Task Management Techniques for Enforcing ED Scheduling on Periodic Task Set

Aloysius K. Mok †
Department of Computer Science
University of Texas at Austin
Austin, TX 78712

Extended Abstract

Introduction

A well known algorithm for enforcing hard deadline constraints on tasks is the ED (Earliest Deadline) policy. In this policy, the task with the nearest deadline is always selected for execution whenever a scheduling decision is made. A classical result in [Liu & Layland 73] showed that for the case of independent periodic tasks, the ED policy can achieve 100% CPU utilization on a single processor without missing any deadline. (In the independent periodic tasks model, every task is parameterized by an ordered pair \((c, p)\) where \(c\) and \(p\) are respectively the computation time and period of the task, and every task can preempt any other task at any time. The CPU utilization is the sum of the ratios \(c/p\) of the tasks.) In [Mok 84], the ED policy was extended and shown to be optimal for the case where both task synchronization (in the style of the ADA‡ rendezvous) and mutual exclusion (under certain restrictions) are allowed. In [Mok et al 87], another variation of the ED policy was shown to be optimal also for a dataflow model with data-driven timing constraints. Hence, there is practical interest in the efficient implementation of the ED policy. For periodic tasks, however, the method of using a heap (e.g., see [Aho, Hopcroft & Ullman 74]) to organize task control blocks will in the worst case require \(O(n)\) operations to select the task with the nearest deadline where \(n\) is the number of tasks. In this note, we shall describe a \(O(\log n)\) solution to the task management problem.

The Problem and a \(O(n)\) Solution

For the purpose of enforcing the ED policy, there are two parameters that the operating system keeps track of in a task control block: the current deadline \(d\) and the ready time \(r\) (the start of the current period). We say that a task is not ready if it has completed its execution for the current period and the next period has not yet started. Suppose the task control blocks are organized and stored in a data structure \(D\). The relevant task

---

† Supported by ONR under contract number N00014-85-K-0117, by a University Research Initiative grant under contract number N00014-86-K-0763, and by a DoD University Research Instrumentation Program under contract number N00014-86-G-0199.

‡ ADA is a trademark of the United States Department of Defense.
management operations for ED scheduling are:

- **SELECT (D)**: Return the ready task with the smallest \( d \).
- **POST (π,D)**: Insert the task \( π \) which has been preempted or which has just completed execution into \( D \). In the latter case, set \( d \) to be the end of the next period and set \( r \) to be the start of the next period.
- **WAKEUP (t,D)**: Classify any task in \( D \) as ready whose ready time \( r = t \).

Since the ED policy requires a dynamic priority scheduler, a "textbook solution" is to use a heap (priority queue, e.g., see [Aho, Hopcroft & Ullman 74]) to implement the data structure \( D \). This solution, however, has a \( O(n) \) worst-case time complexity. Consider the following periodic task set which has 5 tasks. Four of them have the same period \( p=5 \) and computation time \( c=1 \). Task #5 has parameters \( p=15 \), \( c=2 \). The first four tasks will be executed according to the ED policy after about 4 time units. At this point, the heap will look like:

- **T1**: \( T1:(5,10) \)
- **T2**: \( T2:(5,10) \)
- **T3**: \( T3:(5,10) \)
- **T4**: \( T4:(5,10) \)
- **T5**: \( T5:(0,15) \)

![Figure 1](image)

Heap of Tasks at Time=4

When SELECT(D) is performed at around time=4, the only ready task (#5) is at the rightmost leaf on the lowest level. Thus in the worst case, the whole heap will have to be searched before SELECT(D) can find the ready task with the smallest deadline, a \( O(n) \) operation. For large \( n \), this overhead can be unacceptable. In the above scenario, for example, it might be after time=5 before SELECT(D) can return task #5 to be executed next. By then, however, the first four tasks will have become ready again.
It should be noticed that having two separate heaps for ready and not ready tasks will not help in reducing the worst-case time complexity, since in the worst case, all of the tasks in the not-ready heap may become ready at the same time, and O(n) operations are required to move all of them to the ready heap.

The HOH (Heap-of-Heaps) Data Structure

We shall present a O(log n) solution to the task management problem. In this solution, we make use of a data structure which is a heap of heaps (HOH). A HOH is a heap whose elements can be heaps themselves. Like a heap, a HOH has the heap property: the priority of any element cannot be greater than that of any of its ancestors. A HOH supports priority queue operations:

\textbf{INSERT} (x,H) :- Insert x into the HOH H; x may be a simple node or a heap with two or more nodes.

\textbf{DELETEMIN} (H) :- Remove and return from H the node with the highest priority (smallest integer). Reorganize H so that it remains a HOH.

HOH operations are implemented very much like heap operations and are described below.

\textbf{INSERT} (x,H):

Place x as far left as possible on the lowest level of H, starting a new level if necessary. If x has a higher priority (smaller integer) than its parent, exchange it with its parent. When comparing priority, use the priority of the root if the element under comparison is a heap, else use the priority of the simple node. Repeat the comparison as many times as necessary until either the root of H is reached or no exchange is needed.

