

The PRESTO Users Manual

*Brian N. Bershad*¹

University of Washington
Department of Computer Science
Seattle, Washington 98195
(206) 545-2675

January 1988, October 1991

Abstract

PRESTO is an environment for writing object-oriented parallel programs in the C++ programming language. It makes few assumptions about the behavior of these programs and the models of parallelism to which they adhere. This paper describes the basic PRESTO primitives and provides examples of their use. A simple programming environment based on Mesa monitors and threads is provided as part of the standard PRESTO distribution, but this may easily (and efficiently) be abandoned in preference to environments based on other models.²

1 Introduction

PRESTO is an object-oriented parallel programming environment for shared-memory multiprocessors. The system provides the programmer with a set of basic classes useful for writing parallel programs in the C++ programming language. These include threads for concurrency, and locking mechanisms for synchronization. The structure of these more primitive classes and the basic PRESTO run-time environment are sufficient for writing many types of parallel programs. Nevertheless, the system is structured so that the programmer may extend, modify, or completely replace arbitrary pieces of the environment.

This manual is divided into two parts. The first addresses the mechanics of writing PRESTO programs using the structures provided by the system. Programming examples and short explanations about how the system works are also given. The second part discusses in more depth the internals of PRESTO and how to modify the system to meet the needs of specific applications. The reader is expected to have a good understanding of object-oriented programming concepts in general, and the C++ programming language in particular. To help clarify explanations, class definitions and code fragments are presented throughout this document. A more thorough motivation for the system, its structures, and its relationship to parallel programming in general can be found in [?] and [?].

2 Information For PRESTO Programmers

2.1 Parallel Programming On Top Of UNIX

A UNIX process provides a single thread of control within an address space. To the UNIX kernel, that process is the smallest schedulable entity. Unfortunately, relying directly on heavyweight UNIX processes for building parallel applications can be severely restrictive. Problems include the high cost of context switching, the limited number of simultaneous threads possible, and difficulty in sharing many types of resources between UNIX processes.

PRESTO works by considering UNIX processes as physical processors. When a PRESTO program begins, some number of UNIX processes are created. A PRESTO thread can be scheduled to execute within any one of these processes. Since all PRESTO objects live in a single address space, that address space can be shared amongst all the UNIX processes, allowing a thread to move between UNIX processes. Although this structure is hidden from the

¹ Revised for Presto 1.0 by Paul Barton-Davis, University of Washington

² This work is supported by the National Science Foundation under Grants No. DCR-8420945, CCR-8619663, and CCR-8703049, and by grants from the Naval Ocean Systems Center, US WEST Advanced Technologies, the Washington Technology Center, the USENIX Association, and Digital Equipment Corporation's Systems Research Center and External Research Program.

PRESTO programmer (who sees only threads and a single address space), understanding how the system works can be useful during debugging. Further, understanding will make clearer the reasons for some of the system's current limitations, which (will be discussed in this document as they arise in context).

On true multiprocessors, PRESTO creates multiple UNIX processes in which to run threads. On uniprocessors, all threads are scheduled within the context of a single UNIX process. Again though, this distinction is generally transparent to the programmer who may construct applications without concern for the underlying processor structure.

2.2 C++ On A Sequent

C++ on the Sequent multiprocessor is almost identical to generic C++. The Sequent C compiler understands the two storage class identifiers `shared` and `private`. Since the latter conflicts with a C++ keyword, they have been changed for the C++ compiler to be `shared_t` and `private_t`.³

For example,

```
static shared_t int x = 100;           // declare x to be static shared
extern private_t Thread* thisthread;   // thisthread is private to each processor
```

By default, the loader assumes that declarations not specifying the storage class are to be private and not shared across processors. Since all PRESTO objects live in a single, shared address space, there should be no connection between a physical processor and an object. The programmer must either ensure that the `-Y` flag is given to the loader (to change the default behavior), or explicitly declare all static objects as shared (`shared_t`). Both methods are recommended as they ensure that the right thing always happens (the loader will complain if the storage class for an object is inconsistently declared). Failure to properly declare the storage class will cause strange things to happen when a static object is referenced from more than one processor. The most common types of PRESTO programming errors result from misusing the global shared memory. As a rule, PRESTO programs should almost never use private objects since threads can migrate between processors. PRESTO programs written on the Sequent can be recompiled directly on other machines, where the storage class specifiers are pre-processed away.

2.3 PRESTO Components

PRESTO consists of a run-time library and a set of header files that define system objects. The library is called `libpresto.a`. All of the necessary header files can be pulled in by including only the file `presto.h`. These files should probably be placed in `/usr/local/lib/libpresto.a` and `/usr/include/presto`. In addition to the PRESTO library, Sequent programmers will also need to include Sequent's parallel programming library dealing with the shared-memory structures on that machine. Eventually, PRESTO will provide its own memory management, eliminating the need for the Sequent library.

PRESTO relies heavily on inline expansions for simple, but frequently called functions (such as lock acquisition). These functions are defined within the PRESTO header files and are subject to change. Without inline functions, a change in the libraries only requires that user programs be relinked. In-line functions require that they be recompiled as well. For this reason, programmers are encouraged to specify the complete header dependencies within their makefiles. The following is a sample PRESTO makefile that takes care of creating these dependencies.

