Chapter 6

Memory Patterns

The following patterns are presented in this chapter.

- Static Allocation Pattern: Allocates memory up front
- Pool Allocation Pattern: Preallocates pools of needed objects
- Fixed Sized Buffer Pattern: Allocates memory in same-sized blocks
- Smart Pointer Pattern: Makes pointers reliable
- Garbage Collection Pattern: Automatically reclaims lost memory
- Garbage Compactor Pattern: Automatically defragments and reclaims memory

6.1 Memory Management Patterns

Much of the difficulty in building complex real-time and embedded system centers around managing shared resources in ways that are simultaneously efficient and robust. The patterns in this chapter focus on efficient management of memory as a resource and the robust sharing of general software resources (modeled as objects) to ensure schedulability of the overall system.

6.2 STATIC ALLOCATION PATTERN

The Static Allocation Pattern applies only to simple systems with highly predictable and consistent loads. However, where it *does* apply, the application of this pattern results in systems that are easy to design and maintain.

6.2.1 Abstract

Dynamic memory allocation has two primary problems that are particularly poignant for real-time and embedded systems: nondeterministic timing of memory allocation and deallocation and memory fragmentation. This pattern takes a very simple approach to solving both these problems: *disallow dynamic memory allocation.* The application of this pattern means that all objects are allocated during system initialization. Provided that the memory loading can be known at design time and the worst-case loading can be allocated entirely in memory, the system will take a bit longer to initialize, but it will operate well during execution.

6.2.2 Problem

Dynamic memory allocation is very common in both structured and object design implementations. C++, for example, uses *new* and *delete,* whereas C uses *malloc* and *free* to allocate and deallocate mem-

ory, respectively. In both these languages, the programmer must explicitly perform these operations, but it is difficult to imagine any sizable program in either of these languages that doesn't use pointers to allocated memory. The Java language is even worse: *All* objects are allocated in dynamic memory, so all object creation implicitly uses dynamic memory allocation. Further, Java invisibly deallocates memory once it is no longer used, but when and where that occurs is not under programmer control.[1]

As common as it is, dynamic memory allocation is somewhat of an anathema to real-time systems because it has two primary difficulties. First, allocation and deallocation are nondeterministic with respect to time because generally they require searching data structures to find free memory to allocate. Second, deallocation is not without problems either. There are two strategies for deallocation: explicit and implicit. Explicit deallocation can be deterministic in terms of time (since the system has a pointer to its exact location), and the programmer must keep track of all conditions under which the memory must be released and explicitly release it. Failure to do so correctly is called a *memory leak,* since not all memory allocated is ultimately reclaimed, so the amount of allocable storage decreases over time until system failure. Implicit deallocation is done by means of a *Garbage Collector*—an object that either continuously or periodically scans memory looking for lost memory and reclaiming it. Garbage collectors *can* be used, but they are more nondeterministic than allocation strategies, require a fair amount of processing in and of themselves, and may require *more* memory in some cases (for example, if memory is to be compacted). Garbage collectors *do,* on the other hand, solve the most common severe defects in software systems.

The third issue around dynamic memory is fragmentation. As memory is allocated in blocks of various sizes, the deallocation order is usually unrelated to the allocation order. This means that what was once a contiguous block of free memory ends up as a hodgepodge of free and used blocks of memory. The fragmentation increases the longer the system runs until eventually a request for a block of memory cannot be fulfilled because there is no single block large enough to fulfill the request, even though there may be more

1. This is not true in the two competing real-time Java specifications but is true for generic Java.

than enough total memory free. This is a serious problem for all real-time and embedded systems that use dynamic memory allocation—not just for those that must run for longer periods of time between reboots.

6.2.3 Pattern Structure

Figure 6-1 shows the basic structure for this pattern. It is structurally very simple but can handle systems of arbitrary size via nesting levels of abstraction. The *System Object* starts the initialization process and creates the highest-level *Composite Objects.* They have composition relations to other *Composite Objects* or to *Primitive Objects.* The latter are defined to be objects that do not create other objects dynamically. Composition relations are used because they clearly identify the creation/deletion responsibilities.

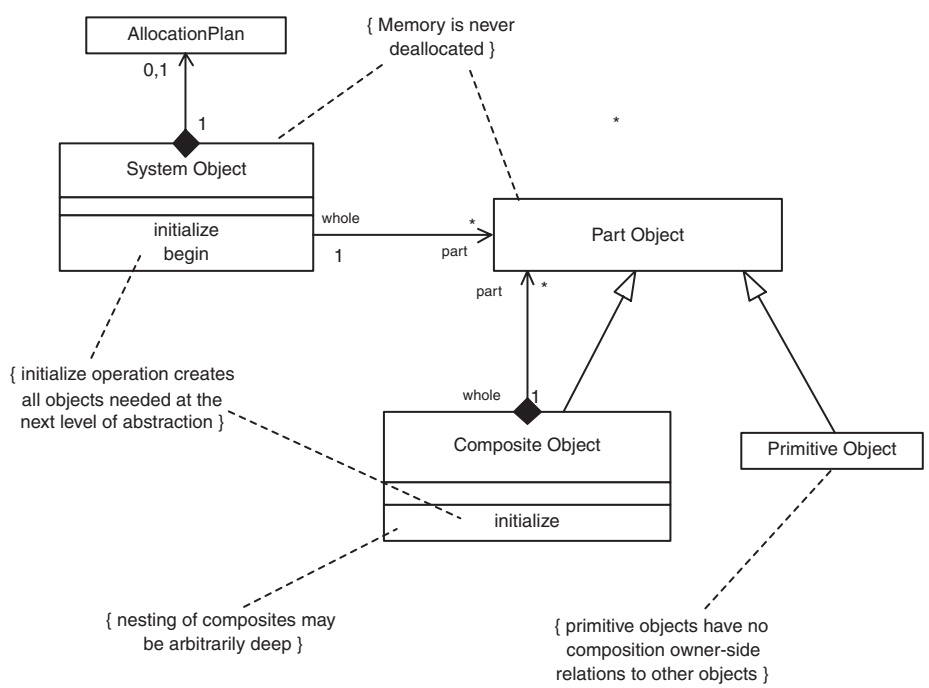


Figure 6-1: *Static Allocation Pattern*

6.2.4 Collaboration Roles

- *Allocation Plan*
  The *Allocation Plan*, if present, identifies the *order* in which the largest system composite objects should be allocated. If not present, then the system can allocate objects in any order desired.
- *Composite Object*
  A *Composite Object*, by definition within this pattern, is an object that has composition relations to other objects. These other objects may either be composites or primitive objects. A *Composite Object* is responsible for the creation of all objects that it owns via composition. There is a constraint on this object (and the *System Object* as well) that memory cannot be deallocated. Composites may be composed of other composites, but, as stated in the UML specification, each object owned via a composition relation may only belong to one such relation. This means that the creation responsibility for every object in the system is clearly identified in the pattern.
- *Part Object*
  This is a superclass of *Composite Object* and *Primitive Object*. This allows both *System Object* and *Composite Object* to contain, via composition, both *Composite Objects* and *Primitive Objects*.
- *Primitive Object*
  A *Primitive Object* is one that does not allocate any other objects. All *Primitive Objects* are created by composites.
- *System Object*
  The *System Object* is structurally the same as a *Composite Object*, except that it may own an *Allocation Plan* and is the highest abstraction possible in the system. Its responsibility is to "kick-start" the system by creating and initializing the primary pieces of the system (the highest level *Composite Objects*), which in turn create *their* pieces, and so on. Once the objects all are created, the *System Object* then begins system execution by running the *begin* operation.

6.2.5 Consequences

The Static Allocation Pattern is useful when the memory map can be allocated for worst case at run-time. This means that (1) the worst

case is known and well understood, and (2) there is enough system memory to handle the worst case. Systems that work well with this pattern typically don't have much difference between worst and average case; that is, they have a consistent memory load for all execution profiles. System behavior is likewise relatively simple and straightforward. This means that the systems using this pattern will usually be small. The need to allocate the memory for all possible objects means that it can easily happen that more memory will be required than if dynamic allocation was used. Therefore, the system must be relatively immune to the cost of memory. When the cost of memory is very small with respect to overall cost, this pattern may be applicable.

There are a number of consequences of the Static Allocation Pattern. First of all, because creation of all objects takes place at startup, the execution of the system after initialization is generally faster than when dynamic allocation is used, sometimes *much* faster. Run-time execution is usually more predictable as well because of the removal of one of the primary sources for system nondeterminism. Further, since no deallocation is done, there is no memory fragmentation whatsoever.

Since all allocation is done at start time, there may be a noticeable delay from the initiation of startup until the system becomes available for use. In some systems that must have a very short startup time, this may not be acceptable. An ideal system run-time profile is that the system can handle a long start time but must provide minimum response time once operation has begun.

6.2.6 Implementation Strategies

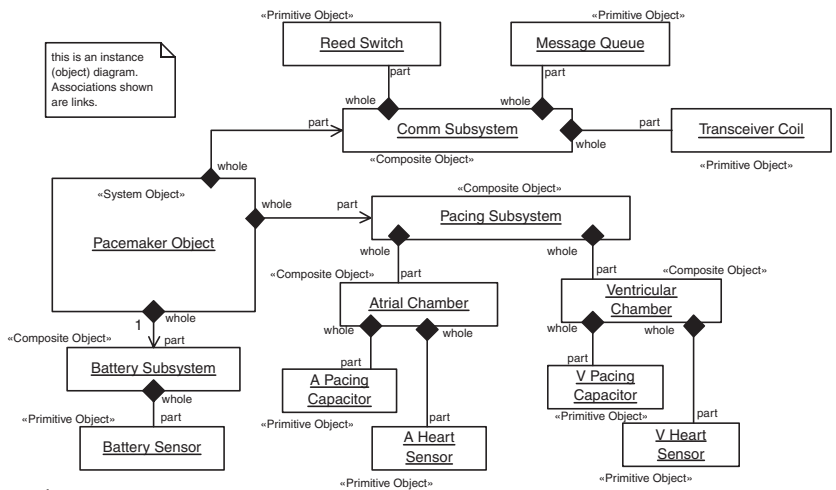
This pattern is very easy to implement. In many cases, a separate initialize method may not be required—and the constructor of each of the composites may be used.

6.2.7 Related Patterns

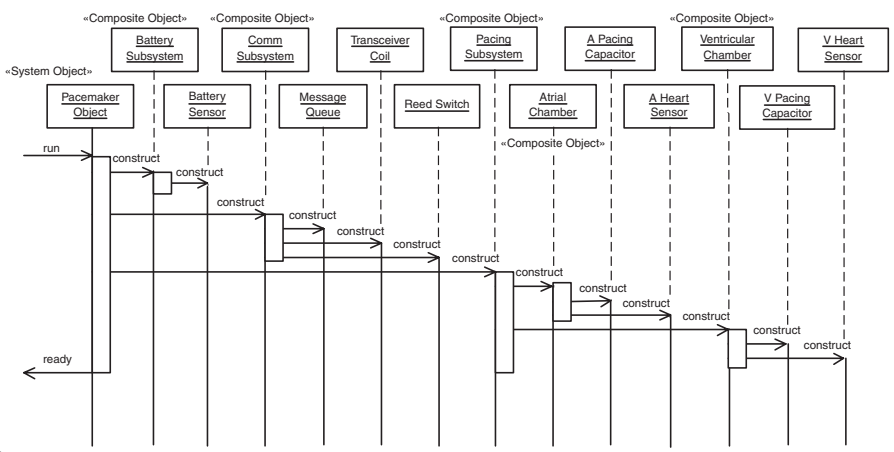
Other patterns in this chapter address these same issues but provide somewhat different benefits and consequences. See, for example, Pool Allocation, Fixed Sized Buffer, Garbage Collector, and Garbage Compactor Patterns.

6.2.8 Sample Model

Figure 6-2 shows a simple example using instances of a fully constructed system. Figure 6-2a shows the instance structure with an object diagram, and Figure 6-2b shows how the Static Allocation Pattern works on startup. You can see how the creation process is delegated to the composite objects of decreasing abstraction until the primitive objects are constructed.



a. Structure



b. Scenario

Figure 6-2: *Static Allocation Pattern Example*

6.3 POOL ALLOCATION PATTERN

The Static Allocation Pattern only works well for systems that are, well, *static* in nature. Sometimes you need sets of objects for different purposes at different times during execution. When this is the case, the Pool Allocation Pattern works well by creating pools of objects, created at startup, available to clients upon request. This pattern doesn't address needs for dynamic memory but still provides for the creation of more complex programs than the Static Allocation Pattern.

6.3.1 Abstract

In many applications, objects may be required by a large number of clients. For example, many clients may need to create data objects or message objects as the system operates in a complex, changing environment. The need for these objects may come and go, and it may not be possible to predict an optimal dispersement of the objects even if it *is* possible to bound the total number of objects needed. In this case, it makes sense to have pools of these objects—created but not necessarily initialized and available upon request. Clients can request them as necessary and release them back to the pool when they're done with them.

6.3.2 Problem

The prototypical candidate system for the Pooled Allocation Pattern is a system that cannot deal with the issues of dynamic memory allocation, but it is too complex to permit a static allocation of all objects. Typically, a number of similar, typically small, objects, such as events, messages, or data objects, may need to be created and destroyed but are not needed a priori by any particular clients for the entire run-time of the system.

6.3.3 Pattern Structure

Figure 6-3 shows the Pooled Allocation Pattern. The parameterized class *Generic Pool Manager* is instantiated to create the specific *Pooled-*

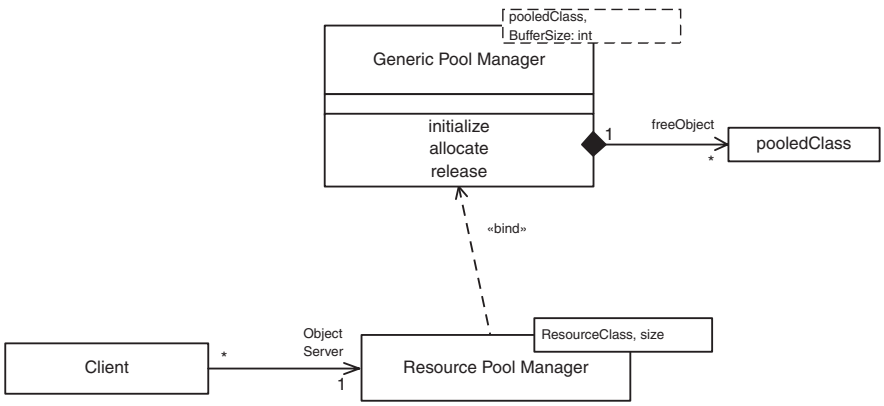


Figure 6-3: *Pooled Allocation Pattern*

Class required. The instantiated class is shown as *Resource Pool Manager.* Typically, there will be a number of such instantiated pools in the system, although only one for any specific *PooledClass* type. Each pool creates and then manages its set of objects, allocating them (and returning a pointer, reference, or handle to the allocated object to the client) and releasing them (putting the released object back into the freeObject list) upon request. Usually, the entire set of pools is created at system startup and never deleted. This removes the problems associated with dynamic memory allocation but preserves many of the benefits.

6.3.4 Collaboration Roles

- *Client*
  The *Client* is any object in the system that has a need to use one or more objects of class *resourceClass.* To request an object, they call *ResourcePool::allocate()* and give the object back to the pool by calling *ResourcePool::release().* As we will see later in the Implementation Strategies section, in C++, the *new()* and *delete()* operators may be overridden to call the *allocate()* and *release()* operations to hide the infrastructure from the client.
- *Generic Pool Manager*
  Generic Pool Manager is a parameterized (template) class that uses the formal parameters *pooledClass* and *BufferSize* to specify the

class of the objects pooled and the number of them to create, respectively. The operations of *Generic Pool Manager* are written to work in terms of these formal parameters.

- *PooledClass*
  PooledClass is a formal parameter to the *Generic Pool Manager* parameterized class. Practically, it may be realized with just about any object desired, but most often, they are simple, small classes used by a variety of *clients*.
- *Resource Pool Manager*
  The *Resource Pool Manager* class is the instantiated *Generic Pool Manager,* in which a specific class *(ResourceClass)* and a specific number of objects (*size)* are passed as actual parameters. There may be any number of such instantiations in the system but only one per *Resource Class.*

6.3.5 Consequences

The Pooled Allocation Pattern has a number of advantages over the Static Allocation Pattern. Since memory is allocated at startup and never deleted, there is no problem with the nondeterministic timing of dynamic memory allocation during run-time or memory fragmentation. The system, however, can handle nondeterministic allocation of certain classes of objects. Thus, the pattern scales to more complex systems than does the Static Allocation Pattern. It is especially applicable to systems where a number of common objects may be needed by many different clients, but it cannot be determined at design time how to distribute these objects. This pattern allows the objects to be distributed on an as-needed basis during the execution of the system. Because all pooled objects are created at system startup, the decision about the optimal number of different kinds of objects must be made at design time. For example, it might be decided that as a worst case, 1000 message objects, 5000 data sample objects, and so forth, may be required. If this turns out to be an erroneous decision, the system may fail at startup or later during execution. Further, the system cannot grow to meet new system demands, so the pattern is best applied to systems that are well understood and relatively predictable in their demands on system resources.

A note for Java programmers: This pattern is particularly helpful for Java applications because memory is never released. This pattern

avoids fragmentation and the run-time overhead for object allocation, but the garbage collector will still take up some time even though it won't find objects to delete.

6.3.6 Implementation Strategies

This common pattern is fairly easy to implement. To make the pattern easier to use in C++, it is common to rewrite *new* and *delete operators* to use the pool manager for the various *pooledClass* types. That way, the issue of dynamic versus pooled allocation can be hidden from the application programmer.

Code Segment 6-1: C++ Pooled Allocation Implementation Strategy

```
#include <list>
using namespace std;

class PoolEmpty {
}; // exception type to be thrown

// note: list is a container from the C++ STL
template <class Resource, int nElements>
class GenericPool {
       list<Resource* > freeList;
public:
       GenericPool(void) {
       for (int j=0; j<nElements; j++) {
              freeList.push_back(new Resource);
              };
       };

      Resource* allocate(void) {
              Resource* R;
              if (freeList.size() > 0) {
                     R = freeList.begin();
                     // get the first one.
                     freeList.pop_front();
                     // remove it from the free list
             return R
                     // and pass it back to the client
                   } else {
             throw new PoolEmpty;
                   };
      };

      void release(Resource* R) {
             freeList.push_back(R);
```

```
        };
};
class BusMessage {
        string s;
};
int main(void)
{
        GenericPool<BusMessage, 1000> busMessagePool;
        return 0;
}
```

Additionally, this pattern can be mixed with the Factory Pattern of [1] to create the correct subtypes, if desired.

Java has no parameterized types, but it does have collections (arrays) plus some methods for manipulating arrays (in java.util) that are modeled after the Standard Template Library of C++. There are many implementation solutions available. A very simple one was used in Code Segment 6-2. In this example, the LinkedList class from java.util was used to hold the created *BusMessage* objects. When a client allocates it, the object is removed from the list and passed back to the client. When the client wishes to return the object to the pool, it merely calls *BusMessagePool.release(),* and the object is reinserted into the pool.

Code Segment 6-2: Java Implementation Strategy for Pools

```
import java.util.*;
class BusMessage {
  private String s;
  };
class PoolEmpty extends Exception {
};
public class BusMessagePool{
  private LinkedList freeList = new LinkedList();
  public BusMessagePool() {
    for (int j=0; j<1000; j++)
      freeList.addLast(new BusMessage());
  };
  //
```

```
  // allocate() gives the client a reference to a
  // BusMessage object and removes it from the free list
  public BusMessage allocate() throws PoolEmpty {
    BusMessage B;
    if (freeList.size() > 0) {
      B = (BusMessage) freeList.getFirst();
                              // get the first one.
      freeList.removeFirst(); // remove it from the
                              // free list and pass
      return B;               // it back to the
                              // client
      } else {
      throw new PoolEmpty();
      };
  };
  // release() returns the passed BusMessage object back
into the pool
  public void release(BusMessage Carcass) {
    freeList.addFirst(Carcass);
    };
}
```

Whatever the underlying basis for the pools, there must be a separate *Resource Pool Manager* for each kind of *pooledClass.* In C++, this is simply a matter of binding a different class to the formal parameter list of the *ResourcePool* template. In Java, this can be done by creating different lists using the LinkedList containers.

6.3.7 Related Patterns

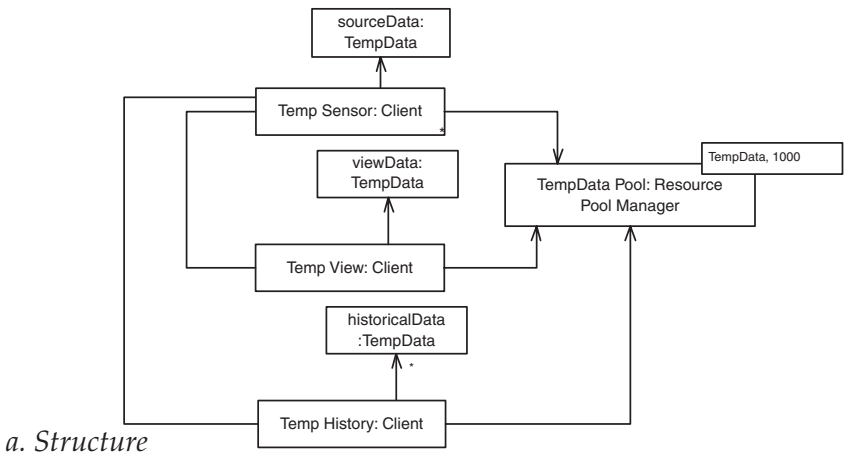
The Pooled Allocation Pattern is but one of many approaches to managing memory allocation. The Static Allocation Pattern can be used for systems that are simpler, and the Dynamic Allocation Pattern and its variants can be used for more complex needs. The Abstract Factory Pattern [1] can be used with this pattern to provide a means for Pooled Allocation for different environments.

6.3.8 Sample Model

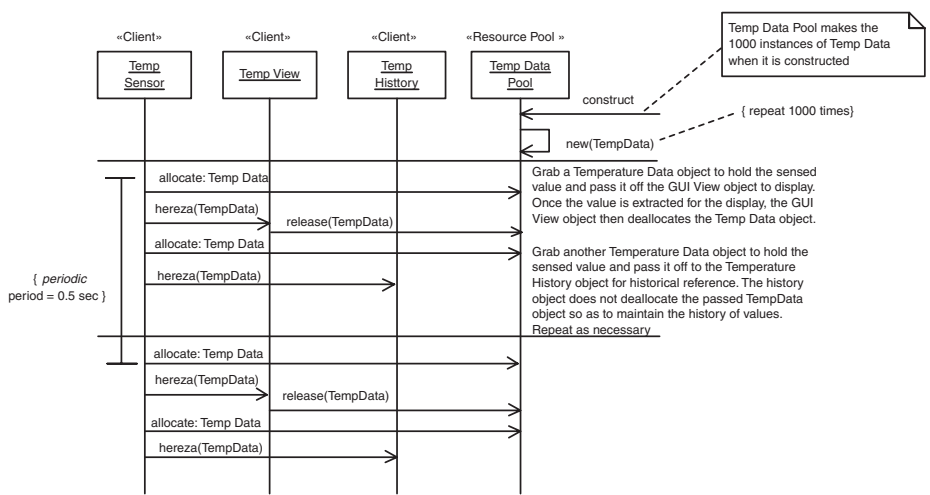
Figure 6-4a shows the object model for a system running a class model derived from the pattern shown in Figure 6-3. Figure 6-4b shows a scenario of the objects as they run. The first message shows the creation of the TempDataPool object, which in turn creates the

1000 TempData objects that it will manage. The other part of the object model shows three clients of the TempDataPool.

- *TempSensor* This is a thermometer that records the temperature every ½ second and in doing so, allocates a TempData object to store the information. This object reports the temperature (by passing a reference to the allocated TempData object), first to the



a. Structure



b. Scenario

Figure 6-4: *Pooled Allocation Pattern Example*

TempView, a GUI view object, and then to TempHistory, which manages the history of Temperature over the last several seconds.

