# Scheduling and Synchronization in Embedded Real-Time Operating Systems

Sanjeev Khushu and Johnathan Simmons
CSE 221, March 5, 2001

## Abstract

Scheduling and synchronization are the two mainstays of embedded real-time operating system development. This paper presents research on these two topics. On the topic of schedulers we explain the scheduling constraints faced by embedded real-time systems and present scheduling techniques that can efficiently meet these constraints. On the topic of synchronization, we give a simple synchronization method commonly used today in embedded real-time systems and follow that with several more sophisticated approaches.

## 1 Introduction

An *embedded real-time system*, as its name implies, possesses the characteristics of both an embedded system and a real-time system. An *embedded system* is one that is special purpose, often uniprocessor, and generally is not user-programmable. When it is user-programmable, the "programming" is more at the level of system configuration. Because an embedded system is special purpose and is not user-programmable, the applications are trusted. This means that the operating system and the applications can share portions of their address spaces to communicate. Another result of an embedded system being special purpose is that it typically does not have a file system or standard peripherals.

A *real-time system* is one that must perform operations within rigid timing constraints. Real-time systems are further subdivided into hard real-time and soft real-time. Hard real-time means that that a failure will be of great consequence. An example of this is a real-time system controlling a nuclear reactor. A soft real-time system must act within timing constraints for its operation to be correct, although a timing failure in this kind of system is more of an annoyance. An example of this kind of system is a bank's automated teller machine.

An example of an embedded real-time system is a hypothetical helicopter sensor monitoring system. The system's software is only be loadable at the factory and is stored on a socketed ROM chip. The purpose of this system is to acquire sensor data on regular intervals, process that data, and store it to a solid state device for later playback for perform usage tracking and off-line mechanical problem detection. The sensors are a combination of polled sensors and sensors which signal the CPU by interrupt when they have data ready. The real-time nature of the system depends on a clock interrupt, and consequently this interrupt has highest priority. Multiple tasks are used to acquire the data, process the data and write it to the display, and store the data. The first task is the highest priority, which must acquire data on the regular intervals as closely as possible for an accurate usage profile. Processing of the data and writing it to the display is of secondary importance, but the user still wants to see the processed data regularly update. Storing the data is of the lowest priority because the data is queued until written. Critical sections of code are safely executed by disabling interrupts before the section and reenabling them after.

In this paper, we present recent research on two very important topics in embedded real-time operating systems: scheduling and synchronization. Applications of this research can provide improvements, either in correctness or performance, to systems like the one described above. An important topic in embedded real-time system research not covered in this paper is the use of software-managed caches to guarantee a performance boost. However, the literature presents this as the responsibility of the compiler or the application programs themselves, rather than the operating system [1].

## 2 Scheduling Tasks

Scheduling is the designing of the order and/or execution of a set of tasks with certain known characteristics on a limited set of processing units. In embedded systems there are different resources like the CPU, memory, peripherals, etc. for which scheduling techniques are required. In this paper some of the scheduling principles and algorithms are general but they are more relevant to the CPU as a resource of contention. The CPU is the most important resource for which a contention arises and the design of a good scheduler goes a long way in determining the performance of embedded systems, and any system for that matter.

In general, an operating system scheduler works on a broader principle of fairness. In real-time systems the main goal is to achieve timeliness. In an embedded real-time system, a scheduler has to work under a number of constraints such as concurrent execution of tasks with the constraint of meeting deadlines, using very little CPU power, low memory constraint, size constraint, economy of scale and host of other factors. Therefore, embedded real-time schedulers are characterized by the need for running several tasks in a constraint environment with minimum overheads, even at the cost of losing generality.

Most embedded systems are based on single processor, so our discussion in this paper is mostly based on uniprocessor architecture. With multiprocessor architecture comes the factor of cost and complexity that is avoided in embedded systems.

In the following sections we will describe different scheduling paradigms, scheduling algorithms followed by their analysis. We will also provide a hybrid solution suitable for embedded systems. Before discussing embedded real-time system schedulers, we provide an introduction to certain system concepts that carry a lot of significance in embedded real-time systems.

