1998

A Multithreaded Scheduler for a High-Speed Simulator

Gene Saghi

Kirk Reinholtz

Paul Savory
University of Nebraska at Lincoln, psavory2@gmail.com

G. Saghi, K. Reinholtz, and P. Savory (1998), "A Multithreaded Scheduler for a High-Speed Simulator," ***Software Practices and Experiences,*** Volume 28, No. 6, pp. 641-656

A Multithreaded Scheduler for a High-Speed Spacecraft Simulator[*]

Gene Saghi
Advanced Designs, Inc.
1495 Garden of the Gods Road
Colorado Springs, CO. 80919
gsaghi@a-d-inc.com

Kirk Reinholtz
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, MS 303/310
Pasadena, CA 91109-8099
kirk@zipcode.jpl.nasa.gov

Paul A. Savory
Industrial and Management Systems Engineering
University of Nebraska – Lincoln
175 Nebraska Hill
Lincoln, NE 68588-0518
savory@engrs.unl.edu

October 1997

Submitted to Software Practice and Experience

[*] For this research effort, Dr. Saghi and Dr. Savory were supported through Summer Faculty Fellowships provided by the American Society of Engineering Educators (ASEE) in combination with the National Aeronautic and Space Administration (NASA). Portions of the work described were performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration.

G. Saghi, K. Reinholtz, and P. Savory (1998), "A Multithreaded Scheduler for a High-Speed Simulator," ***Software Practices and Experiences,*** Volume 28, No. 6, pp. 641-656

Summary

The Cassini spacecraft is on its journey to Saturn to perform a close-up study of the Saturnian system; its rings, moons, magneto-sphere, and the planet itself. Sequences of commands will be sent to the spacecraft by ground personnel to control every aspect of the mission. To validate and verify these command sequences, a bit-level, high-speed simulator (HSS) has been developed. To maximize performance, the HSS is implemented with multiple threads and runs on a multiprocessor system. A key component of the HSS is the scheduler, which controls the execution of the simulator. The general framework of the scheduler can be adapted to solve a wide variety of scheduling problems. The architecture of the scheduler is presented first, followed by a discussion of issues related to performance and multiple threads. Second, the avoidance of deadlocks and race conditions is discussed and an informal proof for the absence of both in the scheduler is described. Finally, a study of various scheduling policies is provided.

Key Words - deadlock, multiprocessing, multithreaded, object-oriented, scheduling policy, simulation.

Introduction

An international endeavor involving the National Aeronautic and Space Agency (NASA), the European Space Agency, and the Italian Space Agency has developed the Cassini spacecraft to learn more about Saturn's atmosphere, magnetic field, rings, and moons. Unlike the Voyagers that flew past Saturn on their way out of the solar system, Cassini's mission is a four-year, close-up study of the Saturnian system. The mission represents a rare opportunity to gain significant insights into major scientific questions about the creation of the solar system and the conditions that led to life on Earth. (A detailed description of the spacecraft and its mission can be found at the NASA Cassini home page: http://www.jpl.nasa.gov/cassini/)

During Cassini's mission, sequences of commands will be sent to the spacecraft by ground control personnel. Each sequence instruction will direct the spacecraft to perform some operation such as firing a thruster or sending a control command to one of the on-board scientific instruments. To validate and verify these command sequences, a "bit-level" high-speed simulator (HSS) has been developed by NASA.

The HSS provides models of each of the thirty-one data system hardware components on the spacecraft. Examples of hardware components include: the 1750a central processing unit, the intercommunication bus, the Reed-Solomon downlink, and the solid state data recorder. The HSS is implemented as a "bit-level", software-only simulator derived directly from the hardware specifications. That is, the thirty-one HSS models are actually hardware emulators for the Cassini spacecraft data systems. Because the emulator models are bit-level representations of the actual hardware, the HSS directly executes the flight software that is to be loaded on the spacecraft.

One of the key control components in the HSS is the scheduler, whose task is to schedule the execution of models so as to minimize execution time while keeping the models synchronized with one another. To meet its high-performance goal, the Cassini HSS is implemented with multiple threads that run on multiple processors. Thus, the scheduler must manage how "threads" execute each of the thirty-one hardware emulators. Race conditions and deadlock are potential problems with any parallel program. The

HSS scheduler has the additional goals of being provably free of race conditions and deadlocks.

This paper describes the design and development of a general-purpose, multithreaded scheduler that has high performance and is provably free of race conditions and deadlocks. In addition, the selection of a scheduling heuristic is described. Although this paper focuses on the Cassini HSS scheduler, the framework described here can be easily applied to many other scheduling problems where a multithreaded approach is desired. The following sections describe some of the key issues associated with multithreaded programming, the HSS scheduler's object-oriented framework, the scheduler algorithm, the prevention of race conditions and deadlocks, and the choice of a scheduling policy.

Multithreaded Programming Considerations

Before discussing the design of the HSS scheduler, it is necessary to briefly discuss the multithreaded programming paradigm. Readers that are familiar with multithreaded programming issues may wish to bypass this section. Those interested in reading more about multithreaded programming are referred to references [1] and [2]. Our discussion on threads is based on Sun Microsystem's Solaris 2 threads implementation.