³The appropriate diffs to `cfront` are included in the appendix.

```

LIBS      =      /usr/local/lib/libpresto.a  -lpps
MAKEFILE  =      Makefile
CC        =      CC
CFLAGS    =      -g
LDFLAGS   =      -Y                      # if Sequent
SRCS      =      qs.c qsmain.c
OBJS      =      qs.o qsmain.o
PROG      =      qs
$(PROG):   $(OBJS) $(LIBS)
           $(CC) $(CFLAGS) $(LDFLAGS) $(OBJS) $(LIBS) -o $(PROG)

#
# Construct header dependencies
#
depend:
    $(CC) -M $(SRCS) | sort | uniq > makedep
    cp $(MAKEFILE) $(MAKEFILE).sav
    sed -n '1,/^\# DO NOT DELETE THIS LINE/p' $(MAKEFILE).sav >$(MAKEFILE)
    echo '# stuff after here goes away' >>$(MAKEFILE)
    cat makedep >> $(MAKEFILE)
    echo '# DEPENDENCIES MUST END AT END OF FILE' >> $(MAKEFILE)
    echo '# IF YOU PUT STUFF HERE IT WILL GO AWAY' >> $(MAKEFILE)
# DO NOT DELETE THIS LINE
# stuff after here goes away

```

The rules for making depend will update the makefile to properly reflect all header dependencies. It is important to keep these up-to-date. The only other interesting point about the makefile is the definition of LDFLAGS for the Sequent. The -Y will force static objects to be shared, *with the exception* of static class members. These need to be explicitly declared as shared.

```

class Any    {
    static shared_t int x;
};

```

2.4 Writing PRESTO Programs

A PRESTO program is essentially a C++ program having more than one thread of execution. A thread represents a virtual processor. There may be many more threads than physical processors when executing a PRESTO program. It is best to think of a thread as always executing in the context of one object or another. When an object synchronously invokes another object, the invoking object's thread is *borrowed* by the other, wherein it executes (possibly passing on to other objects) until it eventually returns. For consistency, global functions can be considered member functions of some single anonymous global object.

All PRESTO programs have three phases: initialization, execution and termination. These phases are defined in terms of member functions on the class Main, which is used to transform a program from a single-threaded UNIX process into a multi-threaded PRESTO program.

```

class Main    {
    int numprocessors;           // # scheduling processors
    int nummainthreads;         // # threads in Main::main()
    int mainstacksizes;         // how big each stack
    int quantum;                // scheduling quantum
    int argc;                   // argc from main
    char **argv;                // argv from main
    char **envp;                // envp from main
public:
    Main(int ac, char **av, char **ep);
    ~Main();
    int init();                 // user provides any or all of
    int main();                 // init, main, done...
    int done();                 //
};

```

The programmer provides the implementation for the functions `init()`, `main()` and `done()`, much as the UNIX programmer provides the implementation for the subprogram `main()` (which should not be provided in a PRESTO program). The initializing `Main::init()` is called in the context of a single-threaded UNIX program by the PRESTO run-time system (PRTS). In it, the programmer can specify PRTS parameters or create alternative system objects. `Main::init()` should return non-zero on failure, which will be passed through to `::exit()`. If the initialization does not fail, `Main::nummainthreads` will begin executing in `Main::main()`, running on top `Main::numprocessors` processors. If not set by `Main::init()`, the system will default to having one thread on one processor. For sanity's sake, `Main::numprocessors` is limited to one fewer than the number of physical processors on the machine. When the system runs out of runnable threads, it returns to a single-threaded UNIX mode and invokes `Main::done()`. The programmer can use this function to perform any extra cleanup that might be necessary. The function's return value is passed on to `::exit()`. The PRTS provides default implementations for the three functions `Main::init()`, `Main::main()` and `Main::done()`. These implementations do nothing, so the programmer who needs nothing can use them.

2.5 Basic Objects

The two main PRESTO classes for dealing with threads and synchronization are derived from the very basic object `Object`.

```
// objects.h
class Object {
    int      o_type;           // object type
    char     *o_name;          // object name
    Object   *o_next;          // linked list next field
public:
    Object(int type, char* name, Object* next = 0)
    int type()
    virtual void error(char* s);    // all objects handle their own errors
    char* name()
};
```

An instance of class `Object` has a name, type, an error handler, and the ability to "live" in an object queues (such as class `Oqueue`). PRESTO defines several object types in the file `objects.h`. Users can define their own objects having type values greater than `OBJ_END`. The main reason for the type is to allow error checking when dealing with objects in different capacities. The error handler provides a convenient way to lay the blame on an object when something goes wrong. The object can deal with the error in an appropriate manner. Most PRESTO objects abort the program when an error occurs simple, but effective.

