A Users Guide to AWESIME:

An Object Oriented Parallel Programming and Simujlation System

Dirk Grunwald

CU-CS-552-91 November 1991

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Users Guide to AWESIME: An Object Oriented Parallel Programming and Simulation System

Dirk Grunwald
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

CU-CS-552-91 November 1991

University of Colorado at Boulder

Technical Report CU-CS-552-91
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

A Users Guide to AWESIME: An Object Oriented Parallel Programming and Simulation System

Dirk Grunwald
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

November 1991

Abstract

AWESIME (*A Widely Extensible SIMulation Environment*) is an object-oriented library for parallel programming and process-oriented simulation on computers with a shared address space. AWESIME is written in the C++ language.

AWESIME is similar to other toolkits, such as Presto [Bershad et al. 88], but provides additional classes that simplify building process oriented simulations. Much of the additional functionality was modeled after the CSIM simulation package [Schwetman 86].

The object library of AWESIME is structured to simplify porting the environment to new architectures. AWESIME can currently be configured for the MIPS R2000/R3000, Sun SPARC, Motorola 680x0, Intel i386, National Semiconductor NS32K family and Motorola 88K. The library is designed to be easy to extend.

This manual is an introduction to using AWESIME. It is self-contained, but incomplete; an acompanying AWESIME reference manual provides more detail than available here.

Contents

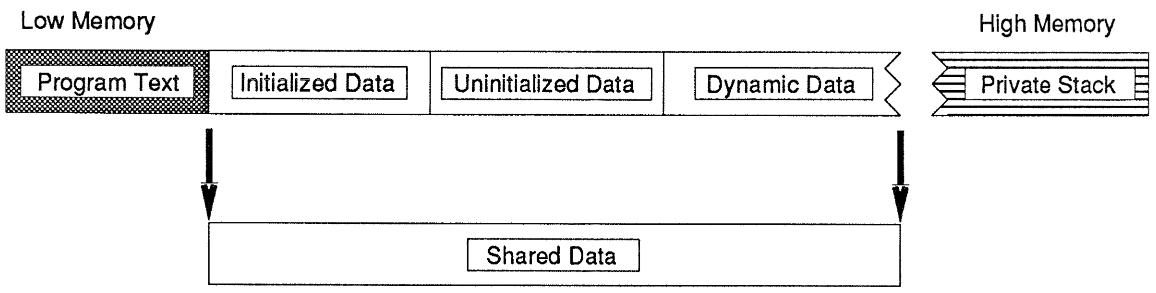

Figure 1: Memory Architecture Assumed by AWESIME

1 Introduction

Programs for parallel MIMD architectures with a shared address space can be written using special parallel language mechanisms, functions supplied by the native operating system or subroutine libraries. Subroutine libraries are commonly chosen because they can offer better performance than native operating systems and are more portable than language mechanisms or a particular operating system. Some subroutine libraries are tightly integrated to the architecture, and the programmer can only control a limited number of execution streams. Others attempt to abstract the architecture, allowing the programmer to manage *threads* [Bershad et al. 88, Weiser et al. 89], or independent execution streams sharing the same address space.

Traditionally, subroutine libraries are difficult to use, because they are not integrated into the programming environment. For example, when implementing a monitor, the programmer must explicitly place preamble and postable subroutine calls into each member of the monitor module. Languages such as Concurrent Pascal or Path Pascal simplify the use of common constructs; however, they usually require compilers that are difficult to port to new environments. Consequentially, parallel programming libraries are more commonly used; although portions of the library must be reimplemented for new architectures, this is considerably easier than retargeting a compiler.

Object oriented languages, such as SmallTalk, Objective C or C++, can provide a middle ground. Although the parallel programming constructs are not tightly integrated into the language, object-oriented languages simplify construction of parallel programs using subroutine libraries, providing a reasonable syntax and the ability to extend an existing design.

This report describes how to use the AWESIME parallel programming environment, written in C++. The environment is designed to be extensible and customizable. The design has evolved over three years, reflecting the refinement of the class hierarchy and expanded utility. Care was taken to allow a parallel program to execute efficiently on a single processor system while maintaining compatibility with multiprocessor systems.

2 Architectural Assumptions

AWESIME was designed for architectures with a shared address space running some variant of the Unix operating system. A prime goal in the design of of AWESIME was portability, necessitating certain assumptions in the underlying memory architecture of the the host machine. The AWESIME library creates multiple Unix processes that are ideally mapped to individual processors. The child processes inherit the memory and file configuration of the initial process. Within AWESIME, each Unix process is term a CPU.

The memory layout assumed by AWESIME is illustrated by Figure 1. AWESIME was developed for traditional Unix memory architectures; program text is in a read-only, shared text segment (*text*), initialized data is in a mutable data segment (*data*), uninitialized data is in a mutable blank storage segment (*bss*) and stack space is allocated from an expandable, mutable stack segment (*stack*).

AWESIME maps all data (data and bss) to a single shared segment, preserving the existing values of the memory space. Because most variants of Unix have limited support for shared memory segments, it is usually very difficult to "grow" the shared segment once multiple processors are using it. Thus, before the child processes are created, the shared data segment is "grown" to a specific size and can not be modified thereafter.

In AWESIME, all non-stack memory references are shared. A modified memory allocator supports dynamic allocation of memory in a parallel environment. No mechanism is provided for dynamically allocating unshared data.

Some Unix systems, such the Sequent Symmetry, provide language keywords to place data in distinct shared and private segments. This was not adopted in AWESIME, because it requires changes to the C++ compiler and the system loader to be implemented correctly.

3 Classes for Programming with Threads

The AWESIME library is structured as groups of C++ lass definitions that manipulate or specify different aspects of a parallel programming environment. This sections briefly covers the class definitions used when programming with threads, stating their purpose and explaining the motivation for the class organization used. Examples are used throughout this user manual; for more detailed explanations of the semantics of particular classes or member functions, consult the source or the AWESIME Reference Manual. In the text of this user manual, class names and member functions are are denoted by typeface. Thus, HARDWARECONTEXT refers to a specific class name, and switchTo refers to a specific method. If the class of a method is ambiguous, the fully qualified name, e.g., HardwareContext::switchTo, is used.

