Augustus P. Lowell
Member of Technical Staff
Abbott Critical Care and Control Systems
1212 Terra Bella Avenue
Mountain View, CA 94043

Overview

This article discusses the problems of implementing a watchdog software monitor in multiple-thread applications, and describes several software architectures for overcoming these problems. It begins with an overview of the capabilities and requirements of watchdog monitors, and a synopsis of multiple-thread schedulers. It then describes the limitations imposed on a watchdog monitor by a multiple-thread scheduler. Finally, it develops an architecture for a flexible system monitor to avoid these limitations,

1. Watchdog Monitors

High reliability embedded systems generally require some mechanism for detecting and stopping errant software sequences which may occur due to EMI, hardware failure, spurious noise, or incorrect input. There are two approaches to the problem of software error detection which reflect the reliability, performance, and cost goals of the systems which use them.

One approach for verifying proper software operation, used extensively in the field of *fault tolerant systems*, is called *checkpointing*. It focuses on the results produced by a segment of code, but not the actual sequence of operation. A system which uses checkpoints duplicates the functionality (though not necessarily the implementation) of each segment of code, and allows the original and duplicate to operate independently; the results produced by the two segments are compared, and any discrepancies are assumed to be the result of improper operation. Typically, the two segments are executed on independent processors to protect against single-point hardware failures and increase throughput, but sequential operation on a single processor may achieve many of the same advantages, at the cost of more than doubling compute resources required for the system. In many cases, additional software and hardware are used to enable the system to correct faults, rather than simply detect them. Fault-tolerant systems using checkpoint techniques are used for such applications as flight control and nuclear reactor control, because they can guarantee detection and correction of single-point failures. The main disadvantage of such systems is cost and complexity: software redundancy more than doubles the necessary computing capacity, and the checkpoints require synchronization between the independent code segments. Additional hardware redundancy for sensors and timing sources increases complexity and cost still further. For low-cost, compact systems, fault-tolerant techniques are generally not acceptable.

The other mechanism for software error detection focuses on the actual sequence of software operation, and takes the form of a *watchdog timer*, implemented in hardware, which periodically resets the machine unless the software takes some preventive action to indicate it is healthy. The assumption is that properly executing software should "get around to" updating the watchdog within some maximum interval, and that improperly executing software will eventually fail to do so (or, put another way, proper software sequencing is a *necessary* and *sufficient* condition for proper watchdog updating). To this end, many watchdog schemes require some processing intelligence and/or state verification as part of the watchdog hardware update; this minimizes the probability of random instruction sequences properly updating the watchdog circuitry.

For a watchdog to reliably detect errant software sequences, two conditions must always be true:

a) The longest possible execution path between watchdog updates must require less than one watchdog timer interval. (*Sufficiency Condition*)

b) The execution of the watchdog update must depend on correct software sequencing. (*Necessity Condition*)

Condition (a) ensures the software will always "get around to" updating the watchdog within the watchdog timer interval. Condition (b), which is generally more difficult to verify, is what correlates proper watchdog updates with proper software sequencing. In particular, condition (b) prohibits the watchdog update from being generated *automatically* by a timed interrupt, because the interrupt may function properly independent of the state of the software being monitored.

The sufficiency condition may be met in one of two ways:

a) The actual path through the code between watchdog updates is well-defined and guaranteed, a-priori, to require less then one watchdog update period to execute.

b) An external, timed event forces a watchdog update at the required interval.

Implementation (a) we will call the *path criterion*, because it relies on foreknowledge of the execution path through the program to guarantee the code will meet the sufficiency condition. Implementation (b) relies on the accuracy of some time base to make this guarantee, and will be called the *time criterion*. Both criteria are quantifiable and predictable, and allow straightforward calculation of the worst-case time between watchdog updates, assuming correct code sequencing .

Meeting the necessity condition is not as straightforward, because it requires some means to verify proper sequencing at every step during the watchdog interval. Again, there are two ways to meet this condition:

a) The actual path through the code between watchdog updates is well-defined and immutable, such that arrival at the watchdog update point can occur only by proper traversal of the path.

b) The path through the code is documented during execution, such that an independent monitor can track and verify sequencing between watchdog updates.

Implementation (a) can be called the *arrival criterion*, because it assumes that arrival at some point in the code can occur only if the preceding sequence executed correctly. In implementation (b), the *monitor criterion*, some description of the actual path through the code is saved for later analysis. In practice, the monitor function in implementation (b) uses a form of the arrival criterion at discrete points within the code to verify proper sequencing.