*Periodic Tasks* - The period of a task is the rate with which a particular task becomes ready for execution. Periodic tasks become ready at regular and fixed intervals. Periodic tasks are commonly found in applications such as avionics and process control accurate control requires continual sampling and processing data.

*Sporadic tasks* - These tasks have non-periodic arrival patterns. Sporadic tasks are associated with event driven processing such as responding to user inputs or non-periodic device interrupts. These events occur repeatedly, but the time interval between consecutive occurrences varies and can be arbitrarily large.

*Deadlines* - All real time tasks have deadline by which a particular job has to be finished. There are scheduling algorithms designed to allow maximum tasks to meet their deadline.

*Laxity* - Laxity is defined as the maximum time a task can wait and still meet the deadline. It can also be used as a measure of scheduling necessity.

*Jitter* - It is defined as the time between when a task became ready and when it actually got executed. For certain real time systems there is an additional constraint that all the tasks should have minimum jitter.

*Schedulability* - A given set of tasks is considered to be schedulable if all the tasks can meet their deadline. In certain on-line scheduling algorithms a new task is subject to schedulability test, wherein it is verified that the new task is schedulable along with the already existing tasks. If the task is not schedulable the task is not permitted to enter the system.

*Utilization* - It is the factor giving a notion of how much CPU is utilized by a given set of tasks. Liu and Layland [2] gave proof that there is a theoretical upper limit of processor utilization by a given scheduling algorithm. For *earliest deadline first* (EDF -- described later) scheduling, a utilization of 1 (maximum) can be achieved.

## 2.1 Design Issues

The scheduling problem consists of deciding the order of execution and also the period of execution of a set of tasks with certain known characteristics like periodicity and limited set of processing units, which is typically a single processor in embedded systems. There are two kinds of constraint faced by tasks executing in an embedded real-time environment: *time constraint* and *resource constraint*.

In real time world most of the tasks have a time constraint, a deadline in executing a particular job. The tasks are also required to have a good response time to increase the response time of the System and execute in a manner so that other tasks can also meet their deadlines.

The other constraint that affects the design of an embedded real-time system is resource constraint. In embedded systems there is a limited RAM availability, limited CPU speed, power consumption constraint and host of other resource related constraints. An embedded system is designed to work optimally in spite of the resource constraint problems it has.

Due to the above-mentioned constraints (a combination) there is an immense pressure on embedded operating system performance. An embedded system's performance in presence of constraints is highly correlated with how smart the scheduler is.

Up until now, a lot of research has been done on developing schedulers that can meet these constraints. We will go through these in our next section. We will discuss various designs relevant in embedded real-time world and follow them up with our analysis.

## 2.2 Scheduling Paradigms

### 2.2.1 Static scheduling

This is a pre-runtime based scheduler, wherein tasks execute in a statically decided order. The order of execution is decided before the tasks are entered into the system based on statically defined criterion like deadline, criticality, periodicity etc. The advantage of using static scheduling procedure is that it involves almost no overhead in deciding which task to schedule. Static scheduling involves that a task be executed whenever it is expected to be ready or at least be tested for readiness cyclically. Static scheduling of tasks in embedded real-time systems often implies a tedious iterative design process. The reason for this is the lack of flexibility and expressive power in existing scheduling framework, which makes it difficult to both model the system accurately and provide correct optimizations. This causes systems to be over constrained due to statically decided rules of procedures.

#### 2.2.1.1 Round-robin method

The simplest of static scheduling procedures is round-robin method. The tasks are checked for readiness in a predetermined order with ready to execute task getting a CPU slice. Each tasks gets checked for schedulability once per cycle, with scheduling time bound by execution time of other tasks. Apart from simplicity this method has no advantages. The major disadvantage being that urgent tasks always have to wait for their turns, allowing non urgent tasks to execute before the urgent tasks. Also polling tasks for schedulability for readiness is not a good procedure at all. This type of scheduling works well in some simple embedded systems where software in the loop executes quickly and the loop can execute repeatedly at a very rapid rate.