A process is defined here as a running program and all of the state information associated with it. A thread is an independent stream of control within a process. Traditional processes have a single thread of control and possess sole ownership of the process's memory and other resources. In a multithreaded environment, a process can have many active threads, with all threads sharing the process' memory and resources. On a multiprocessor system, different threads may execute on different processors. The threads execute concurrently, which can result in a substantial improvement in performance. Better performance often results on uniprocessors due to better utilization of the processor. For example, when one thread blocks on an I/O request, another thread can execute and the processor remains fully utilized.

A process is made up of several lightweight processes. Each lightweight process (LWP) can be viewed as a virtual processor that is available for code execution. On a multiprocessor system, each LWP can concurrently execute on a different processor. LWPs are scheduled onto the available processors according to their scheduling class and priority. Every LWP can have many threads associated with it, but threads assigned to the same LWP run sequentially with respect to one another. In the general case, the user does not know how threads are distributed among LWPs. However, it is possible for the user to create a bound thread, which is a thread that is permanently tied to a specific LWP.

As stated earlier, threads share memory and other system resources. This can lead to race conditions. Races are events with nondeterministic outcomes in otherwise deterministic programs. For example, consider two threads, *A* and *B* with the code segments:

Thread A	**Thread B**
`x = x + 1;`	`x = x - 1;`

If the value of `x` is 2 before the code segments execute, the value of `x` afterward could be 1, 2, or 3, depending on the order in which the reads and writes associated with the code segments are actually performed. This type of nondeterministic behavior is generally undesirable.

To prevent the above variety of race condition, access to shared data must be serialized. In multithreaded programming, this is done through the use of a mutual exclusion lock, or mutex. The first thread that calls the lock on a mutex gets ownership of the mutex. It can then proceed to access the shared data protected by the mutex. Further calls to lock the mutex will fail, causing the calling thread to sleep. When the mutex owner unlocks the mutex, one of the sleeping threads will be awakened and given the chance to lock the mutex, although another thread could actually obtain ownership of the mutex first.

It is important to realize that the proper use of mutexes alone does not prevent the occurrence of all race conditions. Consider the following example:

```
   Thread A              Thread B
mutex_lock(&m);      mutex_lock(&m);
x = x + 1;           x = x * 2;
mutex_unlock(&m);    mutex_unlock(&m);
```

If the value of `x` is again 2 before the code segments execute, the value of `x` afterward could be 5 or 6 depending on which thread obtained the mutex first.

The amount of code protected by mutexes in a multithreaded program should be minimized because the use of mutexes significantly reduces the concurrency of the program. Although the use of mutexes is necessary to control access to shared resources, their use can lead to another significant problem called deadlock. <u>Deadlock</u> can occur whenever a circular chain of threads exists in which each thread owns one or more mutexes that are requested by the next thread in the chain. For example, consider the case where thread *A* owns mutex m_1 and is waiting to obtain mutex m_2, while thread *B* owns mutex m_2 and is waiting to obtain mutex m_1. Neither thread can continue. Deadlock also occurs when one thread already owns a mutex, but tries to obtain ownership of the mutex again.

Among the goals for the HSS scheduler were high performance and the absence of race conditions and deadlocks. In the following sections, the way in which these goals were achieved is discussed.

Scheduler Architecture

The HSS scheduler is implemented in C++ because of that language's runtime efficiency and its support for object-oriented programming. A simplified representation of the overall architecture of the HSS is shown in Figure 1. The simulation elements, or models, are connected to each other via specialized interface objects called <u>splices</u>. Each model splice is a unidirectional communication channel. Additionally, the scheduler is connected to each model by dedicated scheduler splices. There are two user interfaces available in the HSS: a command-line interface and a graphical-user interface. These interfaces are implemented using an embedded Tcl (“tool command language”) interpreter, a freely distributed interpretive language developed at the University of California at Berkeley [3].

The remainder of this section discusses the architecture of the HSS scheduler. A graphical representation of the structure of scheduler is depicted in Figure 2.

Barriers

The HSS scheduler framework is based on a barrier mechanism. A barrier is an object used to synchronize models. Each barrier has an entered set, an entry set, an event set, and an exit set. The entered set is used to track which models have arrived at the barrier. No model in the barrier entry set may proceed beyond the barrier until all models in the entry set have arrived at the barrier. Once all models in the entry set have reached the barrier, the events associated with the event set are performed. Then, the models in the barrier exit set are released, i.e., enabled for execution. The only restrictions on model entry and exit sets are as follows:

Given barriers b_i and b_j that occur at times $b_i(t)$ and $b_j(t)$, respectively,

1. If $b_i(t) = b_j(t)$, model m_k can be a member of at most one of the barrier entry sets and one of the barrier exit sets.
2. If $b_i(t) < b_j(t)$, model m_k can be in the entry set for b_i and b_j if and only if it is also in the exit set for b_i.

Figure 3 illustrates the use of barriers to synchronize model execution.