3.1 The Big Picture

An AWESIME application is normally composed of one or more kinds of threads; each kind of thread may have a number of instances. For example, if you were writing a simulation of a distributed computer network, you might have a kind of thread representing the network, two instances of another kind representing disk drives and two instances of yet another kind representing CPUs. The threads are managed by a "Cpu Multiplexor" that multiplexes the cpu on behalf of the threads. The cpu multiplexor encapsulates scheduling policies and some additional functionality.

3.2 Threads

The abstract class THREAD defines the interface for user threads. Each THREAD instance contains data representing the hardware execution context for that thread. Applications do not define instances of class THREAD; rather, they define *subclasses* that implement specific types of threads. Thus, in our previous example, we might define a NetworkThread as a subclass of Thread.

Initialization arguments to the thread are passed using the thread constructor. The thread constructor is executed in the hardware context of the "parent context" (which is not always a thread); this means it looks just like a regular procedure call. Subclasses of Thread can define an instance name, the stack size for the new thread, a "stack security" method and a thread priority. Normally, the default parameters suffice.

Each thread must define a virtual method main, whi is the "main procedure" or starting point for that subclass of threads. If no main method is provided, an execution error will be reported. Figure 2 illustrates the declaration and definition for class Phil, which will be used in subsequent examples were we solve the Dining Philosophers problem.

Initially, the constructor in Figure 2 gives each Phil instance the same name, ("Philosopher"), and private stack space (5000 words). The constructor then changes the thread name to an explicit name based on an identifier number passed to the thread, using the Thread::name method. If assigned, thread names are used when printing information about a particular thread. The Thread class does not allocate any storage for names, but you can use the name method to set or get the current name. The main method is the starting point for any thread instances of class Phil. Without delving into the details right off, method main represents the actions of each philosopher. The dining philosopher problems sits N philosophers around a table of N forks. Philosophers need two forks to each, thus only $\lfloor N/2 \rfloor$ philosophers can eat at one time. The problem is to orchestrate the actions of the philosophers to avoid deadlock. For example, if everyone grabbed for their left fork and refused to release it until they acquired the right fork, the entire table of philosophers would deadlock, leading to starvation. If philosophers pick up one fork and put it down when they find the other fork busy, the philosophers might livelock, again leading to starvation.

In our example solution to the problem, we mediate the use of forks by only allowing $\lfloor N/2 \rfloor$ philosophers to the table at any one time. Figure 2 shows these steps:

1. Each philosopher determines which two forks they will use to eat.

2. Until the philosopher eats their fill, they

   (a) Approach the table and acquires two forks (using the Table class).

   (b) Eats a mouthful.

   (c) Return the forks to the table.

At the moment, we won't concern ourselves with the details of the Table method; suffice it to say that it arbitrates the use of the forks in a deadlock-free and livelock-free manner.

Users of other C++ thread libraries, such as Presto or the AT&T tasking library, may be concerned about the proliferation of thread subclasses. In those libraries, there are no subclasses of Thread. Each thread instance is an instance of Thread, and each instance is passed an initial function and parameter list.

This approach limits portability, and, more importantly, is less intuitive and "object oriented" than the approach use in Awesime. Portability is increased because only a single parameter, the pointer to the class instance variable, must be passed to the new thread. Initial parameters are passed to the new thread by constructing a series of function call frames that cause the arguments to the new thread to appear as if a conventional function call had been

```
class Phil : public Thread {
    int pid;
    char nameBuffer[128];
    static SpinFetchAndOp mouthFulls;
public:
    Phil(int pid);
    virtual void main();
};

Phil::Phil(int xpid) : ( "Philosopher", 5000 )
{
    pid = xpid;
    sprintf(nameBuffer,"Phil-%d", pid);
    name(nameBuffer);
}

void
Phil::main()
{
    extern int PHILOSOPHERS;
    int toTheRight = (pid + 1) % PHILOSOPHERS;

    cerr << name() << " using forks " << pid << " and " << toTheRight << "\n";

    while( mouthFulls.value() < 10 ) {
        TheTable -> getForks(pid, toTheRight);
        cerr << name() << " gets to eat\n";
        mouthFulls += 1;
        TheTable -> returnForks(pid, toTheRight);
    }
    cerr << name() << " wanders off to burb awhile\n";
}
```

Figure 2: Sample Class Declaration and Definition

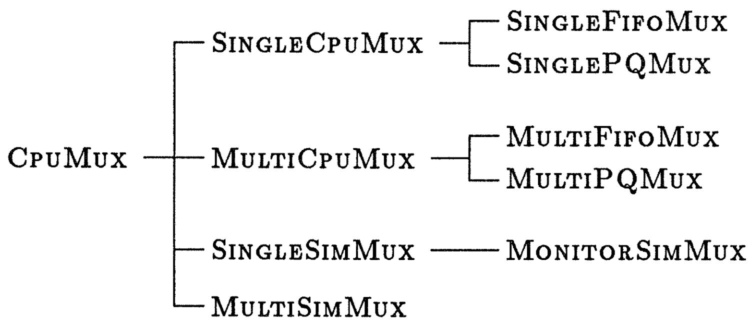

Figure 3: CPU Multiplexor Class Tree

made to the thread method main. Modern architectures, such as the SPARC or MIPS architectures, pass some parameters via registers. Reducing the number of arguments greatly simplifies the thread creation code.

The thread structure used by AWESIME is also more consistent with object-oriented design philosophy, and it is easy to create new threads. Member functions can implicitly refer to thread instance variables and thread subclasses can define other methods that abstract operations on the specific thread class. These variable references and methods are then subject to the normal C++ type checking rules. In practice, most variables referenced by a thread are class variables or local variables. In AWESIME, one simply refers to those variables. In other libraries, threads much explicitly dereference per-thread variables via a pointer.

In practice, the number of distinct thread subclasses is typically fairly small, although the number of thread instances may be large. For example, a complex AWESIME application, simulating an elaborate load distribution system for message-passing network architectures, creates only five thread subclasses, yet the application creates several thousand thread instances.

3.3 Multiplexing Threads

A parallel program may create many more threads than the number of physical processors, or CPUs, available. Thus, the CPUs must execute different threads, switching between threads when synchronization constraints delay a specific thread. We term this *CPU multiplexing.*

Figure 3 shows the class structure for CPU multiplexors defined in AWESIME. Certain cases, such as multiplexing threads on a single CPU, lend themselves to certain optimizations, implemented by the SINGLECPUMUX classes. The more general situation is threads multiplexed across multiple CPUs, implemented by the MULTICPUMUX classes. In both cases, subclasses define the scheduling policy. For example, the MULTIFIFOMUX class implements a first-in, first-out scheduling policy using multiple CPUs, while the SINGLEPQMUX implements a priority scheduling thread multiplexor on a single CPU.