- *TempView* This is a GUI object that displays the temperature to a user on a display. Once it has displayed the value, it releases the TempData object that was passed back to the pool
- *TempHistory* This maintains a history of the last ten seconds of temperature data. Thus, for the first 20 samples, it does not delete the TempData objects passed to it, but subsequently, it releases the oldest TempData object it owns when it receives a new one.

6.4 FIXED SIZED BUFFER PATTERN

Many real-time and embedded systems are complex enough to be unpredictable in the order in which memory must be allocated and too complex to allocate enough memory for all possible worst cases. Such systems would be relatively simple to design using dynamic memory allocation. However, many such systems in the real-time and embedded world must function reliably for long periods of time—often years or even decades—between reboots. That means that while they are complex enough to require dynamic random allocation of memory, they cannot tolerate one of the major problems associated with dynamic allocation: fragmentation. For such systems, the Fixed Sized Buffer Pattern offers a viable solution: fragmentation-free dynamic memory allocation at the cost of some loss of memory usage optimality.

6.4.1 Abstract

The Fixed Sized Buffer Pattern provides an approach for true dynamic memory allocation that does not suffer from one of the major problems that affect most such systems: memory fragmentation. It is a pattern supported by most real-time operating systems directly. Although it requires static memory analysis to minimize nonoptimal memory usage, it is a simple and easy to implement approach.

6.4.2 Problem

One of the key problems with dynamic memory allocation is memory fragmentation. Memory fragmentation is the random intermixing of free and allocated memory in the heap. For memory fragmentation to occur, the following conditions must be met.

- The order of memory allocation is unrelated to the order in which it is released.
- Memory is allocated in various sizes from the heap.

When these preconditions are met, then memory fragmentation will inevitably occur if the system runs long enough. Note that this is not a problem solely related to object-oriented systems, functionally decomposed systems written in C are just as affected as those written in C++.[2] The problem is severe enough that it will usually lead to system failure if the system runs long enough. The failure occurs even when analysis has demonstrated that there is adequate memory because if the memory is highly fragmented, there may be more than enough memory to satisfy a request, but it may not be in a contiguous block of adequate size. When this occurs, the memory allocation request fails even though there is enough total memory to satisfy the request.

6.4.3 Pattern Structure

There are two ways to fix dynamic allocation so that it does not lead to fragmentation: (1) correlate the order of allocation and deallocation, or (2) do not allow memory to be allocated in any but a few specific block sizes. The basic concept of the Fixed Sized Buffer Pattern is to *not* allow memory to be allocated in any random block size but to limit the allocation to a set of specific block sizes.

Imagine a system in which you can determine the worst case of the total number of objects needed (similar to computing the worst-case memory allocation) as well as the largest object size needed. If the entire heap was divided into blocks equal to the largest block

2. Although it should be noted that the problem is potentially even worse with Java because *all* objects in Java are allocated on the heap, whereas "automatic variable" objects are allocated on the stack in C++.

ever needed, then you could guarantee that if there is *any* memory available, then the memory request could be fulfilled.

The cost of such an approach is the inefficient use of available memory. Even if only a single byte of memory were needed, a worst-case block would be allocated, wasting most of the space within the block. If the object sizes were randomly distributed between 1 byte and the worst case, then overall, ½ of the heap memory would be wasted when the heap was fully allocated. Clearly, this is wasteful, but the advantage of this approach is compelling: There will *never* be failure due to the fragmentation of memory.

To minimize this waste, the Fixed Sized Buffer Pattern provides a finite set of fixed-sized heaps, each of which offers blocks of a single size. Static analysis of the system can usually reveal a reasonable allocation of memory to the various-sized heaps. Memory is then allocated from the smallest block-sized heap that can fulfill the request. This compromise requires more analysis at design time but allows the designer to "tune" the available heap memory to minimize waste. Figure 6-5 shows the basic Fixed Sized Buffer Pattern.

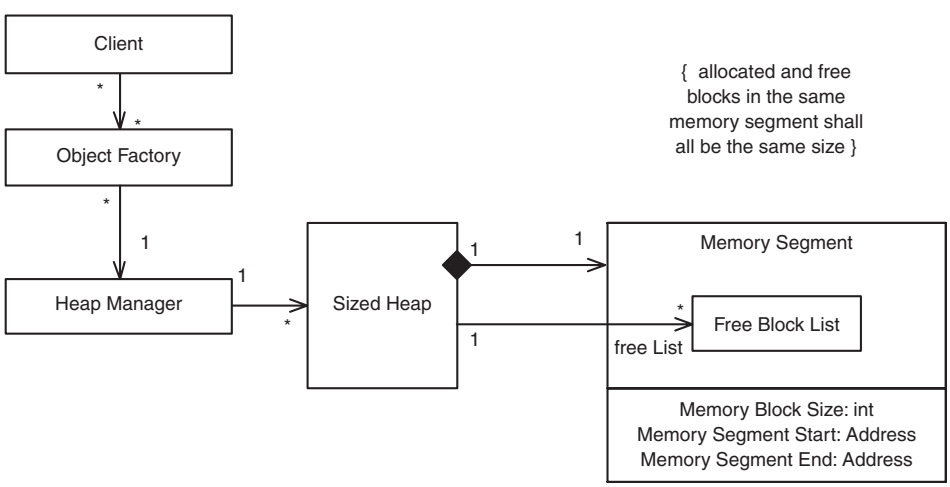


Figure 6-5: *Fixed Sized Buffer Pattern*

6.4.4 Collaboration Roles

- *Client*
  The *Client* is the user of the objects allocated from the fixed sized heaps. It does this by creating new objects as needed. In C++ this can be done by overwriting the global new and delete operators. In other languages, it may be necessary to explicitly call ObjectFactory.new() and ObjectFactory.delete().
- *Free Block List*
  This is a list of the unallocated blocks of memory within a single *Memory Segment.*
- *Heap Manager*
  This manages the sized heaps. When a request is made for a block of memory for an object, it determines the appropriate *Sized Heap* from which to request it. When memory is released, the *Heap Manager* can check the address for the memory block to determine which memory segment (and hence which free list) it should be added back into.
- *Memory Segment*
  A *Memory Segment* is a block of memory divided into equal-sized blocks, which may be allocated or unallocated. Only the free blocks must be listed, though. When memory is released, it is added back into the free list. The *Memory Segment* has attributes that provide the size of the blocks it holds and the starting and ending addresses for the *Memory Segment.*
- *Object Factory*
  The *Object Factory* takes over the job of allocation of memory on the heap. It does this by allocating an appropriately sized block of memory from one of the *Sized Heaps* and mapping the newly created object's data members into it and then calling the newly created object's constructor. Deleting an object reverses this procedure: The destructor of the object is called, and then the memory used is returned to the appropriate *Free Block List.*
- *Sized Heap*
  A *Sized Heap* manages the free and allocated blocks from a single *Memory Segment.* It returns a reference to the memory block when allocated, moves the block to the allocated list, and accepts a reference to an allocated block to free it. Then it moves *that* block to the free list so that subsequent requests can use it.

6.4.5 Consequences

The use of this pattern eliminates memory fragmentation. However, the pattern is suboptimal in terms of total allocated memory because more memory is allocated than is actually used. Assuming a random probability of memory size needs, on the average, half of the allocated memory is wasted. The use of *Sized Heaps* with appropriately sized blocks can alleviate some of this waste but cannot eliminate it. Many RTOSs support fixed sized block allocation out-of-the-box, simplifying the implementation.

6.4.6 Implementation Strategies

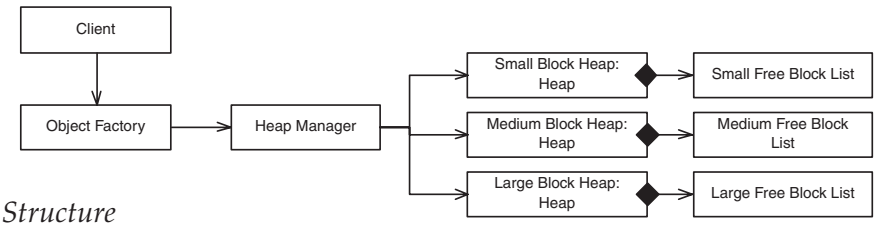
If you use an RTOS, then most of the pattern is provided for you by the underlying RTOS. In that case, you need to perform an analysis to determine the best allocation of your free memory into various-sized block heaps. If you rewrite global new and delete operators so that they use the *Object Factory* object rather than the default operators, then the use of sized heaps can be totally hidden from the clients.

6.4.7 Related Patterns

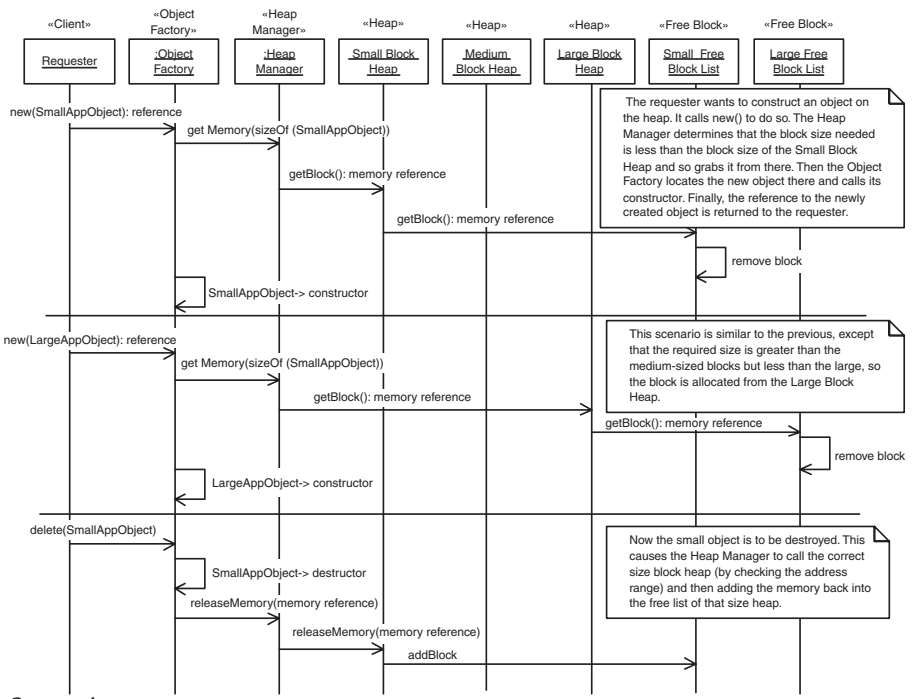
This pattern allows true dynamic allocation but without the problems of memory fragmentation. The issues of nondeterministic time are minimized but still present. However, there is no protection against memory leaks (clients neglecting to release inaccessible memory), inappropriate access to released memory, and the potentially critical issue of wasted memory. In simpler cases, the pooled allocation, or even static allocations patterns, may be adequate. If time predictability is not a major issue, then the Garbage Collector pattern may be a better choice, since it does protect against memory leaks.

6.4.8 Sample Model

Figure 6-6a shows a structural example of an instance of this pattern. In this case, there are three block-sized heaps: 128-byte blocks, 256-byte blocks, and 1024-byte blocks. Figure 6-6b presents a scenario in which a small object is allocated, followed by a larger object. After this, the smaller object is deleted.



a. Structure



b. Scenario

Figure 6-6: *Fixed Sized Buffer Pattern Example*

6.5 SMART POINTER PATTERN

In my experience over the last couple of decades leading and managing development projects implemented in C and C++, pointer problems are by far the most common defects and the hardest to identify.

They are common because the pointer metaphor is very low level and requires precise management, but it is easy to forget about when dealing with all possible execution paths. Inevitably, somewhere a pointer is destroyed (or goes out of scope), but the memory is not properly freed (a memory leak), memory is released but nevertheless accessed (dangling pointer), or memory is accessed but not properly allocated (uninitialized pointer). These problems are notoriously difficult to identify using standard means of testing and peer reviews. Tools such as Purify and LINT can identify "questionable practices," but sometimes they flag so many things it is virtually impossible to use the results. The Smart Pointer Pattern is an approach that is mechanistic (medium scope) rather than architectural (large scope) but has produced excellent results.

6.5.1 Abstract

Pointers are by far the most common way to realize an association between objects. The most common implementation of a navigable association is to use a pointer. This pointer attribute is dereferenced to send messages to the target object. The problem with pointers per se is that they are not objects; they are just data. Because they are not objects, the primitive operations you can perform on them are not checked for validity. Thus, we are free to access a pointer that has never been initialized or after the memory to which it points has been freed. We are also free to destroy the pointer without releasing the memory, resulting in the loss of the now no-longer-referenceable memory to the system.

The Smart Pointer Pattern solves these problems by making the pointer itself an object. Because a Smart Pointer is an object, it can have constructors and destructors and operations that can ensure that its preconditional invariants ("rules of proper usage") are maintained.

6.5.2 Problem

In many ways, pointers are the bane of the programmer's existence. If they weren't so incredibly useful, we would have discarded them a long time ago. Because they allow us to dynamically allocate, deallocate, and reference memory dynamically, they form an important

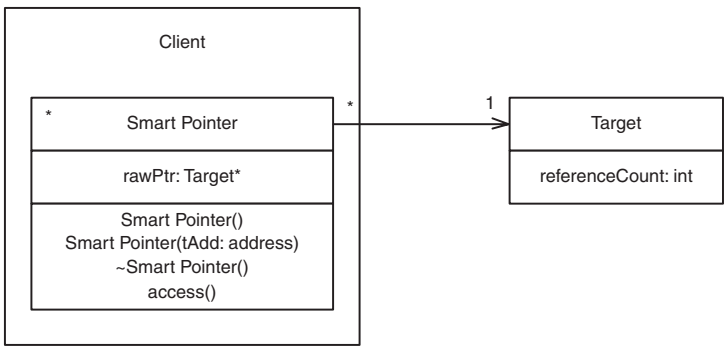
part of the programmer's toolkit. However, their use commonly results in a number of different kinds of defects.

- Memory leaks—destroying a pointer before the memory they reference is released. This means that the memory block is never put back in the heap free store, so its loss is permanent, at least until the system is rebooted. Over time, the available memory in the heap free store (that is, memory that can now be allocated by request) shrinks, and eventually the system fails because it cannot satisfy memory requests.
- Uninitialized pointer—using a pointer as if it was pointing to a valid object (or memory block) but neglecting to properly allocate the memory. This can also occur if the memory request is made but refused.
- Dangling pointer—using a pointer as if it was pointing to a valid object (or memory block) but *after* the memory to which it points has been freed.
- Pointer arithmetic defects—using a pointer as an iterator over an array of values but inappropriately. This can be because the pointer goes beyond the bounds of the array (in either direction), possibly stepping on memory allocated to other objects, or becoming misaligned, pointing into the middle of a value rather than at its beginning.

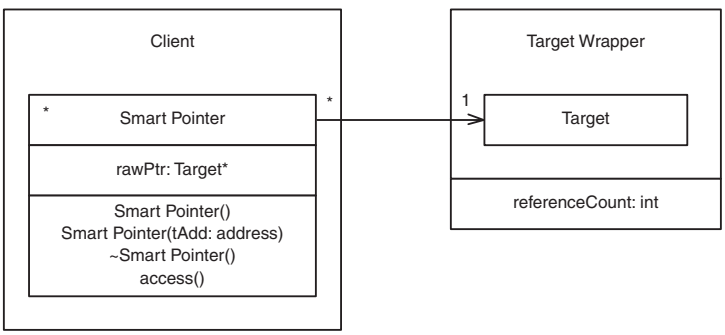
These problems arise because pointers are inherently *stupid.* They are only data values (addresses), and the operations defined on them are primitive and without checks on their correct use. If only they were *objects,* their operations could be extended to include validity checks and they could identify or prevent inappropriate use.

6.5.3 Pattern Structure

The basic solution of the Smart Pointer Pattern is to reify the pointer into an object. Once a pointer comes smart, or potentially smart, its operations can ensure that the preconditions of the pointer (it points to valid memory) are met. Figure 6-7a shows the simple structure of this pattern, and Figure 6-7b shows a common variant.



a. Basic



b. Wrapper Variant

Figure 6-7: *Smart Pointer Pattern*

6.5.4 Collaboration Roles

- *Client*
  The *Client* is the object that at the analysis level simply has an association to the *Target* object. If this is a bidirectional association, then *both* these objects have smart pointers to the other.
- *Smart Pointer*
  The *Smart Pointer* is an object that contains the actual pointer *(rawPtr)* as an attribute, as well as constructor, destructor, and access operations. The access operations will usually be realized by overriding pointer dereference operators ([] and →) in C++, to hide the fact that a smart pointer is being used. The *Target:: referenceCount* attribute keeps track of the number of smart pointers that are referring to the specific target object. It's important to know

this so you can determine when to destroy the dynamically created *Target.*

The *Smart Pointer* has two constructors. The default constructor creates a corresponding *Target* and sets *referenceCount* to the value 1. The second constructor initializes the *rawPtr* attribute to the value of the address passed in and increments the *Target::referenceCount.* The destructor decrements the *Target::referenceCount;* if it decrements to 0; then the *Target* is destroyed. In principle, the *Target::referenceCount* must be referred to by all *Smart Pointers* that point to the *same* object.

- *Target*
  The *Target* is the object providing the services that the *Client* wishes to access. In the basic form of the pattern (Figure 6-7a), *Target* also has a *reference count* attribute that tracks the number of *Smart Pointers* currently referencing it.
- *Target Wrapper*
  In the Smart Pointer Pattern variant in Figure 6-7b, the *Target* object is not at all aware of Smart Pointers or reference counts. The *Target Wrapper* object contains via composition, the *Target* object, and it owns the *referenceCount* attribute.

6.5.5 Consequences

This is a mechanistic-level design pattern; that means it optimizes individual collaborations. The main advantage of applying this pattern is that it is a simple means to ensure that objects are destroyed when they are no longer accessible—that is, when all references to them have been (or are being) destroyed. This requires some discipline on the part of the programmers. If the *Target* object is being referenced by *both* smart and raw pointers, then this pattern will break with potential catastrophic consequences. On the other hand, using the pattern can be codified into an easily checked rule: Use no raw pointers; that is, validate during code reviews.

To ensure robustness in the presence of multithreaded access to an object (*Smart Pointers* exist in multiple threads that reference the same *Target*), then care must be taken in the creation of constructors and destructors. The simplest way to handle them is to make them atomic (prevent task switching during the construction or destruction of a *Smart Pointer*). You can do this easily by making the first operation in

the constructor a call to the OS to prevent task switching (just don't forget to turn it back on when you're done!). The destructor must be similarly protected. Otherwise, there is a possibility that the object may be destroyed *after* you checked that it was valid and a *Smart Pointer* is now pointing to a *Target* that no longer exists.

Finally, there is one situation in which *Smart Pointers* may be correctly implemented but still may result in memory leakage. The *Smart Pointer* logic ensures that whenever there is no *Smart Pointer* pointing to a *Target,* the *Target* will be deleted. However, it is possible to define small cycles of objects that contain *Smart Pointers,* but the *entire cycle* cannot be accessed by the rest of the application. In other words, it is *still* possible to get a memory leak if the collaboration of objects has cycles in it. Figure 6-8 shows how this can happen.

Object *Obj3* and *Obj5* form a cycle. If *Obj2* and *Obj4* are destroyed, the reference counts associated with *Obj3* and *Obj5* decrement down to 1 rather than 0, and these two objects are unable to be referenced by the remainder of the application. Since their reference counts are greater than 1, they cannot be destroyed, but neither can the application invoke services of these objects because there are no references to these objects outside the cycle itself.

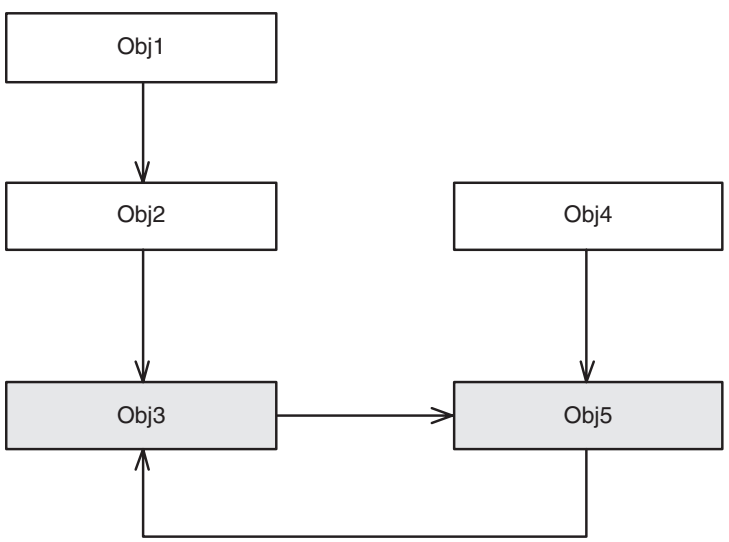


Figure 6-8: *Smart Pointer Cycles*

The easiest way to handle the problem is to ensure that no *Target* itself references another object that could ever reference the original. This can usually be deduced from drawing class diagrams of the collaborations and some object diagrams resulting from the execution of the class diagram. If cycles cannot be avoid, then it might be better to avoid using the Smart Pointer Pattern for those cycles specifically.

6.5.6 Implementation Strategies

This pattern is simple and straightforward to implement and should create no problems. If you desire a Smart Pointer Pattern that can handle cyclic object structures, then this can be solved at the cost of increased complexity and processing resource usage. A good discussion of these methods is provided in [2].

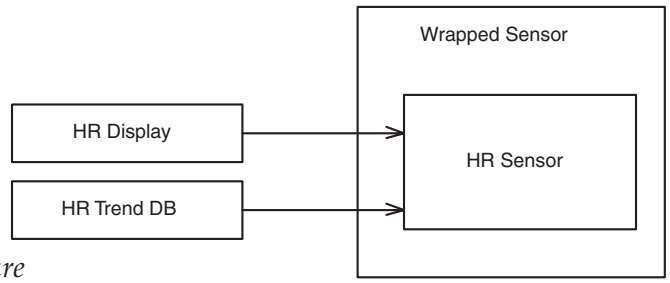
6.5.7 Related Patterns

There are more elaborate forms of the Smart Pointer in [2], although they are expressed as algorithms defined on the Smart Pointer rather than patterns per se, as it is here. When cycles are present but the benefits of the Smart Pointer Pattern (protection against pointer defects) are strongly desired, the Garbage Collector or Compacting Garbage Collector Patterns may be indicated.

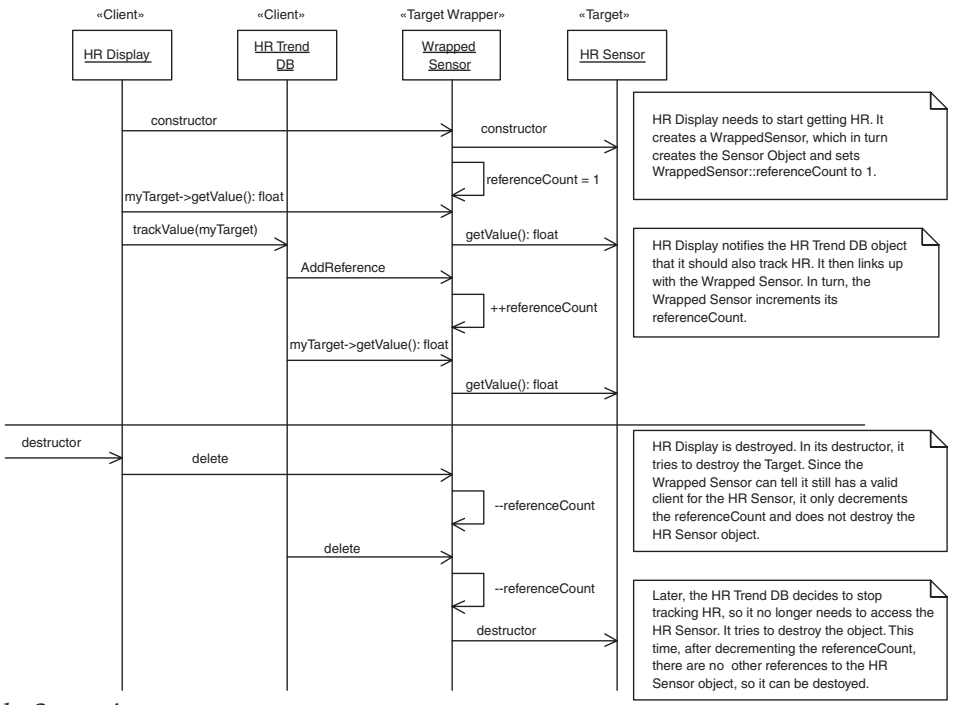
6.5.8 Sample Model

Figure 6-9 shows a simple application of this pattern. Two clients of the HR Sensor object exist: one object that displays the values and another that tracks values to do trending analysis. When the HR Display object runs, it creates an HR Sensor object via a Wrapped Sensor object. The HR Display object also notifies the HR Trend object to begin tracking the heart rate information (via the Wrapped Sensor object).