The Cassini spacecraft is a real-time system. To simulate real-time behavior, the HSS must track the passage of time from the view point of the Cassini system. This is done by storing within each barrier object the simulated time at which the barrier will be encounterred. In addition, each barrier includes a flag that indicates the presence of a barrier breakpoint. The use of this flag is discussed further in the next section.

Barriers can be made periodic. Each barrier has three parameters to control the barrier's periodicity. The phase describes the offset of the barrier's first occurrence from time zero. The cycles and nanoseconds parameters describe the period of the barrier in cycles per number of nanoseconds. In the Cassini spacecraft, there are two reference clocks that are available to all of the system modules. These clocks are used by the modules for the purpose of synchronization. In the Cassini HSS, the same function is accomplished with two periodic barriers.

Models and the Model Queue

As discussed earlier, every hardware entity to be emulated in the spacecraft has an associated model object. These models emulate the hardware in response to the flight software and other models. In the scheduler, a model object stores:

- the identity of the next barrier at which it must stop,
- the time of that barrier,
- a list of barrier stop times for every known barrier the model will encounter in the future,
- the current time for the model in cycles,
- the model state,
- and the cycle rate for the model expressed in cycles per number of nanoseconds.

Models cannot be stopped in the middle of an instruction execution and some models have variable length instructions. The cycle rate information and the instruction to be executed determine the points in time at which a model can be stopped. Thus, the concept of a barrier is relaxed such that a model that reaches a barrier will actually stop after that barrier if the current instruction is not yet complete when the barrier occurs.

A model can be "waiting" for a thread on which to run, "running" on a thread, "at_barrier", or "suspended." A model that encounters a breakpoint becomes suspended. The user can then interrogate the model to determine its state information. In addition, the scheduler will stop executing all other models. The user can then interrogate the simulation models for troubleshooting purposes. Although execution is suspended by the scheduler when any model hits a breakpoint, the models are not guaranteed to be completely synchronized. However, barriers can be used to set bounds on the degree to which models may be out of synchronization.

The model queue holds models waiting to be executed. Figure 4 shows the model queue and demonstrates its use. Models are put on the model queue only when a barrier has been reached, i.e., when every model in the barrier entry set has reached the barrier. At that time, the scheduler reads the barrier exit set, which is the list of models that will continue execution beyond the barrier. Then, the scheduler orders the list of models according to its scheduling policy. The models are then placed on the model queue in the order in which they should be executed. A thread that is ready for more work will remove the next available model from the model queue and execute it. A detailed description of the scheduling policy and its effect on the performance of the scheduler is provided in a later section.

Tasks, Threads, and the Thread Queue

Because the scheduler's goal is to minimize runtime by maximizing parallelism, the scheduler uses bound threads to execute models. This allows each scheduler thread to be scheduled globally by the operating system and results in good performance when the number of LWPs is equal to or slightly greater than the number of processors in the system. For single-processor systems, it is generally better to use unbound threads in order to reduce the operating system overhead.

Every scheduler thread has an associated task object. A task can be in one of three states: idle, running, dying. When a task has a model to run, it is in the running state. When a thread is marked for deletion, the associated task is put in the dying state. Otherwise, the task is in the idle state. Every task also stores the number of the model to run and the time of the next barrier at which the model must stop (both valid only when the task is in the running state). The task also stores its unique identity number.

Every task has a mutex associated with it. This mutex is used to implement a sleep-wait barrier for the task. When a task (thread) is created, its mutex is locked. The thread then executes the code shown below for a sleep-wait barrier.

When it tries to re-acquire the mutex, it is put to sleep. When the user is ready to advance the simulation, a run command is issued to the scheduler, which causes the task mutex to be released. Now the thread will acquire the mutex and then proceed do useful work. Once the thread has done all the work it can, it returns, goes to the top of the loop, and tries to re-acquire the mutex. Once again, it will be put to sleep until another user run command is issued.

Sleep-Wait Barrier

```
while(!done) {
  task.lock();
  if (task.state == dying)
    done = 1;
  else
    do_work();
};
```

Spin-Wait Barrier

```
while(!done) {
  while (task.cond == 0){};
  if (task.state == dying)
    done = 1;
  else {
    do_work();
    task.cond = 0;
  };
};
```

Another possible approach is the spin-wait barrier (shown above). With this approach, the task condition would initially be set to 0. The thread would stay in the inner-most while loop until the user run command changed the task condition to some other value. Then the thread could go on to do useful work. Once finished, the task would once again enter the spin-wait barrier. The difference between these two approaches is that a sleeping thread does not compete for execution time on a processor. This is not an issue if every thread in the HSS has its own processor. However, when there are more threads than processors, the sleep-wait barrier has been found to be a far superior approach in terms of performance.

The thread queue holds the task identity numbers for threads that are currently sleeping at the sleep-wait barrier. During initialization of the scheduler, all created tasks are placed on the thread queue. When a thread is needed, a task identity number is removed from the thread queue and the corresponding task mutex is released. When a thread has no work left, it puts its task identity number back on the thread queue before entering the sleep-wait barrier.

