadline for completion is soonest will get control of the CPU first. Completion deadlines are abstractly represented by priorities. Higher-priority threads are assumed to have nearer completion deadlines, so they are scheduled first.

# Definitions

## Thread

An execution thread is a section of program data that can be run independently. Helium tracks each thread’s stack pointer and program counter throughout the course of its execution. At any time during a thread’s execution, Helium may preempt the thread and give control of the CPU to a different thread. Processes (commonly used in high-end systems such as PCs) differ from threads in that they have independent code and data sections. Such constructs require the CPU to have memory management hardware not commonly found in embedded systems.

## Context Switch

A context switch is the process of taking control of the CPU away from one thread and giving it to another. The thread that loses control of the CPU is said to be *preempted*. During a context switch, Helium saves the contents of all CPU registers so that when the thread that was preempted is again given control of the CPU, it can begin executing code in the same place it left off.

## Clock Tick

A clock tick is a periodic interrupt to the CPU. Helium’s clock tick interrupt service routine (ISR) contains the scheduler as well as a short routine for handling sleep requests[[1]](#footnote-2). Clock tick periods can vary by implementation, but they are generally in the range of 10-100ms. If some event has occurred that has made a new thread ready to run since the last clock tick interrupt, the scheduler will perform a context switch and reschedule the new thread.

# Implementation

## Organization

Helium represents each thread with a data structure of type THREAD that holds several system registers (see Table 1). THREAD objects are populated when they are initialized by the ThreadCreate function. Some members are referenced from source code written in assembler, so any new additions to the THREAD object should be made at the end of the list of member objects.

|  |  |
| --- | --- |
| **Name** | **Description** |
| struct thread \*next | Next THREAD object in the linked list. |
| struct thread \*prev | Previous THREAD object in the linked list. |
| int priority | Priority of the thread. |
| int numTicksRunning | Number of clock ticks the thread has been running for. This is used for statistics calculations. |
| int PC | Program counter of the thread. Saved by the scheduler every time the thread is preempted. |
| int SR | Status register of the thread. Saved by the scheduler every time the thread is preempted. |
| int SP | Stack pointer of the thread. Saved by the scheduler every time the thread is preempted. |
| int numTicksWaiting | The number of ticks remaining that the thread needs to wait for (set by the sleep API call and decremented each clock tick). |
| struct thread \*sidenext | Next thread in a side chain that is branched off of readyList. |
| struct thread \*sideprev | Previous thread in a side chain branched off of readyList. |

Table : THREAD object members.

The thread’s data structure is linked into one of the following lists based on its status:

|  |  |
| --- | --- |
| **List Name** | **Status of Threads in the List** |
| readyList | Ready to run. Sorted by thread priority with the highest priority thread ready to run at the beginning of the list. readyList is actually a binary tree (see the readyList subsection in this chapter). |
| waitList | Blocked because of a sleep call. This list is not sorted. |
| Mutex Waiting Lists | Waiting for a mutex resource. Each mutex object has its own waiting list. Threads themselves are not actually linked into these lists, but they are pointed to by MUTEX objects that are linked into the lists. See the document on mutexes for more information. |

Table : Linked lists that determine the status of threads.

## Helium 2 Status Lists

Threads are linked into one of the following lists based on their status.

### readyList

The readyList is a list of threads that are ready to run. The scheduler uses readyList to determine which thread should be run at any given time. When a new thread is inserted into the readyList, it is linked into a location based on its priority. The highest priority thread is located at the beginning of the list, and the lowest priority thread is located at the end. Threads that share a priority level are linked into sublists of the readyList (using the pointers sideprev and sidenext).

An insert into the ready list is an O(n) operation, but once a thread is in the ready list, the task of running the highest priority thread is O(1). Because readyList has sublists that are pointed to by elements on the main list, the structure of readyList is more accurately described as a tree than a linked list.

### waitList

 The waitList holds a list of threads that are blocked because of a call to the sleep function. waitList is not sorted in any particular order. Inserts and deletes from waitList are both O(1) operations.

### Mutex Waiting Lists

Each mutex object in Helium 2 has an associated waiting list (see the document on mutexes for further explanation). The waiting list is a linked list of objects of type MUTEX[[2]](#footnote-3), each of which point to a thread that is waiting for the mutex. Mutex objects are inserted at the end of the waiting list, so the last thread to request access to the mutex is the last thread to be given access[[3]](#footnote-4). Helium uses a separate object type to represent threads in a mutex waiting list because the first element of the mutex waiting list has access to the mutex, and therefore must be made ready to run by linking it into the readyList. Since an object cannot be linked into two lists at once, there must be a placeholder object to represent the thread in the mutex waiting list.

# User Programming Considerations

## Real-Time Scheduling Theory

A considerable amount of research has been done on the topic of real-time thread scheduling, and understanding a few important conclusions generally helps programmers write more efficient code. The most important result gives a hard upper limit on real-time software’s ability to load the hardware it runs on. For a real-time system with n periodic threads (each having unique periods), there exists a schedule that will always meet the deadline of each thread provided that the CPU’s utilization is:

$$U= \sum\_{i=1}^{n}\frac{C\_{i}}{T\_{i}} \leq n\left(\sqrt[n]{2}-1\right)$$

Where *Ci* is the computation time and *Ti* is the release period of the thread. By this formula, a real-time system with four threads must use no more than 75.7% of the total CPU time to guarantee that a scheduling algorithm exists that will always meet the threads’ deadlines. As the number of threads tends toward infinity, the maximum CPU utilization tends toward ln2 ≈ 0.6931. The remaining 30% of the CPU time may be used by non-real time system functions.

Note that this principle only guarantees that **some scheduling scheme** exists that will satisfy each thread’s deadline requirements given the CPU load is low enough. It does not guarantee that Helium’s scheduling policy will use that scheme. This has largely to due with the fact that the scheduling policies capable of attaining the theoretical maximum do not allow for thread interaction of any kind, and they prohibit the dynamic priority reassignments required for priority inversion[[4]](#footnote-5). Since adhering to these conditions would make the operating system nearly unusable except for the simplest applications, Helium was architected to sacrifice some efficiency for the sake of usability. To accommodate this condition, programmers should not expect real-time threads to load the CPU to more than 50%.

1. sleep is a Helium API function that threads can use to voluntarily relinquish control of the CPU for a number of clock tick intervals. [↑](#footnote-ref-2)
2. struct mutex is defined in helium.h. [↑](#footnote-ref-3)
3. Accesses to mutexes is provided using a FIFO policy. [↑](#footnote-ref-4)
4. Priority inversion occurs when a thread with a high priority must wait for a thread with a low priority to release a shared resource. The priorities of the two threads are swapped until the low priority thread finishes using the resource. [↑](#footnote-ref-5)