Later, the HR Display is destroyed. It calls the delete operation on the Wrapped Sensor class. The Wrapped Sensor decrements its reference count but does not delete the HR Sensor because the reference count is greater than zero (the HR Trend DB still has a valid reference to it). Later on, when the HR Trend DB removes the last pointer to the HR Sensor object, the HR Sensor object is finally deleted.



a. Structure



b. Scenario

Figure 6-9: *Smart Pointer Pattern Example*

6.6 GARBAGE COLLECTION PATTERN

Memory defects are among the most common and yet most difficult to identify errors. They are common because the programming languages provide very low access to memory but do not provide the means to identify when the memory is being accessed properly. This can lead to memory leaks and dangling pointers. The insidious aspect of these defects is that they tend to have global, rather than local, impact, so while they can crash the entire system, they leave no trace as to where the defect may occur. The Garbage Collection Pattern addresses memory access defects in a clean and simple way as far as the application programmer is concerned. The standard implementation of this pattern does not address memory fragmentation (see the Garbage Compactor Pattern to get that benefit), but it does allow the system to operate properly in the face of poorly managed memory.

6.6.1 Abstract

The Garbage Collection Pattern can eliminate memory leaks in programs that must use dynamic memory allocation. Memory leaks occur because programmers make mistakes about when and how memory should be deallocated. The solution offered by the Garbage Collection Pattern removes the defects by taking the programmer out of the loop—the programmer no longer explicitly deallocates memory. By removing the programmer, that source of defects is effectively removed. The costs of this pattern are run-time overhead to identify and remove inaccessible memory and a loss of execution predictability because it cannot be determined at design time when it may be necessary to reclaim freed memory.

6.6.2 Problem

The Garbage Collection Pattern addresses the problem of how we can make sure we won't have any memory leaks. Many high-availability or high-reliabilty systems must function for long periods of time without being periodically shut down. Since memory leaks lead to

unstable behavior, it may be necessary to completely avoid them in such systems. Furthermore, reference counting Smart Pointers (see Smart Pointer Pattern, earlier in this chapter) have the disadvantages that they require programmer discipline to use correctly and cannot be used when there are cyclic object references.

6.6.3 Pattern Structure

Figure 6-10 shows the pattern for what is called *Mark and Sweep* garbage collection. In Mark and Sweep, garbage collection takes place in two phases: a marking phase, followed by a reclamation phase. When objects are created, they are marked as *live objects.* The marking phase is begun in response to a low memory or an explicit request to perform garbage collection. In the marking phase, each of the *root* objects is searched to find all live objects. Objects that cannot be reached in this way are marked as dead. In the subsequent sweep phase, all the objects marked as dead are reclaimed. The garbage collector must stop normal processing before performing its duties, reducing the predictability of real-time systems.

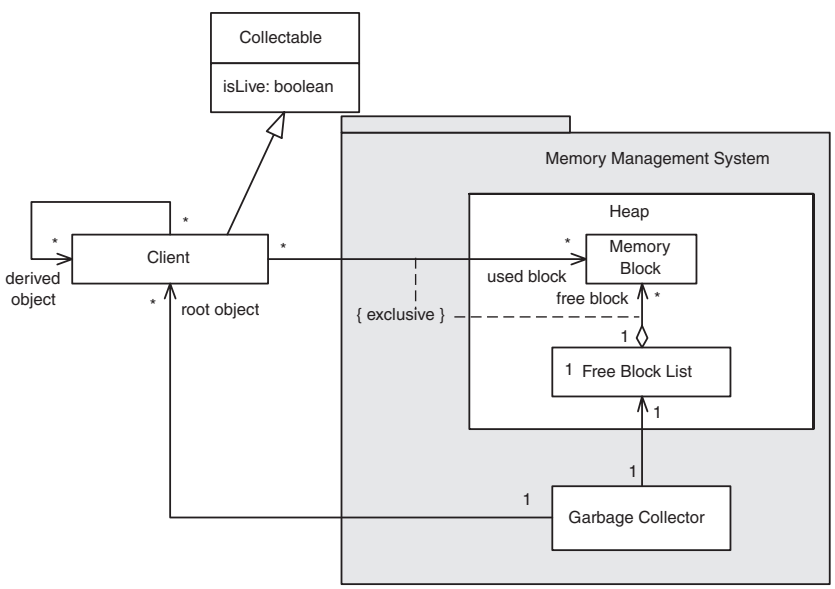


Figure 6-10: *Garbage Collection Pattern (Mark and Sweep)*

6.6.4 Collaboration Roles

- *Client*
  The *Client* is the user-defined object that allocates memory (generally, although not necessarily, in the form of objects). It is a subclass of *Collectable* and contains pointers to *derived objects,* allowing the *garbage collector* to search from the so-called *root objects* to all derived objects. When created, the object is marked as live with is *isLive* attribute, inherited from *Collectable.* On the second pass, all objects not marked as live are removed—that is, added back to the heap free memory.
- *Collectable*
  This is the base class for *Client,* and it provides the *isLive* Boolean attribute used during the garbage collection process.
- *Free Block List*
  A list of free blocks from which requests for dynamic memory are fulfilled.
- *Garbage Collector*
  The *Garbage Collector* manages the reclamation of memory by searching the object space starting with the root objects, looking at all blocks to ensure their liveness, and removing all those that are no longer live—in other words, those that cannot be reached in some fashion from a root or derived object.
- *Heap*
  The *Heap* is the owner of all the *Memory Blocks* and the *Free Block List.*
- *Memory Block*
  The *Memory Block* is just like it sounds: a block (normally of arbitrary size, in which case it contains a *size* parameter) of memory, usually, although not necessarily, associated with an object. *Memory Blocks* may be pointed to by the *Free Block List,* in which case they are not currently being pointed to by a *Client* or may be pointed to by a *Client,* in which case they are not pointed to by the *Free Block List.* Hence, the {exclusive} constraint on the relations to those classes.

6.6.5 Consequences

This architectural pattern removes the vast majority of memory-related problems by effectively eliminating memory leaks and dan-

gling pointers. It is *still* possible to do bad pointer arithmetic, but they account for a relatively small number of defects compared to the first two memory-management defects. Further, there is much less need to do pointer arithmetic when memory is collected and managed for you. The use of this pattern removes these user defects by eliminating reliance on the user to correctly deallocate memory. The garbage collector runs episodically when a "low-memory" condition is detected and deallocates all inaccessible memory. Following garbage collection, all non-NULL pointers and references are valid, and all unreferenced memory is freed. The pattern correctly identifies and handles circular references, unlike the *Smart Pointer Pattern.*

Since this pattern uses a two-pass mark-and-sweep algorithm, it takes a nontrivial amount of time to do a complete memory cleanup. This has two negative consequences. First, considerable processing time and effort may be required to perform the memory reclamation, and it cannot, in general, be predicted how much time and effort will be required. Second, because it is done in response to a low-memory condition (such as a request for memory that cannot be fulfilled), *when* it occurs is likewise unpredictable. This means that while the approach scales up to large-scale system well in terms of managing complexity, it may not work well in systems with hard real-time constraints.

Another difficulty with this approach is that it does not affect *fragmentation,* a key problem in systems that must run for long periods of time. Memory will be reclaimed properly, but it will result in fragmented free space. This means that although enough memory may be free to fulfill a request for memory, there may not be a single contiguous block available to fulfill the request. In fact, with this pattern (and most other memory management patterns) fragmentation increases monotonically the longer the system runs. The *Garbage Compactor Pattern,* described in the next section, addresses this need.

6.6.6 Implementation Strategies

As with all such patterns, the simplest way to use this pattern is to *buy* it. Some languages, such as Java, provide memory management systems that use garbage collection out of the box. Where such languages are not available, the implementation of such a memory management schema can be done easily in the naïve case and with more difficulty in the more optimized case.

A common optimization is to allow the application objects to explicitly request a garbage collection pass when it is convenient for the application, such as when the application is quiescent. For example, if the concurrency model is managed by a cyclic executive (see the *Cyclic Executive Pattern*), then at the end of the cycle, if there is sufficient time, a memory cleanup may be performed. If it cannot be guaranteed that the garbage collection will complete before the next cycle occurs, the garbage collector may be preemptable, so that it is stopped prior to completion, allowing the application to run and meet its deadlines. When using such a strategy, be careful that you do not assume that the object marked as live on the previous pass has remained live.

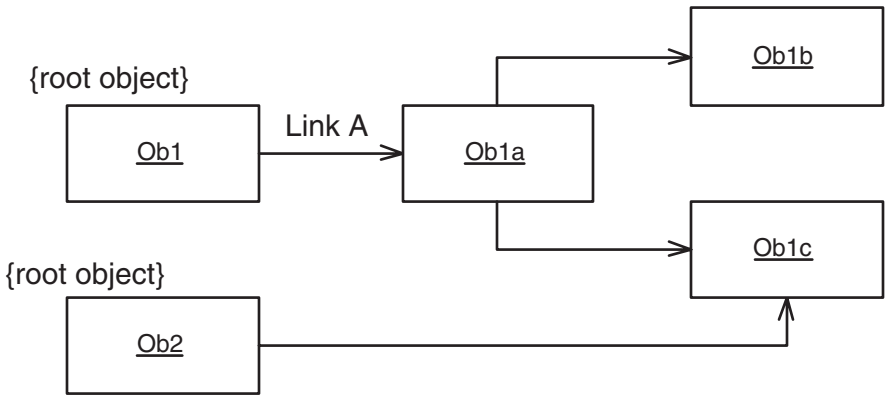
6.6.7 Related Patterns

When the inherent unpredictability of the system cannot be tolerated, another approach, such as the *Smart Pointer Pattern* or *Fixed Size Allocation Pattern,* should be used. To eliminate memory fragmentation, the *Garbage Compactor Pattern* works well. The Static Allocation Pattern does not have fragmentation, and the Fixed Sized Allocation Pattern does its best to minimize fragmentation. The Smart Pointer Pattern cannot handle circular references, but the Garbage Collection and Garbage Compactor Patterns do.

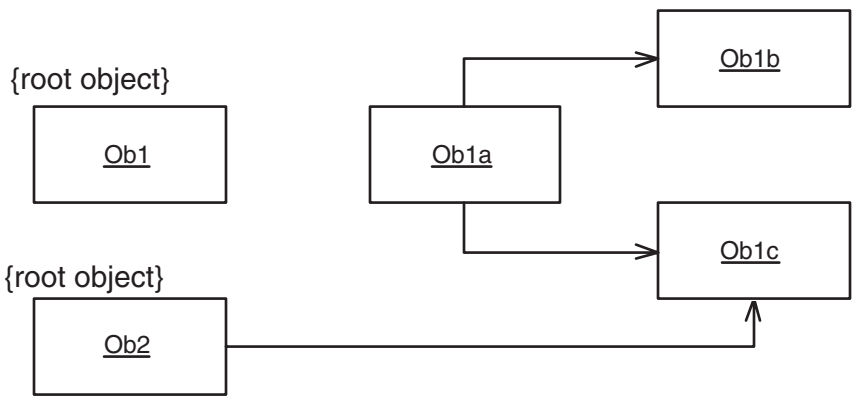
6.6.8 Sample Model

Figures 6-11a, b, and c show instance snapshots of allocated memory. In the figures, Ob1 and Ob2 are root objects, known to the Garbage Collector. These might be, for example, initial instances created in the main() of the application. Objects Ob1a, Ob1b, and Ob1c are *derived* objects that can be found by traversing the links from Ob1 and Ob2 in Figure 6-11a. In Figure 6-11b, the link from Ob1 to Ob1a is broken. This means that Ob1a and Ob1b are no longer accessible to the system, since they cannot be found through a traversal of links from root objects. Note that Ob1c remains accessible via the link through the root object Ob2. In Figure 6-11c, we see that the memory used by Ob1a and Ob1b is reclaimed, and only accessible objects remain.

Figure 6-12 shows how the garbage collector proceeds. First, every object in the heap is marked as dead. Subsequently, each root



a. Instance Model Snapshot 1—Starting Instance Model



b. Instance Model Snapshot 2—Before Garbage Collection

c. Instance Model Snapshot 3—After Garbage Collection

Figure 6-11: *Garbage Collection Pattern*

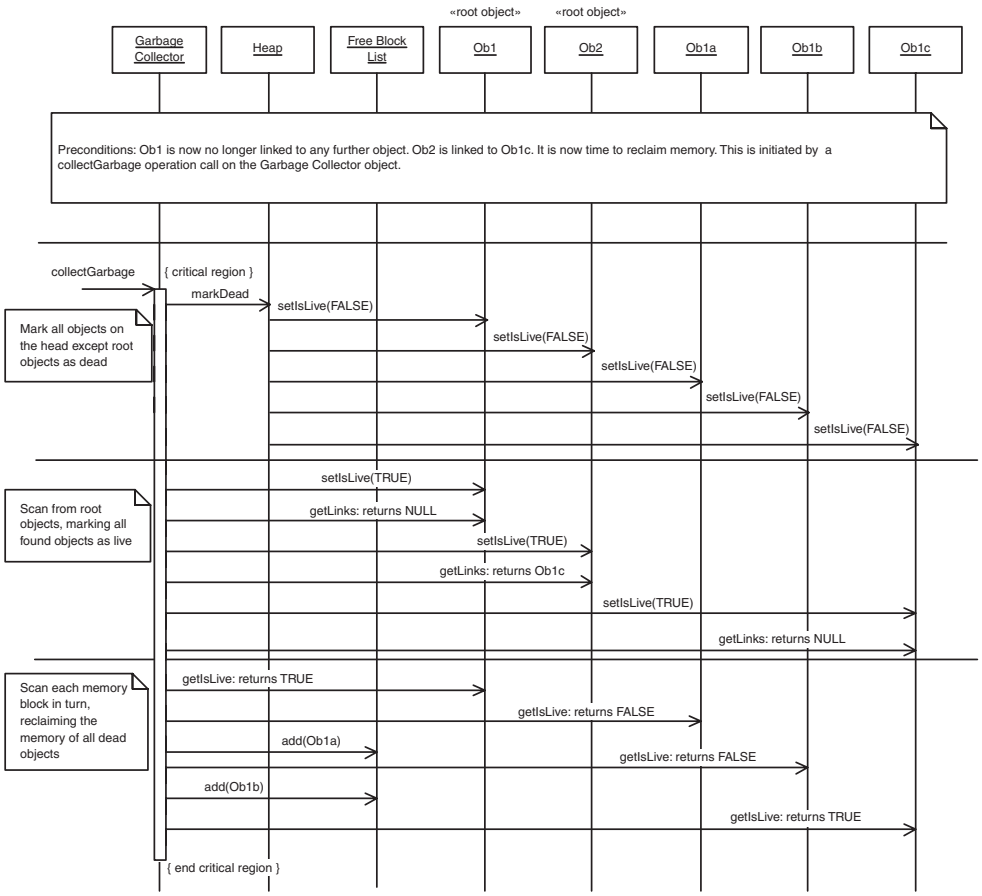


Figure 6-12: *Garbage Collection Pattern Example Scenario*

object is searched. As objects are found, they are marked as live by setting the *isLive* attribute to TRUE. In the second pass, the garbage collector does a linear search through all the allocated memory blocks, removing those that are still marked dead. This is done by first calling the object's destructor (if one exists) and then adding the object to be deleted to the *Free Block List*.

6.7 GARBAGE COMPACTOR PATTERN

6.7.1 Abstract

The Garbage Compactor Pattern is a variant of the Garbage Collection Pattern that also removes memory fragmentation. It accomplishes this goal by maintaining two memory segments in the heap. During garbage collection, live objects are moved from one segment to the next, so in the target segment, the objects are juxtapositioned adjacent to each other. The free memory in the segment then starts out as a contiguous block.

6.7.2 Problem

The Garbage Collection Pattern solves the problem of programmers forgetting to release memory by every so often finding inaccessible objects and removing them. The pattern has a couple of problems, including maintaining the timeliness of the application and fragmentation. Fragmentation means that the free memory is broken up into noncontiguous blocks. If the application is allowed to allocate blocks in whatever size they may be needed, most applications that dynamically allocate and release blocks will eventually get into the situation where although there is enough total memory to meet the allocation request, there isn't a single contiguous block large enough. At this point, the application fails. Garbage collection per se does not solve this problem just because it finds and removes dead objects. To compact memory, the allocated blocks must be moved around periodically to leave the free memory as a single, large contiguous block.

6.7.3 Pattern Structure

Figure 6-13 shows the structural pattern for *copying garbage collection.* A copying garbage collector works in a single phase. It is initiated in the same way as a mark and sweep garbage collector. As it searches the object space from the root objects, it copies all the live objects it finds to another memory space. It is more efficient than mark and sweep because a single phase is all that is necessary, and it also

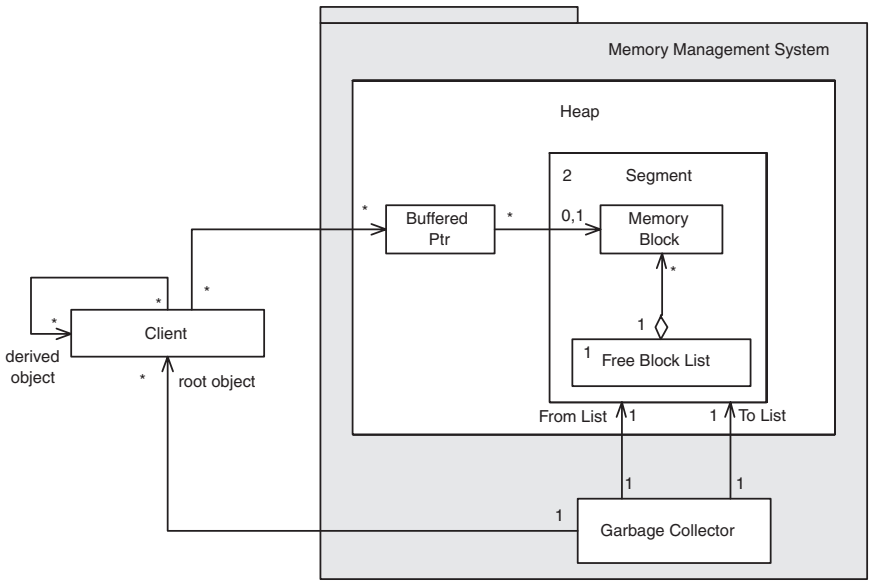


Figure 6-13: *Garbage Compactor Pattern*

eliminates memory fragmentation because it compacts memory as it moves the referenced objects. The copying garbage collector must update object references as it moves objects. This pattern requires twice as much memory as the mark and sweep pattern because it must always maintain both a *from space* and a *to space* (although they will reverse roles on subsequent invocations of the garbage collector). In addition, a copying garbage collector requires that user objects reference heap objects via double buffering—that is, their pointers must point to pointers owned by the heap. This allows the garbage collector to update its internal pointer references to the actual location of the heap object as it moves around. Either that, or the garbage collector must have references to the user objects and modify their pointers *in vivo* when the referenced heap object is relocated.

6.7.4 Collaboration Roles

- *Buffered Ptr*
  The *Buffered Ptr* is an intermediary between one object's reference to the object being referenced. This is required because the

Garbage Compactor must update the references to the objects as it moves them. This is far simpler if the actual references to the memory location are under its control rather than the object's clients.

- *Client*
  The *Client* is the user-defined object that allocates memory (generally, although not necessarily, in the form of objects). During the collection process, root objects are searched, and objects found during the search are moved as they are found.
- *Free Block List*
  A list of free blocks from which requests for dynamic memory are fulfilled.
- *Garbage Compactor*
  The *Garbage Compactor* manages the reclamation of memory by searching the object space starting with the root objects and copying the found objects from the current memory segment to the target memory segment. Dead objects are not copied and are thus automatically reclaimed.
- *Heap*
  The *Heap* is the owner of all the *Segments* and *Buffered Ptrs.* The heap fills all memory requests from the currently *active segment* and ignores the *inactive segment.* The roles the two segments play swap each time the *Garbage Compactor* performs the garbage compaction process.
- *Memory Block*
  The *Memory Block* is just like it sounds: a block (normally of arbitrary size, in which case it contains a *size* parameter) of memory usually, although not necessarily, associated with an object. *Memory Blocks* may be pointed to by the *Free Block List*—in which case they are not currently being pointed to by a *Client*—or may be pointed to by a *Client* (via a *Buffered Ptr*)—in which case they are not pointed to by the *Free Block List.*
- *Segment*
  The heap maintains two segments that are alternatively swapped in terms of use. The one in use is called the *active segment,* and the other is called the *inactive segment.* From the *Garbage Compactor's* point of view, which is taken during the compaction process, one

is the *from segment* and the other is the *target segment*. The *Active Segment* is used to fill all requests for dynamic memory allocation. During compaction, all live objects are copied from the *from segment* to the *target segment*, and then the *target segment* is set to be the *Heap's active segment.*

6.7.5 Consequences

The most noticeable consequence of using the Garbage Compactor Pattern is that the programmers don't need to deallocate their objects (the Garbage Compactor does it for them) and that fragmentation does not monotonically increase the longer the system runs. Fragmentation increases for a while but is reduced to zero when the Garbage Compactor runs. Since the Garbage Compactor runs when a request for memory cannot be satisfied, this means that if there is enough total memory to meet a request, the request will always be satisfied.

Another highly noticeable consequence of this pattern, at least in terms of memory requirements, is that the pattern requires twice as much memory as the *Garbage Collection Pattern.* Assuming that each *Segment* is large enough to hold the worst-case memory needs at any moment in time, the pattern requires two such segments. This makes this approach inappropriate when there are tight memory size requirements.

Doing pointer arithmetic with the Garbage Compactor Pattern is a chancy thing for a number of reasons. However, since the main reason for doing pointer arithmetic is to manage memory, this should not cause many difficulties.

Of course, the length of time necessary to run the compactor is an issue for any application in which timeliness is a concern. There is a small amount of overhead for the double buffering of the pointers, but the major timeliness impact comes from the time and cycles necessary to identify the live objects and copy them to the target memory segment. This pattern requires more CPU cycles to run than the Garbage Collection Pattern because of the overhead of copying objects, but with care, it may be possible to run the garbage collector piecemeal to implement an incremental garbage compactor at the cost of recomputing which objects in the From segment are still live.

Because the reclaimed objects are not destroyed per se, their destructors are not called. If there are finalizing behaviors required of the objects (other than the normal release of memory), then the programmer must still manually ensure these behaviors are invoked before removing their references to the objects.