2.6 Threads

Although a thread represents the computational power of a virtual processor, a thread itself is represented by a normal C++ object. Creating and starting threads within an object's member functions are the most common operations.

```
//
// Abridged Thread interface.  FILE threads.h
//
class Thread : public Object {
    Thread(char* name, int tid = 0, long ssiz = DEFSTACKSIZ, int musthavestack = 0);
    ~Thread();
    int      start(Object obj, PFany pf, ...);
};
```

A thread has, among other things, a name, a thread id, and a stack. Creating a new thread is done with the standard new operator.

```
Thread *t = new Thread("billy", 100);
```

will create a new thread by the name "billy" with an id of 100. Thread names and thread id's are currently not used for anything in the PRTS other than to identify an offending thread when an error occurs. These values can be obtained using the operations `char *name()` and `int tid()`.

```
cout << "t's name is " << t->name() << " and id is " << t->tid();
```

The default stack size is 16k bytes; to modify this, set the variable `stacksize` in `Main::init()`. For simplicity, stack sizes should be specified in increments of ONEK (or left to the default). The current implementation does no checking for stack overflow, although it probably should. The last argument to the thread constructor should only be given (as non zero) when a complete thread must be constructed. If a complete thread is not needed, the system may perform certain storage optimizations in order to minimize the amount of new shared data that must be created. These optimizations are discussed in detail in a later section.

A newly created thread is essentially a passive data object. That is, it has everything a thread needs *except* something to do. The operation `start()` enqueues a thread to begin executing within the member function of some object.

```
Thread *t = new Thread("billy");           // defaults are sufficient
Matrix *m = new Matrix;
Matrix *n = new Matrix;
t->start((Objany)m, (PFAny)m->multiply, n);
```

is equivalent to the asynchronous invocation `m->multiply(n)`. `start()` returns immediately, after having scheduled `t` for execution. A thread may only be started once or an error will occur. If the first argument to `start()` is NULL, the thread will be started in the global function named in the second parameter.

```
{
    extern int write(int fd, char* buf, int len);
    Thread *t = new Thread("writer");
#define HW      "Hello World"
    t->start(0, (PFAny)write, fileno(stdout), HW, sizeof(HW));
}
```

Although C++ has a reasonably strict typing system, `start()` is essentially untyped since it can be used to start a thread within any object's member function having any number of any type of parameters. The implementation of `start()` blindly copies its arguments onto the thread's stack without checking their types. Caveat Programmer.⁴

Older versions of Presto relied on a now anachronistic feature of C++ compilers that allowed pointers to member functions to be cast into pointers to functions. Current and future compiler implementations either do or will prohibit use of such a cast, which in violation of the C++ ARM [?]. Because of this, you should obey the following guidelines concerning the function pointer passed to `start()`:

- The function should be a `static` member function **OR**
- The function should be a global function taking the desired object as its first argument.

The following examples illustrate legal and illegal calls to `start()`:

```
class Foo {
    . . .
public:
    static void static_member_function (Foo *, int arg);
    void non_static_member_function (int arg);
    . . .
```

⁴On RISC processors, such as the MIPS R3000, the use of registers for argument passing prevents such untyped calls from being easily decipherable by the PRTS. Thus, on such processors, `start` has only 3 arguments: a `Thread`, a function pointer and a single argument of type `void *`. The final argument can obviously be a pointer to an array of arguments, to be deciphered by the function.

```

};

void global_function (Foo *f, int arg) {
    f->non_static_member_function (arg);
}

Thread *t = new Thread ("Fred");
Foo      *f = new Foo ();
int      arg = 4;

t->start ((Objany)f, (PFany)Foo::static_member_function, arg);    // legal
t->start ((Objany)f, (PFany)global_function, arg);                // legal
t->start ((Objany)f, (PFany)Foo::non_static_member_function, arg); // illegal

```

5

The asynchronous nature of `start ()` affects the styles of parameter passing that may be used in a started function. Default and reference parameters will not work properly, although virtual and inline functions work fine. Overloaded functions will also not work, since the compiler can not discern the types of the arguments to the function (or its return value) at compile time.

Disallowing reference parameters implies that all parameters (including pointers) are passed by value. Consequently, the programmer should be careful that pointers reference objects residing in the global address space (static or created by `new ()`) and do not point to objects on the stack of the thread that is starting the new thread. Failure to abide by this may result in the passed pointer referencing garbage (consider the stack's behavior on an asynchronous invocation). Each of the following `start ()` statements is erroneous.

```

class Foo      {
public:
    static int use_pointer(Foo *, int *x);
    static int use_reference(Foo *, int &x);
    static int use_default(Foo *, int x = -123);
    static int use_overload(Foo *, int x);
    static int use_overload(Foo *, float x);
};
....
{
    Foo      *f = new Foo;
    Thread *t = new Thread("timmy");
    t->start((Objany)f, (PFany)f->use_any, &x);                // pointer to stack variable
/*or*/ t->start((Objany)f, (PFany)f->use_reference, x);        // reference parameter
/*or*/ t->start((Objany)f, (PFany)f->use_default);             // default parameter
/*or*/ t->start((Objany)f, (PFany)f->use_overload,12);         // overloaded function
/*or*/ t->start((Objany)f, (PFany)f->use_overload, 8.5);       // overloaded function
}

```

A thread begins executing on its own stack. The variable `thisthread` always refers to the thread referencing it. For example, an object can query the name of the thread by which it is being animated.

```

cout << thisthread->name() << " - id == " << thisthread->tid();

```

When a thread returns from the function in which it was started, it will be reclaimed by the system. The programmer should be careful when referencing a thread after it has been started (garbage collection may cause the thread to become something different).

It is possible to wait on a thread's completion by expressing interest in the started function's return value after the thread has been created, but before it has been started.