Scheduler Algorithm

This section describes some of the details of the Cassini HSS scheduler implementation. Particular attention is focused on measures taken to improve performance and to prevent deadlock and race conditions. The paragraphs that follow refer to the simplified pseudo code provided in Figures 5–8.

In the Cassini spacecraft, the hardware components synchronize with one another every one-eighth second, at a real-time interrupt (RTI). The HSS mimics this behavior through the use of a periodic barrier with the period set to one-eighth second. Other barriers are added as needed to provide additional synchronization points among models. The user can specify to the scheduler the length of emulated time or the number of RTIs that the scheduler should execute. In response, the scheduler sets up a special barrier, known as a breakpoint barrier, that has the desired stop time and stop flag set.

When the scheduler is initiated, a user-specified number of threads are created. The corresponding task identifiers are loaded into the thread queue. The task mutexes are locked and each thread then begins to execute the sleep-wait barrier code in the `threadloop()` function (Figure 5). Each thread will attempt to lock its task mutex again and will thus sleep on the call to `lock()`. This method of rendezvous incurs overhead costs on the order of microseconds on a Sun SPARCstation 10. As mentioned earlier, there are other methods to cause the scheduler threads to wait for work, such as spin-wait barriers and condition variables. However, we found that other system-provided methods of rendezvous result in greater overhead costs (often much greater).

As the result of a user "run" command, issued through a Tcl interface, the scheduler creates a breakpoint barrier and then loads all of the models (all active models must belong to the entry set and exit set of the breakpoint barrier) onto the model queue. Finally, the user interface thread calls `model_onto_thread()` (Figure 6). This function obtains a thread from the thread queue and then a model from the model queue. Next, the model is assigned to the thread and the task mutex is unlocked. That thread can now move beyond the sleep-wait barrier in the `threadloop()` function. The Tcl interface thread will continue to pair threads with models in this fashion until the thread queue is empty. At that time the Tcl interface thread will return and the scheduler threads will continue executing on their own.

In `threadloop()`, a thread that obtains its mutex proceeds to call `do_work()` (Figure 7). When the thread returns from `do_work()`, it will again try to obtain its mutex and will thus sleep-wait at that point. An outside event, such as a user command, must unlock the task mutex to allow the thread to continue.

In `do_work()`, a thread will run the model it was assigned in `model_onto_thread()`. One of three results can occur:

1. the model may not complete all of its assigned cycles (i.e., the model does not hit a barrier),

2. the model executes enough cycles to hit a barrier, or
3. the model hits a breakpoint (a model breakpoint set by a user for troubleshooting purposes).

In the first case, the model is left on the thread and will be run again. In the second case, `model_hit_barrier()` is called. In the third case, the scheduler sets a global suspended flag and the model state is set to "suspended".

In `model_hit_barrier()` (Figure 8), the model is added to the entered set for the barrier. If some models in the entry set have not yet been added to the entered set, the function returns. Otherwise, if all of the models in the entry set have arrived at the barrier, the barrier entered set is cleared, the events associated with that barrier are performed, and the next stop time for the barrier is established. If the barrier has its stopped flag set, no models are loaded onto the model queue. Otherwise, the models in the exit set for the barrier are all loaded onto the model queue and `model_onto_thread()` is called to start up the other threads.

Because the model queue and the thread queue are both accessed by multiple threads, each must be protected by a mutex. When accessing either queue, a thread first obtains the correct mutex, then pushes or pops an element from the queue, and then immediately releases the mutex. When using mutexes, the best code performance is achieved by keeping those mutexes locked for as short a time as possible. As will be shown in the next section, the fact that a thread will not own more than one mutex at a time when accessing the model and thread queues means that no deadlock can occur.

Each barrier object is also protected by a mutex unique to itself. The barrier mutex guarantees that only one thread can access the barrier object at a time. Thus, only one thread can find that its model was the last to enter a barrier. The other threads return from `model_hit_barrier()` knowing that the barrier has not yet been hit. Each of these threads attempts to obtain another model from the model queue. If successful, each thread will then execute that model and the process will continue. However, if the model queue is empty, the thread will put its identifier on the thread queue and return from `do_work()` only to perform a sleep-wait in `threadloop()`. The thread that had the model that was last to hit the barrier, the master thread, is thus responsible for loading the model queue and starting the other threads up by calling `model_onto_thread()`.

After doing so, the master thread will return to `do_work()`, where it will attempt to obtain a model to execute.

When the master thread encounters a barrier that has its breakpoint set, it will put its identifier on the thread queue and will signal the Tcl interface thread that the scheduler is finished. The master thread will then return from `do_work()` and end up at the sleep-wait barrier in `threadloop()`. Thus, all of the scheduler threads are back to the same state they were in prior to the launch of the scheduler.

Deadlocks and Race Conditions

Being provably free of deadlocks and race conditions was an important goal during the development of the HSS scheduler. This section describes how deadlocks and race conditions can be avoided. Informal proofs for the absence of deadlocks and race conditions in the scheduler are also presented.