6.7.6 Implementation Strategies

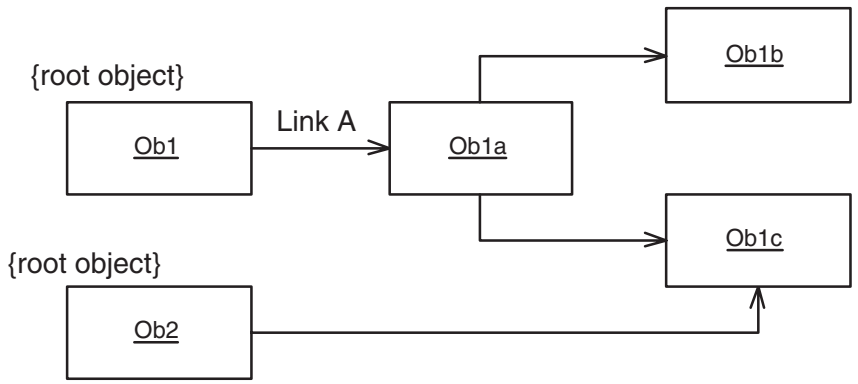
There are a number of small details to be managed by the memory management system in this pattern. The use of *Buffered Ptrs* allows the *Garbage Compactor* to move the objects and then update the location in a single place. If the source language is interpreted, such as Java, then the virtual machine can easily manage the double pointer referencing required of the client objects (in other words, their pointers are to *Buffered Ptrs,* which ultimately point to the actual memory used). If the source language produces native code, then the *new* operator must be rewritten to not only allocate the *Memory Block* per se but also create a *Buffered Pts* as well.

6.7.7 Related Patterns

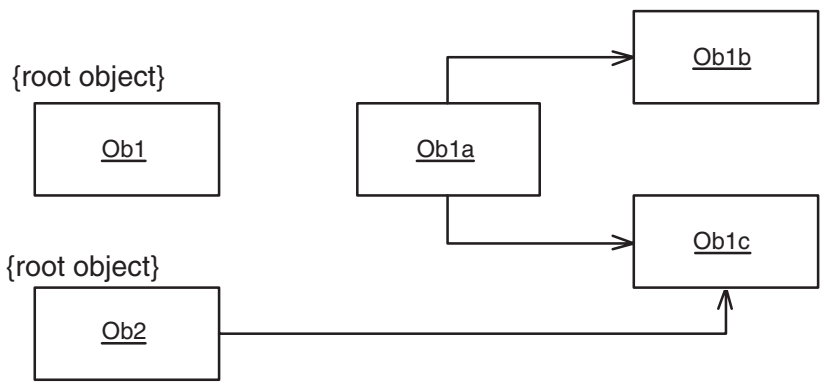
As with the Garbage Collection Pattern, this pattern can seriously impact the predictability of timeliness of systems using it. When timeliness is a primary concern, the Static Allocation, Fixed Sized Buffer, or Smart Pointer Patterns may be better. The Garbage Collection Pattern has the benefit of removing memory leaks, and it requires less memory than the Garbage Compactor Pattern, but it doesn't address memory fragmentation. The Static Allocation and Fixed Sized Allocation Patterns remove or reduce fragmentation but are not immune to memory leaks.

6.7.8 Sample Model

The example shown in Figure 6-14 is the same as for the previous pattern. Figure 6-15 shows a scenario of the system as it collects the garbage. We see that when the Garbage Collector is started, it first requests Segment 2 (the target segment) to initialize itself so that it is ready to begin copying memory into itself. Because it always starts empty, there is no fragmentation within the segment



a. Instance Model Snapshot 1—Starting Instance Model



b. Instance Model Snapshot 2—Before Garbage Collection

c. Instance Model Snapshot 3—After Garbage Collection

Figure 6-14: *Garbage Compactor Pattern*

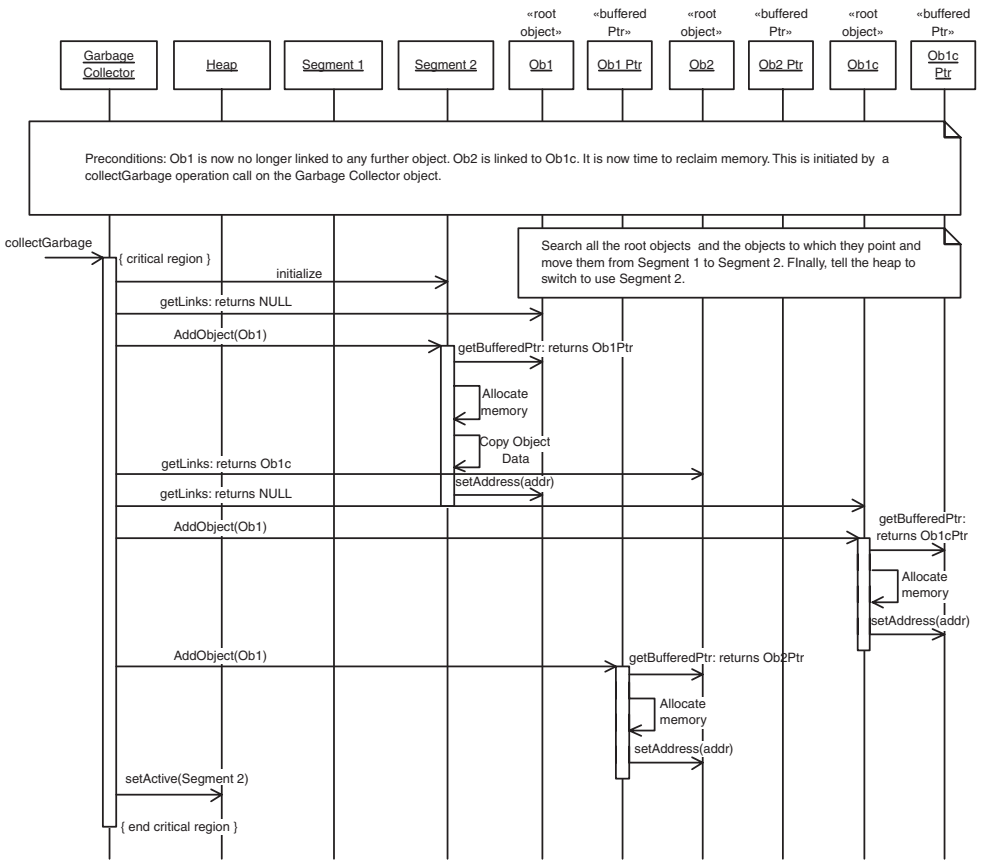


Figure 6-15: *Garbage Compactor Pattern Example Scenario*

as memory blocks are allocated one after another in a contiguous fashion.

Then the garbage collector searches the root objects. As it finds a root object, it asks if it has any valid links. First, in the case of Ob1, it finds two valid links, so the Garbage Collector can pass the object Ob1 off to the AddObject operation of the target segment (Segment 2). The segment, in turn, gets the location of the buffer pointer for the memory, allocates a new memory block to store Ob1's data, and then copies the object from the original segment. Finally it updates Obj1's buffered pointer to point to its data's new location.

6.8 References

[1] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA: Addison-Wesley, 1995.

[2] Jones, R., and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management,* West Sussex, England: John Wiley & Sons, 1996.

[3] Noble, J., C. Weir. *Small Memory Software: Patterns for Systems with Limited Memory,* Reading, MA: Addison-Wesley, 2001.

Chapter 7

Resource Patterns

The following patterns are presented in this chapter.

- Critical Section Pattern: Uses resources run-to-completion
- Priority Inheritance Pattern: Limits priority inversion
- Highest Locker Pattern: Limits priority inversion
- Priority Ceiling Pattern: Limits priority inversion and prevents deadlock
- Simultaneous Locking Pattern: Prevents deadlock
- Ordered Locking Pattern: Prevents deadlock

7.1 Introduction

One of the distinguishing characteristics of real-time and embedded systems is the concern over management of finite resources. This chapter provides a number of patterns to help organize, manage, use, and share such resources. There is some overlap of concerns with the patterns in this and other chapters. For example, the Smart Pointer Pattern provides a robust access to resources of a certain type: those that are dynamically allocated. However, that pattern has already been discussed in Chapter 6. Similarly, the synchronization of concurrent threads may be thought of as resource management, but it is dealt with using the Rendezvous Pattern from Chapter 5. This chapter focuses on patterns that deal with the sharing and management of resources themselves and not the memory they use. To this end, we'll examine a number of ways to manage resources among different, possibly concurrent, clients.

A *resource,* as used here, is a thing (an object) that provides services to clients that have finite properties or characteristics. This definition is consistent with the so-called Real-Time UML Profile [1], where a resource is defined as follows.

> An element that has resource services whose effectiveness is represented by one or more Quality of Service (QoS) characteristics.

The QoS properties are the quantitative properties of the resource, such as its capacity, execution speed, reliability, and so on. In real-time and embedded systems, it is this quantifiable finiteness that must be managed. For instance, it is common for a resource to provide services in an atomic fashion; this means that the client somehow "locks" the resource while it needs it, preventing other clients from accessing that resource until the original client is done. This accomplishes the more general purpose of *serialization* of resource usage, crucial to the correct operation in systems with concurrent threads. This is often accomplished with a mutex semaphore (see the Guarded Call Pattern in Chapter 5) or may be done by serializing the requests themselves (see the Message Queuing Pattern, also in Chapter 5).

The management of resources with potentially many clients is one of the more thorny aspects of system design, and a number of patterns have evolved or been designed over time to deal specifically with just that.

The first few patterns (Priority Inheritance, Highest Locker, Priority Ceiling) address the schedulability of resources in a priority-based preemptive multitasking environment, which can be a major concern for real-time systems design. In static priority scheduling approaches (see, for example, the Static Priority Pattern in Chapter 5), the priorities of the tasks are known at design time. The priority of the task determines which tasks will run preferentially when multiple tasks are ready to run—the highest-priority task that is ready. This makes the timing analysis of such systems very easy to compute, as long as certain assumptions are not violated too badly. These assumptions are the following.

- Tasks are periodic with the deadlines coming at the end of the periods.
- Infinite preemptibility—a lower-priority task can be preempted immediately when a higher-priority task becomes ready to run.
- Independence—tasks are independent from each other.

When these conditions are true, then the following standard rate monotonic analysis formula may be applied.

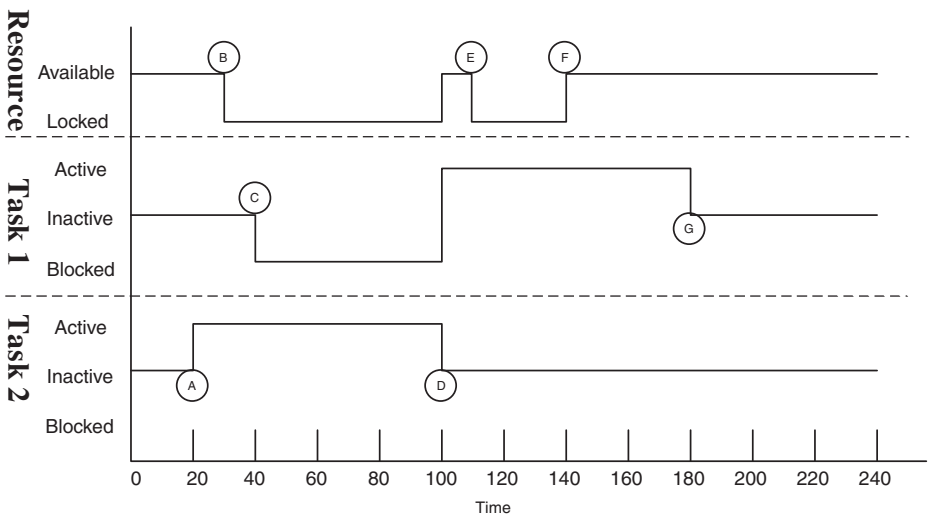
$$\sum_{n} \frac{C_j}{T_j} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

Note that it is "2 raised to the power of (1/n)", where C_j is the worst-case amount of time required for task j to execute, T_j is its period, and n is the number of tasks. [2], [3] If the inequality is true, then the system is *schedulable*—that is, the system will *always* meet its deadlines. Aperiodic tasks are generally handled by assuming they are periodic and using the minimum arrival time between task invocations as the period, often resulting in an overly strong but sufficient condition. The assumption of infinite preemptibility is usually not a problem if the task has very short critical sections during which it cannot be preempted—short with respect to the execution and period times. The problem of independence is, however, much stickier.

If resources are sharable (in the sense that they can permit simultaneous access by multiple clients), then no problem exists. However many, if not most, resources cannot be shared. The common solution to this problem was addressed in the Guarded Call Pattern of Chapter 5 using a mutual-exclusion semaphore to serialize access to the resource. This means that if a low-priority task locks a resource and then a higher-priority task that needs the resource becomes ready to run, it must *block* and allow the low-priority task to run until it can release the resource so that the

higher-priority task can run. A simple example of this is shown in the timing diagram in Figure 7-1.

In the figure, Task 1 is the higher-priority task. Since Task 2 runs first and locks the resource, when Task 1 is ready to run, it cannot because the needed resource is unavailable. It therefore must block and allow Task 2 to complete its use of the resource. During the period of time between marks C and D, Task 1 is said to be *blocked.* A task is blocked when it is prevented from running by a lower-priority task. This can only occur when resources are shared via mutual exclusion.



Legend:
Task 1 is High Priority. Task 2 is Low Priority.
A: Task 2 is ready to run and starts.
B: Task 2 locks the Resource.
C: Task 1 is ready to run but needs the Resource. It is blocked and must allow Task 1 to complete.
D: Task 2 completes and releases the Resource. Task 1 now unblocks and runs.
E: Task 1 locks the Resource.
F: Task 1 releases the Resource.
G: Task 1 completes.

Figure 7-1: *Task Blocking*[1]

1. A note about notation. These are *timing diagrams.* They show linear time along the X-axis and "state" or "condition," a discrete value, along the Y-axis. For more information, see [4].

The problem with blocking is that the analysis of the timeliness becomes more difficult. When Task 1 is blocked, the system is said to be in a state of *priority inversion* because a lower-priority task has the thread focus even though a higher-priority task is ready to run. One can imagine third and fourth tasks of intermediate priority that don't share the resource (and are therefore able to preempt Task 2) running and preempting Task 2, thereby lengthening the amount of time before Task 2 releases the resource and allowing Task 1 to run. Because an arbitrary number of tasks can be fit in the priority scheme between Task 1 and Task 2, this problem is called *unbounded priority inversion* and is a serious problem for the schedulability of tasks. Figure 7-2 illustrates this problem by adding intermediate-priority Tasks X and Y to the system. Note that for some period of time, Task 1, the highest-priority task in the system, is blocked by *all three* remaining tasks.

To compute the schedulability for task sets with blocking, the modified RMA inequality is used.

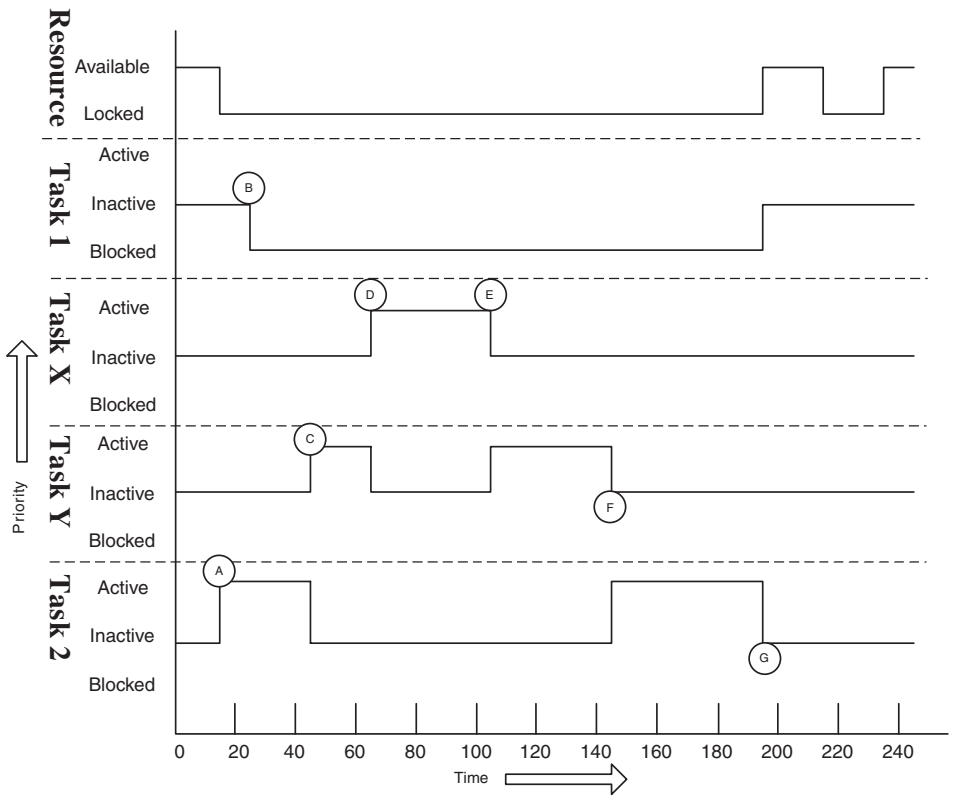
$$\sum_j \frac{C_j}{T_j} + \max\left(\frac{B_1}{T_1}, \cdots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n\left(2^{\frac{1}{n}} - 1\right)$$

where B_j is the blocking time for task *j*—that is, the worst-case time that the task can be prevented from execution by a lower-priority task owning a needed resource. The problem is clear from the inequality—unbounded blocking means unbounded blocking time, and nothing useful can be said about the ability of such a system to meet its deadlines.

Unbounded priority inversion is a problem that is addressed by the first three patterns in this chapter. Note that priority inversion is a necessary consequence of resource sharing with mutual exclusion locking, but it can be bounded using these patterns.

These first three patterns solve, or at least address, the problem of resource sharing for schedulability purposes, but for the most part they don't deal with the issue of deadlock. A deadlock is a condition in which clients of resources are waiting for conditions to arise that cannot in principle ever occur. An example of deadlock is shown in Figure 7-3.

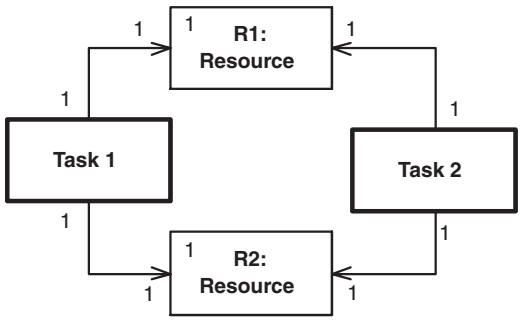
In Figure 7-3, there are two tasks, Task 1 and Task 2, that share two resources, R1 and R2. Task 1 plans to lock R2 and then lock R1 and release them in the opposite order. Task 2 plans to lock R1 and then R2 and release them in the reverse order. The problem arises when Task 1 preempts Task 2 when it has a single resource (R1) locked. Task 1 is a higher



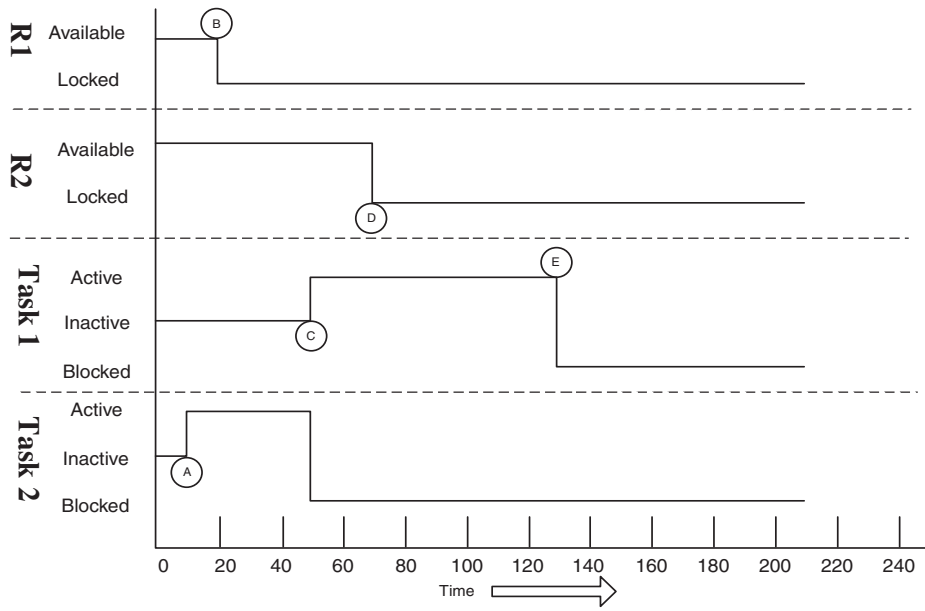
Legend:
Priorities: Task 1 > Task X > Task Y > Task 2
A: Task 2 is ready to run and starts.
B: Task 1 is ready to run but needs the Resource. It is blocked and must allow Task 2 to complete.
C: Task Y, which is a higher priority than Task 2, is ready to run. Since it doesn't need the resource, it preempts Task 2. Task 1 is now effectively blocked by both Task 2 and Task Y.
D: Task X, which is a higher priority than Task Y, is ready to run. Since it doesn't need the resource, it preempts Task Y. Task 1 is now effectively blocked by 3 tasks.
E: Task X completes, allowing Task Y to resume.
F: Task Y completes, allowing Task 2 to resume.
G: Task 2 (finally) completes and releases the resource, allowing Task 1 to access the resource.

Figure 7-2: *Unbounded Task Blocking*

priority, so it can preempt Task 1, and it doesn't need a currently locked resource, so things are fine. It goes ahead and locks R2. Now it decides that it needs the other resource, R1, which, unfortunately is locked by the



a. Deadlocked Class Structure



Legend:
Priority: Task 1 > Task 2
A: Task 2 runs with the intent of locking R1, then R2.
B: Task 2 locks R1 and is about to lock R2, when...
C: Task 1 runs, preempting Task 2, with the intent of locking R2, then R1.
D: Task 2 locks R2.
E: Now Task 2 needs to lock R1, but R1 is already locked. So Task 2 must block until Task 1 can release R1. However, Task 1 cannot run to release R1 because it needs R2, which is locked by Task 2.

b. Timing

Figure 7-3: *Deadlock*

blocked task, Task 2. So Task 1 cannot move forward and must block in order to allow Task 2 to run until it can release the now needed resource (R1). So Task 2 runs but finds that it now needs the other resource (R2) owned by the blocked Task 1. At this point, each task is waiting for a condition that can never be satisfied, and the system stops.

In principle, a deadlock needs the following four conditions to occur.

1. Mutual exclusion (locking) of resources
2. Resources are held (locked) while others are waited for
3. Preemption while holding resources is permitted
4. A circular wait condition exists (for example, P1 waits on P2, which waits on P3, which waits on P1)

The patterns for addressing deadlock try to ensure that at least one of the four necessary conditions for deadlock cannot occur. The Simultaneous Locking Pattern breaks condition 2, while the Ordered Locking Pattern breaks condition 4. The Priority Ceiling Pattern is a pattern that solves both the scheduling problem and the deadlock problem.

7.2 CRITICAL SECTION PATTERN

The Critical Section Pattern is the simplest pattern to share resources that cannot be shared simultaneously. It is lightweight and easy to implement, but it may prevent high priority tasks, even ones that don't use *any* resources, from meeting their deadlines if the critical section lasts too long.

7.2.1 Abstract

This pattern has been long used in the design of real-time and embedded systems whenever a resource must have at most a single owner at any given time. The basic idea is to lock the Scheduler whenever a resource is accessed to prevent another task from simul-

taneously accessing it. The primary advantage of this pattern is its simplicity, both in terms of understandability and in terms of implementation. It becomes less applicable when the resource access may take a long time because it means that higher-priority tasks may be blocked from execution for a long period of time.

7.2.2 Problem

The main problem addressed by the Critical Section Pattern is how to robustly share resources that may have, at most, a single owner at any given time.

7.2.3 Pattern Structure

Figure 7-4 shows the basic structural elements in the Critical Section Pattern.