The arrival criterion, which is the classic approach to watchdog monitoring, actually provides a rather questionable "guarantee" of the necessity condition. Consider the structure of the requirement:

IF we arrive at the update point THEN we followed the specified path

which is logically equivalent to its contrapositive:

IF we did not follow the specified path THEN we did not arrive at the update point

Obviously, a clever critic could probably find some path by which misexecuting software (which, by definition, does not follow a path defined by the programmer) can arrive at the update point in the proper time, and thereby escape detection by the watchdog monitor (**figure 1**). The counter to that argument is that any such path is bound to leave behind some corrupted state or data, which will ultimately result in an error trap or failure to update the watchdog at a later time; and if it doesn't, it was probably indistinguishable from properly executing code, and needn't have been trapped anyway. The critic argues, correctly, that our system for trapping unforeseen errors is blind to a class of errors which, though perhaps unlikely, can occur with potentially disastrous results; the counter argument, though probably accurate, is based more on intuition than science, because estimating the actual probabilities involved is impractical, if not impossible.

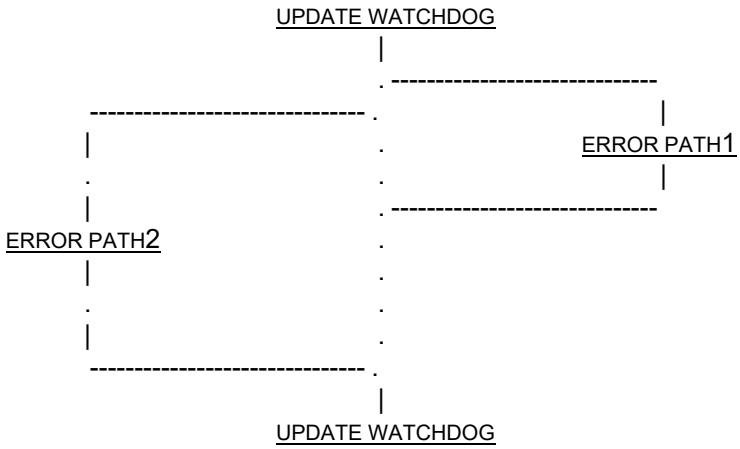

Figure 1: Undetectable Sequence Errors

An enhancement to the arrival criterion, designed to increase the likelihood of detecting errant software, adds the generation and transformation of some state information at *status points* in the code between the watchdog update points. In this scheme, a watchdog update can occur only if the code arrives at the proper point *and* has properly transformed the state information along the way, perhaps by incrementing a counter or manipulating a bit pattern (**figure 2**). This scheme is not perfect, however, because errant sequences between status points will still not be detectable; it eliminates a number of possible errant sequences, but not all.

In general, the closer together the checkpoints, the better the confidence in the result; for example, for the simple failure mode of a random jump from one point to another, decreasing the distance between checkpoints from N opcodes to M opcodes decreases the probablilty of not detecting the error by a factor of approximately (M/N). The theoretical limit, though not a practical one, is a checkpoint after each opcode; even in this case, however, a single opcode error may pass undetected, and the scheme is not perfect. In a real system, the designer must trade off confidence in the error detection scheme with system performance. If any possibility of undetected error, however small, is unacceptable, fault-tolerant techniques must be used instead of a watchdog monitor.

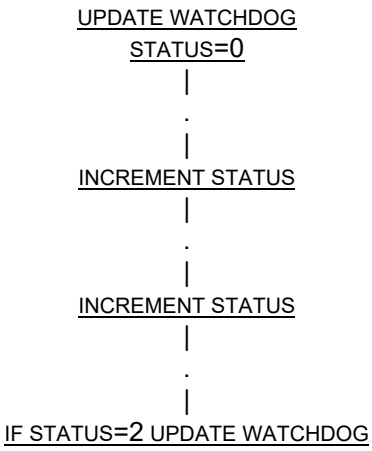

Figure 2: Watchdog Update with Status Points

2. Program Threads

A program *thread*, or *task,* is an independent sequence of instructions which implement some program function. As the metaphor implies, several threads may be joined end-to-end (executed in sequence) to form another, longer thread. In general, a thread may be thought of as the *logical path* through the program code from some starting point to some ending point. In a typical embedded system, the logical path through the code may not be the actual sequence in which instructions are executed, because hardware interrupts or operating system functions will interject other sequences (or threads) between logical steps.

3. Thread Scheduling

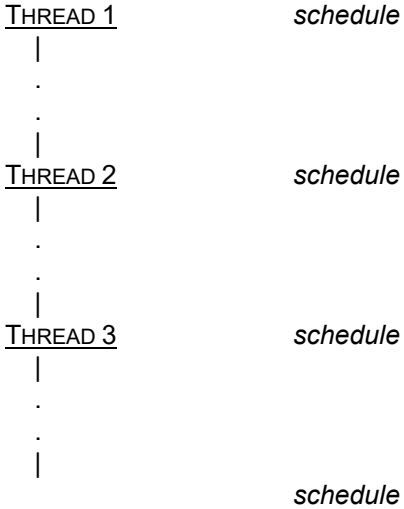

Figure 3: Control Loop Scheduling

A typical program will consist of many threads implementing different functions of the application. *Scheduling* is the process by which the execution of the various threads is controlled, and a *scheduler* is the mechanism which implements scheduling. At a particular time a thread may be either *active*, *ready*, or *suspended*. The thread currently executing is the active thread; only one thread may be active at any time. Threads which may be executed by the scheduler, but are not currently executing, are in a ready state; a suspended thread may not be executed. Typically, a thread will be suspended when data or machine resources it requires to function are not available, or when its function is not required, and become ready when resources are available or its function is required.