Deadlocks

One important requirement for the scheduler is that it must be free of deadlocks. There are three classical approaches to deadlocks in the literature: deadlock prevention, deadlock avoidance, and deadlock detection and recovery [4]. Deadlock avoidance and deadlock detection and recovery schemes involve runtime overhead to detect and avoid unsafe states or to detect when deadlock has occurred. Deadlock prevention, on the other hand, can be implemented without runtime overhead. Thus, in the interest of maximizing performance, deadlock prevention was chosen as the approach to deadlock for the Cassini HSS scheduler.

Reference [5] provides the following four necessary conditions that must be in effect for deadlock to exist:

1. Processes claim exclusive control of the resources they require.
2. Processes hold resources already allocated to them while waiting for additional resources.
3. Resources cannot be removed from the processes holding them until the resources are used to completion.

4. A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain.

Because all four of the above conditions are necessary for deadlock to occur, the absence of any one or more of the conditions eliminates the possibility of deadlock. The first condition cannot be avoided when multiple processors cooperate to solve a problem and they must share one or more resources. Deadlock prevention seeks to eliminate the occurrence of one or more of conditions (2), (3), or (4). For the scheduler, we chose to concentrate on preventing condition (2) and condition (4).

The shared objects controlled by the scheduler are the model queue, the thread queue, the barrier objects, the task objects, and the model objects. As discussed above, the model and thread queues are each protected by unique mutexes. In all of the scheduler code, when the scheduler obtains either of these mutexes, it accesses the protected data and then releases the mutex without attempting to obtain any other mutexes. To help insure this, there are no function calls performed between the acquire and release of the queue mutexes.

In `model_hit_barrier()`, a thread must obtain the mutex for the barrier that the executed model hit, prior to adding the model to the entered set. The barrier mutex is released before the function returns. Only two functions called from within `model_hit_barrier()` attempt to obtain mutexes. These are `put_models_onto_queue()`, and `model_onto_thread()`. Both of these functions must obtain the model queue mutex. Thus condition (2) is met. However, because the model queue mutex is obtained and released without attempting to acquire other mutexes, there can be no circular chain of requests for mutexes, and condition (4) is denied.

Task objects are not actually protected by mutexes. There is a one-to-one correspondence between a task and a thread. Only the thread corresponding to a task object ever manipulates that task's data. Thus, no real sharing of data takes place. The task mutex is instead used to put the thread to sleep and to wake it up later (as presented in the Scheduler Architecture Section and the Scheduler Algorithm Section). When all of the scheduler threads go to sleep, the scheduler can in fact make no progress without outside intervention. However, a user thread can unlock the task mutexes to start scheduler operation. Outside threads never lock task mutexes, with the exception of scheduler initiation, where each task mutex is locked only when the corresponding thread is created.

Model objects are not protected by mutexes even though they are shared objects. Thus, deadlock is not possible with regard to contention for model objects among threads. Instead, race conditions are the concern with models and are discussed in the next subsection.

Race Conditions

From the discussion above, it should be clear that with the exception of model objects, all shared data are protected by mutexes. In the case of models, the barrier mechanism combined with the existence and use of the model queue guarantees that no two threads will try to access a model object at the same time. The restrictions on barriers presented in the Scheduler Architecture Section insure that only one barrier can stand in the way of a model at any one time. Further, a model is placed on the model queue only when a barrier has been reached and only one thread will ever "reach" a barrier, because only one thread can own the last model to enter a barrier. This implementation of barriers combined with the use of a model queue that is protected by a mutex, guarantees that a model can only be assigned to one thread at any point in time. Thus, there is never any contention between threads for a particular model object.

Recall that in the Scheduler Architecture Section, it was shown that race conditions can occur even with the proper use of mutexes. This is possible when two threads write different values to a shared variable and the ultimate result depends on the order in which the threads obtain access to that shared variable. In the case of the scheduler, we can show that this will never occur.

Consider the model queue. Model identities are put onto the model queue either by a user Tcl thread or by a scheduler thread when a barrier has been reached. In the first case, the scheduler does not accept run commands from a user thread until the scheduler has finished with the previous run command (a simple handshake scheme assures this). In the latter case, recall that only one thread can have the model that is last to reach a barrier. The other threads arrive earlier and, upon finding that the model queue is empty, put themselves to sleep. Thus, regardless of which thread loads the model queue, the model queue always ends up with the same models on it (those models in the exit set of the last barrier encountered).

In regard to the thread queue, the one-to-one correspondence between a thread and a task object insures that a thread can only write its corresponding task identifier to the thread queue. Further, it does not overwrite other task identifiers already on the queue. Because every thread is created equal in the eyes of the scheduler, the order that task identifiers appear on the thread queue makes no difference.

Once a barrier object is created, only two data items associated with the object are updated. These are the time at which the barrier is to occur and the entered set for the barrier. Regardless of which thread reaches the barrier, the time for the next occurrence of the barrier will be updated in exactly the same way. As for the entered set, it is represented as a bit vector, so each model that reaches a barrier results in changing the state of one unique bit in the vector. Thus, no race condition can occur.

To summarize this section, we have shown that the HSS scheduler is free from deadlocks and race conditions. While the approach used depends on the particular algorithm and data structures involved, the techniques used here can be readily applied to other software design efforts.