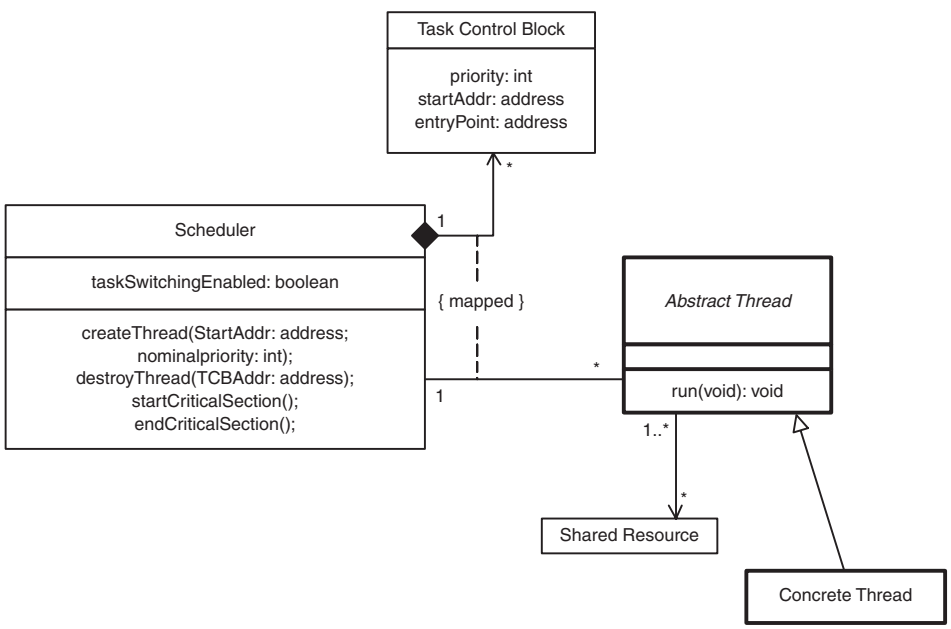


Figure 7-4: *Critical Section Pattern*

7.2.4 Collaboration Roles

- *Abstract Thread*
  The *Abstract Thread* class is an abstract (noninstantiable) superclass for *Concrete Thread. Abstract Thread* associates with the *Scheduler.* Since *Concrete Thread* is a subclass, it has the same interface to the *Scheduler* as the *Abstract Thread.* This enforces interface compliance. The *Abstract Thread* is an «active» object, meaning that when it is created, it creates an OS thread in which to run. It contains (that is, it has composition relations with) more primitive application objects that execute in the thread of the composite «active» object.

- *Concrete Thread*
  The *Concrete Thread* is an «active» object most typically constructed to contain passive "semantic" objects (via the composition relation) that do the real work of the system. The *Concrete Thread* object provides a straightforward means of attaching these semantic objects into the concurrency architecture. *Concrete Thread* is an instantiable subclass of *Abstract Thread.*

- *Scheduler*
  This object orchestrates the execution of multiple threads based on some scheme requiring preemption. When the «active» *Thread* object is created, it (or its creator) calls the *createThread* operation to create a thread for the «active» object. Whenever this thread is executed by the *Scheduler,* it calls the StartAddr address (except when the thread has been blocked or preempted—in which case it calls the *EntryPoint* address).

  In this pattern, the *Scheduler* has a Boolean attribute called taskSwitchingEnabled and two operations, startCriticalSection() and endCriticalSection(), which manipulate this attribute. When FALSE, it means that the *Scheduler* will not perform any task switching; when TRUE, tasks will be switched according to the task scheduling policies in force.

- *Shared Resource*
  A resource is an object shared by one or more *Threads* but cannot be reliably accessed by more than one client at any given time. All operations defined on this resource that access any part of the resource that is not simultaneously sharable (its nonreentrant parts) should call *Scheduler.startCriticalSection()* before

they manipulate the internal values of the resource and should call *Scheduler.endCriticalSection()* when they are done.

- *Task Control Block*
  The TCB contains the scheduling information for its corresponding *Thread* object. This includes the priority of the thread, the default start address, and the current entry address if it was preempted or blocked prior to completion. The *Scheduler* maintains a TCB object for each existing *Thread.* Note that TCB typically also has a reference off to a call and parameter stack for its *Thread,* but that level of detail is not shown in Figure 7-4.

7.2.5 Consequences

The designers and programmers must show good discipline in ensuring that every resource access locks the resource before performing any manipulation of the source. This pattern works by effectively making the current task the highest-priority task in the system. While quite successful at preventing resource corruption due to simultaneous access, it locks out all higher-priority tasks from executing during the critical section, even if they don't require the use of the resource. Many systems find this blocking delay unacceptable and must use more elaborate means for resource sharing. Further, if the initial task that locks the resource neglects to deescalate its priority, then all other tasks are permanently prevented from running. Calculation of the worst-case blocking for each task is trivial with this pattern: It is simply the longest critical section of any single task of lesser priority.

It is perhaps obvious, but should nevertheless be stated, that when using this pattern a task should never suspend itself while owning a resource because task switching is disabled so that in a situation like that no tasks are permitted to run at all. This pattern has the advantage in that it avoids deadlock by breaking the second condition (holding resources while waiting for others) as long as the task releases the resource (and reenables task switching) before it suspends itself.

7.2.6 Implementation Strategies

All commercial RTOSs have a means for beginning and ending a critical section. Invoking this Scheduler operation prevents all task

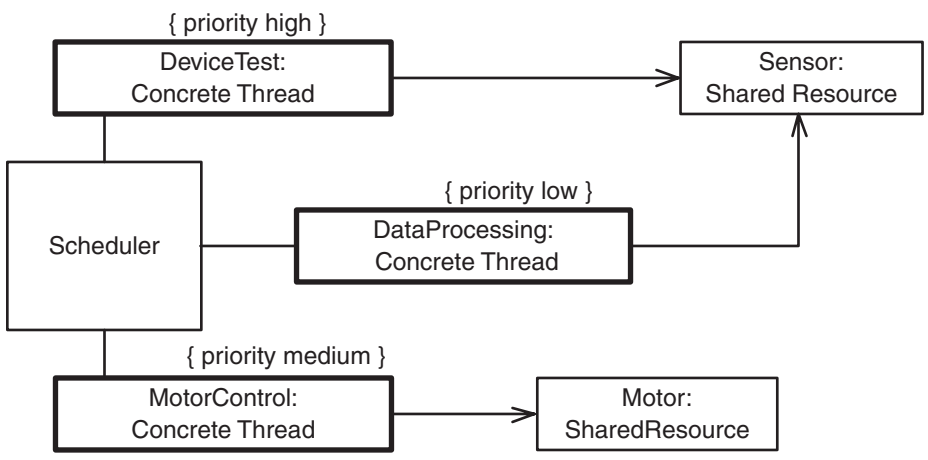
switching from occurring during the critical section. If you write your own RTOS, the most common way to do this is to set the Disable Interrupts bit on your processor's flags register. The precise details of this vary, naturally, depending on the specific processor.

7.2.7 Related Patterns

As mentioned, this is the simplest pattern that addresses the issue of sharing nonreentrant resources. Other resource sharing approaches, such as Priority Inheritance, Highest Locker, and Priority Ceiling Patterns, solve this problem as well with less impact on the schedulability of the overall system but at the cost of increased complexity. This pattern can be mixed with all of the concurrency patterns from Chapter 5, except the Cyclic Executive Pattern, for which resource sharing is a nonissue.

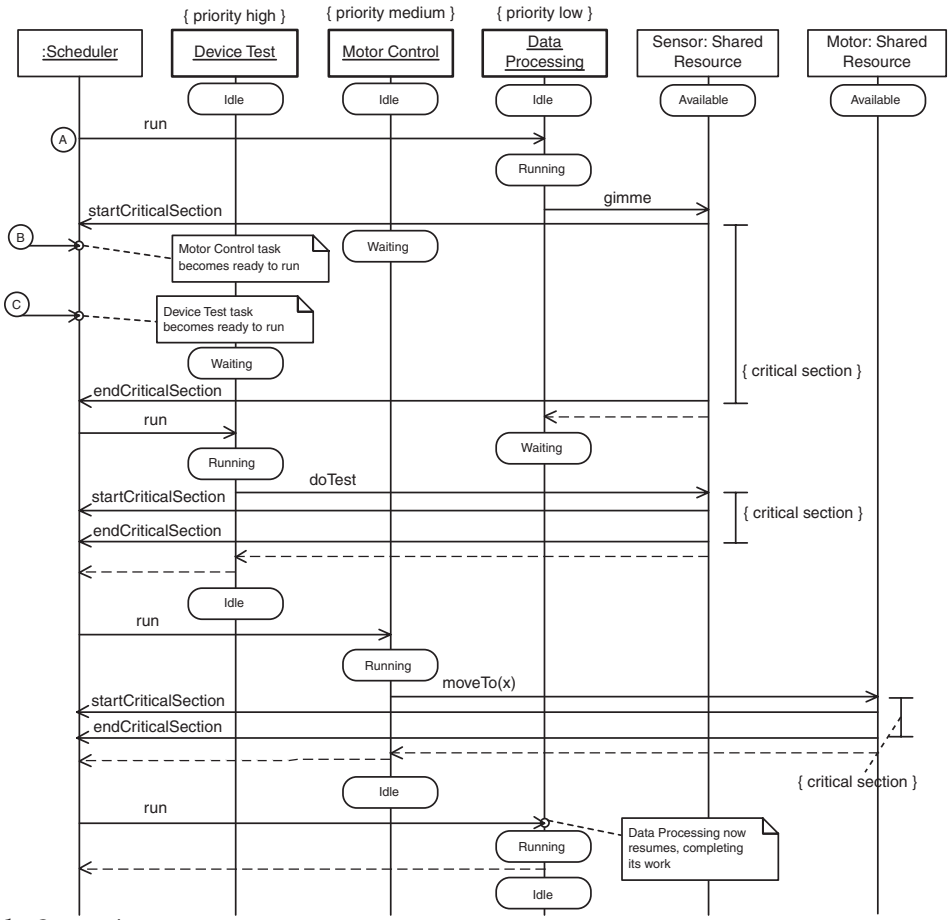
7.2.8 Sample Model

An example of the use of this pattern is shown in Figure 7-5. This example contains three tasks: Device Test (highest priority), Motor Control (medium priority), and Data Processing (lowest priority). Device Test and Data Processing share a resource called Sensor, whereas Motor Control has its own resource called Motor.



a. Structure

Figure 7-5: *Critical Section Pattern Example*



b. Scenario

Figure 7-5: *Critical Section Pattern Example (continued)*

The scenario starts off with the lowest-priority task, Data Processing, accessing the resource that starts up a critical section. During this critical section both the Motor Control task and the Device Test task become ready to run but cannot because task switching is disabled. When the call to the resource is almost done, the *Sensor.gimme()* operation makes a call to the scheduler to end the critical section. The scenario shows three critical sections, one for each of the running tasks. Finally, at the end, the lowest-priority task is allowed to complete its work and then returns to its Idle state.

7.3 PRIORITY INHERITANCE PATTERN

The Priority Inheritance Pattern reduces priority inversion by manipulating the executing priorities of tasks that lock resources. While not an ideal solution, it significantly reduces priority inversion at a relatively low run-time overhead cost.

7.3.1 Abstract

The problem of unbounded priority inversion is a very real one and has accounted for many difficult-to-identify system failures. In systems running many tasks, such problems may not be at all obvious, and typically the only symptom is that occasionally the system fails to meet one or more deadlines. The Priority Inheritance Pattern is a simple, low-overhead solution for limiting the priority inversion to at most a single level—that is, at most, a task will only be blocked by a single, lower-priority task owning a needed resource.

7.3.2 Problem

The unbounded priority inversion problem is discussed in the chapter introduction in some detail. The problem addressed by this pattern is to bound the maximum amount of priority inversion.

7.3.3 Pattern Structure

Figure 7-6 shows the structure of the pattern. The basic elements of this pattern are familiar: Scheduler, Abstract Task, Task Control Block, and so on. This can be thought of as an elaborated subset of the Static Priority Pattern, presented in Chapter 5. Note the use of the «frozen» constraint applied to the Task Control Block's *nominalPriority* attribute. This means the attribute is unchangeable once the object is created.

7.3.4 Collaboration Roles

- *Abstract Thread*
  The *Abstract Thread* class is an abstract (noninstantiable) superclass for *Concrete Thread. Abstract Thread* associates with the *Scheduler.*

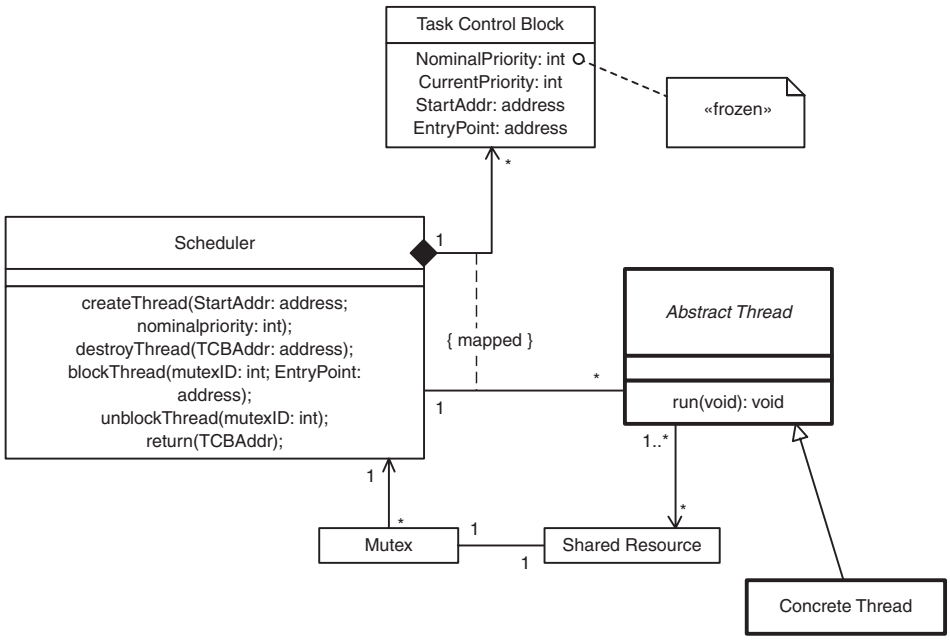


Figure 7-6: *Priority Inheritance Pattern*

Since *Concrete Thread* is a subclass, it has the same interface to the *Scheduler* as the *Abstract Thread.* This enforces interface compliance. The *Abstract Thread* is an «active» object, meaning that when it is created, it creates an OS thread in which to run. It contains (that is, it has composition relations with) more primitive application objects that execute in the thread of the composite «active» object.

- *Concrete Thread*
  The *Concrete Thread* is an «active» object most typically constructed to contain passive "semantic" objects (via the composition relation) that do the real work of the system. The *Concrete Thread* object provides a straightforward means of attaching these semantic objects into the concurrency architecture. *Concrete Thread* is an instantiable subclass of *Abstract Thread.*
- *Mutex*
  The *Mutex* is a mutual exclusion semaphore object that permits only a single caller through at a time. The operations of the *Shared Resource* invoke it whenever a relevant service is called, locking it

prior to starting the service and unlocking it once the service is complete. *Threads* that attempt to invoke a service when the services are already locked become blocked until the *Mutex* is in its unlocked state. This is done by the *Mutex* semaphore signaling the *Scheduler* that a call attempt was made by the currently active thread, the *Mutex* ID (necessary to unlock it later when the *Mutex* is released), and the entry point—the place at which to continue execution of the *Thread.*

- *Scheduler*
  This object orchestrates the execution of multiple threads based on their priority according to a simple rule: Always run the ready thread with the highest priority. When the «active» *Thread* object is created, it (or its creator) calls the *createThread* operation to create a thread for the «active» object. Whenever this thread is executed by the *Scheduler,* it calls the StartAddr address (except when the thread has been blocked or preempted, in which case it calls the *EntryPoint* address).

  In this pattern, the *Scheduler* has some special duties when the *Mutex* signals an attempt to access a locked resource: Specifically, it must block the requesting task (done by stopping that task and placing a reference to it in the *Blocked Queue* (not shown—for details of the *Blocked Queue,* see Static Priority Pattern in Chapter 5), and it must elevate the priority of the task owning the resource to that of the highest priority *Thread* being blocked. This is easy to determine since the *Blocked Queue* is a priority FIFO—the highest-priority blocked task is the first one in that queue. Similarly, when the *Thread* releases the resource, the *Scheduler* must lower its priority back to its nominal priority.

- *Shared Resource*
  A *Shared Resource* is an object shared by one or more *Threads.* For the system to operate properly in all cases, all shared resources must either be reentrant (meaning that corruption from simultaneous access cannot occur) or they must be protected. In the case of a protected resource, when a *Thread* attempts to use the resource, the associated *Mutex* semaphore is checked, and if locked, the calling task is placed into the *Blocked Queue.* The task is terminated with its reentry point noted in the TCB.

- *Task Control Block*
  The TCB contains the scheduling information for its corresponding *Thread* object. This includes the priority of the thread, the default start address and the current entry address, if it was preempted or blocked prior to completion. The *Scheduler* maintains a TCB object for each existing *Thread.* Note that TCB typically also has a reference off to a call and parameter stack for its *Thread,* but that level of detail is not shown in Figure 7-6. The TCB tracks both the current priority of the thread (which may have been elevated due to resource access and blocking) and its nominal priority.

7.3.5 Consequences

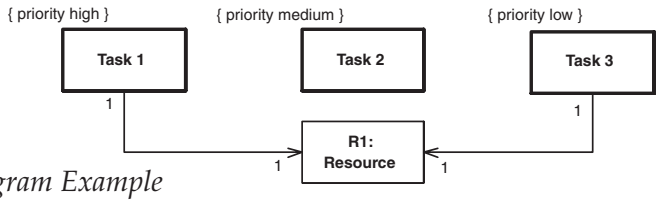
The Priority Inheritance Pattern handles well the problem of priority inversion when at most a single resource is locked at any given time and prevents unbounded priority inversion in this case. This is illustrated in Figure 7-7. With naïve priority management, Task 1, the highest-priority task in the system, is delayed from execution until Task 2 has completed. Using the Priority Inheritance Pattern, Task 1 completes as early as possible.

When there are multiple resources that may be locked at any time, this pattern exhibits behavior called *chain blocking.* That is, one task may block another, which blocks another, and so on. This is illustrated in the only slightly more complex example in Figure 7-8. The timing diagram in Figure 7-8b shows that Task 1 is blocked by Task 2 *and* Task 3 at Point G.

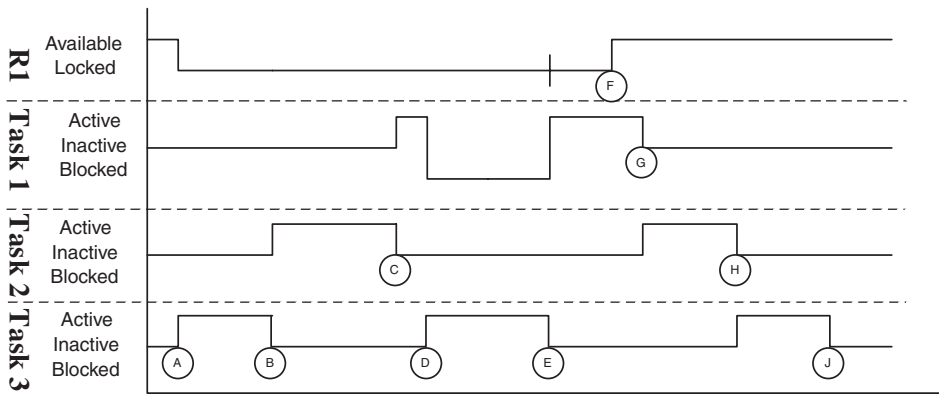
In general, the Priority Inheritance Pattern greatly reduces unbounded blocking. In fact, though, the number of blocked tasks at any given time is bounded only by the lesser of the number of tasks and the number of currently locked resources. There is a small amount of overhead to pay when tasks are blocked or unblocked to manage the elevation or depression of the priority of the tasks involved. Computation of a single task's worst-case blocking time involves computation of the worst-case chain blocking of all tasks of lesser priority.

This pattern does not address deadlock issues at all, so it is still possible to construct task models using this pattern that have deadlock.

Another consequence of the use of the priority inheritance patterns (Priority Inheritance Pattern, Highest Locker Pattern, and Priority Ceiling Pattern) is the overhead. The use of semaphores and



a. Task Diagram Example

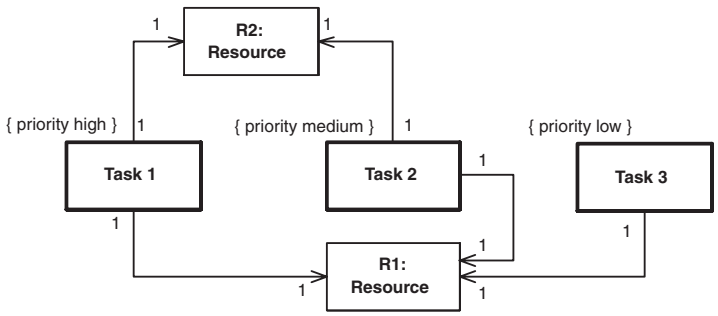


Legend:
Priority: Task 1 > Task 2 > Task 3
A: Task 3 runs at its nominal (i.e., the lowest) priority and locks R1.
B: Task 2 runs, and since it is a higher priority and doesn't need R1, it preempts Task 3.
C: Task 1 runs, preempting Task 2.
D: Task 1 attempts to lock R1. Because it is owned by Task 3, Task 1 must block. Task 3's priority is elevated to that of Task 1. Task 3 now continues.
E: Task 3 releases R1. Its priority is reduced, and Task 1 unblocks and runs, locking R1.
F: Task 1 releases R1.
G: Task 1 completes. Task 2, the highest-priority task ready to run, now continues.
H: Task 2 completes. Task 3 now can run to completion.
J: Task 3 completes.

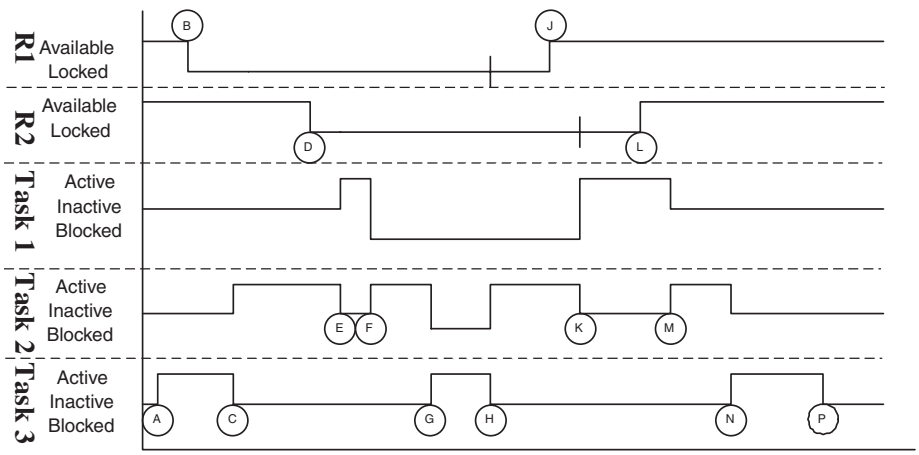
b. Behavior

Figure 7-7: *Priority Inheritance Pattern*

blocking involves task switching whenever a locked mutex is requested and another task switch whenever a waited-for mutex is released. In addition, the acts of blocking and unblocking tasks during those task context switches involves the manipulation of priority queues. Further, the use of priority inheritance means that there is some overhead in the escalation and deescalation of priorities. If blocking occurs infrequently, then this overhead will be slight, but if there is a great deal of contention for resources, then the overhead can be severe.



a. Task Diagram with Chain-Blocking Example



Legend:
Priority: Task 1 > Task 2 > Task 3
A: Task 3 runs.
B: Task 3 locks R1.
C: Task 2 becomes ready to run and preempts Task 3.
D: Task 2 locks R2.
E: Task 1 becomes ready to run and preempts Task 2.
F: Task 1 attempts to lock R2. But it must block and allow Task 2 to run. Task 2's priority is elevated to that of Task 1, and Task 2 runs.
G: Task 2 attempts to access R1, which is locked by Task 3. It must block to allow Task 3 to run. Task 3 runs at Task 1's priority. Task 1 is now blocked by both Task 2 and Task 3.
H: Task 3 releases R1. Task 2 immediately preempts Task 3 and locks R1 and runs.
J: Task 2 releases R1 and continues to run.
K: Task 2 releases R2 (the resource Task 1 is blocked on). Task 1 immediately preempts Task 2 and locks R2.
L: Task 1 releases R2.
M: Task 1 completes. This allows Task 2 to run.
N: Task 2 completes. This allows Task 3 to run.
P: Task 3 completes.

b. Chain-Blocking Behavior

Figure 7-8: *Priority Inheritance Pattern*

7.3.6 Implementation Strategies

Some RTOS directly support the notion of priority inheritance, and so it is very little work to use this pattern with such an RTOS. If you are using an RTOS that does not support it, or if you are writing your own RTOS, then you must extend the RTOS (many RTOSs have API for just this purpose) to call your own function when the mutex blocks a task on a resource. The *Scheduler* must be able to identify the priority of the thread being blocked (a simple matter because it is in the Task Control Block for the task) in order to elevate the priority of the task currently owning the resource.

