Chapter 2. Support for multithreading

Authors: *Joseph T. Buck*

Multithreading means that there are multiple *threads of control*, or *lightweight processes*, in the same Ptolemy process. The principal consequence of the existence of multithreading is that it is necessary to provide mechanisms that guarantee exclusive access to resources. The Ptolemy kernel does not provide a multithreading library, as this is currently a very OS and CPU-specific operation. There are a variety of such libraries that might be used; Sun's lightweight processes library and the University of Colorado's Awesime package are two examples. What the kernel does provide is a locking mechanism for implementing *critical regions*, noninterruptable regions of code in which only one thread can be active at a time. This facility is used to protect critical resources in the kernel that might be accessed by multiple threads.

2.1 Class PtGate

Objects of classes derived from PtGate are used as semaphores to obtain exclusive access to some resource. PtGate is an abstract base class: it specifies certain functionality but does not provide an implementation. Derived classes typically provide the desired semantics for use with a particular threading library. PtGate has three virtual functions that must be implemented by each derived class. The first is a public method:

```
virtual PtGate* makeNew() const = 0;
```

The `makeNew` method returns a new object of the same class as the object it is called for, which is created on the heap. For example, a hypothetical SunLWPGate object would return a new SunLWPGate. The other two methods are protected. They are:

```
virtual void lock() = 0;
virtual void unlock() = 0;
```

The first call requests access for a resource; the second call releases access. If code in another thread calls `lock()` on the same PtGate after `lock()` has already been called on it, the second call will block until the first thread does an `unlock()` call. Note that two successive calls to `lock()` on the same PtGate from the same thread will cause that thread to hang. It is for this reason that these calls are protected, not public. Access to PtGates by user code is accomplished by means of another class, CriticalSection. The CriticalSection class is a friend of class PtGate.

2.2 Class CriticalSection

CriticalSection objects exploit the properties of constructors and destructors in C++ to provide a convenient way to implement *critical sections*: sections of code whose execution can be guaranteed to be atomic. Their use ensures that data structures can be kept consistent even when accessed from multiple threads. The CriticalSection class implements no methods other than constructors and a destructor. There are three constructors:

```
CriticalSection(PtGate *);
```

```
CriticalSection(PtGate &);
CriticalSection(GateKeeper &);
```

The function of all of these constructors is to optionally set a lock. The first constructor will set the lock on the given PtGate unless it gets a null pointer; the second form always sets the lock. The third form takes a reference to an object known as a GateKeeper (discussed in the next section) that, in a sense, may "contain" a PtGate. If it contains a PtGate, a lock is set; otherwise no lock is set. The lock is set by calling `lock()` on the PtGate object. The CriticalSection destructor frees the lock by calling `unlock()` on the PtGate object, if a lock was set. CriticalSection objects are used only for their side effects. For example:

```
PtGate& MyClass::gate;
...
void MyClass::updateDataStructure() {
        CriticalSection region(MyClass::gate);
        code;
        ...
}
```

The code between the declaration of the CriticalSection and the end of its scope will not be interrupted.

2.3 Class GateKeeper

The GateKeeper class provides a means of registering a number of PtGate pointers in a list, together with a way of creating or deleting a series of PtGate objects all at once. The motivation for this is that most Ptolemy applications do not use multithreading, and we do not wish to pay the overhead of locking and unlocking where it is not needed. We also want to have the ability to create a number of fine-grain locks all at once. GateKeeper objects should be declared only at file scope (never as automatic variables or on the heap). The constructor takes the form

```
GateKeeper(PtGate *& gateToKeep);
```

The argument is a reference to a pointer to a GateKeeper, and the function of the constructor is to add this reference to a master list. It will later be possible to "enable" the pointer, by setting it to point to a newly created PtGate of the appropriate type, or "disable" it, by deleting the PtGate object and setting the pointer to null. The GateKeeper destructor deletes the reference from the master list and also deletes any PtGate object that may be pointed to by the PtGate pointer. The public method

```
int enabled() const;
```

returns 1 if the GateKeeper's PtGate pointer is enabled (points to a PtGate) and 0 otherwise (the pointer is null). There are two public static functions:

```
static void enableAll(const PtGate& master);
```

This function creates a PtGate object for each GateKeeper on the list, by calling `makeNew()` on the master object.

```
static void disableAll();
```

This function destroys all the PtGate objects and sets the pointers to be null. This function must never be called from within a block controlled by a CriticalSection, or while multi-

threading operation is in progress. A GateKeeper may be used as the argument to a CriticalSection constructor call; the effect is the same as if the PtGate pointer were passed to the constructor directly.

2.4 Class KeptGate

A KeptGate object is simply a GateKeeper that contains its own PtGate pointer. It is derived from GateKeeper, has a private PtGate pointer member, and a constructor with no arguments. Like a GateKeeper, it should be declared only at file scope and may be used as an argument to a CriticalSection constructor call.