Scheduling Policy

To empirically explore the impact of various scheduling policies for the scheduler, we developed a discrete-event simulation model [6]. The model was developed to mimic the operation of the scheduler. The key performance variable we measured was the total estimated time for the HSS to execute all thirty-one models for 100 RTIs.

Once the simulation model was verified and validated, we developed fifty-six variations of it. Each differed by the number of threads available to process models (1 thread, 2 threads, 3 threads, 4 threads, 8 threads, 12 threads, and 16 threads) and the queue priority rule for the model queue. By running forty replications of each simulation model, we performed a total of 2240 simulation runs. For these simulation runs, it was assumed that every thread was bound to a unique processor.

Table 1 summarizes the experimental design used for our study. The data in the table is the estimated average time for the HSS to run all of the thirty-one hardware models for the varying scenarios. For example, our simulation model estimates that with two threads and Rule 4, the HSS will have an execution time of 38.8 seconds. That is, it is estimated that it will take the HSS 38.8 seconds to simulate 12.5 seconds (100 RTI x 1/8 second per RTI) of the spacecraft data control system. These results indicate that the HSS is slower than real-time. However, since the time that these execution-time measurements were made, significant improvements have been made to the code of the hardware models. Currently, the HSS is faster than real time.

Table 1: Average runtime estimates for the HSS to execute all thirty-one emulators for 100 RTIs. The results are indexed by the queue priority rule for the model queue and the number of threads to process models.

Rule	Description	Number of Threads						
		1	2	3	4	8	12	16
1	First-in, first-out - priority given to model arriving at model queue *first*	76.4	42.1	38.2	23.5	23.0	22.9	22.9
2	Last-in, first-out - priority given to model arriving at model queue *last*	76.4	39.4	33.3	22.9	22.9	22.9	22.9
3	Priority given to model that has been in process for the *least* time	76.5	42.1	38.2	23.5	23.0	22.9	22.9
4	Priority given to model that has been in process for the *most* time	76.4	38.8	32.8	22.9	22.9	22.9	22.9
5	Priority given to model having waited *least* total time in model queue	76.4	41.3	37.9	23.7	23.0	22.9	22.9
6	Priority given to model having waited *most* total time in model queue	76.4	41.3	35.1	23.3	23.0	22.9	22.9
7	Priority given to model having waited *least* total time in thread queue	76.4	40.4	33.2	22.9	22.9	22.9	22.9
8	Priority given to model having waited *most* total time in thread queue	76.4	42.1	38.2	23.5	23.0	22.9	22.9

An analysis of variance was performed to test whether there is a statistical difference among the mean times to completion. For one thread, there is no difference. For two and more threads, the best performing rule is Rule 4. A close second is Rule 7, followed by Rule 5. The results also indicate that as the number of threads increase, the run time decreases. Though there is a limit on the improvement.

Our study shows that after four threads, adding more threads has minimal impact. This is a direct result of the run times for the individual models. In the Cassini HSS, four of the models account for over 98% of the total model execution time. Therefore, little benefit can be derived from utilizing more than four processors.

The fact that Rule 4 had the best performance indicates that models that required the longest execution times in the past are likely to require the longest execution times in the future. This rule is relatively easy to implement. Cumulative execution-time data is maintained for each model. Models are placed onto the model queue in order from highest cumulative execution time to lowest.

Summary and Conclusions

This paper described the design and development of a general-purpose multithreaded scheduler that is free of deadlocks and race conditions. The techniques for avoiding deadlock and races presented here can be applied to other multithreaded program design efforts.

The multithreaded HSS complete with the scheduler described in this paper has been in use at the NASA Jet Propulsion Laboratory for more than one year. It has proven to be reliable and free of deadlock and race conditions. Currently, the HSS executes flight code at approximately twice the real-time rate. On-going work to reduce the processor model execution time through optimization of the code and through the application of block optimization of the flight software is expected to yield further execution time improvements. The availability of ever faster computing hardware will also improve the HSS performance as compared to real-time.

The Cassini HSS, with its embedded scheduler, is a valuable software tool that will provide valuable support for the Cassini mission to Saturn over the years to come.

Acknowledgments: The authors gratefully acknowledge Jeff Nesheiwat and the entire HSS team at NASA. A preliminary version of portions of this research was presented at the *16th Digital Avionics Systems Conference* in October 1997.

REFERENCES

[1] S. Kleiman, D. Shah, and B. Smaalders, *Programming with Threads*, SunSoft Press, Mountain View, CA, 1996.

[2] B. Lewis and D. J. Berg, *Threads Primer: A Guide to Multithreaded Programming*, SunSoft Press, Mountain View, CA, 1996.

[3] J. K. Osterhout, *An Introduction to Tcl and Tk*, Addison-Wesley, Reading, MA, 1993.

[4] H. M. Deitel, *Operating Systems*, 2nd edition, Addison-Wesley, Reading, MA, 1990.

[5] E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani, “System deadlocks,” *Computing Surveys*, Vol. 3, No. 2, Jun. 1971, pp. 67-78.