It is possible to build in the nominal priority as a constant attribute of the Concrete Thread. When the Concrete Thread always runs at a given priority, then the constructor of the «active» object should do exactly that. Otherwise, the creator of that active object should specify the priority at which that task should run.

In virtually all other ways, the implementation is very similar to the implementation of standard concurrency patterns, such as the Static Priority Pattern presented in Chapter 5.

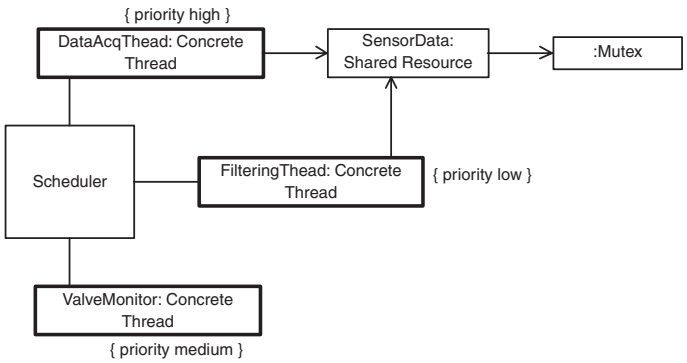
7.3.7 Related Patterns

The Priority Inheritance Pattern exists to help solve a particular problem peculiar to priority-based preemption multitasking, so all of the concurrency patterns having to do with that style of multitasking can be mixed with this pattern.

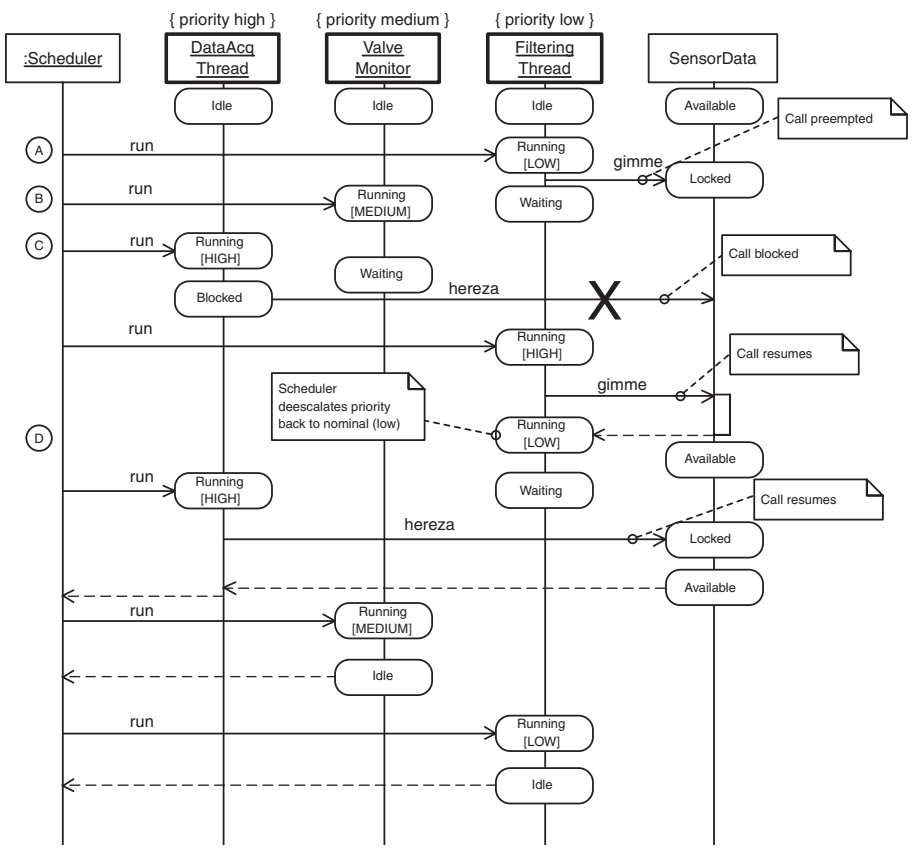
While this pattern is lightweight, it greatly reduces priority inversion in multitasking systems. However, there are other approaches that can reduce it further, such as Priority Ceiling Pattern and Highest Locker Pattern. In addition, Priority Ceiling Pattern also removes the possibility of deadlock.

7.3.8 Sample Model

Figure 7-9 provides an example to illustrate how the Priority Inheritance Pattern works. States of the objects are shown using standard UML—that is, as state marks on the instance lifelines. Some of the returns are shown, again using standard UML dashed lines. Showing that a call cannot complete is indicated with a large X on the call—*not* standard UML, but clear as to its interpretation.



a. Priority Inheritance Pattern Example



b. Scenario

Figure 7-9: *Priority Inheritance Pattern*

The flow of the scenario in Figure 7-9b is straightforward. All tasks begin the scenario in the Idle state. Then, at point A, the *FilteringThread* task becomes ready to run. It runs at its nominal priority, which is LOW (the priority of the thread is shown inside square brackets in the Running state mark—again, not quite standard UML, but parsimonious). It then calls the resource *SensorData* that then enters the Locked state.

At point B, the *ValveMonitor* task becomes ready to run. It preempts the *FilteringThread* because the former is of higher priority. The *ValveMonitor* task runs for a while, but at point C, task *DataAcqThread* becomes ready to run. Since it is the highest priority, it preempts the *ValveMonitor* thread. *DataAcqThread* object then tries to access the *SensorData* object and finds that it cannot because the latter is locked with a *Mutex* semaphore (not shown in the scenario). The *Scheduler* then blocks the *DataAcqThread* thread and runs the *FilteringThread* at the same priority as *DataAcqThread* because the *FilteringThread* inherits the priority from the highest blocking task—in this case the *DataAcqThread* task. Note at this point, the medium-priority task, *ValveMonitor,* is in the state Waiting. Without priority inheritance, if *DataAcqThread* is blocked, the *ValveMonitor* would run because it has the next highest priority.

At point D, *FilteringThread's* use of the resource is complete, and it releases the resource (done at the end of the *SensorData.gimme* operation). As it returns, the *Mutex* signals the *Scheduler* that it is now available, so the Scheduler deescalates *FilteringThread's* priority to its nominal value (LOW) and unblocks the highest-priority task, *DataAcqThread.* This task now runs to completion and returns. The *Scheduler* then runs the next highest-priority waiting task, *ValveMonitor,* which runs until it is done and returns. Finally, the lowest-priority task, *FilteringThread,* gets to complete.

The worst-case blocking time for the *DataAcqThread* task is then the amount of time that *FilteringThread* locks the *SensorData* resource. Without the Priority Inheritance Pattern, the worst-case blocking for *DataAcqThread* task would be the amount of time *FilteringThread* locks the *SensorData* resource plus the amount of time that *ValveMonitor* executes.

7.4 HIGHEST LOCKER PATTERN

The Highest Locker Pattern defines a *priority ceiling* with each resource. The basic idea is that the task owning the resource runs at the highest-priority ceiling of all the resources that it currently owns, provided that it is blocking one or more higher-priority tasks. This limits priority inversion to at most one level.

7.4.1 Abstract

The Highest Locker Pattern is another solution to the unbounded blocking/unbounded priority inversion problem. It is perhaps a minor elaboration from the Priority Inheritance Pattern, but it is different enough to have some different properties with respects to schedulability. The Highest Locker Pattern limits priority inversion to a single level as long as a task does not suspend itself while owning a resource. In this case, you may get chained blocking similar to the Priority Inheritance Pattern. Unlike the Priority Inheritance Pattern, however, you cannot get chained blocking if a task is preempted while owning a resource.

7.4.2 Problem

The unbounded priority inversion problem is discussed in the chapter introduction in some detail. The problem addressed by this pattern is to limit the maximum amount of priority inversion to a single level—that is, there is *at most* a single lower-priority task blocking a higher-priority task from executing.

7.4.3 Pattern Structure

The Highest Locker Pattern is shown in Figure 7-10. The structural elements of the pattern are the same as for the Priority Inheritance Pattern, with the addition of an attribute *priorityCeiling* for the *SharedResource.*

The pattern works by defining each lockable resource with a priority ceiling. The priority ceiling is just greater than the priority of the

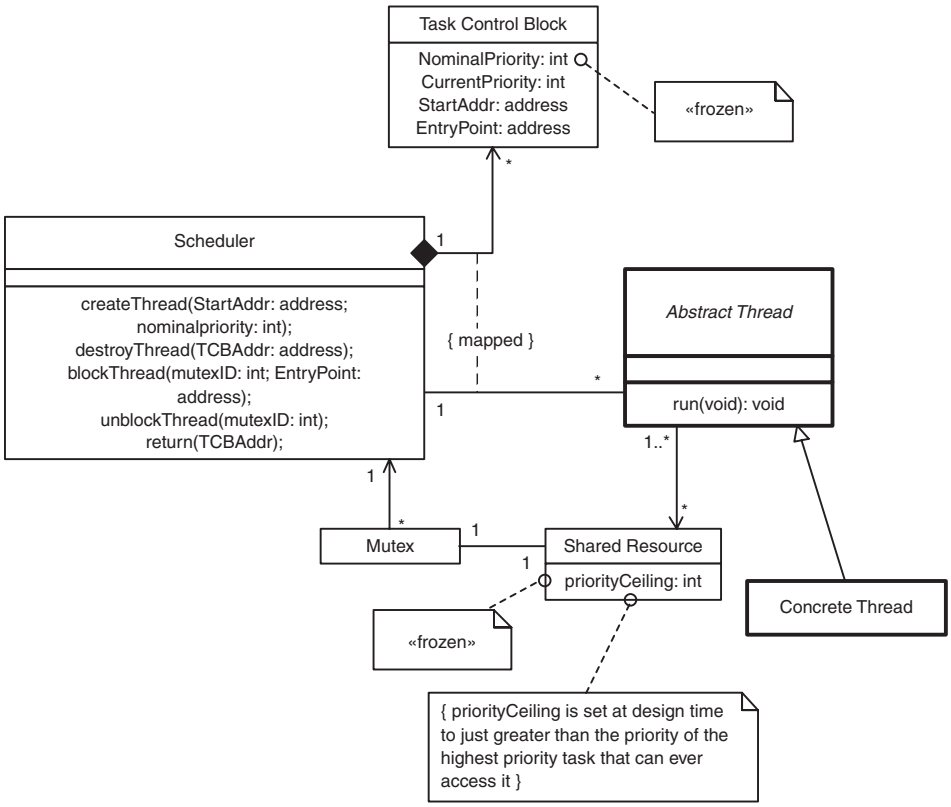


Figure 7-10: *Highest Locker Pattern*

highest-priority client of the resource—this is known at design time in a static priority scheme. When the resource is locked, the priority of the locking task is augmented to the priority ceiling of the resource.

7.4.4 Collaboration Roles

- *Abstract Thread*
  The *Abstract Thread* class is an abstract (noninstantiable) superclass for *Concrete Thread*. *Abstract Thread* associates with the

Scheduler. Since *Concrete Thread* is a subclass, it has the same interface to the *Scheduler* as the *Abstract Thread.* This enforces interface compliance. The *Abstract Thread* is an «active» object, meaning that when it is created, it creates an OS thread in which to run. It contains (that is, it has composition relations with) more primitive application objects that execute in the thread of the composite «active» object.

- *Concrete Thread*
  The *Concrete Thread* is an «active» object most typically constructed to contain passive "semantic" objects (via the composition relation) that do the real work of the system. The *Concrete Thread* object provides a straightforward means of attaching these semantic objects into the concurrency architecture. *Concrete Thread* is an instantiable subclass of *Abstract Thread.*

- *Mutex*
  The *Mutex* is a mutual exclusion semaphore object that permits only a single caller through at a time. The operations of the *Shared Resource* invoke it whenever a relevant service is called, locking it prior to starting the service and unlocking it once the service is complete. *Threads* that attempt to invoke a service when the services are already locked become blocked until the *Mutex* is in its unlocked state. This is done by the *Mutex* semaphore signaling the *Scheduler* that a call attempt was made by the currently active thread, the *Mutex* ID (necessary to unlock it later when the mutex is released), and the entry point—the place at which to continue execution of the *Thread.*

- *Scheduler*
  This object orchestrates the execution of multiple threads based on their priority according to a simple rule: Always run the ready thread with the highest priority. When the «active» *Thread* object is created, it (or its creator) calls the *createThread* operation to create a thread for the «active» object. Whenever this thread is executed by the *Scheduler,* it calls the StartAddr address (except when the thread has been blocked or preempted—in which case it calls the *EntryPoint* address).

  In this pattern, the *Scheduler* has some special duties when the *Mutex* signals an attempt to access a locked resource. Specifically, it must block the requesting task (done by stopping that task and placing a reference to it in the *Blocked Queue* (not shown—for

details of the *Blocked Queue*, see the Static Priority Pattern in Chapter 5), and it must elevate the priority of the task owning the resource to the *Shared Resource's priorityCeiling*.

- *Shared Resource*
  A resource is an object shared by one or more *Threads*. For the system to operate properly in all cases, all *Shared Resources* must either be reentrant (meaning that corruption from simultaneous access cannot occur), or they must be protected. In the case of a protected resource, when a *Thread* attempts to use the resource, the associated *Mutex* semaphore is checked, and if locked, the calling task is placed into the *Blocked Queue*. The task is terminated with its reentry point noted in the TCB.

  The *SharedResource* has a constant attribute (note the «frozen» constraint in Figure 7-10), called priorityCeiling. This is set during design to just greater than the priority of the highest-priority task that can ever access it. In some RTOSs, this means that the priority will be one more (when a larger number indicates a higher priority), and in some it will be one less (when a lower number indicates a higher priority). This ensures that when the resource is locked, no other task using that resource can preempt it.
- *Task Control Block*
  The TCB contains the scheduling information for its corresponding *Thread* object. This includes the priority of the thread, the default start address, and the current entry address if it was preempted or blocked prior to completion. The *Scheduler* maintains a TCB object for each existing *Thread*. Note that TCB typically also has a reference off to a call and parameter stack for its *Thread*, but that level of detail is not shown in Figure 7-10. The TCB tracks both the current priority of the thread (which may have been elevated due to resource access and blocking) and its nominal priority.

7.4.5 Consequences

The Highest Locker Pattern has even better priority inversion-bounding properties than the Priority Inheritance Pattern. It allows higher-priority tasks to run, but only if they have a priority higher than the priority ceiling of the resource. The priority ceiling can be determined at design time for each resource by examining the clients

of a given resource and identifying to which active object they belong and selecting the highest from among those. The priority ceiling is this value augmented by one. Computation of worst-case blocking is the length of the longest critical section (that is, resource locking time) of any task of lesser priority as long as a task never suspends itself while owning a resource.

The pattern has the disadvantage that while it bounds priority inversion to a single level, that level happens more frequently than with some other approaches. For example, if the lowest-priority task locks a resource with the highest-priority ceiling, and during that time an intermediate priority task becomes ready to run, then it is blocked even though in this case one would prefer that the normal priority rules apply. One way to handle that is to elevate the priority of the task owning the resource only when another task attempts to lock it; until then, the locking tasks runs at its nominal priority.

In this pattern, care must be taken to ensure that a task never suspends itself while owning a resource. It is fine if it is preempted, but voluntary preemption while owning a resource can lead to chain blocking, a problem previously identified with the Priority Inheritance Pattern in the previous section. If the system allows tasks to suspend themselves while owning a resource, then the computation of worst-case blocking is computed in the same way as with the Priority Inheritance Pattern—the longest case of chain blocked must be traversed.

This pattern avoids deadlock as long as no task suspends itself while owning a resource because no other task is permitted to wait on the resource (condition 4). This is because the locking task runs at a priority higher than any of the other clients of the resource. As previously noted, there is also a consequence of computational overhead associated with the Highest Locker Pattern.

7.4.6 Implementation Strategies

Fewer RTOSs support the Highest Locker Pattern more than the basic Priority Inheritance Pattern. Implementation of this pattern in your own RTOS is fairly straightforward, with the addition of priority ceiling attributes in the *Shared Resource.* When the mutex is locked, it must notify the *Scheduler* to elevate the priority of the locking task to that resource's priority ceiling.

7.4.7 Related Patterns

The Highest Locker Pattern exists to help solve a particular problem peculiar to priority-based preemption multitasking, so all of the concurrency patterns having to do with that style of multitasking can be mixed with this pattern.

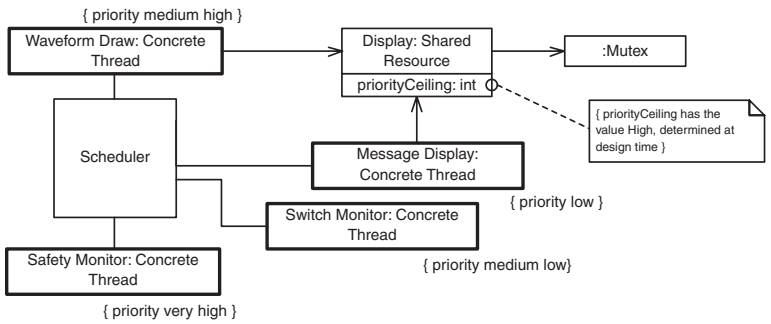
7.4.8 Sample Model

In the example shown in Figure 7-11, there are four tasks with their priorities shown using constraints, two of which, *Waveform Draw* and *Message Display,* share a common resource, *Display.* The tasks, represented as active objects in order of their priority, are *Message Display* (priority Low), *Switch Monitor* (priority Medium Low), *Waveform Draw* (priority Medium High), and *Safety Monitor* (priority Very High), leaving priority High unused at the outset. *Message Display* and *Waveform Draw* share *Display,* so the priority ceiling of *Display* is just above *Waveform Draw* (that is, High).

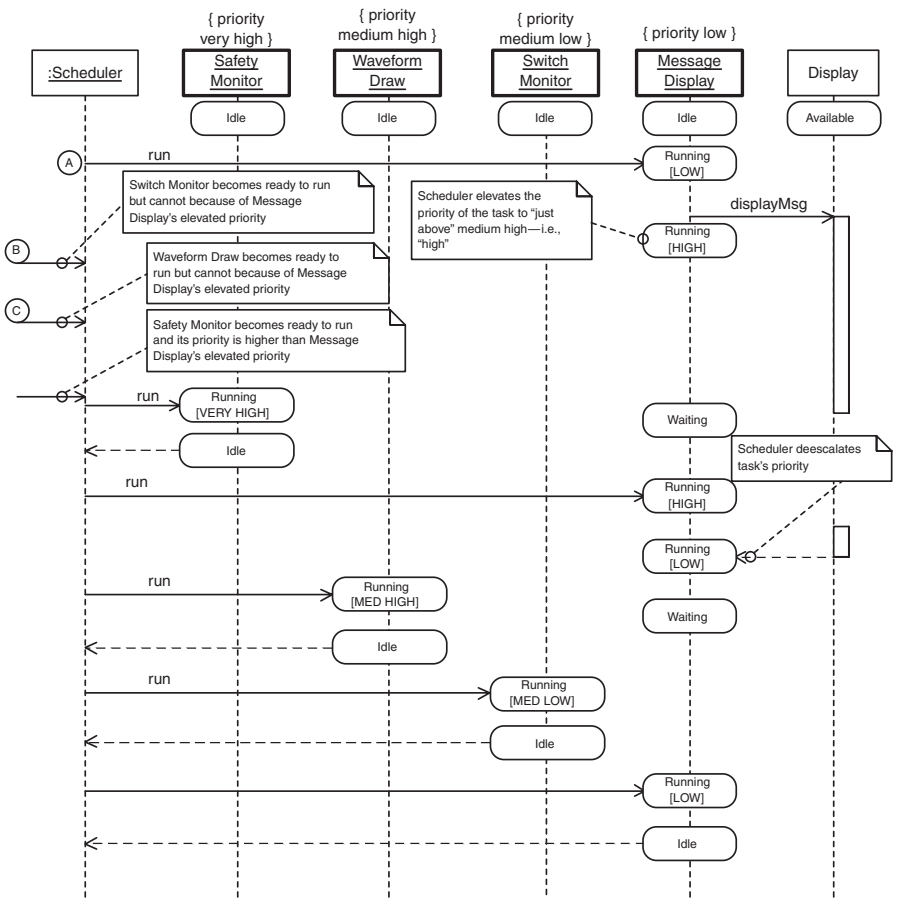
The scenario runs as follows: First, the lowest-priority task, *Message Display,* runs, calling the operation *Display.displayMsg().* Because the *Display* has a mutex semaphore, this locks the resource, and the *Scheduler* (not shown in Figure 7-11) escalates the priority of the locking task, *Message Display,* to the priority ceiling of the resource—that is, the value High.

While this operation executes, first the *Switch Monitor* and then the *Waveform Draw* tasks both become ready to run but cannot because the *Message Display* task is running at a higher priority than either of them. The *Safety Monitor* task becomes ready to run. Because it runs at a priority Very High, it can, and does, preempt the *Message Display* task.

After the *Safety Monitor* task returns control to the *Scheduler,* the *Scheduler* continues the execution of the *Message Display* task. Once it releases the resource, the mutex signals the *Scheduler,* and the latter deescalates the priority of the *Message Display* task to its nominal priority level of Low. At this point, there are two tasks of a higher priority waiting to run, so the higher-priority waiting task (*Waveform Draw*) runs, and when it completes, the remaining higher-priority task (*Switch Monitor*) runs. When this last task completes, the *Message Display* task can finally resume its work and complete.



a. Highest Locker Pattern Example



b. Scenario

Figure 7-11: *Highest Locker Pattern*

7.5 PRIORITY CEILING PATTERN

The Priority Ceiling Pattern, or Priority Ceiling Protocol (PCP) as it is sometimes called, addresses both issues of bounding priority inversion (and hence bounding blocking time) and removal of deadlock. It is a relatively sophisticated approach, more complex than the previous methods. It is not as widely supported by commercial RTOSs, however, and so its implementation often requires writing extensions to the RTOS.

7.5.1 Abstract

The Priority Ceiling Pattern is used to ensure bounded priority inversion and task blocking times and also to ensure that deadlocks due to resource contention cannot occur. It has somewhat more overhead than the Highest Locker Pattern. It is used in highly reliable multitasking systems.

7.5.2 Problem

The unbounded priority inversion problem is discussed in the chapter introduction in some detail. The Priority Ceiling Pattern exists to limit the maximum amount of priority inversion to a single level and to completely prevent resource-based deadlock.

7.5.3 Pattern Structure

Figure 7-12 shows the Priority Ceiling Pattern structure. The primary structural difference between the Priority Ceiling Pattern and the Highest Locker Pattern is the addition of a System Priority Ceiling attribute for the Scheduler. Behaviorally, there are some differences as well. The algorithm for starting and ending a critical section is shown in Figure 7-13.