\textbf{DELETEMIN} (H):

Suppose the root of H is a heap with two or more nodes. Call this heap h. Remove and return the root of h and rearrange h. Call this new heap h’. Push h’ as far down the HOH H as it will go. When comparing priority, use the priority of the root if the element under comparison is a heap, else use the priority of the simple node.

If the root of H is a simple node, remove and return that node and rearrange H as follows: Take the rightmost element at the lowest level of H and put it at the root of H. Then push this element as far down H as possible.

\textbf{Theorem 1}: Both \textbf{INSERT} and \textbf{DELETEMIN} operations on a HOH have O(log n) worst-case time complexity where n is the number simple nodes.

\textbf{O(log n) Solution to the Task Management Problem}

This solution uses a HOH to store ready tasks and a 2-3 tree (see, e.g., [Aho, Hopcroft & Ullman 74]) to store the not ready tasks. Each ready task is stored as a simple node in the HOH. The leaves of the 2-3 tree are heaps and each not ready task is stored as a node in one of the leaves. All the tasks in the same leaf (heap) have the same ready
time (the parameter \( r \) in the task control block). A leaf in the 2-3 tree is accessed by using the corresponding ready time of the tasks in it as the key. The data structure \( D \) for task management is a pair \((H, T)\) where \( H \) is a HOH and \( T \) is a 2-3 tree. The relevant operations are implemented as follows.

**SELECT** \((D)\) :-

Perform a DELETEMIN on \( H \). The task returned is to be executed next.

**POST** \((\pi, D)\) :-

If \( \pi \) has just been preempted, perform \( \text{INSERT}(\pi, H) \). If \( \pi \) has just completed execution, update its parameters by setting \( d \) to the end of the next period and \( r \) to the start of the next period. Locate the leaf in \( T \) which has the same ready time \( r \) and insert \( \pi \) into this heap, using the deadline \( d \) as the key. If \( T \) does not contain a leaf with the same ready time as \( \pi \), create a heap with \( \pi \) as its only node and insert this heap into \( T \).

**WAKEUP** \((t, D)\) :-

Locate a leaf in \( T \) with ready time = \( t \). If one exists (call it \( h \)), delete \( h \) from \( T \) and insert \( h \) into \( H \) by performing \( \text{INSERT}(h, H) \).

**Theorem 2**: The task management operations SELECT, POST, WAKEUP can be implemented with a \( O(\log n) \) worst-case time complexity where \( n \) is the number of tasks.

**Representation of Clock Values**

The data structure operations discussed above involve mostly arithmetic operations on time values. It is important for efficiency reasons to be able to represent time values as integers. For a 32-bit computer, a 1 millisecond basic time unit allows about 50 days before an arithmetic overflow occurs as a result of clock update. If 50 days is not a sufficiently long interval or if the mission length of an application is unknown, the following scheme may be used for representing clock values with only finite variables. The applicability of this scheme is predicated on the assumption that the values of the time parameters \( d \) (current deadline) and \( r \) (ready time) are bounded by the maximum of the periods of the tasks. This assumption holds when for example, the CPU utilization for the periodic task set does not exceed 1 in the independent periodic tasks model.

In this scheme, all time values are stored in finite counters (e.g., integer variables) and all arithmetic operations are performed modulo the size of the counter, say \( m \). We use a variable: \( fc \) (finite clock) to keep track of the passage of time. \( fc \) is initialized to 0 and is incremented \( \mod m \) at every clock tick (timer interrupt). Suppose \( r \) (ready time parameter in task control block) and \( d \) (deadline parameter) of a task which has just completed execution are respectively \( t \) and \( t+p \mod m \) where \( p \) is its period. Then \( r \) and \( d \) will be updated to \( t+p \mod m \) and \( t+2p \mod m \) respectively. It is easy to see that the ready time and deadline of the task are actually \( r-fc \mod m \) and \( d-fc \mod m \) respectively from the current moment. These values can then be used in comparing deadlines or ready times among tasks. Thus this scheme poses no limit on how long the real-time application
can run without an infinite range calendar clock.

**Conclusion**

In enforcing the ED scheduling discipline on periodic tasks, the straightforward way to organize task control blocks by means of a heap will incur a worst-case time complexity of $O(n)$ per operation. We showed that a data structure with $O(\log n)$ worst-case time complexity can be used to solve the task management problem. An interesting question is whether inexpensive hardware can be built to perform ED scheduling in constant time for large but not arbitrarily large task sets.

**Bibliography**


**Postscript Added After RTSOS’88**

We proposed to use a HoH to store the ready tasks and a 2-3 tree to store the not-ready tasks. In a practical implementation, it may be more efficient to use a timing wheel (a circular array of pointers to ) whose slots contain pointers to heaps such that all the not-ready tasks with the same ready time can be stored in the same heap pointed to by the time slot that corresponds to their common ready time. The size of the timing wheel should be big enough for a completed task to be stored in the heap at the time slot corresponding to its next wake-up time and is bounded by the largest period of the periodic tasks. If there is a large variation in the size of the periods, then a hierarchical timing wheel may be considered.