#### 2.2.1.2 Static Cyclic Scheduling

In this method tasks are checked in a predetermined order and the tasks that are found ready are executed. There can be a multi rate procedure where tasks can appear more than once in a cycle depending upon their urgency. This method is better than former but still has disadvantage of overheads involved with frequent readiness checks. This procedure still has a problem that an execution of a particular task is dependent upon execution time of former tasks.

### 2.2.2 Dynamic Scheduling

In embedded real-time systems a *dynamic scheduling* [5] policy is based on priority. In a dynamic scheduling policy the tasks are dynamically chosen based on their priority dynamically, generally from ordered prioritized queue. The priorities can be assigned statically or dynamically based on different criterions like, deadline, criticality, periodicity etc. Dynamic scheduling can be preemptive or non preemptive. There are two approaches to dynamically schedule tasks. The first is to dynamically schedule tasks with an optimistic criterion to minimize the number of tasks that fail to make the deadline, or minimize the maximum deadline violations. The second, as each task occurs in the system the existing schedule is checked to see if the new task can be added to the schedule so that it's deadline can be met. If adding the task results in previous essential tasks to miss the deadline, then application is informed about it.

Meeting deadlines, achieving high CPU utilization with minimum resource and time utilization are considered as the main goals of task scheduling. Both preemptive and non-preemptive algorithms can be used to satisfy these objectives. Non-preemptive schedulers are easier to implement and analyze. It has advantages of saving extra context switches and save overheads involved in supporting mutual exclusion over preemptive scheduling. The problem with non pre-emptive scheduling is that it can cause some tasks to miss their deadlines as non-preempted tasks can have varying release times.

A hybrid approach can also be used wherein tasks are divided into different queues at different scheduling levels. A scheduler could consist of two levels of scheduler queues, the upper queue consists of very high priority (critical) non-preemptive tasks and a lower queue has pre-emptive tasks.

### 2.2.3 Synchronous Scheduling

In synchronous scheduling algorithms the available processing time is divided by hardware clock into intervals called frames. In each frame, the set of tasks allocated are guaranteed to be completed by the end of the frame. If a task is too big to fit into a frame, it is artificially divided into a set of highly independent tasks such that smaller tasks can be scheduled into the frames.

### 2.3 Scheduling algorithms

#### 2.3.1 Rate Monotonic Scheduling (RMS)

Static priority dynamic scheduling has received tremendous interest after pioneering work done by Liu and Layland. For tasks that were periodic (with deadlines at the end of each period), perfectly preemptible, independent (or noninteracting) Liu and Layland provided mathematical conditions for assessing schedulability of a set of tasks. The rate monotonic scheduling algorithm simply assigns highest priority to the task with the highest rate (or with shortest period) and assigned the priorities of the remaining tasks monotonically in the order. The Liu-Layland's theorem states that a set of independent periodic tasks scheduled by the rate monotonic algorithm will always meet the deadlines, for all the task phasings, if

$$\mathbf{C_1/T_1 + C_n/T_n <= U(n) = n(2\text{^}1/n - 1)}$$

$C_i$ = worst case task execution time of $\text{Task}_i$, $T_i$ = period of $\text{Task}_i$. U(n) = utilization bound of n tasks.

| Periodic task | Execution Time | Period | Deadline |
|---|---|---|---|
| Task1 | 20 ms | 100 ms | 100 |
| Task2 | 40 ms | 150 ms | 150 |
| Task3 | 100 ms | 350 ms | 350 |

Example above: 20/100 + 40/150 + 100/350 <= U(3) = 3(2^1/3 –1)

The total utilization in this problem is 75.3% that is below the bound for 3 tasks of 77.9 % utilization. So this tells that periodic tasks are schedulable.