[6] P. A. Savory and G. Saghi, “Simulating Queue Scheduling Policies for a Spacecraft Simulator,” *Interfaces*, Vol. 27, No. 5, Spet.-Oct. 1997.

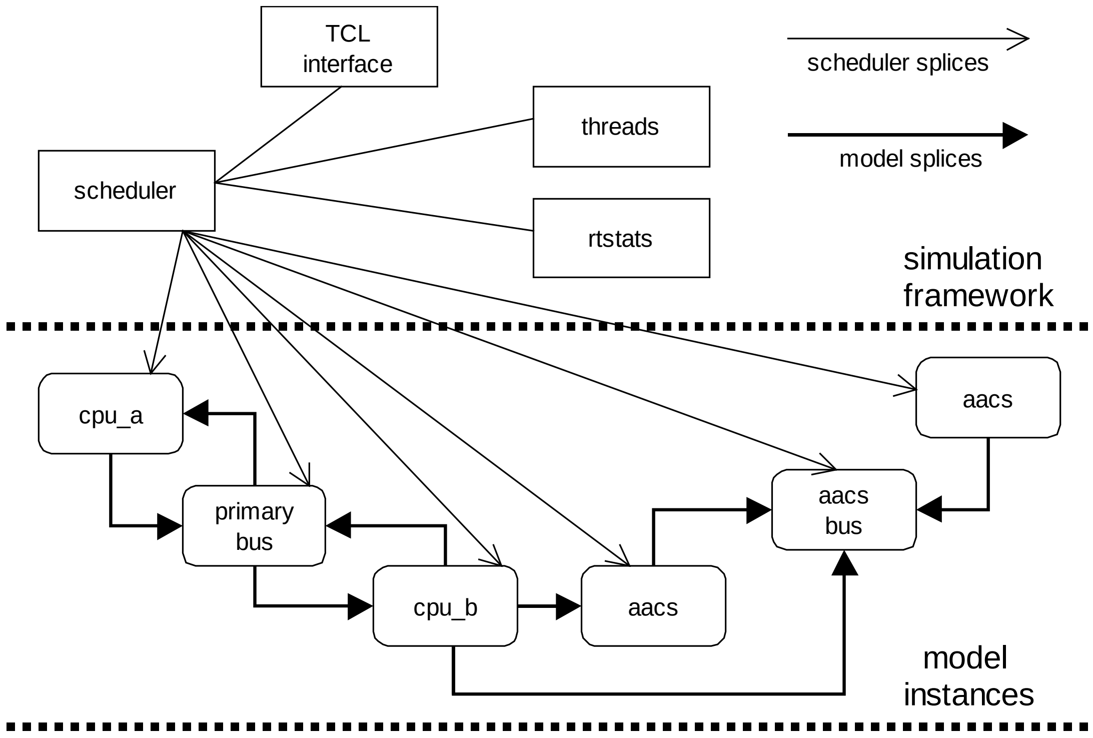

Figure 1: Simplified representation of HSS overall architecture.

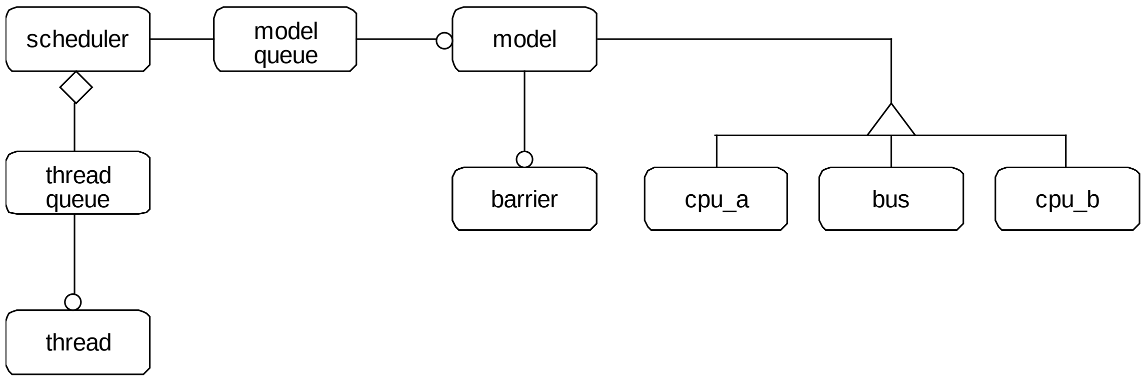

Figure 2: Simplified representation of HSS overall architecture.