The simplest form of scheduler is a *sequencer*, also called a *cyclic executive* or a *control loop*, which executes each thread, in its entirety, in a sequence pre-determined by program design (**figure 3**). For example, a program may read a keyboard (thread 1), filter out non-ascii characters (thread 2), then write the received characters to a screen (thread 3). Because the threads are executed sequentially and without interruption, the sequence of threads may be thought of as a single, larger, thread; for this reason, programs scheduled in this way are referred to as *single-thread* programs. Control loops are *cooperative* schedulers, because the threads are not pre-empted during execution -- they cooperate with the scheduler by indicating when they no longer require control (eg. by terminating or returning when they are done).

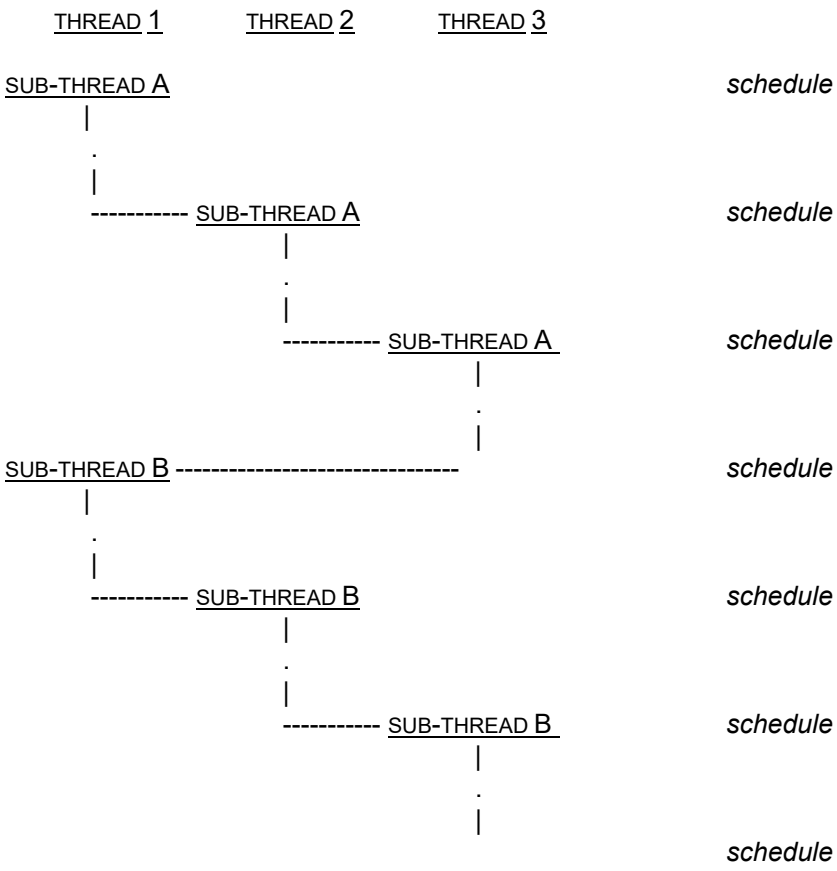

Figure 4: Round Robin Scheduling

An alternative scheduling scheme, the *round-robin* scheduler, intertwines threads by allowing each to execute some portion of its instructions, then interrupting it to allow another thread to run (**figure 4**). This execution/interruption cycle continues until all threads have partially executed, then repeats for the next portion of each thread. In this way, the threads seem to run simultaneously (viewed over a time frame of several cycles), rather than in sequence; for this reason, programs scheduled in this way are referred to as *multiple-thread*, or *multi-tasked*, programs. In general, each time a thread is interrupted the scheduler must save the state of the machine, or the *context*, so that when the thread is reactivated it will start with the same conditions it had when it stopped.

