

The Use of Multiple Threads in An Object-Oriented Real-Time Simulation

David W. Geyer

Unisys Corporation
NASA Langley Research Center
MS 169
Hampton, Virginia 23681

Abstract

As real-time simulations become more complex, a single processor is no longer sufficient to perform all of the necessary computations. It becomes desirable to execute independent sections of a real-time simulation on different processors. The traditional approach to utilize multiple processors on a symmetric multiprocessor machine involves writing a program composed of one process per processor that interact through shared memory. A program that uses threads can utilize multiple processors without the overhead of full processes or the use of a cumbersome shared memory mechanism. This is due to the fact that threads *implicitly* share a common address space. A multithreaded real-time program is organized as a main thread controlling zero or more auxiliary real-time threads. Each processor to be used for real-time computations is assigned a single auxiliary real-time thread. The main thread monitors the real-time clock and informs the auxiliary threads when the clock ticks. An auxiliary thread can also be used to monitor and control the simulation via a graphical user interface (GUI). These object-oriented designs were implemented in C⁺⁺ for the NASA Langley Standard Real-Time Simulation (LaSRS⁺⁺) Application Framework.

Introduction

Real-time simulations are continuous simulations that have a strict and direct correspondence between simulation time and real-world time: one unit of time in the simulation is equal to one unit of time in the real world. This type of simulation is usually used to represent or study a real dynamic phenomenon as it occurs. Real-time control of dynamic processes requires a human-in-the-loop or simulated controller (e.g. a pilot flying a flight simulator). Real-time simulations often involve modeling the evolution of continuous processes over time. This is accomplished by dividing time into small fixed intervals and solving differential equations at each successive interval of time. This type of computation is very time critical. It must be possible to perform all of the necessary computations in the allotted amount of time.

As real-time simulations become more complex, a single processor is no longer sufficient to perform all of the necessary computations. It becomes desirable to execute independent sections of the simulation model on different processors. The traditional approach to utilizing multiple processors on a symmetric multiprocessor machine involves writing a program composed of one *process* per processor that interact

through shared memory. This involves using an explicit shared memory mechanism that must be managed by the program. This approach also presents a problem when using C⁺⁺ [7] virtual functions: virtual functions for a given object are only available in the process that created the object (i.e., the virtual function table is bound to the address space of the creator process).

A program that uses *threads* can utilize multiple processors without the overhead of full processes or the use of a cumbersome shared memory mechanism. This is due to the fact that threads *implicitly* share a common address space. Because the virtual function tables are bound to the shared address space, C⁺⁺ virtual functions for a given object are available in *any* thread.

Real-Time Simulation

In order to perform real-time simulations, it is necessary to synchronize the simulation execution with real world time. This is normally accomplished by system level software that interacts with an external clock. The external clock must be accurate and have a precision on the order of microseconds or even nanoseconds. The system level software divides computer operation into fixed, equal intervals of time called *frames*. The simulation code waits for a frame to start, performs its computations, and informs the system level code when it is done. The simulation time is incremented by one frame time and the cycle repeats. As long as the simulation code takes less time to execute than a single frame, the simulation will continue to operate in real-time.



Figure 1: real-time frame

Figure 1 illustrates the concept of a frame. The synchronous input period of the frame is a variable length of time where information from the outside world is deposited into the memory space of the application code. Similarly, during the synchronous output period, application information is transferred to the outside world. These periods are considered synchronous since their execution is dependent on the system clock for start (input) and stop (output) times. The actual code that simulates the dynamic system runs during the application compute time.

If, for some reason, the dynamic system code takes longer to execute than the allotted application compute time, the simulation will have a deterministic behavior based on the nature of the real-time constrain. A *hard* real-time system will halt with a critical error if the real-time constraint is violated. A *soft* real-time system will allow computations longer than a frame and take some appropriate action without halting. Flight simulations that involve a human-in-the-loop control mechanism are hard real-time systems.

Process Model

The process model is an abstraction used by traditional operating systems to handle program execution. Each process is composed of an addressable memory space, a single sequence of machine instructions to

execute, and other resources maintained by the operating system. Each process has its own, private address space, isolated from all other processes in the system. The process model was specifically designed in this way to allow programmers to write applications in isolation, unconcerned with the memory usage of other execution programs.