⁵ Some compilers will still accept the illegal form; cfront 3.0 however, as well as other C++ compilers that conform to the C++ ARM [?] will not compile such code, or will cause a core dump when the program is run. *Do not use non-static member functions as the second argument to `start ()`*

```

t->willjoin();                // preserve the return value
t->start(anyobject, anyfunction, args); // go
...                          // stuff here
Objany tval = t->join();      // wait here

```

Objany is a typedef for void* and can be used to name any four byte value. A thread may be joined by only one other. If a thread terminates, but has been marked as "joinable" with willjoin(), the thread will not be garbage collected until another thread joins with it. A thread may prematurely terminate itself⁶ with

```

thisthread->terminate(obj)

```

where obj is an optional (default NULL) parameter of type Objany. If thisthread has been marked as joinable, the argument to terminate() is returned to the joining thread. The call to terminate() never returns.

An object may start a thread within itself, as well as within other objects.

```

t = new Thread("self");
t->start((Objany)this, (PFany)this->func, arg1, arg2);

```

Among other things, this allows objects to animate themselves. In particular, a self-threading object can take advantage of static construction to form itself into an object that is always executing. The following example demonstrates this.

```

# include <stream.h>
# include "presto.h"
class Busy {
    char *n;
public:
    Busy(char *name, int count)
    { n = name;
      Thread *t = new Thread(n);
      t->start((Objany)this, (PFany)this->wait, count*1000);
    }
    int wait(int delay);
    ~Busy()
    { cout << "BYE BYE from " << n; }
};
int
Busy::wait(int delay)
{
    while (delay-- > 0)
        if (delay % 1000 == 0)
            cout << n << ":" << delay;
}
Main::init()
{
    numprocessors = 3;                // run on three processors
    return 0;
}
// STATIC CONSTRUCTORS
shared_t Busy    busyone("jim", 25 );
shared_t Busy    busytwo("fred", 40 );
shared_t Busy    busythree("isaac", 80 );
shared_t Busy    busyfour("bill", 15 );

```

The key point in this example is that it is not necessary to even provide an implementation for Main::main(). The declarations of the statics busyone through busyfour will cause the constructor for Busy to be invoked even before the PRTS begins.

⁶Threads may terminate themselves only. There is no support (yet) for terminating other threads. Stay tuned.

There are some limitations to what can be done inside constructors that are called during static initialization. The order in which static constructors is called is not specified by the C++ language, so it's possible for one static constructor to reference another static object that has not yet been constructed. This will likely cause a segmentation fault. In addition, since the PRTS may not yet be initialized during a static constructor's execution, certain operations are not guaranteed to work (or guaranteed not to work). Any operation that involves the scheduler (such as waiting on a condition variable) will cause a segmentation fault. The result of operations on `thisthread` are also unsafe. Objects may query a global variable to determine the current state of the system,

```
// presto.h
extern shared_t int prestoState;
#define STATIC_CTOR      0      /* running static constructors */
#define MAIN_INIT        1      /* inside Main::init() */
#define MAIN_MAIN        2      /* inside Main::main() */
#define MAIN_DONE        3      /* inside Main::done() */
#define STATIC_DTOR      4      /* inside static destructors */
```

or use the macro `MULTITHREADED()` to determine the legitimacy of certain operations.

2.7 Forking

Creating and starting a thread can be combined into a single `fork` operation applied to the current thread `thisthread`. For example,

```
Thread* t = thisthread->fork(NOJOIN, (Objany)this, (PFany)this->wait, count*1000);
```

is equivalent to the code in the last example of a `Busy` constructor. If the first argument to `fork` is `WILLJOIN` then the return value can be used to join with the forked thread. It is not possible to fork on a thread other than `thisthread`.

2.8 Preemption

By setting the `Main::quantum` variable in `Main::init()`, PRESTO programs can take advantage of a preemptive scheduler. A preemptive scheduler imposes slightly more overhead, but can provide a more realistic mapping from virtual to physical processor. The quantum is specified in milliseconds. A quantum of zero implies no preemption. As the quantum decreases, program throughput may also decrease due to the overhead of dealing with many asynchronous scheduling interrupts. A quantum of 500 ms is reasonable, 100 is excessive, and 10 (the minimum) is absurd.

A thread may mark itself as non-preemptable,

```
thisthread->nonpreemptable();
```

or preemptable,

```
thisthread->preemptable();
```

to insulate itself from the preemptive scheduler.

2.9 Synchronization

Concurrent threads must be able to synchronize their actions. To allow this, PRESTO provides two basic types of synchronization: non-relinquishing and relinquishing.

2.10 Spinlocks

Non-relinquishing synchronization relies on spinlocks to force a thread to busywait on an event (generally, the release of a critical section of code). Spinlocks can be constructed, locked, unlocked, and conditionally locked. The following code fragment demonstrates how a spinlock might be used to control access to a shared critical region.


```

class MultiThreadedObject {
    Spinlock *sl;
public:
    MultiThreadObject()
        { sl = new Spinlock; }
    ~MultiThreadObject()
        { delete sl; }
    void entryPoint();
};
void MultiThreadedObject::entryPoint()
{
    sl->lock();
    // critical section
    sl->unlock();
    if (sl->checklock()) { // return zero lock acquired
        // critical section
        sl->unlock();
    } else
        // can not acquire lock. Take other action
}

```

When a thread holds a spinlock, that thread is not preemptable. A thread spinning, waiting for a lock, is preemptable though.