#### 2.3.2 Deadline driven Scheduling Algorithm

##### 2.3.2.1 Earliest Deadline First (EDF)

Liu and Layland have also found stronger utilization for a dynamic priority assignment policy called *earliest deadline first* (EDF). A task is assigned highest priority if its deadline is the nearest and will be assigned lowest priority if the deadline is farthest. In EDF scheduling there is no processor idle time prior to a missed deadline condition. The theorem states that for a given set of *m* tasks, the deadline driven scheduling algorithm is feasible if and only if

$$\mathbf{C_1/T_1 + \ldots + C_m/T_m <= 1}$$

The worst-case response time (R) of a task is measured as the time when task is made ready to the time when tasks actually get executed. The task with a deadline D is schedulable if R < D.

$$\mathbf{R_i = S_i + B_i + C_i + E\ [\ R_i/T_k\ ]C_k}$$

Where $S_i$ = Scheduling overheads of $\text{Task}_i$, $T_k$ = Time between when task *k* gets ready again, $B_i$ = blocking time of task *i*, $C_i$ = Execution time of task *i*. [ ] = the ceiling function ( [1.2] = 2 ). The factor $R_i/T_k$ gives the amount of interference when a higher priority task preempts task *i*. So the total time taken by higher priority task is given by $[R_i/T_k]C_k$.

Blocking time is defined as the time for which a low priority task can delay the execution of high priority task. Blocking can lead to tasks missing deadlines. Since blocking is an important phenomenon we will explain the concept and methods to reduce and quantify Blocking times here. *Figure 1* is a timeline in with lowest priority at the bottom. As is evident from the diagram a lower priority task blocks the execution of high priority task. This phenomenon is called *priority inversion*.

A *priority inheritance* mechanism [4] is used to avoid priority inversion. In this algorithm a lower priority task inherits the priority of the highest priority task that gets blocked. As can be seen from the *Figure 2* the priority of lower task is increased once the higher priority task tries to lock the semaphore. The priority inheritance does solve problems of blocking to some extent but does not completely solve the problem of unpredictable delays. The priority inheritance mechanism cannot solve circular blocking which can lead to deadlocks.