In a multiprogramming environment, several processes can be running on the system at the same time. Butenhof [1] defines concurrency as “things that appear to happen at the same time, but which may occur serially.” On a uniprocessor system, processes only appear to execute concurrently. In reality, the operating system is providing each executing process a small amount of execution time on the CPU. Once this time expires, the operating system switches to another process to give it some time. This switching occurs among all running processes to allow them all to progress in execution.

A multiprocessor system not only provides concurrency, it can also provide true parallelism. Parallelism occurs when multiple processes are executing simultaneously on different processors. A parallel program can actually perform multiple tasks at one, while a concurrent program can only provide the illusion of multiple task execution.

In order to achieve true parallelism on a multiprocessor system with the process model, a program must be designed as a group of distinct processes that cooperate and share resources through some form of interprocess communication, such as shared memory. Most modern operating systems provide mechanisms to map, or overlay, a portion of a given process’s memory space with the memory space

from another process. Both processes view the shared memory space as part of their own addressable memory space. As a result, any changes to the contents of the shared memory space by one process are visible to the other process.

Real-time vs. Time-sharing Processes

In a general purpose (a.k.a. non-real-time) operating system, the time-sharing process scheduler tries to ensure that all runnable processes will eventually get a chance to run. An I/O bound process will occasionally block and allow some other process to assume control of the CPU. A compute-bound process, on the other hand, will run until it is preempted by the scheduler. This is usually accomplished by periodically interrupting the running process on each CPU. Each process is usually given a guaranteed amount of time that it can run on a CPU without being preempted. This period of time is called a time slice. At the end of each time slice, the scheduler will choose another process to run on the give CPU. The choice concerning which process to run next is made based upon a priority value that is associated with each process. As a process runs, its priority diminishes or degrades. When the priority of a process degrades to some minimum value the priority will be resorted to a maximum value.

In order to meet the strict timing requirements for real-time simulation, the traditional time-sharing scheduler must be bypassed. The operating system kernel must provide a mechanism to allow normal processes to be scheduled in real-time mode. A real-time simulation will require one or more actual processors to run in real-time mode.

Running a processor in real-time mode consists of:

- Remove the processor from workload consideration by the normal, time-sharing process scheduler
- The ability to assign specific processes to specific real-time processors
- Prevent the processor from servicing any interrupts that can be handled by other non-real-time processors
- Prevent the preemptive scheduling on the processor; i.e. a process running on a real-time processor gives up control of the processor only when it chooses to relinquish control

Thread Model

The term thread refers to a single point of control that executes a sequence of instructions. The process model can be thought of as a specialization of the thread model where there is only a single thread. The thread model expands on the process model by providing the same abstraction as the process model but allowing multiple execution sequences. All of the process resources are shared by all of the threads.

The implicit sharing of process resources in a multithreaded program has benefits and drawbacks. The immediate benefit is the elimination of the need to use an explicit shared memory mechanism to communicate between threads. Another benefit is that a multithreaded program is a

single program, whereas a multiprocess program is several separate programs that must work together. A single program is simpler to design and maintain. Threads consume less system resources than processes, making threads faster and easier to create than processes.

Unlike the process model, however, the thread model does not provide any explicit memory protection between threads. The developer must be conscious of the interaction of different threads on the common address space. If more than one thread can read and/or write a given data object, then some form of synchronization must be used to avoid data corruption. It is also possible for a given thread to corrupt the stack or heap memory used by another thread.

User Threads vs. Kernel Threads

There are two types of thread models available: user threads and kernel threads. User threads allow the developer to use threads as an organizational tool within programs, but do not allow for true parallelism. These types of threads are only visible within the given process; i.e., they are not visible to the operating system. As a result, the operating system cannot schedule these threads to run in parallel on multiple processors. The application itself has to schedule and manage the threads. As long as a user threads application uses non-blocking system calls, such as application can maintain several independently executing tasks. A blocking system call will suspend the calling process until the system call completes.