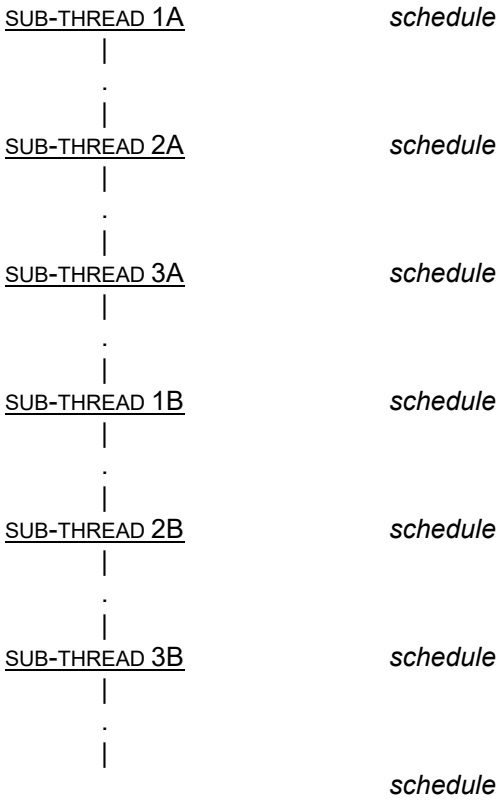

Figure 5: Control Loop Model of Round Robin Scheduler

Round-robin schedulers may be either *cooperative* or *pre-emptive*. In a cooperative round-robin scheduler, each thread is broken down into *sub-threads*, with well-defined entry and exit points, and the scheduler may only switch between threads at the end of a sub-thread. In a sense, each sub-thread is, itself, a thread, and the cooperative round-robin scheduler may be considered a control loop which sequences sub-threads (**figure 5**). Unlike a control loop, however, the order of thread execution for a round-robin scheduler is not built into the program structure; rather, it is controlled by the order in which threads become ready during program execution.

A pre-emptive round-robin scheduler, also known as a *time-slicer*, may interrupt, or pre-empt, a thread at any point in its execution, typically after some pre-determined interval. In this case no well-defined sub-threads exist.

Another type of multiple-thread program scheduler, the *prioritized* scheduler, assigns a priority to each thread, and always allows the highest priority thread to execute in its entirety (**figure 6**). In this scheme, the scheduler switches between threads when the high priority thread suspends itself, or when some event makes a higher priority thread ready or changes the relative priorities of the threads.

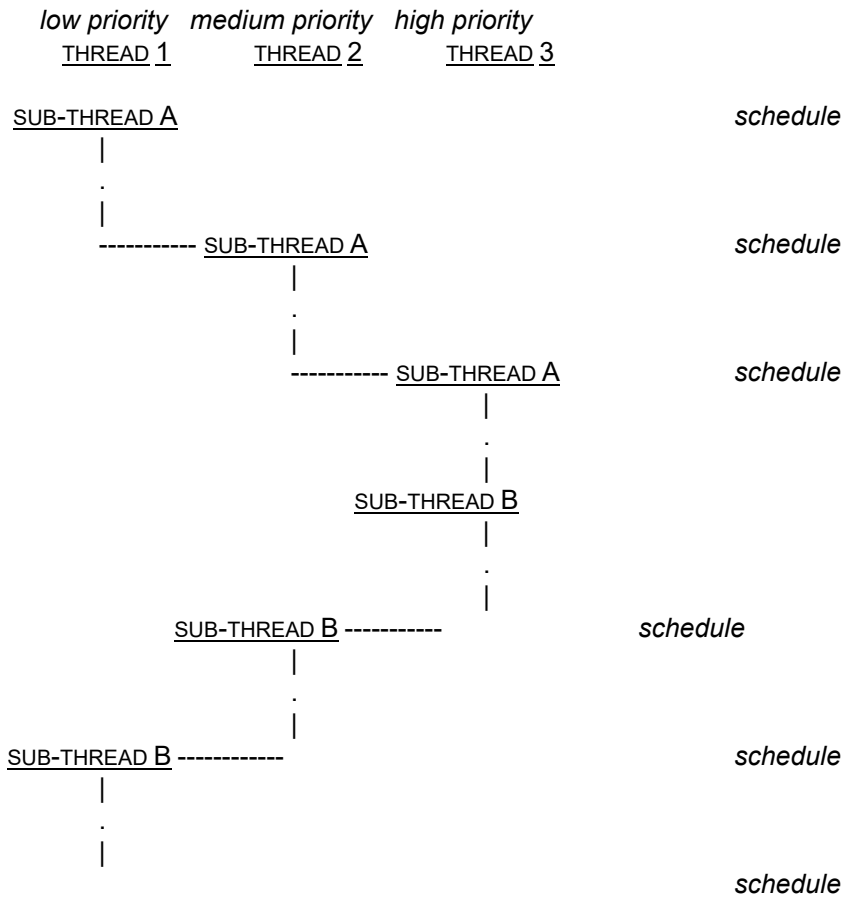

Figure 6: Prioritized Scheduling

As with the round-robin scheduler, a prioritized scheduler may be either cooperative or pre-emptive, depending on whether a switch between threads may take place only at the end of a sub-thread or at any time. If the scheduler is cooperative, it may again be considered a special case of a control loop, which sequences sub-threads; in this case, the order of sub-thread execution is determined dynamically by the relative priorities of ready threads.