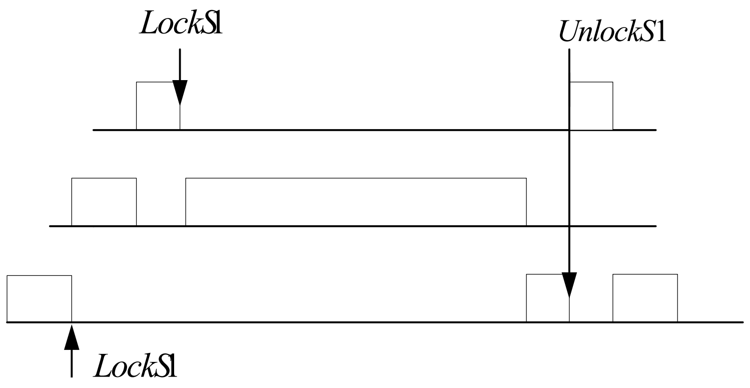

**Figure 1: Priority Inversion**

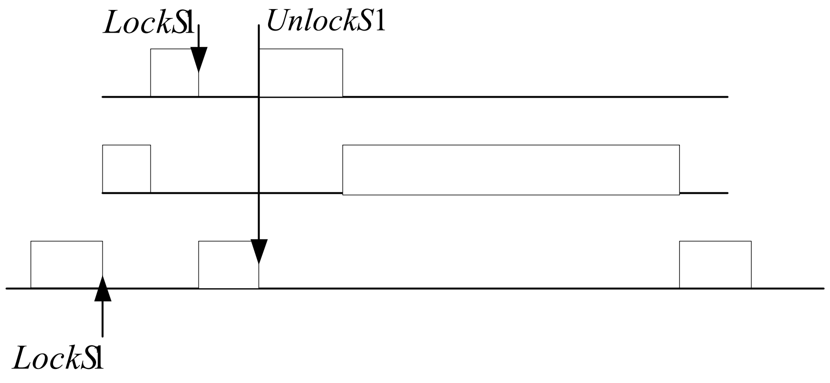

**Figure 2: Priority Inheritance**

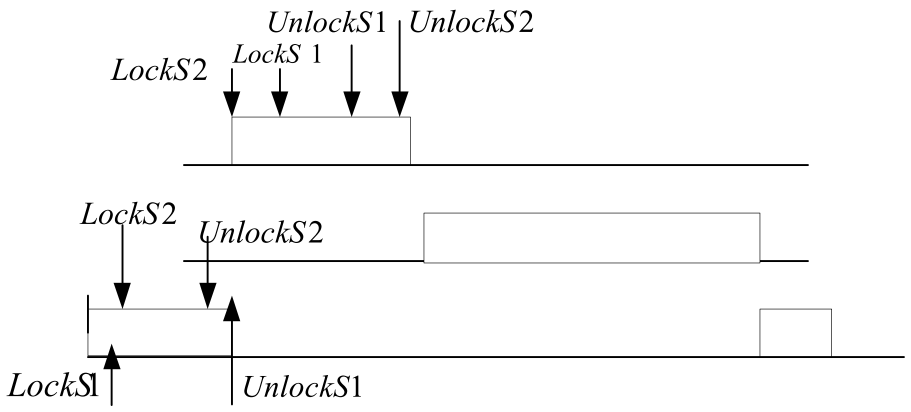

**Figure 3: Priority Ceiling Protocol**

To solve these problems there is a mechanism called the *priority ceiling protocol* (PCP). Each semaphore has an associated ceiling that it attains once it locks that semaphore. When the task releases the semaphore the priority is reverted to old one. The tasks can hold more than one semaphore in a nested pattern (*Figure 3*). Most embedded real-time systems implement a variant of PCP by locking processor interrupts during period of contention so that data can be shared in a mutually exclusive manner. In PCP, blocking times are bounded and can be calculated. The $B_i$ time evaluated in this manner is used in above equation.

### 2.3.2.2 Least Laxity Algorithm

In this approach the process having least laxity is assigned highest priority and is therefore executed first of all. A running process can be preempted by another task whose laxity has decreased below the currently executing task. The laxity of running process is constant.

## 2.4 Analysis

The problem of implementing Liu and Layland's procedures in embedded real-time systems is that a lot of embedded systems often violate the assumptions of Liu and Layland. Rate Monotonic Scheduling (RMS) is very inflexible and is not suited in certain scenarios.

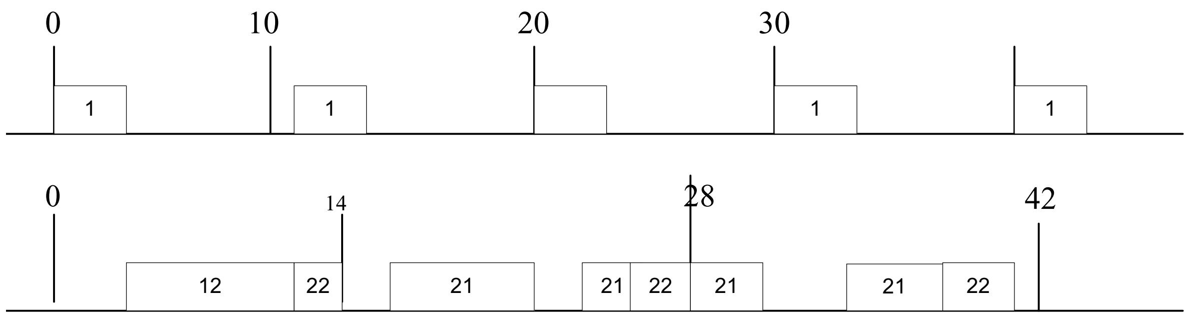

**Figure 4: Subtask Scheduling**

If there are two tasks, $T_1$ and $T_2$ each having one subtask with $C_1 = 4$, $D_1 = 10$, $C_2 = 8$, $T_2 = 14$. According to Liu and Layland, $P_1 > P_2$. It can be clearly seen that due to rate monotonic priority, the task set is not schedulable. However now if $T_2$ is divided into 2 subtasks with a execution time of $C_{21} = 6$ and $C_{22} = 2$ and priorities assigned as $P_{21} < p_1 < p_{22}$ then the task set is schedulable (*Figure 4*). This shows that one may need to check deadlines of more than one job of a particular task. And most real time systems are comprised of subtasks having varying execution times and varying periods. The other disadvantage is run time overheads. The problem with the least laxity approach is when two processes have similar laxity, causing one process to run for a while and then getting preempted by the other task and vice versa. This can result in "thrashing". Least laxity is optimal in the same way as earliest deadline when the cost of context switching is ignored.