Kernel threads, on the other hand, are recognized and managed by the operating system. They can be run in parallel on a multiprocessor machine. Kernel thread applications can also use blocking systems calls, since the operating system will replace any blocked threads with a runnable thread. A kernel thread is usually implemented as two-part object. The user-mode portion is the application interface and data to the thread. The second part is the kernel-mode portion that exists inside the operating system kernel. The kernel-mode part is used by the operating system to schedule the thread as an independent entity. The kernel-mode portion also allows the operating system to bind the thread to one or more specified processors.

Object-Oriented Approach to Threads

A number of different types of objects are necessary for developing multithreaded real-time simulations. These objects can be grouped into two categories: objects that represent the threads themselves and objects used to synchronize the threads. These objects allow the developer to encapsulate all of the low-level details related to using threads.

The C⁺⁺ language does not provide any direct support for multithreaded programming. The approach used by LaSRS⁺⁺ [3] is to use C⁺⁺ wrapper classes around existing C interface libraries. LaSRS⁺⁺ uses classes that encapsulate threads, mutexes and barriers. These classes hide all the low-level details for the actual thread package in use. Currently, LaSRS⁺⁺

only supplies wrappers for SGI Irix share group processes [8]. Preliminary wrappers are in development for the POSIX threads package [1].

Thread Objects

A single wrapper class is used to represent a thread. This class contains all the low-level data and function calls necessary to create, identify and destroy the thread. Since the mentioned thread libraries use a C language interface, the user must provide a C function that the thread will start execution after creation is complete. The developer must provide the wrapper object with function as a constructor argument. Another constructor argument provides the option of allowing the new thread to start executing immediately, or of manually starting the thread at a later time.

Mutexes (or Locks)

The term mutex (short for mutual exclusion), refers to a synchronization primitive used to provide exclusive access to some shared resource. When a given thread wants exclusive access to a given shared resource, the thread tries to lock the mutex. If no other threads currently have exclusive access, the thread acquires the mutex and has exclusive access to the resource. If another thread already has exclusive access to the resource, the original thread will block (suspend) until the resource is released when the controlling thread unlocks the mutex. The mutex is said

to be owned by the thread that has successfully locked the mutex.

Barriers

A barrier is a synchronization primitive used to bring a number of threads together during program execution. A barrier is initialized with the number of threads that it is required to synchronize, call this number n . Then, the barrier will suspend all threads that enter it until n threads are suspended in the barrier. Once the barrier contains N threads, all the suspended threads are allowed to continue.

Threads in a Real-Time Simulation

Kernel threads are required in order to build a multithreaded, real-time simulation. In addition the operating system must provide the ability to bind specific threads to specific processors. A real-time thread is defined to be a kernel thread that has been bound to a processor running in real-time mode. Each processor to be used for real-time computations is assigned a single real-time thread. Each real-time thread assumes complete control of its assigned processor, until the thread decides to relinquish control.

A multithreaded, real-time program is organized as a main real-time thread controlling zero or more auxiliary real-time threads. This is often referred to as the “Work-Crew Model” [1]. The main thread monitors the real-time clock and informs the auxiliary threads when the clock ticks: i.e. frame start. The

auxiliary threads wait for frame start notification from the main thread, perform all computations for the given frame, and notify the main thread when computations are complete. Note that both the main and all auxiliary threads do not make any blocking system calls. An undeterministic duration system could cause a hard real-time simulation to miss a frame deadline.

Figure 2 illustrates the use of kernel threads in a real-time simulation. This approach is known as the one-to-one method [1]; i.e., one kernel thread to one processor. This approach is common for compute-bound threads, where blocking is not an issue.

Figure 3 shows the LaSRS++ classes that correspond to the main and auxiliary threads. A `MainSimulationThread` object contains a variable sized vector of `AuxiliarySimulationThread` objects. Every frame, the `MainSimulationThread` object will coordinate with the attached set of `AuxiliarySimulationThread` objects. Figure 4 shows the runtime interaction between a `MainSimulationThread` object and an `AuxiliarySimulationThread` object.