These three scheduling schemes may be combined (as with a prioritized time-slicer, which allows all high priority threads to run in a round-robin fashion) or specialized for a particular application. In addition, a pre-emptive scheduler may be restricted to behave cooperatively by *locking* and *unlocking* task switching at pre-determined points in the code, thus preventing pre-emption during sequences in which the code is locked. Schedulers may also be combined hierarchically, as is the case when an application, scheduled along with other applications by the operating system, internally implements a scheduler to control its own functions.

Note that an interrupt controller is a special case of a prioritized pre-emptive scheduler, implemented in hardware, in which interrupt handlers have higher priority than other code. In general, a computing system will operate two hierarchically linked schedulers, one in hardware (the interrupt system) at the top of the hierarchy, and one in software which may be considered the lowest priority thread, or *idle task*, handled by the interrupt scheduler.

As a final note, a thread often requires information from other threads to execute its function; for instance, a thread which calculates a number must provide that information to another thread for display. In a single-thread scheduler, the information is frequently passed as a function argument; the function call also synchronizes execution of the threads.

In a multi-tasking system, threads operate independently, and information is generally passed as some form of message between autonomous threads. Typically, the scheduler is responsible for routing these messages, and the arrival of a message may be the event which causes the scheduler to reschedule the threads.

4. Watchdog Monitors and Schedulers

In examining the special requirements for implementing a watchdog monitor in conjunction with a scheduler, it is useful to separate, as much as possible, the various schedulers operating in the system and treat them separately. In particular, the software scheduler operating as the idle task in the interrupt controller may be treated as an autonomous block in designing its watchdog functions. Of course, when determining execution times for blocks of code, it is necessary to account for interrupt handlers which may be activated while that block is executing; fortunately, in all other ways the assumption of autonomy is valid.

The same assumption may be made for all hierarchically linked schedulers; a watchdog monitor for the master scheduler treats the slave scheduler as one of a number of tasks which must be monitored, while the slave scheduler must account for execution time allocated to other tasks by the master scheduler, which appear to the slave as interrupt handlers. Within these constraints, each scheduler may be treated independently.

In the case of multiple threads operating with a cooperative scheduler, the executing code may be treated as a single-thread program, consisting of sequences of sub-threads (**figure 5**). This is true for any of the schedulers described above; while the order in which the sub-threads execute may not be determined prior to program execution, each is guaranteed to finish before the next one begins. In this case, if each sub-thread meets the path criterion and the arrival criterion, the sequence of threads will satisfy both the sufficiency and necessity conditions, and watchdog monitoring will be possible.

Pre-emptive schedulers create much more complex situations, because the execution of a thread may be interrupted at any time, at any point in the sequence. As a result, the execution path during any given interval is not well-defined, and the path criterion cannot be satisfied. For the same reason, the arrival at a single watchdog update point in one thread gives no indication of the actual execution path during that interval, and cannot provide any information about the sequencing of other threads, in violation of the arrival criterion.

This is a very general statement, and not entirely true. There are special cases of pre-emptively scheduled systems for which the combination of length, number, and status of individual threads can guarantee that each thread reaches some status point within each watchdog update interval. In these systems, both the path and arrival criteria are met for all threads if the watchdog update requires each thread individually to meet the arrival criterion. These systems, however, are not extensible; addition of

more threads, or a change in the length of an individual thread, can cause either the path or arrival criterion to fail by extending the sequencing time beyond the watchdog update interval.

Since, in general, pre-emptively scheduled threads will not meet the path criterion, the time criterion must be used to guarantee the watchdog is updated within the required interval. This, however, guarantees the arrival criterion will not be met, because there is no longer an "update point" at which to arrive; rather, there is an update time, which may occur at any point in the code. As a result, any system using a pre-emptive scheduler must implement a *monitor*, which satisfies the monitor criterion, to verify sequencing of each thread independently. This can be done by embedding *milestones* within each thread, points at which the arrival criterion may be assessed by the monitor. Status points between milestones may be used to increase confidence in the result of the monitor check.

Use of the milestones requires some additional complexity. Threads which are suspended will never reach their milestones, and the monitor should not expect them to; threads which are ready, but do not become active, will also not reach their milestones within a given watchdog interval, although they may be expected to do so when they are activated. Threads which are active at some time within the watchdog interval may nonetheless not reach their milestones if they are not allowed to run for sufficient time. In addition to the milestones, the monitor requires some method of detecting which tasks are active and which are supended, to determine which milestones to test during a given interval.

5. Architectures

5.1. Cooperative Scheduler

5.1.1. Arrival Criterion

For threads operating within a cooperative scheduler, the sub-thread structure shown in **figure 7**, which we will call a *local monitor*, meets the arrival criterion to satisfy the necessity condition. The local monitor is the traditional watchdog technique used in single-thread applications. Assuming the size of the application code between watchdog updates, including any possible interrupt handlers and the scheduler itself, is small enough to meet the path criterion, it will also meet the sufficiency condition, and watchdog monitoring will be successful. The *lock* and *unlock* blocks indicate the beginning and end of the sub-thread, where task switching is allowed.