The current Sequent implementation does not consider the acquisition of a hardware lock and the marking of a thread as non-preemptable as an atomic event. To ensure correctness, a thread is marked as non-preemptable when it tries to acquire the hardware lock, rather than waiting until the lock has actually been acquired. Thus, a thread spinning on a lock will not be preempted. To do otherwise would introduce the possibility that a thread, having acquired a lock but not yet marked as non-preemptable, could be preempted. This can quickly lead to a deadlock situation. This problem will be fixed on the Symmetry.

2.11 Relinquishing Synchronization

When the expected waiting time for entering a critical section is high, spinlocks make very ineffective use of the underlying processors. Generally, it is more appropriate to relinquish the processor and reschedule the waiting thread when it would be allowed to enter the critical section. The three operations on threads that permit this are

```

// FILE: threads.h
class Thread ... {
    ...
    void swtch();
    void sleep(SynchroObject* so);
    void wakeup(SynchroObject* so);
    ...
};

```

The first invokes the thread's lowest level scheduling primitive, relinquishing the processor to a system scheduling thread. The scheduling thread then runs any other ready thread that can be found. A thread can only sleep or switch out on itself. Trying to do otherwise raises an error on the thread.

Typically, threads don't just switch out. They go to sleep waiting for some event to occur, at which point they are "woken up". The classes describing these events are inherited from the general class `SynchroObject`.

```

// FILE: synchro.h
class SynchroObject : public Object {
    Spinlock *so_lock;
    ThreadQUnlocked *so_waiting;
public:
    SynchroObject(int t, char *name);
    ~SynchroObject();
};

```

```

void remember(Thread* t);          // remember thread blocking
Thread* recall();                  // get blocked thread
virtual void error(char* s);
inline void lock();                // spin on access to object
inline void unlock();
ThreadQUnlocked *waitingQueue(); // return waiting queue
};

```

The base class `Object` allows `SynchroObjects` to be maintained in queues, have names, etc. (see `objects.h`). `SynchroObjects` use a simple spinlock to control access to the object's data structures. To see why this is needed, consider two threads trying to acquire a monitor at the same time. The `so_waiting` queue keeps track of all threads waiting on a given `SynchroObject`.

2.12 Useful Synchronization Objects

`SynchroObjects` have no usage semantics. That is, a thread can not wait on a `SynchroObject` directly. They exist as a base class for other relinquishing primitives that do have semantics. For example, a simple relinquishing lock is defined in terms of the basic `SynchroObject`.

```

// synchro.h
class Lock : public SynchroObject {
    Thread *lo_owner;          // who owns the lock
    void lock2();              // looping wait
public:
    Lock(char *name);
    ~Lock();
    inline void lock();         // acquire
    inline void unlock();       // release
    void error(char *s);
    Thread *owner()
        { return lo_owner; }
};

```

A `Lock` is nothing more than a `SynchroObject` that has an owner (`lo_owner`) and some operations (`lock()` and `unlock()`).

Performance is the motivation for in-lining the lock function. On the Sequent, the speedup can be as much as one-third. On machines with better support for atomic locking (such as the Symmetry), the improvement due to inline expansions will be less. Eventually, the more complicated inline functions will not be inlined.⁷

Another point relating to inline functions is their inability to describe loops. A looping implementation of `Lock::lock()` would be written as:

```

void Lock::lock()
{
    SynchroObject::lock();          // acquire base spinlock
    while (lo_owner) {
        // update thisthread's state to reflect new waiting status
        remember(thisthread);      // enqueue on SynchroObject
        SynchroObject::unlock();
        thisthread->sleep(this);    // go to sleep here
        SynchroObject::lock();      // reacquire base spinlock
    }
    lo_owner = thisthread;          // it's mine!
    SynchroObject::unlock();
}

```

⁷ A procedure call takes about 15 μ secs. Acquiring and releasing the `SynchroObject`'s spinlock takes about 30 μ secs. An synchronization object that does nothing could take 45 μ secs without inline expansion, but only 30 otherwise.

but the C++ compiler balks at the while loop. To solve this, the lock implementation is broken into two parts: a non-looping part which can be inlined, and a looping part that is called if the lock is already held. The rationale is that if the lock is already held, the overhead of an extra procedure call will be small compared to the waiting time of the lock. This trick is used in several places throughout the system.

```
inline
void Lock::lock()
{
    SynchroObject::lock();
    if (lo_owner == 0) {
        lo_owner = thisthread;
        SynchroObject::unlock();
    } else
        lock2();
}
```

where `lock2()` simply implements the while loop in the first version of `lock()`.

A thread can put only itself to sleep, but any thread can cause another thread to wakeup.

```
inline
void Lock::unlock()
{
    SynchroObject::lock();
    lo_owner = 0;
    Thread *newowner = recall(); // find new lock owner
    SynchroObject::unlock();
    if (newowner)
        newowner->wakeup((SynchroObject*)this);
}
```

`wakeup()` ensures that the thread being awoken is indeed sleeping, and then enqueues it for running. It is an error to wakeup a thread which is not sleeping.