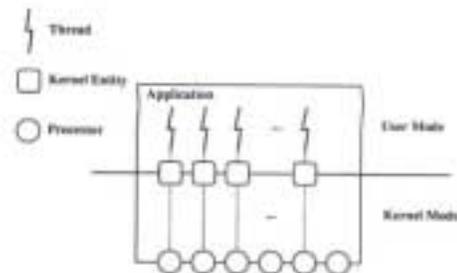


Figure 2: Real-Time Thread Allocation

The easiest way to partition computations between threads is to compute disjoint sections of math models on different threads. This eliminates the need for using time-consuming thread synchronization primitive during the compute phase

of the frame. If thread synchronization during the compute phase is unavoidable, at least try to minimize the usage.

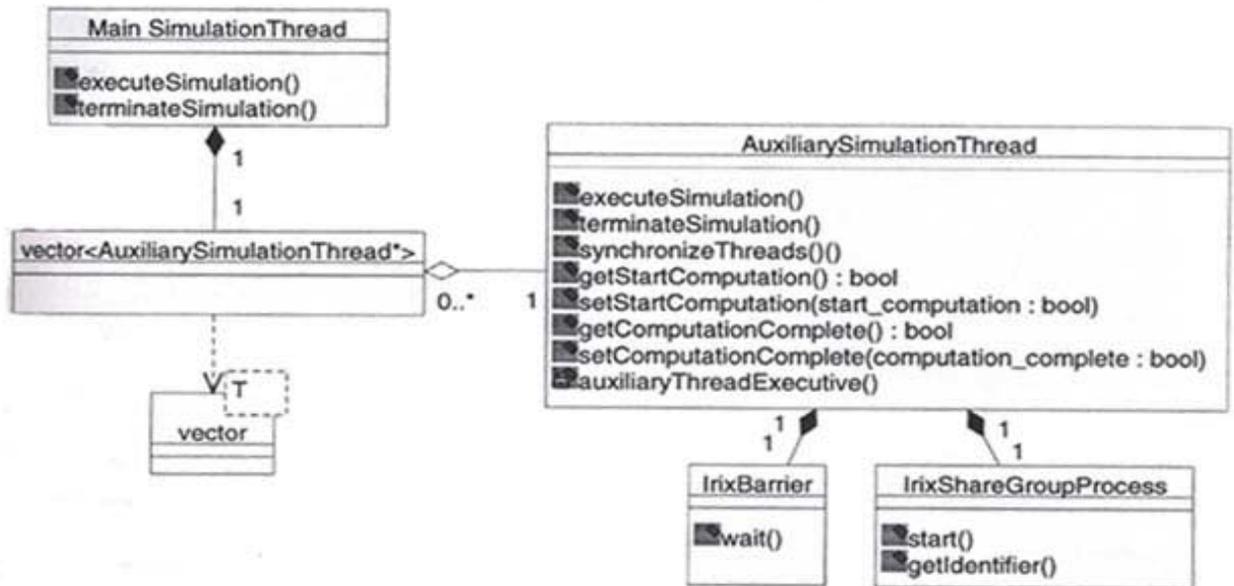


Figure 3: Computation Threads Class Diagram

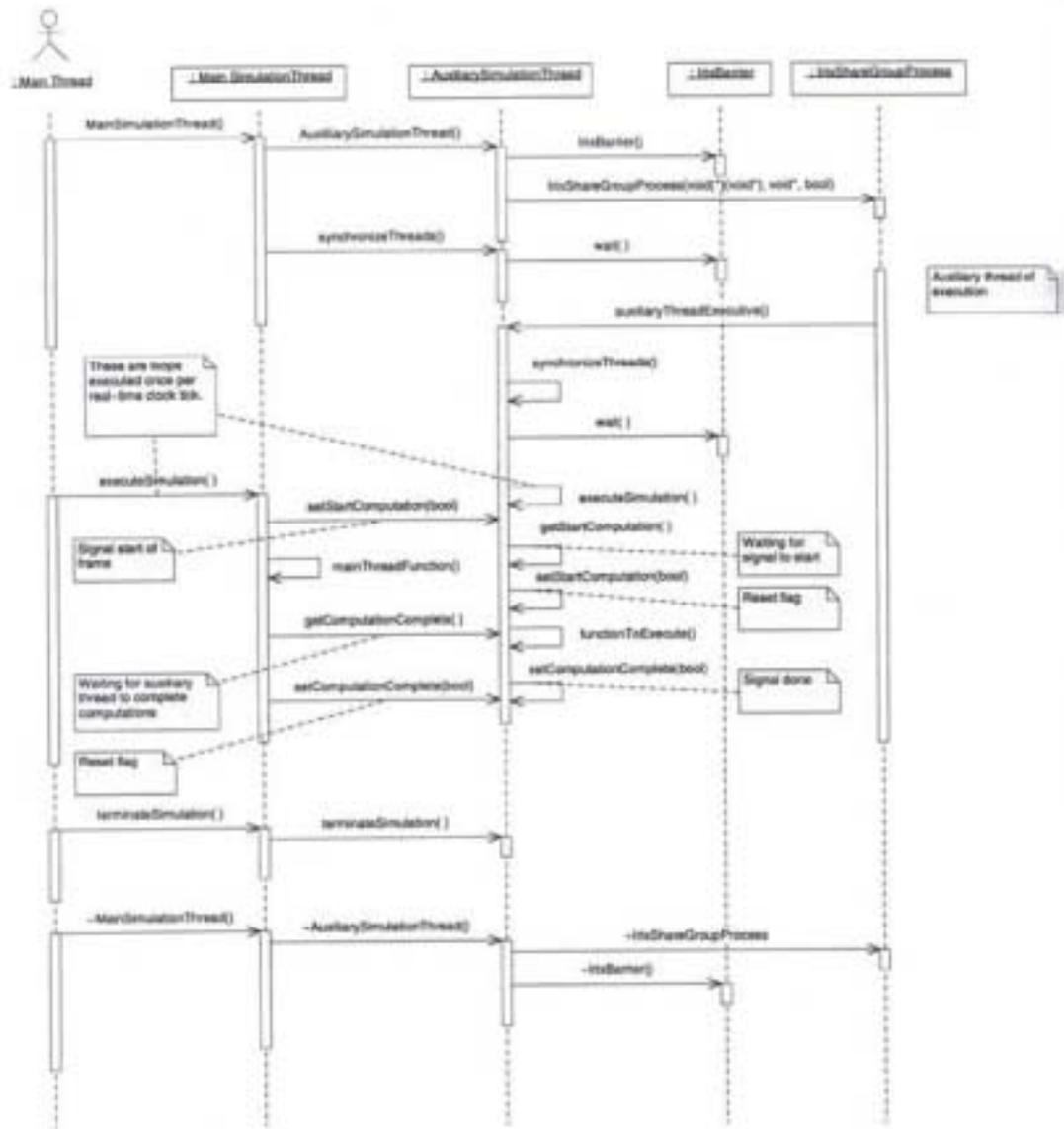


Figure 4: Computation Threads Sequence Diagram

Separate Thread for Graphical User Interface

A separate thread can be used to execute a GUI capable of monitoring and controlling a simulation. The GUI is usually considered non-real-time, so the GUI thread is assigned to run on a non-real-time processor.

Due to the fact that many GUI toolkits are not thread-safe, the GUI is designed such that only a single thread actually executes function calls to the GUI toolkit.

LaSRS++ provides an interface class that simplifies the creation of

separate thread GUIs. Class `SeparateThreadGui` is an abstract class that encapsulates GUI thread creation and synchronization. This class specifies the behavior that all separate thread GUIs must exhibit. All GUI startup synchronization is also handled within the interface class constructor. Since this class specifies the execution sequence, developers can focus on GUI design and implementation issues, without being concerned with thread-specific issues.

A new GUI is created by deriving a new class from `SeparateThreadGui` and providing definitions for the following pure virtual functions:

- `installSignalHandlers()`
- `createGuiObjects()`
- `executeEventLoop()`
- `destroyGuiObjects()`

These member functions contain GUI toolkit function calls that

create, execute and destroy the actual GUI. The actual implementation can be performed without any knowledge of threads. This design allows the GUI developer to focus on GUI design and implementation issues, not thread issues.

`SeparateThreadGui` class contains all the necessary thread and synchronization objects (see Figure 5). The constructor creates a separate thread that executes private member function `guiThreadExecutive()`; this is the predefined execution sequence. In this design the main thread creates an instance of a concrete class derived from `SeparateThreadGui` (see Figure 6). Startup synchronization with the new thread usually occurs in the constructor of the new derived object.