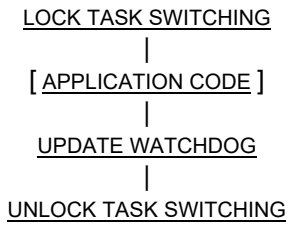

Figure 7: Sub-Thread for Local Monitor

Note that each sub-thread is responsible for updating the watchdog at its exit point. An alternative, is to place the watchdog update routine in the scheduler, such that it is executed automatically each time task switching is unlocked. This makes the invocation of the watchdog update transparent to the application software, though it still must maintain sub-threads small enought to satisfy the path criterion. Status points may be implemented within the application code to increase watchdog monitor reliability.

5.1.2. Monitor Criterion

While the local monitor described above meets all the requirements of a watchdog system, a *global monitor* can potentially provide more information about where software has gone wrong. A sub-thread structure for such a monitor is shown in **figure 8**. The monitor maintains a status register for each thread, and for the scheduler. Each sub-thread begins and ends with its milestone, a routine which updates the status register with its current status.

The monitor is activated at a fixed interval whether or not task switching is enabled, perhaps by a timed interrupt, and checks the status registers of each thread for a valid status message. If it encounters an ***I'm Aleep*** message, it assumes the thread is inactive and accepts the status message as valid without change. If it encounters an ***I'm OK*** message, it again accepts the message as valid, then overwrites the thread status register with a ***Status Unknown*** message and continues. If all registers are found to contain valid status messages, the monitor updates the watchdog and suspends.

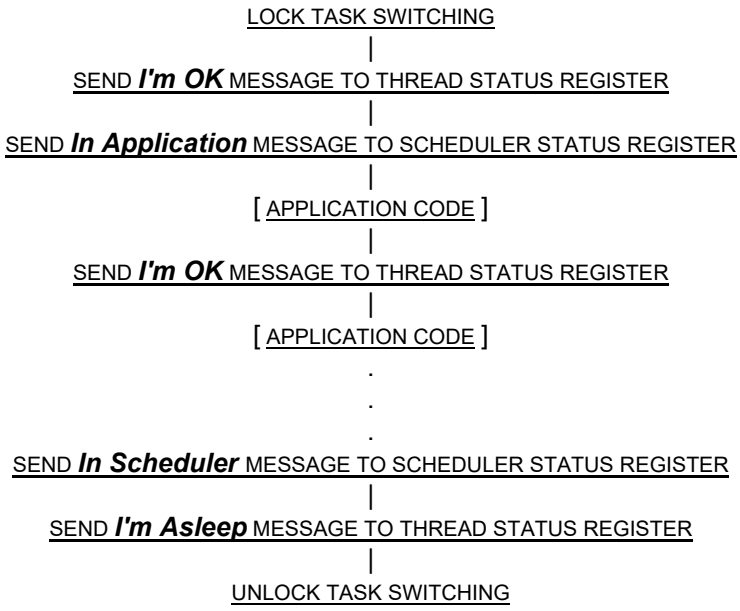

Figure 8: Sub-Thread for Global Monitor -- COOPERATIVE

A ***Status Unknown*** message in a thread status register during the monitor check indicates an active thread which has not reached its milestone within the watchdog interval. The monitor may respond to this error in one of two ways:

a) Fail to update the watchdog timer. In this case, a system reset will occur when the timer reaches its terminal count.

b) Update the watchdog timer, try to recover from the error, and generate a software error message for the user.

In either case, the monitor can generate an error log entry for subsequent debugging, indicating not only that a failure occurred, but in which thread it occurred.

In addition to thread function, the monitor also checks scheduler function via a scheduler status register. From **figure 8** we see that if the monitor is activated during the scheduler operation, between the ***I'm Asleep*** message from one thread and the ***I'm OK*** message from the next thread (or the same thread, if no task switch occurs), all thread status registers will contain ***I'm Asleep*** messages, which the monitor will ignore. A software error which occurs during this period will, therefore, pass untrapped through the monitor. To prevent this, the monitor checks the scheduler status register for a ***In Application*** or ***In scheduler*** message, and replaces a ***In Scheduler*** message with a ***Status Unknown*** message. Again, if the monitor finds a ***Status Unknown*** message in the scheduler status register, it indicates the scheduler did not activate a new thread, and an error should be generated. If the monitor ever finds a ***In Application*** message in the scheduler status register when all threads are reporting ***I'm Asleep***, or if more than one thread is reporting ***I'm OK***, it should generate an error.