In addition to simple relinquishing locks, PRESTO also offers monitors and condition variables. Together, these implement the Mesa synchronization semantics [?]. There is no compiler support for these objects, so it is not possible to declare an entire object or function as "monitored." This fact must be included in the definition. For each function that should be guarded by a monitor, it is necessary to include the code marking the monitor's entry and exit at the appropriate points. The following code replaces the spinlock in the `MultiThreadedObject` class above with a monitor and adds a condition variable.

```
class MultiThreadedObject {
    Monitor      *mto_mon;
    Condition    *mto_cond;
    int          mto_ok;
public:
    MultiThreadObject()
    { mto_mon = new Monitor("mto_monitor");
      mto_cond = new Condition(m, "mto_condition");
      mto_ok = 0;
    }
    ~MultiThreadObject()
    { delete mto_mon; delete mto_cond; }
    void entryPoint();
};
void MultiThreadedObject::entryPoint()
{
    mto_mon->entry();
    // critical section
    while (!mto_ok)
        mto_cond->wait();
}
```

```

        // more critical section
        if (mto_ok)
            mto_cond->signal();
    mto_mon->exit();
}

```

A condition variable must be bound to a monitor. It is an error for a thread to operate on a condition variable if that thread does not currently hold the associated monitor. Similarly, a thread may not exit a monitor (or release a lock) that it does not hold. These run-time restrictions are necessary since the compiler does not enforce special scoping checks for condition variables. There is an advantage to this though: condition variables and monitors may be passed as arguments to functions.

Monitors can be created (new), destroyed (delete), entered, exited, queried for their owner, printed, and laid blame upon.

```

class Monitor : public Lock
{
public:
    Monitor(char* name=0);
    ~Monitor();
    inline void entry()
        { Lock::lock(); }
    inline void exit()
        { Lock::unlock(); }
    Thread *owner()
        { return Lock::owner(); }
    virtual void print(ostream& = cout);
    // error function derived from Lock
};

```

Having to explicitly enter and exit each monitored critical section can be inconvenient and error-prone. To address this, a syntactically sugared construct exists that allows a monitor to control access to a C++ bracketed scope. When flow of control leaves that scope, the monitor is automatically released.

```

// FILE: synchro.h
class MONITOR {
    Monitor *mo_mon;
public:
    MONITOR(Monitor *m)
        { mo_mon = m; m->entry(); }
    ~MONITOR()
        { mo_mon->exit(); }
};

```

Instances of class MONITOR should only be declared on the stack. Their only purpose is to take advantage of the fact that an object's destructor is invoked automatically when that object goes out of scope. This relieves the programmer of the responsibility of having to explicitly exit a monitor. The monitored member function in the previous example can be rewritten as:

```

void MultiThreadedObject::entryPoint()
{
    MONITOR ENTRY(mto_mon); // ENTRY is a nice sounding dummy var
    // critical section
    while (! mto_ok )
        mto_cond->wait();
    // more critical section
    if (mto_ok)
        mto_cond->signal();
}

```

The operations on a condition variable are create, destroy, wait, signal, broadcast and error.

```

// FILE: locks.h
class Condition : public SynchroObject {
    Monitor*      co_monitor;          // bound monitor
public:
    Condition(char* name=0);           // unbound
    Condition(Monitor* boundmon);      // bound, no name
    Condition(Monitor& boundmon);
    Condition(Monitor* boundmon, char* name);
    Condition(Monitor& boundmon, char* name);
    Monitor* monitor()
        { return co_monitor; }
    int threadok()                     // is cond user legit
        { return ((!co_monitor) ||    // unbound is free4all
            (thisthread == co_monitor->owner())); }
    ~Condition();
    void signal();                     // wakeup one
    void broadcast();                  // wakeup all
    void wait();                       // wait for signal (could pickup old)
    virtual void print(ostream& = cout);
};

```

Deleting a monitor or condition variable on which other threads are waiting raises an error on that monitor or condition variable.

Because condition variables obey Mesa semantics, a signal or broadcast should only be considered as a hint that some condition stronger than the monitor invariant had been established. The condition may no longer hold. The general form for condition variables is

```

while (condition_not_true) {
    condition->wait();        // relinquish the monitor
};                           // reacquire the monitor

```

3 Sequent Symmetry Specific Features

3.1 Processor Affinity

If processor affinity is enabled, each Unix process created by PRESTO will be “bound” to a given processor, increasing concurrency, but requiring more control over the CPU resources of the machine. The first process created is bound to processor 0, and so on. A PRESTO program will not exit if a process cannot be bound to a processor, but unusual behaviour and unpredictability will result.

To enable processor affinity, set the variable `affinity` to 1 in `Main::init()`. It is zero by default.

3.2 High Contention Spinlocks

A class of spinlocks designed for use in high contention situations (many threads waiting on the same lock) exists for the Symmetry version of PRESTO. The interface is identical as the original spinlock class; the class name is `HC_Spinlock`.

3.3 High Contention Atomic Integers

In addition to HC spinlocks, there is also a class of HC atomic integers, with the same interface as regular atomic integers. The class is called, suprisingly enough, `HC_AtomicInt`.

4 Debugging PRESTO Programs

For the most part, debugging PRESTO programs is similar to debugging normal C++ programs. The main difference comes about in dealing with multiple threads. The present UNIX debuggers (adb, ddt, dbx) do not know how to deal

with multiple threads. Consequently, inspecting and controlling multiple threads must be done manually. Although not difficult, it does require a bit of finesse. The situation is further complicated by the fact that the existing UNIX debuggers understand only C, but not C++.

4.1 Rules For Debugging C++ Programs

For information on debugging C++ programs in the absence of a symbolic debugger that understands C++, see [?].