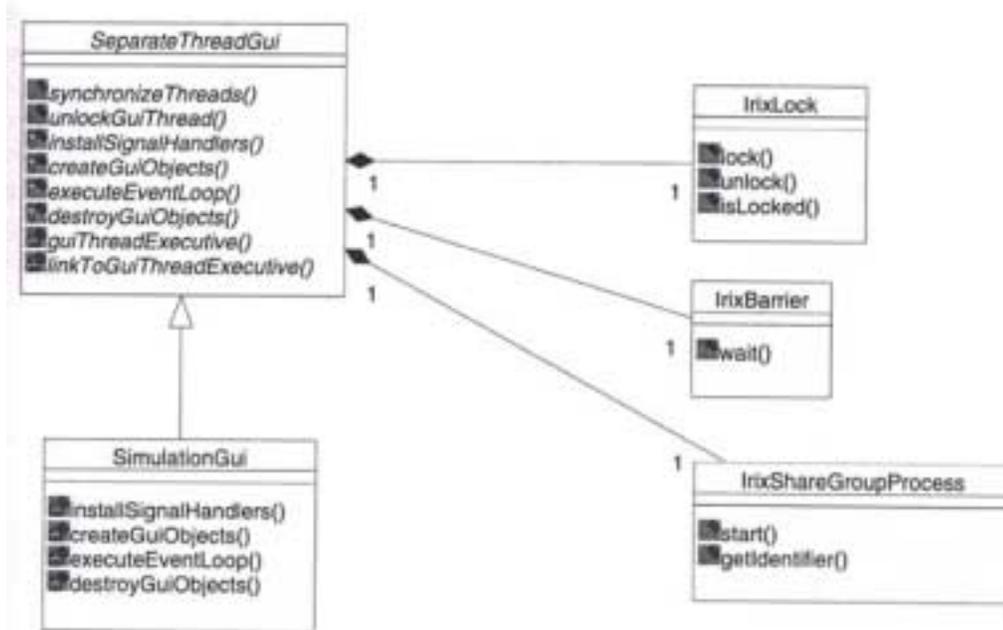


Figure 5: Separate Thread GUI Class Diagram

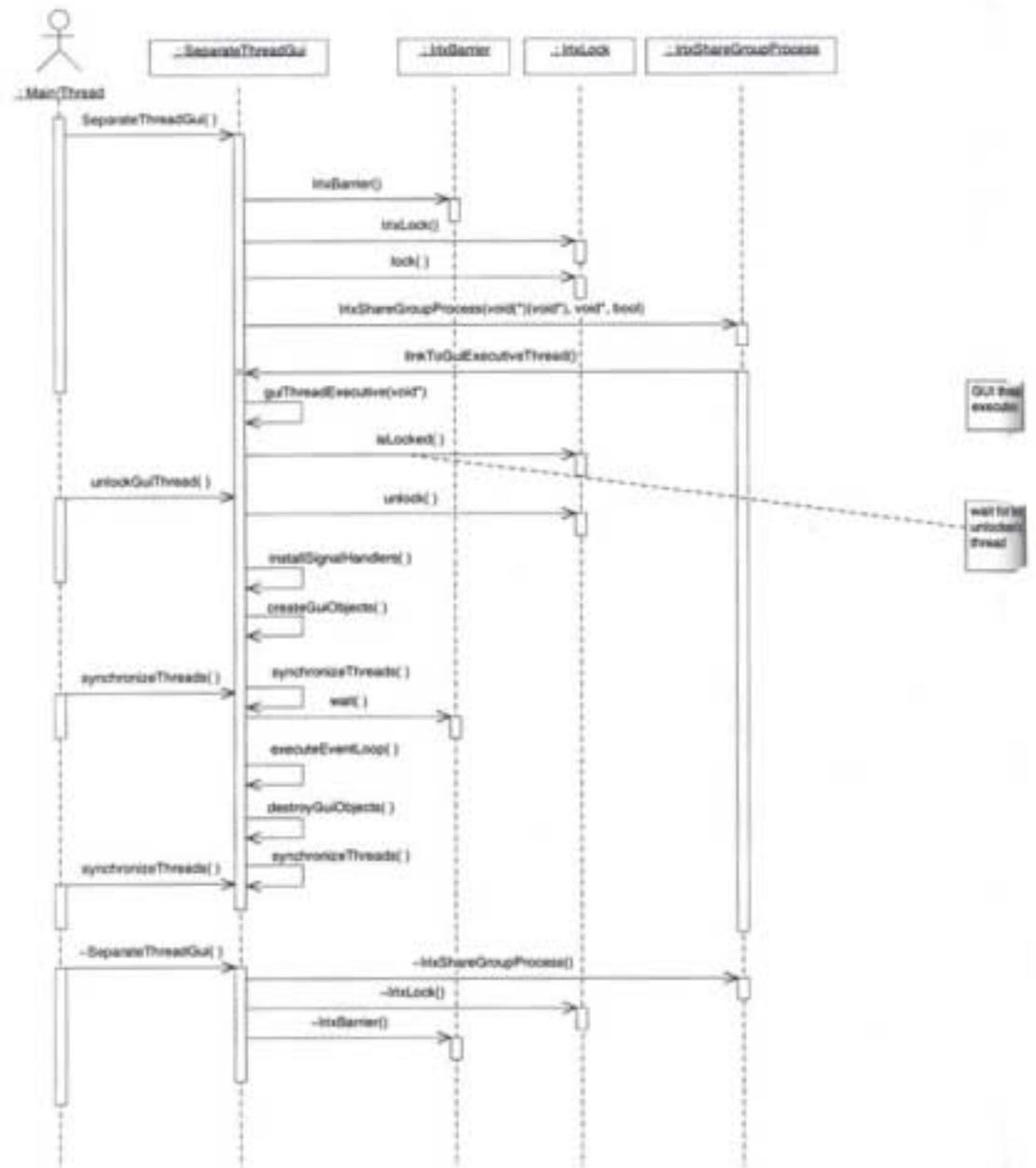


Figure 6: Separate Thread GUI Sequence Diagram

Developers and users want to monitor and modify simulation variables. Monitoring simulation variables does not require any thread synchronization to avoid data corruption. There are several techniques to minimize synchronization between a separate thread GUI thread to only change variables that are read-only to the simulation threads. Another method involves having the GUI modify values in a temporary buffer. The real-time threads can then use the updated buffer values to modify simulation variables when there is no risk of data corruption.

Conclusions

The NASA Langley Standard Real-Time Simulation Framework in C++ (LaSRS++) provides support for multithread programming. The main motivation for using multiple threads is to make parallel, real-time simulation programming easier for the developer. The presented object-oriented designs provide an environment where developers can focus on their specific programming task instead of getting bogged down in thread details. Multithread programs are developed as single entities instead of several cooperating programs. Two of the possible uses of multiple threads include: increasing the number of computations possible during a frame and executing graphical user interfaces for