The SINGLESIMMUX and MULTISIMMUX subclasses, implementing a simple global-time simulation environment. These will be discussed more completely in §5.

The CPUMUX classes allocate per-CPU data. Each CPU must allocate a private copy of the CPUMUX data structures. Since the only truly private memory available is on the stack of the individual CPUs, **applications must allocate a single CPUMUX instance on the main system stack, which is private to each CPU.** When the program "goes parallel," the child

```
Table *TheTable;
main(int argc)
{
    MultiFifoMux Cpu;
    TheTable = new Table;
    for (int i = 0; i < PHILOSOPHERS; i++) {
        CpuMux::add( new Phil(i) );
    }
    Cpu.fireItUp(3, 20000);
    cout << "All done!\n";
}
```

Figure 4: Dining Philosophers: Main Routine

CPUs will also have a CpuMux instance defined at the same place in their private memory. Initially, this will just be a copy of the parents information; as the program continues, this information is changed by each child. All of the provided CPU multiplexors allocate scheduling data structures in shared memory. This is not required, but simplifies load balancing among the schedulers.

Figure 4 shows the process of creating a cpu multiplexor, adding threads and starting the parallel threads. The `main` procedure is executed when a program starts. The variable `Cpu` is introduced as an instance of MultiFifoMux, a cpu multiplexor that can use multiple CPUs (i.e. be truly parallel) and uses a first-in-first-out scheduling discipline. The `for`-loop creates a certain number of philosophers, using the standard C++ operator new. Each philosopher is given their unique philosopher identifier, but is not immediately scheduled to execute. Threads (or, to be more accurate, instances of subclasses of Thread) are not scheduled until they're explicitly added to a CpuMux. If the philosophers were created but not added, they would never execute. Once the threads have been added, the method fireItUp tells the MultiFifoMux to start executing the threads that have been added to the multiplexor. In this example, the MultiFifoMux is told to use three CPUs to execute the program, and that the program may wish to allocate up to allocate $20,000$ additional words of storage once the threads begin execution.[1]

Until fireItUp is called, there is a single stream of execution; this execution stream is resumed when fireItUp returns, indicating that there are no more threads to execute. By that time, no more threads are executing, and any additional CPUs created by fireItUp should have finished. It is possible, although unusual, to call fireItUp several times from the main program; each time, any existing threads will be able to execute.

Thus, the general structure of the `main` subroutine is:

1. Create a CpuMux subclass instance as a local variable.

2. Parse any flags, input files, etc.

[1] Because of the rather lame UNIX interface for sharing memory, you must pre-declare the amount of *additional* storage you need. If possible, you should always create all data structures and threads *before* calling fireItUp, because you will not need to estimate how much storage they require – only additional space need be estimated.

3. Create any data structures that will be used by the threads.
4. Create the initial set of threads (these may create other threads later on, but at least one must be created initially).
5. Add some subset of the threads to the CpuMux.
6. Call fireItUp.
7. After fireItUp returns, collect and report any data or information from the program execution.

If you only use methods defined by the CpuMux, you can change the scheduling policy and execution policy (parallel or sequential execution) without modifying code outside of procedure `main`.

3.3.1 Some useful members of CpuMux

The constructor arguments for the CpuMux classes are normally limited to a single argument indicating if debugging information is desired. Setting this to true generates copious quantities of information documenting the context switching actions of the different CPUs.

The class method Cpu returns a pointer to the current CpuMux instance. There are several static member functions (class members, rather than instance members) that query the CpuMux instance *for the current CPU*. For example, the method name returns a printable string identifying the current CPU, and method cpuId returns a small integer identifying the current CPU number. The CurrentThread returns a pointer to the current Thread instance *for the current CPU*. There is no way to determine what thread is executing on another CPU.

The add method adds a thread to the CPU; this schedules the thread for future execution. The CPU actually executing the thread is determined by the specific CpuMux subclass used. The reschedule method reschedules the current task; it is placed on the pool of available threads, and the next schedulable thread is selected and executed.

The last significant method is fireItUp, which directs the CpuMux instance to "go parallel" (if it is a parallel CPU multiplexor) and start servicing threads. The first argument to fireItUp is the number of CPUs to allocate, and the second is the amount of shared memory to request *in addition to the currently allocated memory*. This later value is problematic because it is machine dependent; thus, it is best to allocate most shared structures *before* calling fireItUp, eliminating the guesswork of estimating memory usage.

3.4 Machine Dependent Classes

In Awesime, concurrent activity is represented by *light-weight threads*; each thread is a distinct parallel execution stream, and the number of threads is unrelated to the number of physical CPUs. There are many possible implementations of thread libraries; an underlying concept in all thread implementations is the notion of *hardware contexts* that encapsulates all hardware resources used by an independent execution stream. In Awesime, this architecture-specific information is represented by class HardwareContext, one of two architecture-specific classes. Normally, users do not directly manipulate HardwareContext objects; rather, those objects are activated by CPUs. Each CPU activates a single HardwareContext at a time.

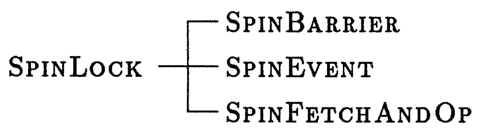

Figure 5: SpinLock Class Tree

Awesime assumes a *locking* synchronization primitive can be implemented. These locks are represented by class SpinLock (see §3.5.1), which is the only other architecture-dependent module. Other synchronization mechanisms are implemented using the SpinLock class. Despite the name, SpinLock need not implement spinlocks, as long as the semantics provide mutual exclusion. For example, some architectures may implement the SpinLock class using fetch&add operations, if those are provided by the underlying architecture.

3.5 Synchronization

This section describes how to synchronize threads at three levels of functionality and expense. The first level locks an entire CPU using hardware synchronization primitives; while the CPU is blocked, other threads are not serviced by that CPU. Thread-level locking is more general; when a thread blocks for a resource, it enqueues itself on a wait-queue, allowing the CPU to service another thread. The highest level generalizes thread locking to resource locking, implementing bounded-buffers for communication between threads.