4.2 Debugging Multiple Threads

A thread can loosely be defined as the combination of a program count (pc) and a stack pointer (sp). At every point in the thread's execution, the pc has a value, and the sp has a current value. The pc's value corresponds to the next instruction to be executed by the thread, and the sp points to the bottom of the stack of call-frames representing the thread's current execution state. The debugger understands only single-threaded programs and always responds to queries using the current execution state; that represented by `thisthread`.

Examining a thread that is not running is possible by changing the debugger's idea about what is the current state. For example, suppose its necessary to examine all threads blocked on a condition variable `*cond`. The condition variable is an instance of a `SynchroObject`, and so maintains a waiting queue. The `dbx` command

```
print *cond
```

will display the address of `cond->_SynchroObject_so_waiting`. The queue is an instance of class `Oqueue`, so it has a head, and each element in the queue is an instance of class `Object`. There are two ways to examine the queue. The first is to plod through the list manually from within `dbx`, looking at each objects, and then the next. The other (simpler) way is to call on an object's print routines directly, passing it the "hidden" first argument this.

```
call _Condition_print(cond->_SynchroObject_so_waiting, &cout)
```

The print function will display the state of the condition variable, and then invoke print on the waiting queue, which will invoke print on each element in the waiting queue. All PRESTO objects can display themselves. If the information displayed when an object is printed is not sufficient to answer a debugging query, then it is necessary to poke at the object directly.

It is possible to view the stack of a blocked thread by changing the debugger's notion of the current frame to the bottom-most frame of the blocked thread. For example, if `_aui_t` is a pointer to a thread in a PRESTO program, that thread's stack can be examined by doing:

```
dbx>set oldfp = $fp
dbx>assign $fp = _aui_t->t_fp
dbx>where
dbx>dump
dbx>assign $fp = oldfp
```

This simply stores the current frame pointer, installs a new one, dumps the stack, and then restores the old frame pointer. A thread's `t_fp` field always refers back to the frame that should be loaded when the thread is next involved in a context switch. So, if the thread is currently active, `t_fp` is the frame pointer for a switch-back scheduling thread. If the thread is blocked, `t_fp` points to the blocked thread's last active frame.

It is not generally possible to single-step a blocked thread, since the thread isn't runnable. It should be possible to force a context switch to any other thread waiting on the ready queue, but a reasonable interface allowing this does not yet exist.

5 Customizing PRESTO

PRESTO can be customized to create any one of a number of parallel programming environments. The loose structure of the system's components make this possible. There are five fundamental PRESTO objects: the scheduler, the

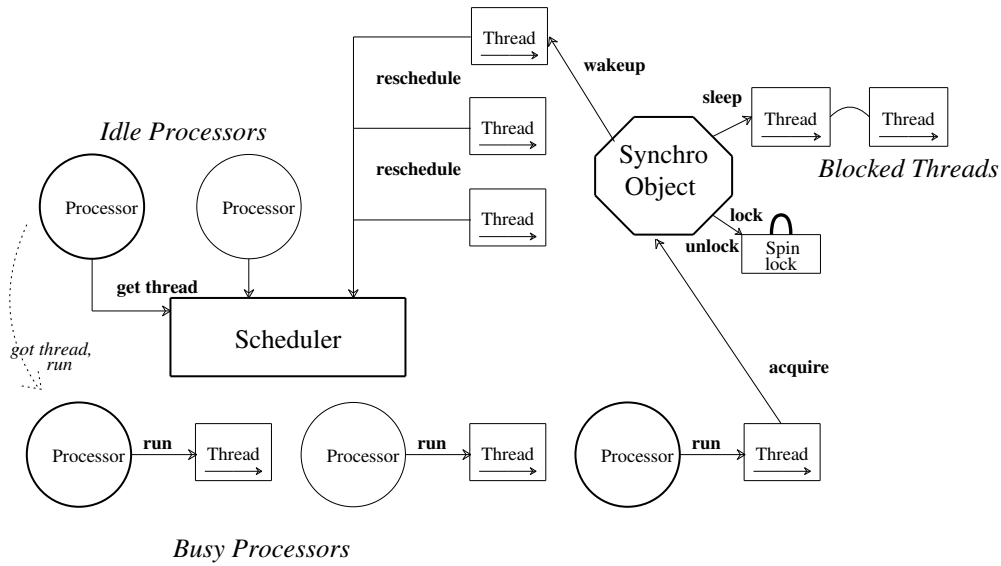


Figure 1: Presto Components

processor, the thread, the spinlock and the synchronization object. Figure I illustrates these objects in terms of their inter-object relationships. From these five, the system supplies a basic parallel programming environment that includes

- a preemptive scheduler,
- the ability to create new threads of control,
- busywaiting synchronization based on hardware atomic locks, and
- primitives allowing a thread to deschedule itself and be rescheduled by another.

Threads can be created, destroyed, put to sleep and awoken. A thread is put to sleep on a synchronization object, which consists of a queue, a spinlock, and whatever other state is needed to implement the synchronization object's semantics. The synchronization objects provided by PRESTO have no semantics in the sense of P and V [?] or signal and wait. The spinlock is needed to guard the critical sections that describe a given synchronization object. Deciding when to block and enqueue a thread that tries to pass through a synchronization object is not PRESTO's responsibility. Synchronization policy belongs to the parallel programming model, not to PRESTO.