program control and data monitoring.

Bibliography

- (1) David R. Butenhof. Programming with POSIX Threads. Addison-Wesley Publishing Company Reading, Massachusetts, 1997
- (2) Michael Madden, et al. Constructing a multiple vehicle, multiple-cpu using object-oriented C++ Number AIAA-98-4530 in AIAA Modeling & Simulation Technologies Conference, August 1998
- (3) Richard A. Leslie, et al. LaSRS++ an object oriented framework for real-time simulation of aircraft. Number AIAA-98-4529 in AIAA Modeling & Simulation Technologies Conference, August 1998.
- (4) Robert Martin. Designing Object-Oriented C++ Applications Using the Booch Method. Prentice-Hall, Inc., 1995 ISBN 0-13-203837-4.
- (5) Scott Meyers, Effective C++. Addison-Wesley Publishing Company, 1992. ISBN 0-201-92488-9.
- (6) Pierre-Allain Muller. Instant UML. Wrox Press LTD., 1997/ ISBN 1-86100-87-1.

- (7) Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley Publishing Company, third edition, 1997. ISBN 0-201-88954-4.
- (8) Topics in irix programming. Technical Report 007-2478-006, Silicon Graphics Incorporated, 1998.
- (9) Uresh Vahalia. UNIX Internals. Prentice Hall, Upper Saddle River, New Jersey, 1996.