3.5.1 CPU-level Locking

The SpinLock class tree is shown in Figure 5. SpinLock objects are initially *unlocked*. Method reserve reserves a lock, and release releases a lock. Method reserveNoBlock attempts to reserve the lock, and returns a bool value indicating success or failure. If the call to reserveNoBlock returns true, the lock has been reserved, else it has not.

The reserve operation blocks the *entire* CPU, not just the active thread using the CPU;§3.5.2 describes synchronization mechanisms that block Thread objects. In general, the SpinLock class and subclasses are only used when you can show the program would not deadlock because the CPU is no longer available. Examples include locking iostream objects during actual I/O, or when locking critical sections. Using SpinLock objects is very efficient; the reserve and release operations are usually inline functions, removing function call overhead. The following illustrates declaring and using a SpinLock to safely execute some critical section of code.

```
foo() {
     static SpinLock fooLock;   // shared across all CPUs
     ......
     fooLock.reserve()
     ...do some foo-like stuff...
     fooLock.release()
}
```

There is also a C language interface to the spinlock primitives. Variables are declared of type SpinLockType, and the functions SpinLockReserve, SpinLockRelease and SpinLockReserveNoBlock perform functions analogous to those implemented by class SpinLock.

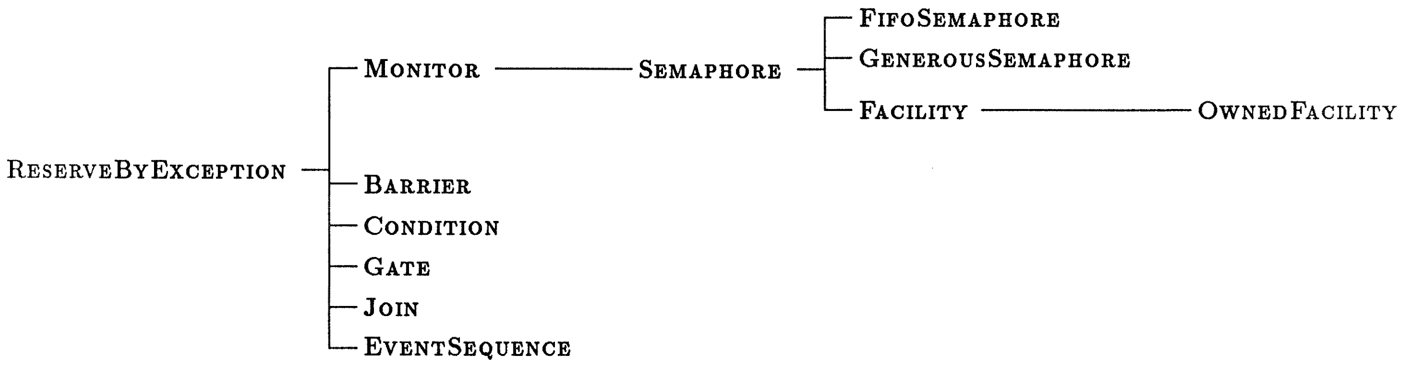

Figure 6: Thread-level Locking Class Tree

For architectures that do not currently support hardware locking, such as the DECstation uniprocessors using the MIPS R2000 and R3000, the SpinLock methods reduce to assertions that insure that deadlock conditions could not have arisen. Since the Awesime library is (usually) non-preemptive, these assertions will only arise because of program error. These same errors would usually arise in a parallel execution.

The SpinBarrier *uses* the SpinLock class; it does not inherit any operations. The SpinBarrier class is initialized with *height*, or the number of CPUs that will rendezvous at the barrier. The rendezvous method enters the current CPU into the current rendezvous group of the barrier. When the number of CPUs in the rendezvous group meets the barrier height, the barrier is lowered, and the CPUs proceed. The SpinGate is similar to a SpinBarrier, but the barrier must always be explicitly lowered for tasks to proceed. CPUs wait at the gate using wait, and are released using release.

The SpinFetchAndOp class encapsulates the concept of locking a data value, recording the current value, performing an update operation, and then returning the previous value. This simplifies many common algorithms, but is typically no more efficient than performing the actions directly using a SpinLock. On certain architectures, the SpinFetchAndOp class may be implemented using special hardware making this more efficient. The constructor specifies the initial value. The set method explicitly sets the value, and value returns the current value. Several methods, such as += perform update operations, returning the previous value. For example, the following code fragment...

```
SpinFetchAndOp foo(5);
int old = (foo += 10);
cout << old << foo.value();
```

should print '5' followed by '10'. The returned value of the update operators can be ignored.

3.5.2 Thread-level Locking

Figure 6 illustrates the thread-level locking currently implemented in Awesime. These locking mechanisms are efficient when the thread does not need to block, which is the normal case.

The Monitor class implements a CPU-blocking mutual exclusion lock; in essence, Monitor defines an alternate interface to the SpinLock class. The Monitor class is used by the

CONDITION synchronization, and is normally used to implement a monitor. Users may want to use the CONDITION class with general counting semaphores or with semaphores having a certain scheduling policy in addition to the more efficient SPINLOCK class. The MONITOR class provides a single interface for these classes.

The SEMAPHORE class implements a general counting semaphore. The SEMAPHORE class has methods similar to the SPINLOCK class, with corresponding meanings. In the SEMAPHORE class, an arbitrary scheduling policy can be specified; the FIFOSEMAPHORE class implements a general counting semaphore with FIFO scheduling discipline; The GENEROUSSEMAPHORE is similar to SEMAPHORE, but a thread removed from the wait-queue is directly scheduled in favor the the currently executing thread. This is used to improve the efficiency of producer-consumer problems, where releasing a thread indicates that thread has work to do, and the releasing thread will shortly block anyway. The FACILITY and OWNEDFACILITY classes are used when building simulations; they are discussed in §5

Class BARRIER implements a barrier with functionality and interface similar to the SPINBARRIER class, save that threads use a wait-queue to avoid blocking the CPU. Likewise, GATE implements a thread-locking synchronization matching the SPINGATE semantics and interface.

The CONDITION class implements condition variables for a MONITOR instance passed to the CONDITION constructor. Calling the wait method enqueues the thread and then releases the MONITOR. The signal method enables an enqueued thread, if one is waiting, and the broadcast method enables *all* enqueued threads. In each case, it is assumed that the thread calling wait, signal or broadcast has reserved the corresponding CONDITION.