The scheduler maintains a pool of runnable threads. Threads enter the pool when they become ready, and processors empty the pool when they become idle. The processor is an inherently active object, while the scheduler and threads are inherently passive. The main body of the processor object supplied by PRESTO does

```

forever do
  ask scheduler for the next ready thread
  if a ready thread is available
    ask the ready thread to run
  else if there will never be a ready thread again
    quit

```

When a processor asks a thread to run, the thread becomes active using the thread of the processor. Once active, the thread is able to execute within any other object. The thread runs until it is preempted, goes to sleep on a synchronization object, or terminates. After any of these actions, the processor object reactivates and continues looking for ready threads. If a processor idles, finding nothing to do, and all other processors are idle, the system halts.

The basic technique for modifying PRESTO objects is

1. Define a new PRESTO object derived from the basic object to be modified.
2. Create a new instance of this object.
3. Inform the PRTS that this new instance should be used instead of the default provided by PRESTO.

For example, to create a new scheduler, one would define a new scheduling object

```
class NewScheduler: public Scheduler {
    // differences described here
};
```

An instance of this object can then be created and bound to the name `sched`.

```
NewScheduler *sched = new Scheduler(/*Constructor arguments here*/);
```

The PRTS will instantiate its own scheduler after `Main::init()` returns and if the PRESTO name `sched` is unbound. Rebinding `sched` in routines other than `Main::init()` will only work properly if the new scheduler knows how to take control from the existing one.

Part of the PRESTO scheduler's job is to create process objects (class `Process`) and scheduling threads to animate those process object. Rather than invoking the new operator on those classes, the scheduler uses the prototypical `thisthread` and `thisproc` to obtain new instances.

```
Thread* t = thisthread->newthread(/*Constructor arguments here*/);
Process* p = thisproc->newprocess(/*Constructor arguments here*/);
```

The operators `newthread()` and `newprocess()` serve as virtual constructors for the process and thread class. Assignment to `thisthread` and `thisproc` in `Main::init()` forces the scheduler to use the virtual constructors for the type of class assigned. These are typically defined as:

```
Thread*
SomeKindOfThread::newthread(char* name, int tid, int stacksiz, int other)
{
    return (Thread*)new SomeKindOfThread(name, tid, stacksiz, other);
}
```

where `SomeKindOfThread` is derived from the basic class `Thread`.

5.1 Appendix A C++ Diffs For The Sequent

It is necessary to add the keywords `private_t` and `shared_t` to the version of `cfront` running on the sequent. The following diffs should be applied to the AT&T version of C++ (Old) to generate a Sequent compatible version (New). These diffs are also included on the distribution tape in the file `C++.diffs.sequent`.

```
diff Old/cfront.h New/cfront.h
339a340,343
> #ifdef ns32000
>     bit        b_private_t;
```



```

>         bit         b_shared_t;
> #endif
diff Old/lex.c New/lex.c
192a193,197
>
> #ifdef ns32000
>     new_key("shared_t", SHARED_T, TYPE);
>     new_key("private_t", PRIVATE_T, TYPE);
> #endif
diff Old/norm.c New/norm.c
129a130,133
> #ifdef ns32000
>     case SHARED_T: b_shared_t = 1; break;
>     case PRIVATE_T: b_private_t = 1; break;
> #endif ns32000
238a243,249
>
> #ifdef ns32000
>     /* not valid to try to save space on shared and private data types */
>     if (b_private_t || b_shared_t)
>         return this;
> #endif
>
diff Old/norm2.c New/norm2.c
149a150,153
> #ifdef ns32000
>     case PRIVATE_T: b_private_t = 1; break;
>     case SHARED_T: b_shared_t = 1; break;
> #endif
diff Old/print.c New/print.c
37a38,45
>
> #ifdef ns32000
>     if (t == PRIVATE_T)
>         putstring("private");
>     else if (t == SHARED_T)
>         putstring("shared");
>     else
> #endif
51c59,60
<
---
>
>
93a103,107
> #ifdef ns32000
>     if (b->b_shared_t) puttok(SHARED_T);
>     if (b->b_private_t) puttok(PRIVATE_T);
> #endif ns32000
>
701a716,717
>
>
706a723
>
708a726
>

```

References

- [1] B.N. Bershad, E.D. Lazowska, and H.M. Levy, "PRESTO: A System For Object-Oriented Parallel Programming", (Submitted for publication), September 1987

- [2] B.N. Bershad, E.D. Lazowska, and D.Wagner, “PRESTO: A Kernel for Parallel Programming Environments”, University of Washington, Department of Computer Science TR 88-XX-YY, January 1987
- [3] E.W. Dijkstra, “The Structure of the ‘THE’-Multiprogramming System”, *Communications of the ACM*, vol. 11, no. 5, pp. 341-346, ACM, 1968
- [4] B.W. Lampson, D.D. Redell, “Experiences with Processes and Monitors in Mesa”, *Communications of the ACM*, vol. 23 no. 2, pp. 104-117, ACM, February 1980
- [5] M.A. Ellis, B. Stroustrup, “*The Annotated C++ Reference Manual*”, Addison-Wesley, 1990
- [6] University of Washington Computer Science Lab, “*A C++ Primer*”, Technical Note 161, September 1991