As in the case of the local monitor, the path criterion must be satisfied by each sub-thread to ensure the status registers are updated within the watchdog interval. To achieve this, the execution time for the sub-thread plus the total execution time for all possible interrupt handlers and the scheduler must be less than one watchdog update interval.

5.2. Pre-Emptive Scheduler

For systems running under pre-emptive schedulers, the global monitor is the only architecture which allows proper function of a watchdog. The monitor used for cooperative schedulers, however, relies on each sub-thread to post its status messages before and after a task switch; this is how it guarantees the active thread will arrive at a milestone, and update its status register, if it becomes inactive before the watchdog update time. Under a pre-emptive scheduler the thread may be switched at any time, and cannot guarantee a message will be posted either before or after the switch. An alternative is to move the posting functions into the scheduler, thus guaranteeing an update; this, however, disassociates the status message from the thread generating the status, and will not satisfy the necessity condition -- as long as the scheduler functions, the status register will be updated, independent of the state of the thread. We need another technique for ensuring active threads update their status registers reliably and punctually.

The problem we have with a pre-emptive scheduler is one of time; the monitor has no way of knowing whether a given thread has been active long enough to have reached its milestone. Typically, we might say that a thread active for two monitor intervals in a row has been active long enough; there will, however, be cases in which the thread was deactivated between updates, and has hardly executed at all.

The solution to the problem is to monitor not only the status register for each thread, but also a running total of its execution time. Using this technique, the monitor can determine at each interval which threads should have reached a milestone, and which should not.

One such technique for monitoring time-sliced threads is shown in **figure 9**. Here, the scheduler maintains a log of execution time for each thread under its control, based on a-priori knowledge of the time-slice interval. Each time the thread is activated and deactivated the scheduler adds one time-slice interval to the elapsed time for that thread. When the monitor is activated, it checks each thread status register for an ***I'm OK*** message, just as in the case of a cooperative scheduler. If, however, it finds a ***Status Unknown*** message, it doesn't immediately assume an error; instead, it checks the elapsed time for the thread and compares it to a known *maximum milestone interval*. Only if the elapsed time exceeds the maximum milestone interval without a valid status message will the monitor generate an error. Whenever the monitor finds a valid status message in the status register, it resets the elapsed time to zero, and the interval for that thread begins again.

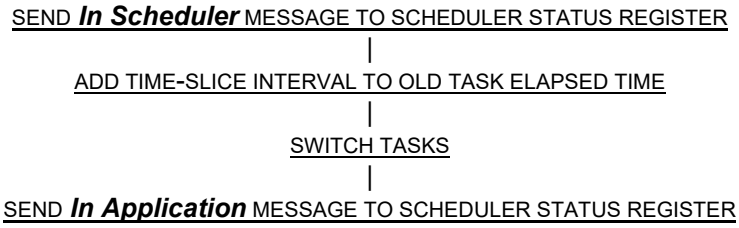

Figure 9: Elapsed Time Monitor for Time-Sliced Scheduler

In general, this scheme relaxes the tolerance on the milestone interval for each thread, because the interval is allowed to extend over several watchdog update periods. In addition, it allows different intervals for different threads, depending on their relative importance to overall system function. It also allows specification of minimum milestone intervals, to trap errors which cause arrival at the milestone too soon.

Unlike the cooperative case, threads need not indicate when they are suspended or ready, because the monitor bases its decisions on elapsed time, which will not change under those conditions. As a result, the thread need only post ***I'm OK*** messages at its milestones; from the perspective of the thread, this looks exactly like the situation for a local monitor under a cooperative scheduler (**figure 7**). The scheduler status register again ensures the scheduler is operating properly, and guarantees the elapsed times for the threads will be incremented properly.

The simplest implementation of this scheme requires that an active thread be able to reach its milestone within one time-slice period. In this implementation, the scheduler need only mark each thread as having been activated or not during the watchdog update interval; the monitor expects valid status messages from all threads that have been marked active.

The elapsed time monitor technique may be generalized to a prioritized scheduler with a slight variation: in a prioritized scheduler there is no fixed interval between task switches, so another method must be used to determine elapsed time for the threads. The obvious solution is a high resolution timer, which may be read to determine the times at which task switches take place. One complication occurs when a single thread holds the processor for an entire watchdog update period. In this case, the scheduler is never activated, and cannot update the elapsed time for the thread. If this occurs, the scheduler status register will not be updated, and the monitor can detect the condition and update the elapsed time itself.

As shown in **figures 10a** and **10b**, the scheduler initially posts a ***Entering Application*** message to the monitor, indicating it has been activated during the interval, and therefore has updated the elapsed time log. The monitor replaces this with a ***In Application*** message to indicate it saw the scheduler message, but did not update the log. If the monitor subsequently finds the ***In Application*** message in the scheduler status register, it knows the scheduler was not activated during the interval, and the elapsed time log was not updated; consequently, the monitor fills the scheduler status register with a ***Old Application*** message and updates the elapsed time for the active task. If the monitor finds a ***Old Application*** message in the scheduler status register, it again knows the scheduler has not been activated and updates the elapsed time for the active task.