The JOIN class implements a thread-join, used to synchronize threads following a fork-join process structure. A JOIN instance is given a pointer to the child thread. The parent thread calls the parentJoin method while the child calls the childJoin method. The child also passes in a return value to be returned to the parent. When both child and parent have met at the JOIN, the child is deleted and the parent is added to the CPUMUX for future execution.

The EVENTSEQUENCE class generates a sequent of events "tickets" and manages an associated event sequence counter and set of threads waiting for specific events. The ticket method gets an event ticket; tickets are monotonically increasing numbers. Threads wait for a particular ticket using await; this waits until an internal counter is greater than the ticket number being awaited. The internal counter is advanced using the advance method.

4 Extended Example – Dining Philosophers

To illustrate how some of the different classes are used, we finish our solution to the Dining Philosophers problem, using different synchronization mechanisms. Our solution has three parts. The main procedure, shown in Figure 4 has already been described.

In Figure 2, we saw that philosophers used a TABLE class to arbitrate access to the forks. The philosophers agree to eat only until each has taken ten mouthfuls. In Figure 2, we elided a few details. In particular, when threads use shared resources, such as files, they must get exclusive use of those resources. One convention for the common use of the `cerr` output stream is to bracket each use of the `cerr` streams with the macros `CERR_ALWAYS_PRE;...;CERR_POST;`. These macros are defined in file containing the CPUMUX class. If output streams are used in an unlocked fashion, output may be garbled. Thus, the main loop of PHIL::MAIN should really look like:

```
void
Phil::main()
{
    extern int PHILOSOPHERS;
    int toTheRight = (pid + 1) % PHILOSOPHERS;

    CERR_ALWAYS_PRE;
    cerr << name() << " using forks " << pid << " and " << toTheRight << "\n";
    CERR_POST;

    while( mouthFulls.value() < 10 ) {
        TheTable -> getForks(pid, toTheRight);
        CERR_ALWAYS_PRE;
        cerr << name() << " gets to eat\n";
        CERR_POST;
        mouthFulls += 1;
        TheTable -> returnForks(pid, toTheRight);
    }
    CERR_ALWAYS_PRE;
    cerr << name() << " wanders off to burb awhile\n";
    CERR_POST;
}
```

The definition for TABLE is show in Figure 7; The constructor Table::Table initializes the table general counting semaphore so up to $\lfloor \text{PHILOSOPHERS}/2 \rfloor$ can be at the table at one time. Each philosopher at the table must still reserve the appropriate forks.

The output from this version of the Dining Philosophers problem is shown in Figure 8. This version shows very unfair behavior, because it acquires the lock using an unfair lock implementation. The SPINLOCK implementations typically do not queue accesses; thus, in this case, some philosophers do not identify their forks until other threads have completely finished eating. We can reduce some of this unfairness by using a BARRIER. We modify the Phil::main method to create a barrier and have each philosopher rendezvous at the barrier:

```
static Barrier sayGrace( PHILOSOPHERS );
sayGrace.rendezvous();
while( mouthFulls.value() < 10 ) {
    TheTable -> getForks(pid, toTheRight);
```

The output for this variant is shown in Figure 9. This variant continues to exhibit unfair behavior, but now, at least, all philosophers can get to the table before one of them eats all the food.

It is possible to provide countless variations on this classic problem, and these make interesting exercises. For example, rather than use a semaphore to guard the TABLE, a GENEROUSSEMAPHORE or even an EVENTSEQUENCE could be used. Modifying this example program, several variants of which are included in the AWESIME distribution, is a good start for using AWESIME.

```
class Table {
    Semaphore table;
    Semaphore fork[ PHILOSOPHERS ];
public:
    Table();
    virtual void getForks(int left, int right);
    virtual void returnForks(int left, int right);
};

Table TheTable;

Table::Table() : table( int(PHILOSOPHERS/2) )
{
}

void
Table::getForks(int left, int right)
{
    table.reserve();
    fork[left].reserve();
    fork[right].reserve();
}

void
Table::returnForks(int left, int right)
{
    fork[right].release();
    fork[left].release();
    table.release();
}
```

Figure 7: Definition of TABLE

```
Phil-0 using forks 0 and 1
Phil-0 gets to eat
Phil-0 gets to eat
Phil-0 gets to eat
Phil-0 gets to eat
Phil-0 gets to eat
Phil-0 gets to eat
Phil-0 gets to eat
Phil-1 using forks 1 and 2
Phil-2 using forks 2 and 3
Phil-2 gets to eat
Phil-2 gets to eat
Phil-2 gets to eat
Phil-2 wanders off to burb awhile
Phil-4 using forks 4 and 0
Phil-4 wanders off to burb awhile
Phil-0 gets to eat
Phil-0 wanders off to burb awhile
Phil-1 gets to eat
Phil-1 wanders off to burb awhile
Phil-3 using forks 3 and 4
Phil-3 wanders off to burb awhile
All done!
```

Figure 8: Output From Dining Philosophers

```
Phil-0 using forks 0 and 1
Phil-3 using forks 3 and 4
Phil-4 using forks 4 and 0
Phil-2 using forks 2 and 3
Phil-1 using forks 1 and 2
Phil-1 gets to eat
Phil-1 gets to eat
Phil-1 gets to eat
Phil-1 gets to eat
Phil-1 gets to eat
Phil-1 gets to eat
Phil-1 gets to eat
Phil-1 gets to eat
Phil-1 gets to eat
Phil-1 gets to eat
Phil-1 wanders off to burb awhile
Phil-4 wanders off to burb awhile
Phil-2 wanders off to burb awhile
Phil-3 gets to eat
Phil-3 wanders off to burb awhile
Phil-0 gets to eat
Phil-0 wanders off to burb awhile
All done!
```