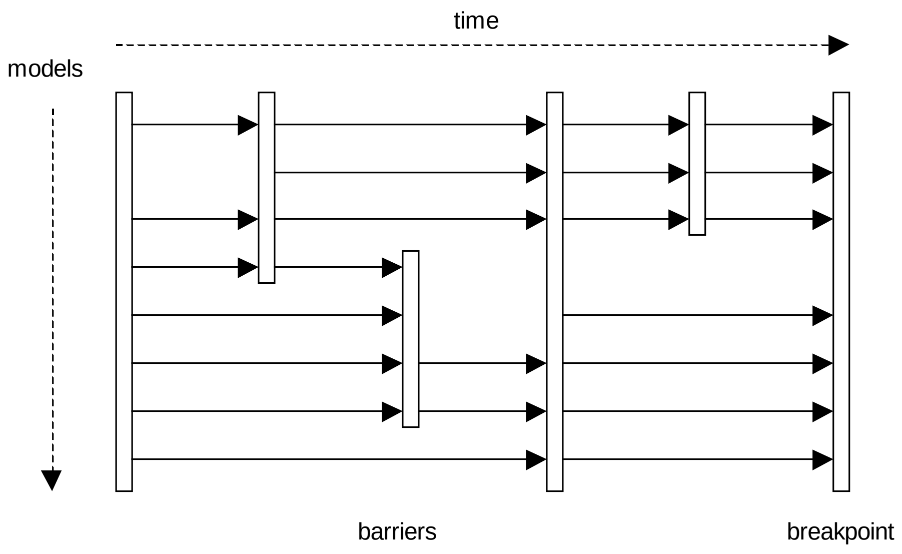

Figure 3: Use of barriers to synchronize model execution.

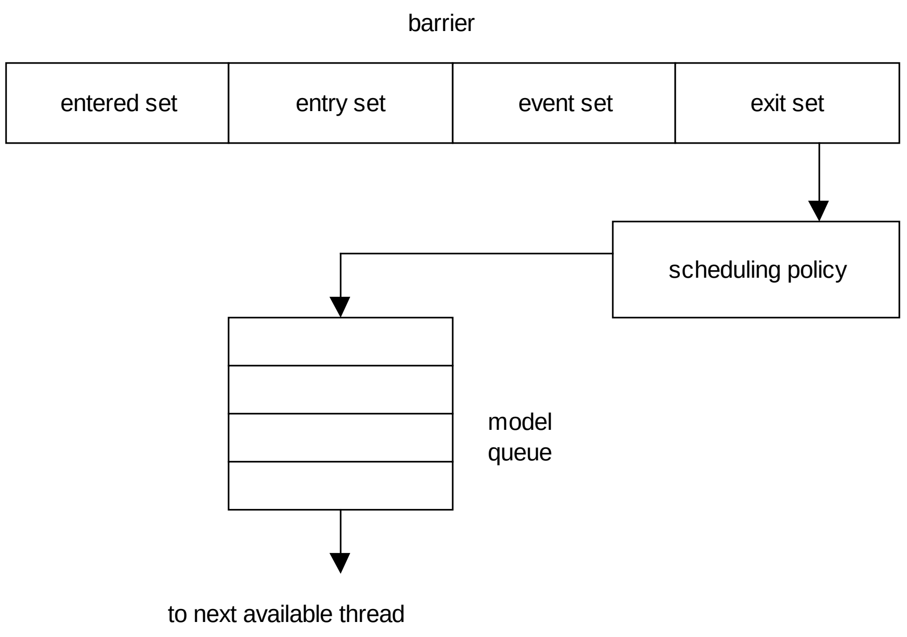

Figure 4: The model queue and its usage.

```
threadloop(task){
  done = 0;
  while(!done){
    task.lock();
    if (task.state == dying)
      done = 1;
    else
      do_work(task);
  };
};
```

Figure 5: Simplified pseudo code for the `threadloop()` function.

```
model_onto_thread(){
  while ((task = get_thread_from_queue()) != NONE){
    while ((model = get_model_from_queue()) != NONE){
      task.model = model;
      task.unlock();
      break;
    };
  };
};
```

Figure 6: Simplified pseudo code for the `model_onto_thread()` function.

```
do_work(task){
  breakpoint = NO;
  while((suspended == NO) & (breakpoint == NO)){
    rv = run_model(task.model);
    model.state = waiting;
    switch(rv){
      case MODEL_NOT_FINISHED:
        model.state = running;
        break;
      case MODEL_REACHED_BARRIER:
        model.state = at_barrier;
        breakpoint = model_hit_barrier(model);
        break;
      case SCHEDULER_SUSPENDED:
        suspended = YES;
        model.state = suspended;
        break;
      case default:
        ERROR();
        break;
    };
    if (model.state != running){
      if ((breakpoint == NO) & (suspended == NO)) {
        model = get_model();
        model.state = running;
        if (model == NONE)
          break;        // The model queue is empty.
    };
  };
  put_task_onto_thread_queue();
};
```

Figure 7: Simplified pseudo code for the `do_work()` function.

```
model_hit_barrier(model){
  breakpoint = NO;
  barrier = model.barrier;
  barrier.insert_into_entered_set(model);
  if (barrier.entered_set == barrier.entry_set){
    breakpoint = barrier.breakpoint;
    barrier.clear_entered_set();
    barrier.do_events(barrier);
    barrier.set_next_stop_time(barrier);
    for (models in barrier.entry_set){
      model.barrier = find_next_barrier_for_model(model);
      model.stoptime = model.barrier->stoptime;
    };
    if (breakpoint == YES)
      break;
    else {
      put_models_onto_queue(barrier.exit_set);
      model_onto_thread();
    };
  };
  return breakpoint;
};
```

Figure 8: Simplified pseudo code for the `model_hit_barrier()` function.