If, when activated, the scheduler reads a ***Old Application*** message from its own status register, it concludes the monitor has updated the elapsed time and does not do so again; for any other message, the scheduler updates the elapsed time log. In either case, the scheduler always posts a ***Entering Application*** message before entering the new active thread.

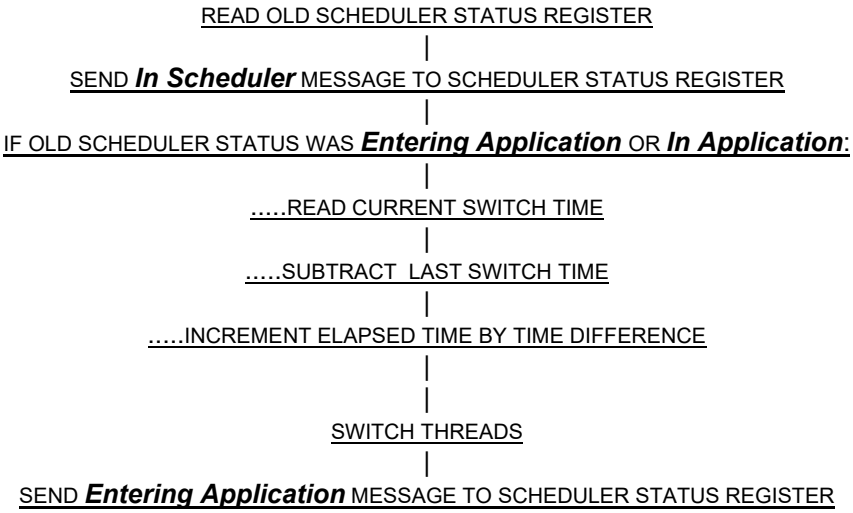

Figure 10a: Scheduler for Generalized Elapsed Time Monitor

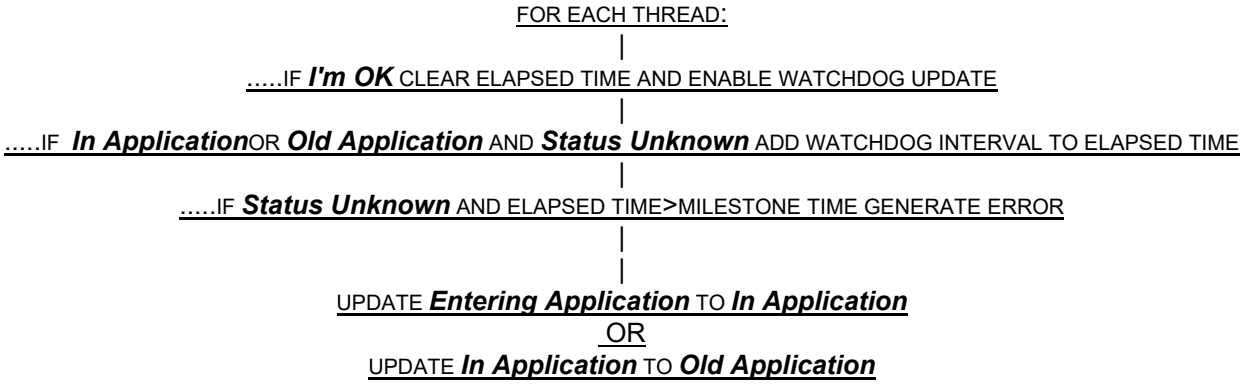

Figure 10b: Monitor for Generalized Elapsed Time Monitor

As with the time-sliced scheduler, each thread need only post ***I'm OK*** messages at its milestones to ensure proper monitor function.

Summary

Proper functioning of a watchdog monitor requires watchdog updates to occur within the watchdog timer period (sufficiency) and to depend on proper software sequencing (necessity). Sufficiency may be guaranteed by analysis. Necessity cannot be guaranteed for all possible failure modes, but probabilities of error detection for certain classes of failure modes may be calculated, and overall probabilities may be assessed qualitatively.

Watchdog monitoring for software operating under a cooperative scheduler may be achieved either with local monitoring at the sub-thread level or global monitoring by an independent task. To achieve either of these monitors, each sub-thread must satisfy both the sufficiency and necessity conditions, and must post status, either directly to the watchdog or to the independent monitor, before activating task switching.

Pre-emptive schedulers complicate watchdog monitoring, and require a global monitor to meet the sufficiency and necessity conditions. The monitor function must track both thread status and thread elapsed execution time to properly assess software function; in prioritized systems, a high-resolution timer is required to track execution time. Under the global monitor, the time between status updates for individual threads may be adjusted to reflect the importance of the thread to overall system performance.