### 2.5 Hybrid Solution

The problem with RMS- and EDF-like scheduling algorithms is that they have significant overhead: run-time overhead and schedulability overhead. In embedded real-time systems, due to limited resources, all sources of overhead should be reduced as much as possible.

The run time overhead is related to parsing the queues and adding/deleting tasks from scheduler queues. Each task blocks and unblocks at least once in a scheduler period and scheduler has to select twice in a single period the task to be scheduled. In RMS method tasks can be maintained in a sorted queue resulting in a selection overhead of just one simple comparison. For EDF the scheduler has to parse the whole list of tasks resulting in high overall run-time scheduler overhead.

Schedulability overhead is the theoretical limit on the task sets that are schedulable under a given scheduling algorithm. For EDF the schedulability overhead is 0 as the processor utilization can be $U=1$. For RM utilization is $U<1$ and hence it always has a schedulability overhead $> 0$.

So a scheduling algorithm that is a hybrid of RM and EDF can combine the low run time overhead of RM and low scheduling overhead of EDF to give an optimized scheduling algorithm [3].

## 3 Synchronization

*Synchronization* in the context of real-time embedded systems normally refers to the controlled access to shared memory within a given subsystem. Messages are used for synchronization as well, but generally this is for communication between subsystems. This section begins with a method of synchronization commonly used in practice followed by a presentation of recent research in mutual-exclusion-type synchronization methods that are suitable for use in a single-processor subsystem.

### 3.1 A Current Method of Synchronization

A simple method of synchronization in embedded real-time systems is disabling interrupts before and reenabling interrupts after a critical section. Because applications are trusted in this environment, they are permitted to execute the CPU instructions to do this. This provides an easy synchronization method when the operating system does not provide any synchronization support. This is a common situation in embedded programming.

The drawback to this approach is that it can negatively affect the real-time nature of the system. For example, suppose an application process needs access to a shared buffer. To guarantee consistent data, all accessing processes must disable the interrupts when accessing the data and reenable them immediately after. As another example, when an interrupt service routine (ISR) must access a shared data structure, it is common practice to execute the entire ISR with interrupts disabled. Disabling the interrupts prevents preemption because the scheduler is triggered by an interrupt. During the time that interrupts are disabled, any pending interrupts are held by a register and will be serviced when interrupts are reenabled. This delay is known as *interrupt latency*. The application programmer assumes the burden of ensuring that interrupts are disabled for a short a period as possible to minimize this.

## 3.2 Interrupt Transparent Synchronization

A synchronization technique that is designed specifically to deal with interrupt latency is *interrupt-transparent synchronization* [8], a feature implemented by the PURE family of embedded operating systems. The basic idea behind this method is to keep interrupts enabled as much as possible to ensure the responsiveness of the system. To do this, servicing of interrupts is decomposed into two parts: a *prolog* and an *epilog*. The prolog is executed immediately when an interrupt is detected and runs with interrupts of lesser priority masked. The epilog is a continuation of the interrupt service routine that does not require instantaneous service and that may contain critical sections. Epilog code run with *all* interrupts enabled. This is the key feature that provides *interrupt transparency.* When an interrupt occurs, the prolog is run which performs some action and then defers the rest of its action by adding an epilog to the *epilog queue*.

The epilog queue is a first-in-first-out (FIFO) structure. Epilogs contained in this queue are run at the lowest physical interrupt priority. Because epilogs are run with interrupt priority, the epilog queue code runs until completion before any normal tasks can run. Because the epilogs are run serially, there is no risk of interprocess conflict.