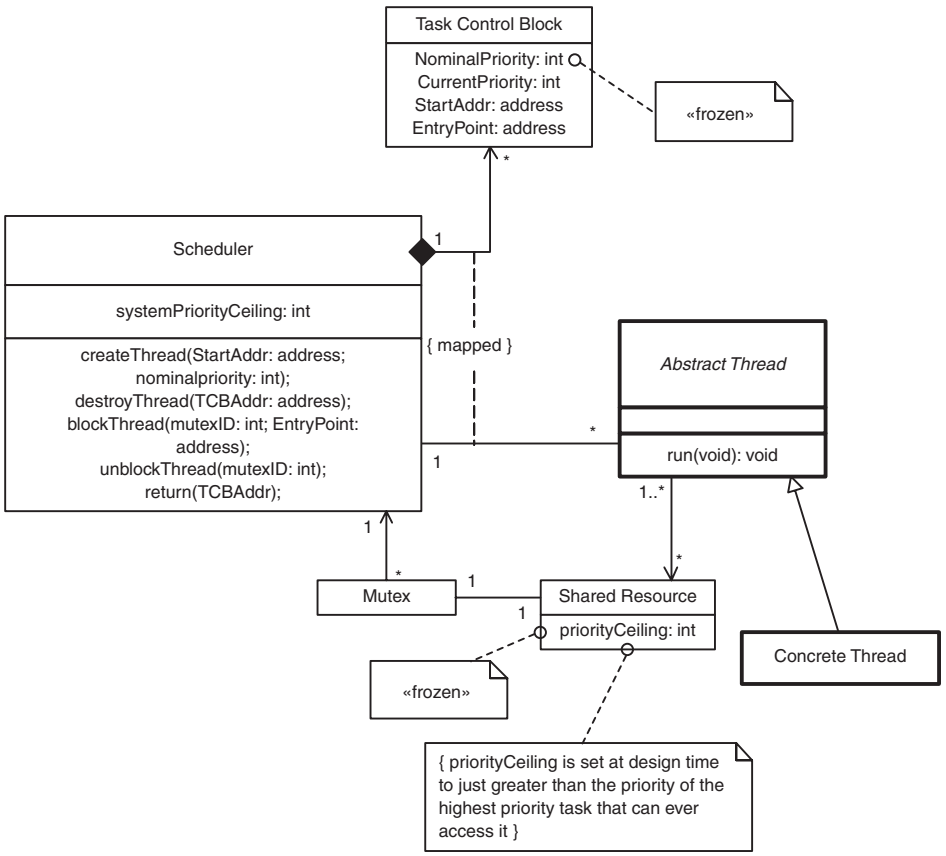


Figure 7-12: *Priority Ceiling Pattern*

7.5.4 Collaboration Roles

- *Abstract Thread*
  The *Abstract Thread* class is an abstract (noninstantiable) superclass for *Concrete Thread*. *Abstract Thread* associates with the *Scheduler*. Since *Concrete Thread* is a subclass, it has the same interface to the *Scheduler* as the *Abstract Thread*. This enforces interface compliance. The *Abstract Thread* is an «active» object, meaning that when it is created, it creates an OS thread in which to run. It

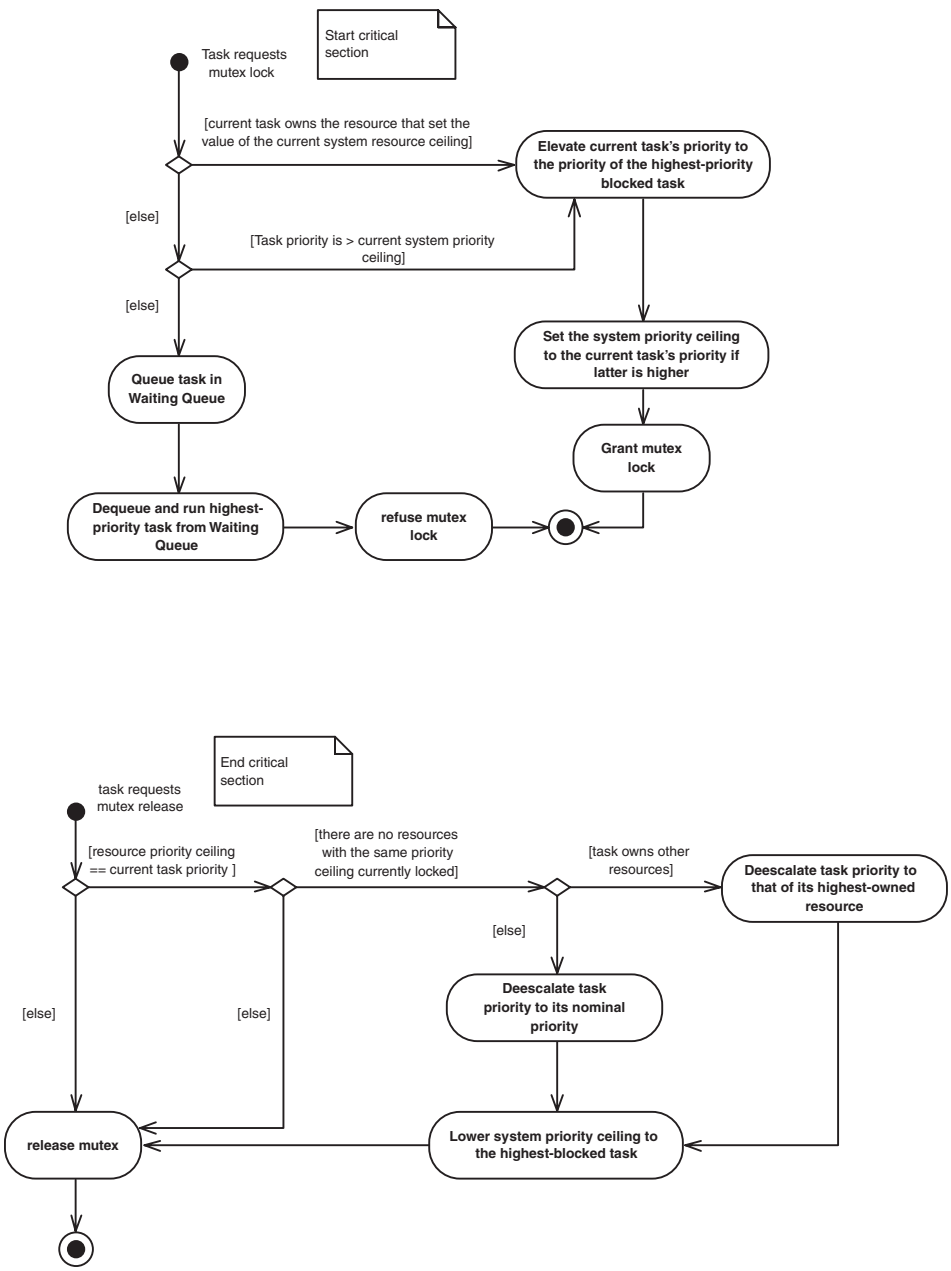


Figure 7-13: *Priority Ceiling Pattern Resource Algorithm*

contains (that is, it has composition relations with) more primitive application objects that execute in the thread of the composite «active» object.

- *Concrete Thread*
  The *Concrete Thread* is an «active» object most typically constructed to contain passive "semantic" objects (via the composition relation) that do the real work of the system. The *Concrete Thread* object provides a straightforward means of attaching these semantic objects into the concurrency architecture. *Concrete Thread* is an instantiable subclass of *Abstract Thread.*
- *Mutex*
  The *Mutex* is a mutual exclusion semaphore object that permits only a single caller through at a time. The operations of the *Shared Resource* invoke it whenever a relevant service is called, locking it prior to starting the service and unlocking it once the service is complete. *Threads* that attempt to invoke a service when the services are already locked become blocked until the *Mutex* is in its unlocked state. This is done by the *Mutex* semaphore signaling the *Scheduler* that a call attempt was made by the currently active thread, the *Mutex* ID (necessary to unblock the correct *Thread* later when the *Mutex* is released), and the entry point—the place at which to continue execution of the *Thread.* See Figure 7-13 for the algorithms that control locking, blocking, and releasing the *Mutex.*
- *Scheduler*
  This object orchestrates the execution of multiple threads based on their priority according to a simple rule: Always run the ready thread with the highest priority. When the «active» *Thread* object is created, it (or its creator) calls the *createThread* operation to create a thread for the «active» object. Whenever this thread is executed by the *Scheduler,* it calls the StartAddr address (except when the thread has been blocked or preempted—in which case it calls the *EntryPoint* address).

  In this pattern, the *Scheduler* has some special duties when the *Mutex* signals an attempt to access a locked resource. Specifically, under some conditions, it must block the requesting task (done by stopping that task and placing a reference to it in the *Blocked Queue* (not shown—for details of the *Blocked Queue,* see the Static Priority Pattern in Chapter 5), and it must elevate the priority of

the highest-priority blocked *Thread* being blocked. This is easy to determine, since the *Blocked Queue* is a priority FIFO—the highest-priority blocked task is the first one in that queue. Similarly, when the *Thread* releases the resource, the *Scheduler* must lower its priority back to its nominal priority. The *Scheduler* maintains the value of the highest-priority ceiling of all currently locked resources in its attribute *systemPriorityCeiling*.

- *Shared Resource*
  A resource is an object shared by one or more *Threads.* For the system to operate properly in all cases, all *Shared Resources* must either be reentrant (meaning that corruption from simultaneous access cannot occur), or they must be protected. In the case of a protected resource, when a *Thread* attempts to use the resource, the associated mutex semaphore is checked, and if locked, the calling task is placed into the *Blocked Queue.* The task is terminated with its reentry point noted in the TCB.

  The *SharedResource* has a constant attribute (note the «frozen» constraint in Figure 7-12), called priorityCeiling. This is set during design to just greater than the priority of the highest priority task that can ever access it. In some RTOSs, this means that the priority will be one more (when a larger number indicates a higher priority), and in some it will be one less (when a lower number indicates a higher priority). This ensures that when the resource is locked, no other task using that resource can preempt it.
- *Task Control Block*
  The TCB contains the scheduling information for its corresponding *Thread* object. This includes the priority of the thread, the default start address and the current entry address if it was preempted or blocked prior to completion. The *Scheduler* maintains a TCB object for each existing *Thread.* Note that TCB typically also has a reference off to a call and parameter stack for its *Thread,* but that level of detail is not shown here. The TCB tracks both the current priority of the *Thread* (which may have been elevated due to resource access and blocking) and its nominal priority.

7.5.5 Consequences

This pattern effectively enforces the desirable property that a high-priority task can at most be blocked from execution by a single critical section of a lower-priority task owning a required resource.

It can happen in the Priority Ceiling Pattern that a running task may not be able to access a resource even though it is not currently locked. This will occur if that resource's priority ceiling is less than the current system resource ceiling.

Deadlock is prevented by this pattern because condition 4 (circular wait) is prevented. Any condition that could potentially lead to circular waiting is prohibited. This does mean that a task may be prevented from accessing a resource even though it is currently unlocked.

There is also a consequence of computational overhead associated with the Priority Ceiling Pattern. This pattern is the most sophisticated of the resource management patterns presented in this chapter and has the highest computational overhead.

7.5.6 Implementation Strategies

Rather few RTOSs support the Priority Ceiling Pattern, but it can be added if the RTOS permits extension, particularly when a mutex is locked or released. If not, you can create your own Mutex and System Resource Ceiling classes that intervene with the priority management prior to handing off control to the internal RTOS scheduler. If you are writing your own scheduler, then the implementation should be a relatively straightforward extension of the Highest Locker Pattern.

7.5.7 Related Patterns

Because this pattern is the most sophisticated, it also has the most computational overhead. Therefore, under some circumstances, it may be desirable to use a less computational, if less capable, approach, such as the Highest Locker Pattern, the Priority Inheritance Pattern, or even the Critical Section Pattern.

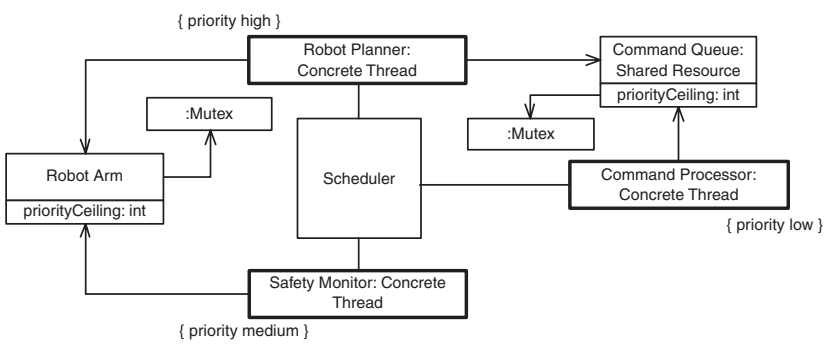
7.5.8 Sample Model

A robotic control system is given as an example in Figure 7-14a. There are three tasks. The lowest-priority task, *Command Processor,* inserts commands into a shared resource, the *Command Queue.* The middle-priority task, *Safety Monitor,* performs periodic safety monitoring, accessing the shared resource *Robot Arm.* The highest-priority task, *Robotic Planner,* accepts commands (and hence must access the *Command Queue*) and also moves the arm (and therefore must access

Robot Arm). Note that the resource ceiling of both resources must be the priority of the highest-priority task in this case because it accesses both of these resources.

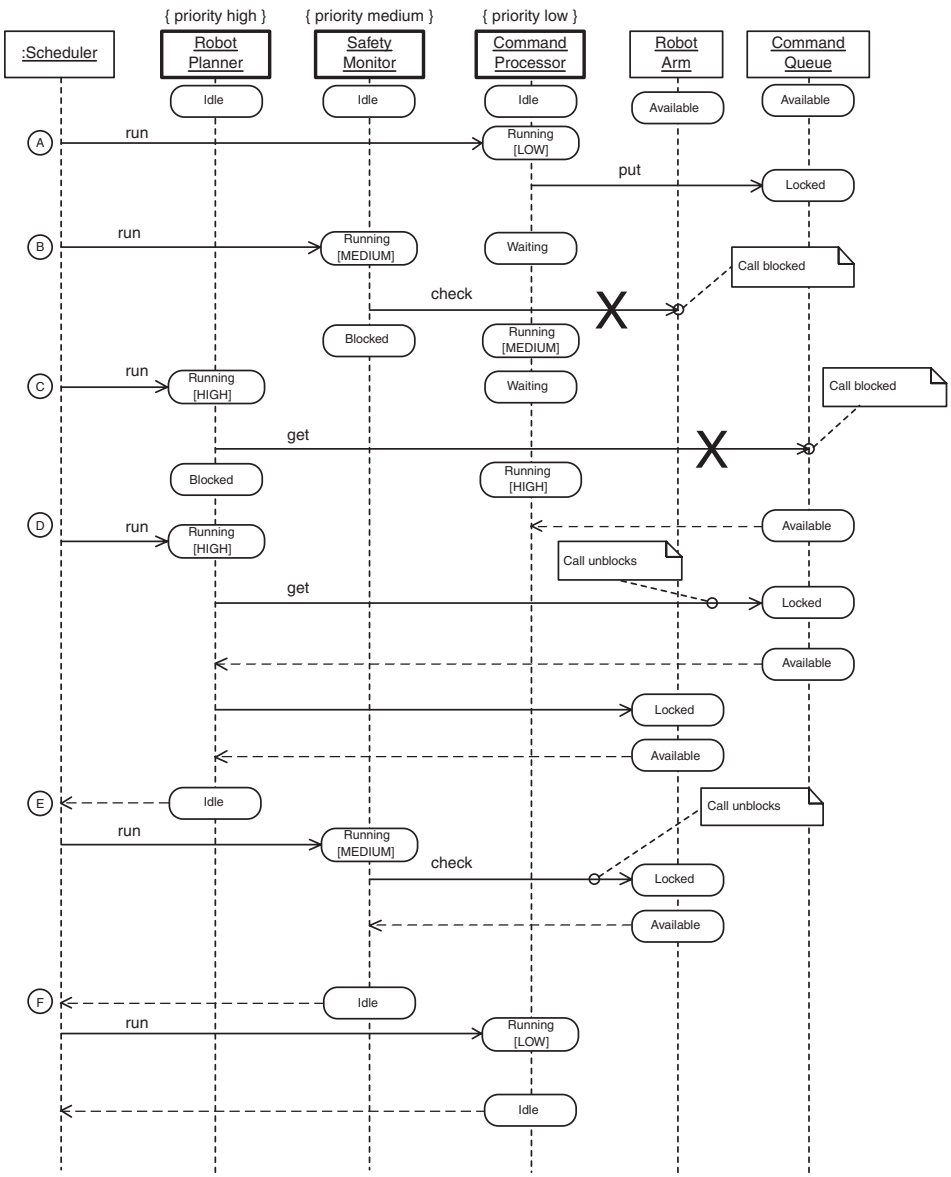
Figure 7-14b shows a scenario for the example. At point A, the *Command Processor* runs, putting set of commands into the *Command Queue.* The call to the *Command Processor* locks the resource successfully because at this point, there are no locked resources. While this is happening, the *Safety Monitor* starts to run at point B. This preempts the *Command Processor* because it is a higher priority, so *Command Processor* goes into a waiting state because it's ready to run but cannot because a higher-priority task is running. Now the *Safety Monitor* attempts to access the second resource, *Robot Arm.* Because a resource is currently already locked with same priority ceiling (found by the Scheduler examining its *systemPriorityCeiling* attribute), that call is blocked. Note that the *Safety Monitor* is prevented from running even though it is trying to access a resource that is not currently locked but *could* start a circular waiting condition, potentially leading to deadlock. Thus, the access is prevented.

When the resource access to *Safety Monitor* is prevented, the priority of the *Command Processor* is elevated to Medium, the same level as the highest-blocked task. At point C, *Robot Planner* runs, preempting the *Command Processor* task. The *Robot Planner* invokes *Command Queue.Get()* to retrieve any waiting commands but finds that this resource is locked. Therefore, its access is blocked, and it is put on the blocked queue, and the *Command Processor* task resumes but at priority High.



a. Priority Ceiling Pattern Example

Figure 7-14: *Priority Ceiling Pattern*



b. Scenario

Figure 7-14: *Priority Ceiling Pattern*

When the call to *Command Queue.put()* finally completes, the priority of the *Command Processor* task is deescalated back to its nominal priority—Low (point D). At this point in time, there are two tasks of

higher priority waiting to run. The higher priority of them, *Robot Planning* runs at its normal High priority. It accesses first the *Command Queue* resource and then the *Robot Arm* resource. When it completes, the next highest task ready to run is *Safety Monitor.* It runs, accessing the *Robot Arm* resource. When it completes, the lowest-priority task, *Command Processor* is allowed to complete its work and return control to the OS.

7.6 SIMULTANEOUS LOCKING PATTERN

The Simultaneous Locking Pattern is a pattern solely concerned with deadlock avoidance. It achieves this by breaking condition 2 (holding resources while waiting for others). The pattern works in an all-or-none fashion. Either all resources needed are locked at once or none are.

7.6.1 Abstract

Deadlock can be solved by breaking any of the four conditions required for its existence. This pattern prevents the condition of holding some resources by requesting others by allocating them all at once. This is similar to the Critical Section Pattern. However, it has the additional benefit of allowing higher-priority tasks to run if they don't need any of the locked resources.

7.6.2 Problem

The problem of deadlock is such a serious one in highly reliable computing that many systems design in specific mechanisms to detect it or avoid it. As previously discussed, deadlock occurs when a task is waiting on a condition that can never, in principle, be satisfied. There are four conditions that must be true for deadlock to occur, and it is sufficient to deny the existence of any one of these. The Simultaneous Locking Pattern breaks condition 2, not allowing any task to lock resources while waiting for other resources to be free.

7.6.3 Pattern Structure

Figure 7-15 shows the structure of the Simultaneous Locking Pattern. The special structural aspect of this pattern is the collaboration role *MultiResource.* Each *MultiResource* has a single mutex semaphore that locks only when the entire set of aggregated *Shared Resources* is available to be locked. Similarly, when the semaphore is released, all the aggregated *Shared Resources* are released.

7.6.4 Collaboration Roles

- *MultiResource*
  This object aggregates an entire set of resources needed (or possibly needed) by a *Resource Client. MultiResource* explicitly locks and unlocks the set of resources. This locking and unlocking action should be a noninterruptible critical section. If any of the aggregated *Shared Resources* is not available during the locking process,

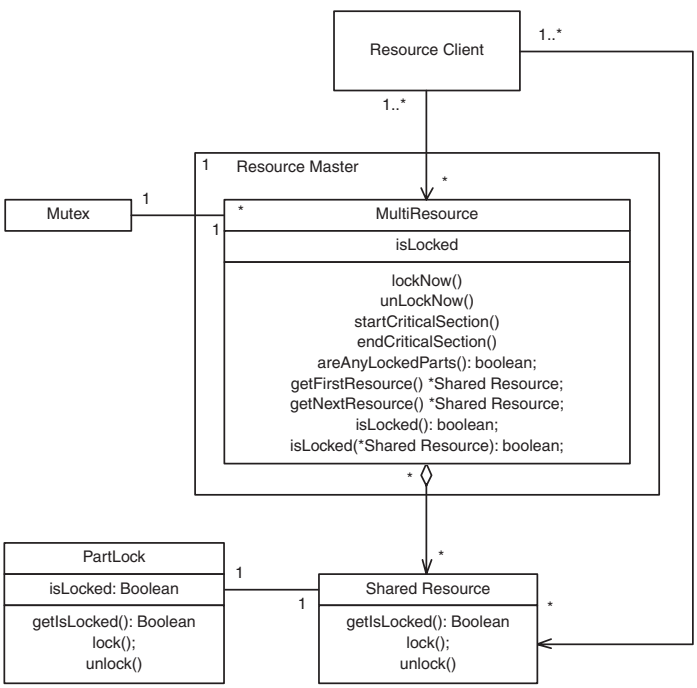


Figure 7-15: *Simultaneous Locking Pattern*

then the *MultiResource* must release all of the Shared Resources it successfully locked. *MultiResource* must define operations *startCriticalSection()* and *endCriticalSection* to prevent task switching from occurring during the locking or unlocking process. Also, *areAnyLockedParts()* returns TRUE if any of the *Shared Resources* aggregated by the *MultiResource* are still locked. For walking through the *Shared Resources,* the *MultiResource* also has the operations *getFirstResource()* and *getNextResource(),* both of which return a pointer to a *Shared Resource* (or NULL if at the end of the list) and *isLocked(*Shared Resource),* which returns TRUE only if the referenced *Shared Resource* is currently locked by the *MultiResource.* If either unlocked or not aggregated by the *MultiResource,* then it returns FALSE. Two more operations, *lockNow()* and *unlockNow(),* simply set the *isLocked* attribute of the *MultiResource* without checking the status of the aggregated parts.

- *Mutex*
  The *Mutex* is a mutual exclusion semaphore object that associates with *MultiResource.* In this pattern the shared resources are locked for a longer duration than with the priority inheritance-based patterns. This is because *Resource Client* needs to own all the resources for the entire critical section so that the *Resource Client* never owns a resource while trying to lock another. The policy is that the *Mutex* is only locked if *all* of the required *Share-Resource PartLocks* are successfully locked. *Mutex* is an OS-level mutex and signals the *Scheduler* to take care of blocking tasks that attempt to lock the *SharedResource.*
- *PartLock*
  The *PartLock* is a special mutual exclusion semaphore that associates to *Shared Resource.* This *Mutex* is queryable as to its lock status, using the *getIsLocked()* operation. This semaphore does not signal the *Scheduler* unlike the *Mutex,* because there is no need; the OS-level locking is done by the *Mutex* and not by the *PartLock.* Nevertheless, the *MultiResource* needs to be able to ascertain the locking status of all the resources before attempting to lock any of them.
- *Resource Client*
  The *Resource Client* is a user of *Shared Resource* objects. It locks potentially multiple *Shared Resources* via the *MultiResource.* The policy enforced in this pattern is that all resources used in a criti-

cal section must be locked at the same time, or the entire lock will fail. The *Resource Client* is specifically prohibited from locking one resource and later, while still owning that lock, attempting to lock another. Put another way, an attempt to lock a set of resources is only permitted if the *Resource Client* currently owns no locks at all, and if any of the requested resources are unavailable, the entire lock will fail and the *Resource Client* must wait and block on the set of resources (that is, it blocks on the mutex owned by its associated *MultiResource*).