Figure 9: Using Barrier To Ensure Fair Start

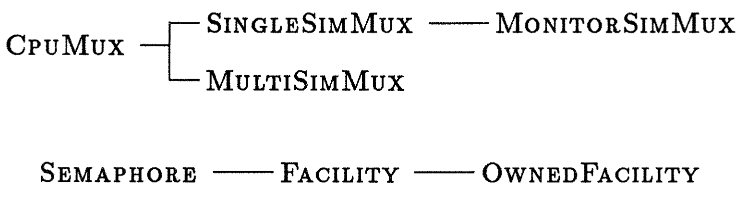

Figure 10: Simulation Thread Oriented Classes

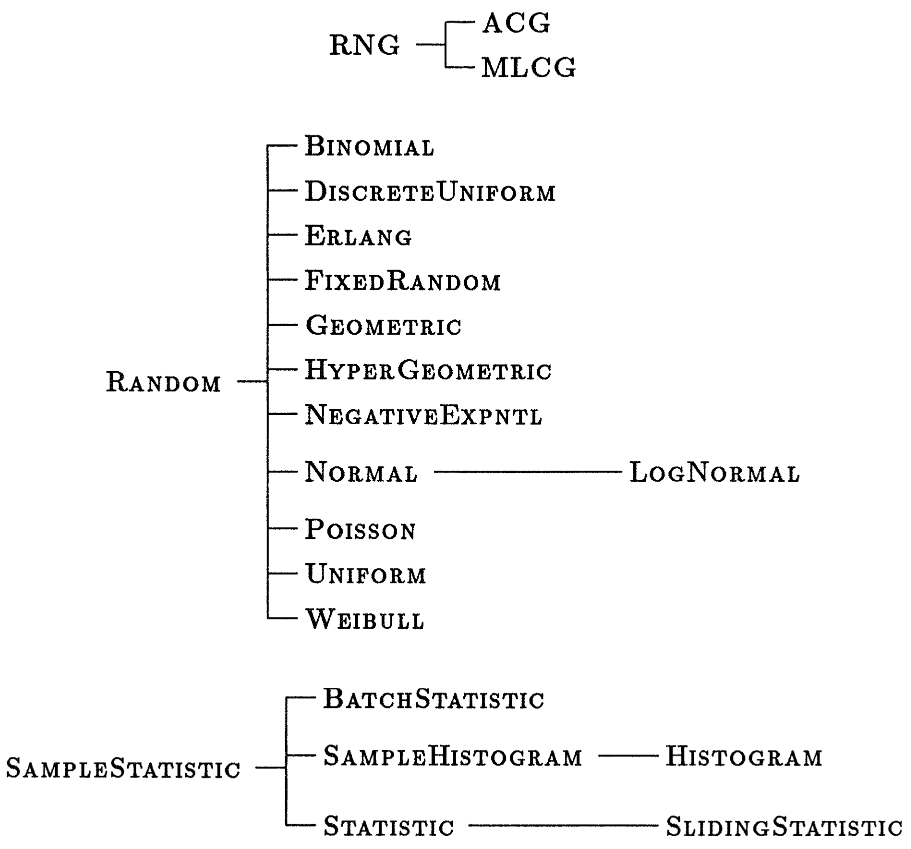

Figure 11: Random Number & Statistics Classes

5 Classes for Process-oriented Simulation

In this section, we discuss the classes shown in Figure 10; these are variants of previously introduced classes that are specific to simulation. We also discuss classes, shown in Figure 11, specific to generating random numbers and transforming those random numbers into distributions. We first discuss the classes and methods provided, and then illustrate their use using a simple $M/M/1$ simulation.

5.1 Thread Classes for Simulation

The goal of AWESIME is to provide a parallel programming environment and a process-oriented simulation environment. The SINGLECPUMUX and MULTICPUMUX, and associated subclasses, are used for parallel programming. The SINGLESIMMUX and MULTISIMMUX classes impose a global simulated time order over threads. The time order is specified in terms specific to a simulation. Threads can await for a specified time, or can be blocked on resources and released at a latter point in time. The MONITORSIMMUX class reports the number of concurrent events per time-unit; this can be used to determine if a simulation should be run in parallel. When running a simulation, all events in a *event set*, i.e. those occurring at a particular time, are executed before time is advanced to the next event set. If an event set has a large number of items, they can be processed in parallel, because they occur "at the same time". If there are few events in each set, then parallel execution will waste resources. The MONITORSIMMUX class reports the mean size of the event set after the simulation terminates. This gives a reasonable estimate to the concurrency of that particular simulation.

The SIMMUX classes multiplex threads with the additional constraint of a global time order. The interface is largely the same as CPUMUX, with the addition of methods to get and and affect the current time. The SIMMUX instance is created in the main procedure of the program, exactly in the same manner used when creating a CPUMUX class.

The CurrentTime method reads the current time. Time advances relative to the initial time, specified by the NullTime method. Time values are stored in the SIMTIMEUNIT class. This normally defaults to DOUBLE values, but can be changed to another; this involves recompiling the library, and generally doesn't yield much speed improvement for modern architectures.

Time changes only when explicitly indicated by *events*. The SIMMUX classes manages events, normally instances of THREAD to be executed at a particular time. For threads, execution resumes at the point of suspension, as with the CPUMUX classes.

Events can be added to the SIMMUX using the addAt method, specifying an event (a THREAD) and the time when the event should be executed. The await method performs a similar function when executed by a THREAD; the thread is scheduled to resume at a specified time, and other events are executed until that time arrives. The addWithDelay class schedules an event relative to the current time. The hold method is similar to await; when executed by a THREAD, the thread delays for a time relative to the current time. The await and hold methods *must* only be executed when a THREAD is running.

The FACILITY class is similar to a SEMAPHORE. A FACILITY can have multiple *servers*. Each server can be used by a single thread at a time. A FACILITY records the amount of time spent being used; the percentage time used over the lifetime of the FACILITY is the facility utilization. In other words, for s servers, assume server i has been busy for b_i time units. If the facility was created at time t_0, then the utilization at time t_n is $u(t_n) = \left(\sum_{0 \leq i \leq s} b_i\right) / s(t_n - t_0)$.

If a THREAD attempts to use a FACILITY that is already in use, it queues until the FACILITY is free. The total time spent waiting for a FACILITY is also recorded. The reserve method reserves the facility while the release method releases it; the THREAD may hold or await while holding the facility. The time elapsing between the reserve and release is the time the FACILITY was used. The use method simplifies this common idiom; it reserves the facility, holds it for a specified period and then releases it.

The utilization method returns the current utilization of the facility. The reset method resets the facility statistics, including the utilization. The queueLength method returns the current number of threads waiting for the FACILITY, while meanQueueLength returns the mean queue length since the server was started or reset. The meanDelay method returns the mean delay experienced when acquiring the server. The FACILITY class defines other methods as well, but these are rarely used; see the actual source code for more information.