Note that access to this queue must be synchronized because higher priority interrupts may trigger additional epilog enqueue operations while one is already in progress. Schön explains that this problem is dealt with by setting the tail pointer of the queue to point to the epilog to be enqueued before the actual insertion is performed. Since the assignment of the pointer is atomic, the enqueue is synchronized. Synchronizing the dequeue operation is somewhat more complicated. If the queue only contains one element and the dequeue operation is overlapped by an enqueue operation, the overlapping enqueue operation(s) will in fact chain the new epilogs to the epilog being dequeued. This results from both operations attempting to modify the tail pointer of the queue at the same time. The dequeue operation must test for this case. If it has occurred, the dequeue operation must unhook all epilogs mistakenly attached to the item it is dequeuing and reinsert them into the epilog queue. The dequeue operation cannot overlap itself because prologs cannot dequeue an epilog once it is queued.

Users of the Java programming language will find familiar the serialization of critical section code for purposes of synchronization. For high performance, *Java Foundation Class* (JFC) user-interface components are not thread-safe. That is, state changes performed on these objects are not explicitly synchronized. This could potentially be dangerous, as Java is a multi-threaded language. To ensure that JFC objects maintain their consistency, all state changes to these objects are executed from a single thread: the *event-dispatch thread*. The event-dispatch thread executes operations queued on the *event queue* in a strict first-come-first-served (FIFO) order. Normally, operations waiting in the event queue result from clicking a button or entering data in an application's graphical user-interface. When the state of a JFC object must be modified from another thread, Java's *SwingUtilities* class provides a method called *invokeLater* to queue the update on the event-dispatch queue. This method returns immediately so the thread requesting the update can continue. Another method called *invokeAndWait* returns only after the update has been applied.

Schön presents interrupt-transparent synchronization as a technique for synchronizing access to operating system kernel data structures. His example shows how it is used for scheduling of real-time tasks triggered by external events (hardware interrupts). However, this idea can be extended to provide general-purpose synchronization in two ways. The first way is for the operating system to provide a software interrupt handler that application programs can use by invoking a software interrupt to queue their own epilogs. This interrupt could take as its parameter a pointer to a function to execute in the epilog that deals with the critical section. The process could block and then at the end of the epilog, there could be a scheduling request to restart the blocked process. The operating system could define this interrupt vector. This is analogous to Java's *invokeAndWait* method described above. As previously mentioned, epilogs run at a higher priority than application processes, so no application process would be blocked waiting for the first process to finish its critical section.

A second simpler, although perhaps architecturally awkward, method of using this mechanism to provide general purpose synchronization is to use the ISR for a particular event queue an epilog that performs a data transfer between shared memory areas before the process that needs to be awoken is started. When it does start, the information that it needs will be available. Note however that the data to copy may not be ready if a lower priority producer process overran its quantum (real-time failure) and has not fully prepared the data it wishes to transmit. To deal with this case, the lower priority process can set a flag that indicates whether the new data is ready. The ISR that triggers the dependent process could make this check before copying. This method only works if the epilog has access to the address spaces of both the producer and consumer processes.

## 3.3 An Efficient Semaphore Implementation

A more conventional method of synchronizing access to shared memory is through the use of semaphores. The EMERALDS operating system implements a high performance blocking semaphore that is designed to reduce the number of context switches on a uniprocessor [9]. Zuberi and Shin state that as much as 40-50% of the overhead of locking and unlocking semaphores is due to context switching.

Consider the case, under a conventional semaphore implementation, where there is a thread $T_1$ that has locked semaphore S. Suppose some event causes a context switch to thread $T_2$. $T_2$ attempts to lock semaphore S, but it is already locked so $T_2$ is blocked. This causes a context switch back to $T_1$ which eventually unlocks S. The unlocking of S causes a context switch back to $T_2$, which is waiting on S. As a result of locking and unlocking S, two context switches were performed.

The method behind the EMERALDS approach is to avoid the first context switch to $T_2$ because $T_2$ will need to lock semaphore S that is already locked by $T_1$. In order for EMERALDS to know which lock $T_2$ will need next, the system call to acquire a lock on a semaphore takes an additional parameter – the next semaphore needed. This way, when an event would normally cause a context switch to $T_2$, a check is made to see if the next lock it needs to acquire is not available. If this is the case, the context switch to $T_2$ is delayed and $T_2$ is placed directly on the wait queue for that semaphore. Otherwise, the context switch is made normally. Using this approach, one context switch from the lock/unlock pair is saved if the next lock needed is unavailable. EMERALDS implements a code parser to instrument application code with this additional information. If the next semaphore to be locked cannot be determined by the code parser, the action taken by EMERALDS reverts to the traditional semaphore approach.