5.2 Random Number Generation

Simulation programs typically use random numbers in some form. The RNG subclasses implement *random number generators*; these produce a number of bits, typically a longword, of random information. These random bits are typically transformed via a RANDOM class to produce a specific random number distribution. The RNG classes use the asLong method to return an unsigned long of random bits. The asDouble extracts two longwords of random bits and transforms it to a floating point value between zero and one; for machines using the IEEE floating point specification, this transformation is very fast.

The central random number generator, ACG, is a variant of a Linear Congruential Generator (Algorithm M) described in [Knuth 81]. This result is permuted with a Fibonacci Additive Congruential Generator to get good independence between samples. This is a very high quality random number generator, although it requires a fair amount of memory for each instance of the generator. The ACG constructor takes two parameters: the seed and the size. The seed is any number to be used as an initial seed. The performance of the generator depends on having a distribution of bits through the seed. If you choose a number in the range of 0 to 31, a seed with more bits is chosen. Other values are modified to give a better distribution of bits. The MLCG class implements a Multiplicative Linear Congruential Generator[L'Ecuyer 88]. This generator has a fairly long period, and has been statistically analyzed to show that it gives good inter-sample independence. The MLCG constructor has two parameters, both of which are seeds for the generator. As in the ACG generator, both seeds are modified to give a "better" distribution of seed digits. Thus, you can safely use values such as '0' or '1' for the seeds. The MLCG generator uses only two longwords (8 bytes) of state, considerably less than the ACG generator.

5.3 Random Number Distribution

The RANDOM classes in Figure 11 transform a sequence of random values from a RNG into a random number distribution. Once a RANDOM is created, the operator() method retrieves the sequence of values. Each RANDOM requires a RNG from which it draws random samples. Most classes require additional parameters, briefly described below. For example, the following code fragment prints two numbers from a normal distribution with mean 10 and variance 2:

```
MLCG gen(0,1);
Normal u(10, 2, &gen);
cout << "first : " << u() << "\n";
cout << "second: " << u() << "\n";
```

The parameters specific to each class are specified below.

BINOMIAL: The first parameter for the BINOMIAL constructor is the number of items in the pool, n, and the second parameter, is the probability of each item being successfully drawn, u. Although it is not checked, it is assumed that $n > 0$ and $0 \leq u \leq 1$. The remaining methods allow you to read and set the parameters.

ERLANG: implements an Erlang distribution with mean **mean** and variance **variance**.

GEOMETRIC: implements a discrete geometric distribution. The first parameter to the constructor is the mean of the distribution. Although it is not checked, it is assumed that $0 \leq$ mean ≤ 1. The method operator() returns the number of uniform random samples that were drawn before the sample was larger than **mean**; this quantity is always greater than zero.

FIXEDRANDOM: returns a stream of constants, specified by the first parameter.

HYPERGEOMETRIC: implements the hypergeometric distribution. The first parameter to the constructor is the mean and the second is the variance.

LOGNORMAL: implements the logarithmic normal distribution. The first parameter to the constructor is the mean and the second is the variance.

NEGATIVEEXPNTL: implements the negative exponential distribution. The first parameter to the constructor is the mean.

NORMAL: implements the normal distribution. The first parameter to the constructor is the mean and the second is the variance.

POISSON: implements the poisson distribution. The first parameter to the constructor is the mean.

RANDOMINTERVAL: implements a uniform random variable over the closed interval [low...high]. The first parameter to the constructor is low value, and the second is the high value. The order of these parameters may be reversed.

RANDOMRANGE: implements a random variable over the open interval [low...high). The first parameter to the constructor is low value, and the second is the high value. The order of these parameters may be reversed.

WEIBULL: implements a weibull distribution with parameters α and β. The first parameter to the class constructor is α, and the second parameter is β.

5.4 Statistics Gathering Classes

The SampleStatistic class is used to compute the mean, variance, standard deviation and confidence intervals from sampled data. The operator+= method adds data items to the SampleStatistic. Various methods provide the mean, variance and so on. Two methods compute the confidence internal. In one, the percentage is expressed as an integer, e.g, "99" corresponds to the 99th percent confidence interval. In the other, fractions, e.g., ".99" serve a similar purpose. The Statistic class is similar, but provides a printing function. For example,

```
Statistic lie;
lie += 2;
lie += 4;
cout << lie << "\n";
```

would print the number of samples, the mean, the standard deviation and the 90%, 95% and 99% confidence intervals. The SlidingStatistic class is similar to Statistic, but records statistics over the most recent samples. For example,

```
SlidingStatistic foo(2);
foo += 2;
foo += 4;
foo += 6;
cout << foo.mean() << "\n";
```

would print the mean of the last two samples, or "5."[2]

The SampleHistogram class is similar to SampleStatistic, but counts the number of samples falling into a number of "buckets" of a specified width between low and high values of interest. The number of low value, high value and width are specified in the constructor. The buckets method returns the number of buckets. Bucket zero holds all the count of samples less than the low value. The highest numbered bucket holds the count of samples greater than the high value. The other buckets hold data values in the range $[w*i+l \ldots w*(i+1)+l)$ for bucket i, where l is the low value and w is the specified width. If no width is specified, it defaults to $(h-l)/10$, giving 10 buckets. The inBucket method returns the number of samples that fall in a particular bucket, and the similarSamples returns the number of samples that fall in the same bucket at the specified value.

This information can be used to print a histogram of the samples using the printBuckets member. The Histogram class can be printed to a stream, much as a Statistic.

The BatchStatistic class collects samples until a batch, specified by a number of samples, is reached. It then records the mean of the current batch as a sample in another Statistic. The mean, variance and so on, can be computed for the current batch in the same manner as for Statistic. Similar methods, such as batchMean, provide the values for the batched means.

5.5 Example: Simulating an $M/M/1$ Queue

In this section we describe an implementation of an $M/M/1$ simulation. Customers arrive at a counter with negative exponential interarrival and service times. The simulation measures the

[2]The existence of the distinct SampleStatistic and Statistic classes is for backwards compatibility. The printing functions were removed from the SampleStatistic class when it was submitted to the Gnu G++ library.

```
Facility counter;

main(int argc, char **argv)
{
    int maxCustomers = (argc > 1) ? atoi(argv[1]) : MAX_CUSTOMERS;
    double meanArrival = (argc > 2) ? atof(argv[2]) : T_ARRIVAL;
    double meanService = (argc > 3) ? atof(argv[3]) : T_SERVICE;
    bool debug = (argc > 4) ? TRUE : FALSE;

    MonitorSimMux Simulator( debug );

    Simulator.add(new Gen(maxCustomers, meanService, meanArrival));

    cout << "\nSINGLE SERVER Queueing System:\n";
    cout << "-----------------------------\n";
    cout << "\tNumber of Customers   \t" << maxCustomers << "\n";
    cout << "\tMean InterArrival Time\t" << meanArrival << "\n";
    cout << "\tMean Service Time     \t" << meanService << "\n";

    Simulator.fireItUp(1,100 * 4196);

    cout << "Server utilization was : " << counter.utilization() << "\n";
    cout << "Server mean delay was  : " << counter.meanDelay() << "\n";
    cout << "Server mean queue was  : " << counter.meanQueueLength() << "\n";

    double mu = 1.0 / meanService;
    double lambda = 1.0 / meanArrival;
    double rho = lambda/mu;

    cout << "Traffic intensity = " << rho << "\n";
    cout << "Expected number of customers in system = ";
    cout << rho / ( 1 - rho ) << "\n";
    cout << "Variance of customers = ";
    cout << (rho / ( (1-rho) * (1-rho) ) ) << "\n";
    cout << "expected response time is " << (1.0 / mu) / ( 1 - rho ) << "\n";

    cout << "\nDONE\n";
    exit(0);
}
```

Figure 12: $M/M/1$ Example: Main Routine

```
class Gen : public Thread {
    int customers;
    MLCG rng;
    NegativeExpntl serviceTime, arrivalTime;
public:
    Gen(int cust, double svc, double arr);
    virtual void main();
};
Gen::Gen(int customers_, double meanService, double meanArrival)
        : ( "CUSTMR", 10000), rng(time(0), time(0)),
          serviceTime(meanService, &rng),arrivalTime(meanArrival, &rng)
{
    customers = customers_;
}
void
Gen::main()
{
    for (int i = 0; i < customers; i++ ) {
        SimMux::add( new Customer( serviceTime() ));
        SimMux::hold( arrivalTime() );
    }
}
```

Figure 13: $M/M/1$ Example: Generator

```
class Customer : public Thread {
    double serviceTime;
public:
    Customer( double serviceTime );
    virtual void main();
};
Customer::Customer( double serviceTime_ )
{
    serviceTime = serviceTime_;
}
void
Customer::main()
{
    extern Facility counter;
    counter.use( serviceTime );
}
```

Figure 14: $M/M/1$ Example: Customer

```
SINGLE SERVER Queueing System:
-----------------------------
Number of Customers    100
Mean InterArrival Time 1
Mean Service Time      0.5
[MonitorSimMux-0] Statistics for threads per tick
[MonitorSimMux-0] 160 1.8625 .363206 .047667 .056901 .075108
[MonitorSimMux-0] Min = 0
[MonitorSimMux-0] Max = 2
Server utilization was : .551157
Server mean delay was  : .576368
Server mean queue was  : 1.11493
Traffic intensity = 0.5
Expected number of customers in system = 1
Variance of customers = 2
expected response time is 1
```

Figure 15: *M/M/*1 Example: Sample Output

average queue length, average queue delay and the server utilization. It compares the simulated results to the theoretical results.

The main subroutine, shown in Figure 12, reads parameters and creates a MONITORSIMMUX. Although a SIMMUX would also work, the MONITORSIMMUX will illustrate any potential for parallelism in this simulation. A GEN, responsible for generating the customers, is created and added to the MONITORSIMMUX. The remainder of the main routine calculates the theoretical values for the queue length and response time.

The generator, shown in Figure 13, creates customers and adds them to the simulation multiplexor. A MLCG is created with initial seed values set to the the current time. The MLCG is used by two NEGATIVEEXPNTL distributions to generate the interarrival and service times. The GEN draws a service time, passing it to a new instance of the CUSTOMER class. The new CUSTOMER is added to the SIMMUX, and the GEN thread delays (using SimMux::hold) execution for the the time specified by the random interarrival variable.

Each customer is represented by a thread, shown in Figure 14. Each customer uses the `serviceTime` value from the generator and uses the globally defined FACILITY, called `counter`, for the specified time.

As coded, this example could be run in parallel, because no resources (the random number generators and distributions) are shared. However, the sample output, in Figure 15, indicates that little would be won by running this simulation in parallel. The simulation was run with the default parameters. The lines labeled `[MonitorSimMux-0]` are produced by MONITORSIMMUX when there are no more events left to process. The values indicate there were 160 "event sets" with each event set having an average of 1.86 events, with a variance of 0.36 events per set. The remaining values give the 90%, 95% and 99% confidence intervals. This indicates that, with perfect execution, 1.86 processors could be kept busy; in practice, the speedup was not be as great as this.

The remaining lines compare the actual simulation, which used only 100 customers, with the theoretical values. The number of customers in the system should match the mean server queue length; differences exist because of the small sample size and the stochastic nature of the simulation.

The $M/M/1$ simulation is intentionally simplistic, but it illustrates the mechanisms for building more complex simulations. Again, the code for the simulation is distributed with the AWESIME distribution, and modifying the simulation is a very good way to become familiar with the AWESIME simulation classes.

References

[Bershad et al. 88] Bershad, B., Lazowska, E., and Levy, H. PRESTO: A system for object-oriented parallel programming. *Software - Practice and Experience*, 18(8):713–732, August 1988.

[Knuth 81] Knuth, D. E. *Seminumerical Algorithms*, volume 2 of *Art of Computer Programming*. Addison Wesley, 2nd edition, 1981.

[L'Ecuyer 88] L'Ecuyer, P. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–749, 1988.

[Schwetman 86] Schwetman, H. *CSIM Reference Manual (Revision 9)*. Microelectronics and Computer Technology Corperation, 9430 Research Blvd, Auston TX, 1986.

[Weiser et al. 89] Weiser, M., Demers, A., and Hauser, C. The portable common runtime approach to interoperability. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 114–122, Litchfield Park, AZ, December 1989.