This implementation is specifically geared for embedded systems because it requires that the set of semaphores in the system be fixed. This assumption does not pose a problem for an embedded real-time system because of the special-purpose nature of the system; the set of semaphores is known. This synchronization method is compatible with priority scheduling with priority inheritance.

### 3.4 Wait-Free Synchronization

Another efficient synchronization approach in recent research is the use of *wait-free synchronization* [10]. In a system that supports wait-free sharing, threads or other independent entities of execution need never block before accessing shared structures. The operating system provides wait-free synchronized objects through the use of special atomic CPU instructions. Herlihy explains the differences in synchronization capability of atomic CPU instructions such as *set* on an atomic register, *test&set*, and *compare&swap*.

Wait-free objects have an associated *consensus number*. Herlihy defines a consensus number to be the maximum number of processes for which the object can solve a simple consensus problem. Objects of lesser consensus numbers cannot be composed to make an object with a higher consensus number. An atomic register has only a consensus number of 1, while stacks and queues has consensus number 2, and *compare&swap* registers have an infinite consensus number.

This approach provides a low cost method of synchronization. The major drawback, however, is that it requires special CPU instructions that support this synchronization. In an embedded real-time system, the CPU may only support a very basic instruction set.

## 4 Conclusion

In this paper, we have presented a survey of recent research on the topic of embedded real-time operating systems in the subareas of scheduling and synchronization. The research on scheduling presents three things. First, it provides a way to analyze tasks for schedulability. Second, a combination of RMS and EDF scheduling creates a balance between static and dynamic scheduling to provide for greater utilization and reduction in scheduling overhead. Third, a hybrid of preemptive and non-preemptive scheduling techniques give the flexibility of both within the same operating system environment.

The research on synchronization presents ways to improve operating system performance. Interrupt-transparent synchronization, used by some members of the PURE family of operating systems, improves system response by keeping interrupts enabled while accessing shared memory. This approach can be extended to provide application level synchronization. The blocking semaphore implementation used by EMERALDS reduces the number of context switches when blocking and unblocking by using knowledge of the next semaphore that a thread needs to lock before making a blocking call. Last, the lock-free and wait-free methods of synchronization provide alternate ways of controlling access to shared memory in a real-time environment.

## References

[1] Bruce L. Jacob and Shuvra S. Bhattacharyya. "Real-Time Memory Management: Compile-Time Techniques and Run-Time Mechanisms that Enable the Use of Caches in Real-Time Systems." Technical Report UMIACS-TR-2000-60, Institute for Advanced Computer Studies, University of Maryland at College Park, September 2000.

[2] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment." *Journal of the ACM*, 1973.

[3] Khawar M. Zuberi, Padmanabhan Pillai, and Kang G. Shin. "EMERALDS: A Small-Memory Real-Time Microkernel." *Operating Systems Review*, December 1999.

[4] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." *IEEE Transactions on Computers*, September 1990.

[5] Felice Balarin, Luciano Lavagno, Praveen Murty, and Alberto S. Vincentelli. "Scheduling for Embedded Real-Time Systems." *IEEE Design and Test of Computers*, 1998.

[6] Robert A. Walker and Samit Chaudhri. "Introduction to the Scheduling Problem." *IEEE Design and Test of Computers*, 1995.

[7] Ken Tindell. "True Real-Time Embedded Systems Engineering." Presented at the *Embedded Systems Conference (ESC)*, Summer 2000.

[8] Friedrich Schön. "On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System." *Proceedings of the Third International Symposium on Object-Oriented Real-Time Distributed Computing*, March 2000.

[9] Khawar M. Zuberi and Kang G. Shin. "An Efficient Semaphore Implementation Scheme for Small-Memory Embedded Systems." *Proceedings of the Real-Time Technology and Applications Symposium*, 1997.

[10] Maurice Herlihy. "Wait-Free Synchronization." *ACM Transactions on Programming Languages and Systems*, January 1991.