- *ResourceMaster*
  The *ResourceMaster* orchestrates the locking and unlocking of *Mutexes* associated with *MultiResources.* Whenever a *MultiResource* locks a *Mutex,* the *ResourceMaster* searches its list of all *MultiResources* and locks any that share one of the *SharedResources.* That way, if a *Thread* tries to lock its *MultiResource* and another one owns a needed *SharedResource,* the *Thread* can block on the *Mutex* of its associated *MultiResource.* Conversely, when a *MultiResource* releases all of its *Shared Resources,* that *MultiResource* notifies the *ResourceMaster* and it tracks down all of the other *MultiResources* and sees if it can unlock them as well (it may not be able to if another *MultiResource* has locked a *SharedResource* unused by the first).
- *Shared Resource*
  A resource is a part object owned by the *MultiResource* object. In this pattern, a *Shared Resource* does not connect to a *Mutex* because it is not locked individually. As implied by its name, the same *Shared Resource* object may be an aggregated part of different *MultiResource* objects. The pattern policy is that no resource that is aggregated by one *MultiResource* is allowed to be directly locked by a *Thread,* although it may be accessed by a *Thread* to perform services. The *Shared Resource* contains operations to explicitly lock, unlock, and to query its locked status, and these simply invoke services in the associated *PartLock.*

7.6.5 Consequences

The Simultaneous Locking Pattern prevents deadlock by breaking condition 2, required for deadlock to occur—namely locking some resources while waiting for others to become available. It does this

by locking all resources needed at once and releasing them all at once. This resource management pattern can easily be used in most scheduling patterns, such as the Static Priority Pattern.

There are two primary negatives to the use of this pattern. First, priority inversion is not bounded. A higher-priority task is free to preempt and run as long as it doesn't use any currently locked resource. This pattern could be mixed in with the priority inheritance pattern to address that problem.

The second issue is that this pattern invokes some computational overhead, which may become severe in situations in which there are many shared resources. Each time a request to lock a resource is made, each of the *Shared Resources* must be locked *and* all of the other *MultiResources* must be checked to see if they aggregate any of these locked *Shared Resources.* Any *MultiResource* that shares one of the just-locked *Shared Resources* must itself be locked. On release of a lock on a particular *MultiResource,* all of its *Shared Resources* must be unlocked, and then each of the other *MultiResources* must be examined using the *areAnyLockedParts()* operation. If it returns TRUE, then that *MultiResource* must remain locked; otherwise is must be unlocked.

Another issue is that programmer/designer discipline is required not to access the *Shared Resources* without first obtaining a lock by going through the *MultiResource* mechanism. Because *Shared Resources* don't use standard OS mutexes for locking (since we don't want *Threads* blocking on them rather then the *MultiResources*), it is possible to directly access the *Shared Resource,* bypassing the locking mechanisms. This is a Bad Idea. One possible solution to enforce the locking is to propagate all of the operations from the resources to the *MultiResource,* make the operations public in the *MultiResource* and private in the *Shared Resource,* and making the *MultiResource* a friend of the *Shared Resource.* This adds some additional computational overhead, but in some languages the propagated operations could be made inline to minimize this. Alternatively, each *Shared Resource* could be told, during the locking process, who its owner is. Then on each service call, the owner would have to pass an owner ID to prove it had rights to request the service.

7.6.6 Implementation Strategies

Care must be taken that the locking of all the resources in *MultiResource.lock()* and *MultiResource.unlock()* must be done in a critical sec-

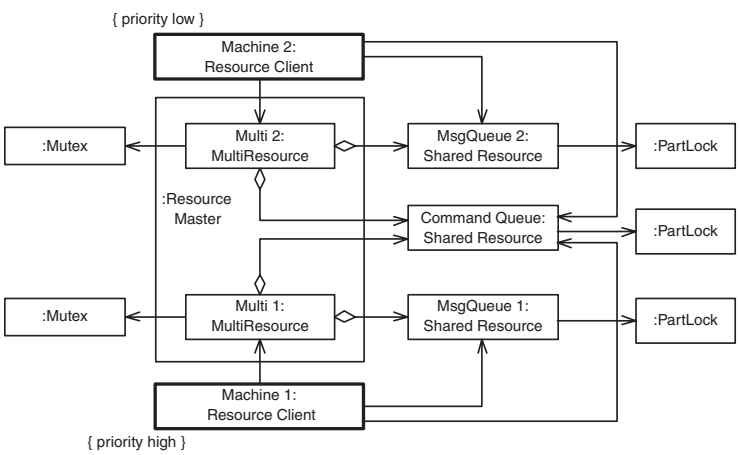
tion to prevent deadlock condition 2 from occurring. Other than that, the implementation of this pattern is straightforward.

7.6.7 Related Patterns

This pattern removes deadlock by breaking condition 2 required for deadlock. There are other approaches to avoiding deadlock. One of this is presented in the Ceiling Priority Pattern and another in the Ordered Locking Pattern, both presented in this chapter. This pattern is normally mixed with a concurrency management policy, such as the Static Priority Pattern, but other patterns can be used as well. If it is desirable to limit priority inversion, then this pattern can be mixed with the Priority Inheritance Pattern.

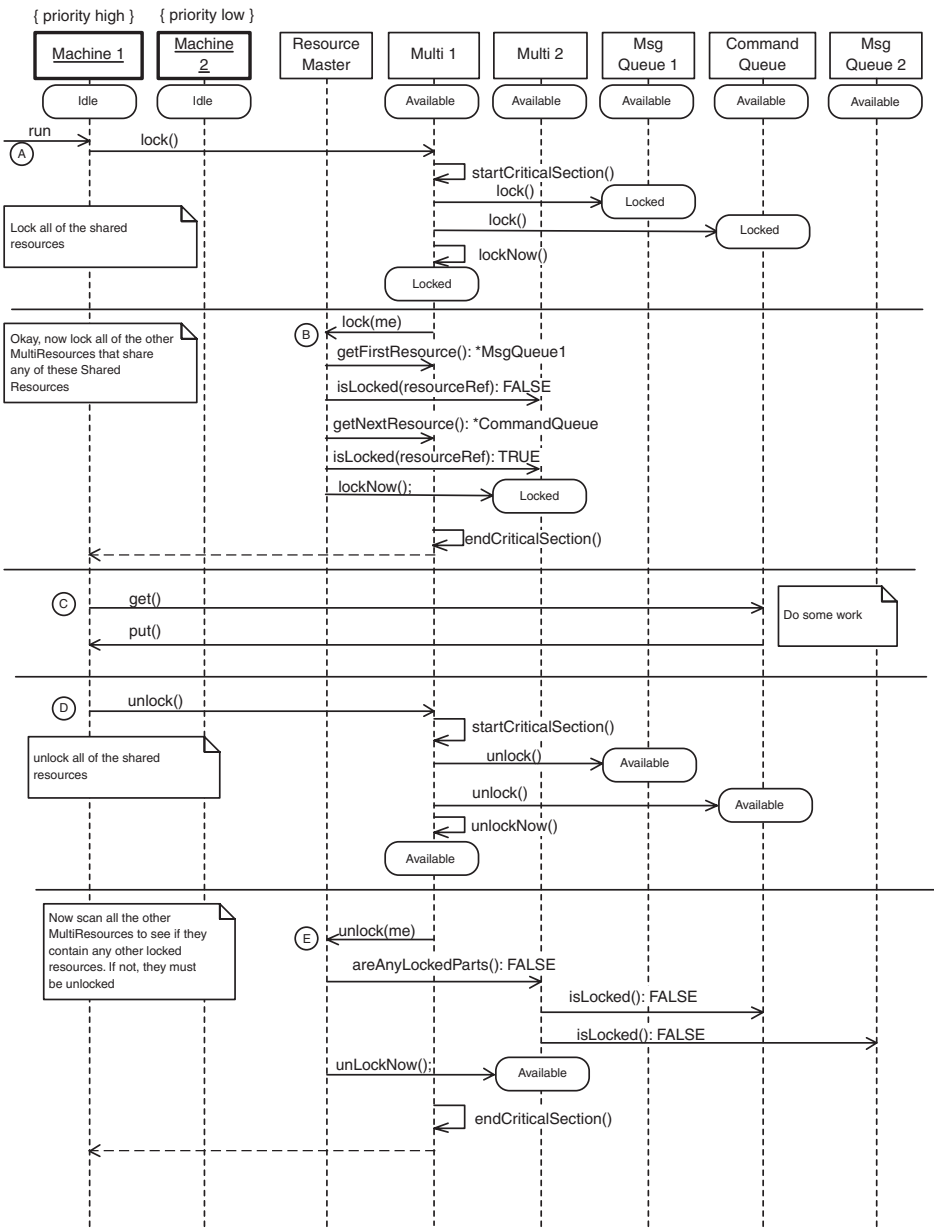
7.6.8 Sample Model

Figure 7-16a shows a simple example of the application of this pattern. Two *Concrete Threads, Machine 1* and *Machine 2,* share three resources: *MsgQueue 1* (Machine 1 only), *Command Queue* (both), and *MsgQueue 2* (Machine 2 only). To avoid the possibility of a deadlock occurring, the Simultaneous Locking Pattern is used. Two *MultiResources* (*Multi 1* and *Multi 2*) are created as composite parts of an instance of *Resource-Master.* Figure 7-16b shows the behavior when *Machine 1* locks its



a. Simultaneous Locking Pattern Example

Figure 7-16: *Simultaneous Locking Pattern (continued)*



b. Scenario

Figure 7-16: *Simultaneous Locking Pattern (continued)*

resources, does some work (moving messages from the *Command Queue* to *MsgQueue 1*), and then unlocks the resources.

What is not shown is what happens if *Machine 2* runs during the execution of the get() and put() operations, but it is clear that as soon as *Machine 2* attempts to lock its *MultiResource,* it will be blocked.

7.7 ORDERED LOCKING PATTERN

The Ordered Locking Pattern is another way to ensure that deadlock cannot occur—this time by preventing condition 4 (circular waiting) from occurring. It does this by ordering the resources and requiring that they always be accessed by any client in that specified order. If this is religiously enforced, then no circular waiting condition can ever occur.

7.7.1 Abstract

The Ordered Locking Pattern eliminates deadlock by ordering resources and enforcing a policy in which resources must be allocated only in a specific order. Unlike "normal" resource access, but similar to the Simultaneous Locking Pattern, the client must explicitly lock and release the resources, rather than doing it implicitly by merely invoking a service on a resource. This means that the potential for neglecting to unlock the resource exists.

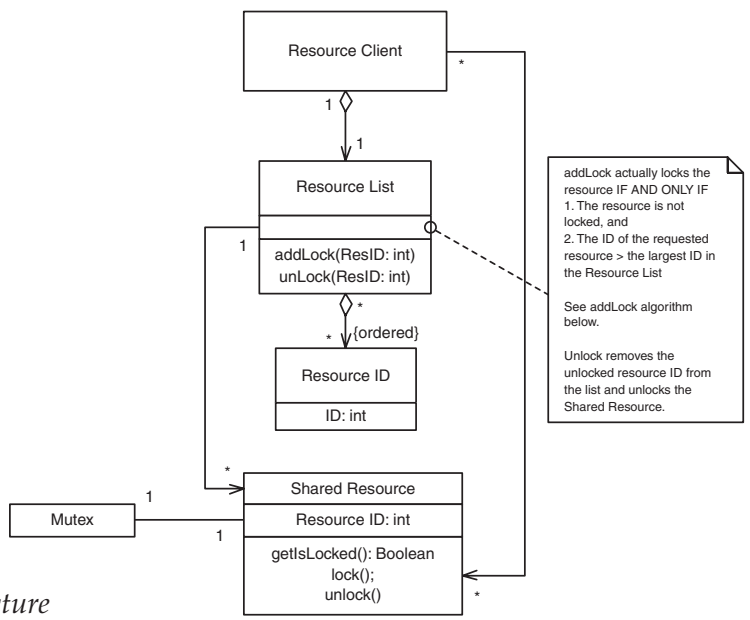
7.7.2 Problem

The Ordered Locking Pattern solely addresses the problem of deadlock elimination, as does the previous Simultaneous Locking Pattern.

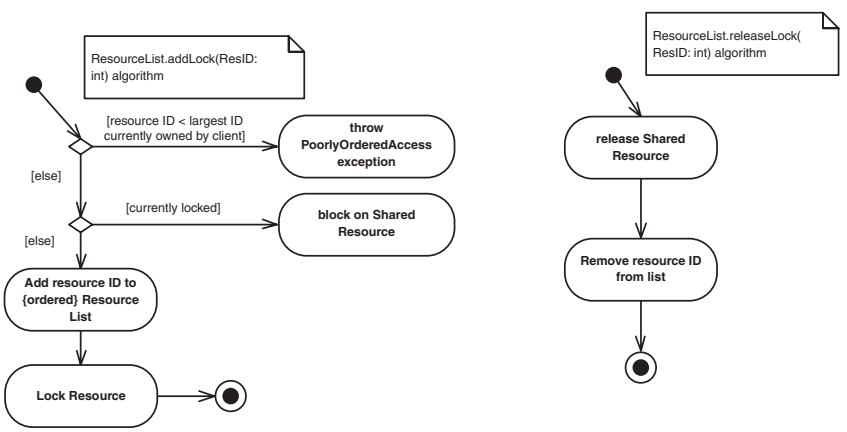
7.7.3 Pattern Structure

Figure 7-17a shows the structural part of the Ordered Locking Pattern. Each *Resource Client* aggregates a *Resource List,* which contains an ordered list of Resource IDs currently locked by the *Thread.*

Figure 7-17b uses UML activity charts to show the algorithms for locking and unlocking the resource. The basic policy of resource locking is that each resource in the entire system has a unique integer-valued identifier, and a *Thread* may *only* lock a resource whose ID is greater than that of the highest resource it currently owns. An attempt to lock a resource with a lower-valued ID than the highest-



a. Structure



b. Access Algorithms

Figure 7-17: *Ordered Locking Pattern*

valued resource you currently own causes the *Resource List* to throw a *PoorlyOrderedAccess* exception, indicating a violation of this policy. Sometimes a *Resource Client* may block on a resource that it needs because it is locked, and that is perfectly fine. But it can never happen that a circular waiting condition can occur (required for deadlock) because it would require that at least one *Resource Client* would have to block waiting for the release of a resource whose ID is lower than its highest-owned resource.[2]

7.7.4 Collaboration Roles

- *Mutex*
  The *Mutex* is a mutual exclusion semaphore object that associates with *Shared Resource.* If a Shared Resource is currently locked when requested by a *Resource Client* (via its *Resource List*), then the *Resource Client* blocks on that resource.
- *Resource Client*
  A *Resource Client* is an object (which may be «active») that owns and locks resources. It aggregates a *Resource List* to manage the locking and unlocking of those resources.
- *Resource ID*
  The *Resource ID* is a simple part objected aggregated by *Resource List.* It merely contains the ID of a corresponding *Shared Resource* currently locked by the *Thread.* When the *Shared Resource* is unlocked by the *Resource List,* its ID is removed from the list.
- *Resource List*
  The *Resource List* manages the locking and unlocking of *Shared Resources* according to the algorithm shown in Figure 7-17b. When a *Resource Client* wants to lock a resource, it makes the request of the *Resource List.* If the ID of the required resource is greater than any currently owned resource, and then if it is unlocked, the *Resource List* locks it and adds it to the list. If it is locked, then the *Thread* blocks on the resource.
- *Shared Resource*
  A resource is an object shared by one or more *Resource Client.* In this pattern, each *Shared Resource* has a unique integer-valued

2. The proof is left as an exercise for the reader.

identifier. This identifier is used to control the order in which *Shared Resources* may be locked. If a *Shared Resource* itself uses other *Shared Resources,* then it may *only* do so if the called *Shared Resource* identifiers are of higher value than its own.

7.7.5 Consequences

This pattern effectively removes the possibility of resource-based deadlocks by removing the possibility of condition 4—circular waiting. For the algorithm to work any ordering of *Shared Resources* will do provided that this ordering is global. However, some orderings are better than others and will result is less blocking overall. This may take some analysis at design time to identify the best ordering of the *Shared Resources.* As mentioned above, if *Shared Resources* are themselves *Resource Clients* (a reasonable possibility), then they should *only* invoke services of *Shared Resources* that have higher-valued IDs than they do. If they invoke a lower-valued *Shared Resource,* then they are in effect violating the ordered locking protocol by the transitive property of locking (if A locks B and then B locks C, then A is in effect locking C).

While draconian, one solution to the potential problem of transitive violation of the ordering policy is to enforce the rule that a *Shared Resource* may never invoke services or lock other *Shared Resources.* If your system design does allow such transitive locking, then each transitive path must be examined to ensure that the ordering policy is not violated. The Ordered Locking Pattern does not address the issue of bounding priority inversion as do some other patterns here.

7.7.6 Implementation Strategies

One memory-efficient implementation for *Resource List* is to use an array of integers to hold the *Resource IDs.* The array only needs to be as large as the maximum number of resources held at any one time. For an even more memory-efficient implementation (but at the cost of some computational complexity), a bit set can be used. The bit set must have the same number of bits as maximum *Resource ID* value. Setting and unsetting the bit is computationally lightweight, but

checking to see if there is a greater bit set is a little more computationally intensive.

7.7.7 Related Patterns

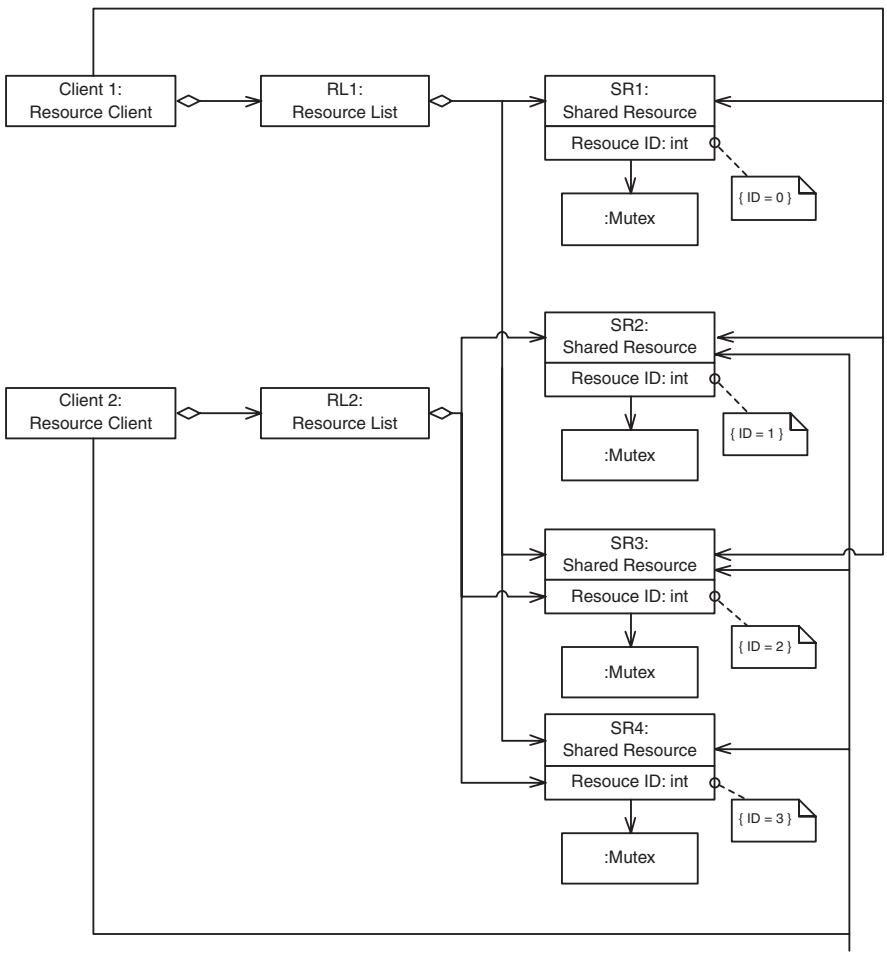
There are two other patterns here that prevent deadlock. The Simultaneous Locking Pattern locks all the needed resources in a single critical section; other *Resource Clients* that need to run can do so as long as they don't request any of the same *Shared Resources.* If a small subset of the resources need to be locked at any given time or if the sets needed for different *Resource Clients* overlap only slightly, then the Simultaneous Locking Pattern works well.

The Priority Ceiling Pattern solves the deadlock problem as well, although the algorithm is significantly more complex. For that added sophistication, the Priority Ceiling Pattern also bounds priority inversion to a single level.

7.7.8 Sample Model

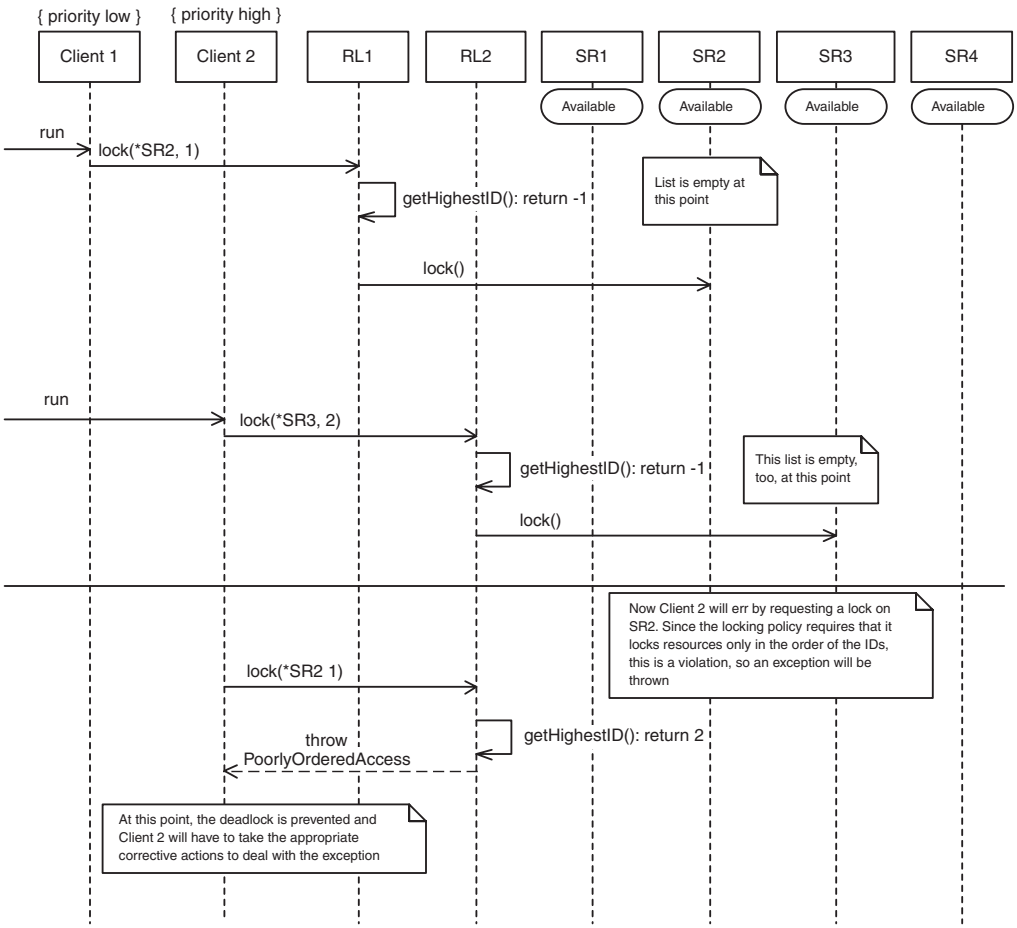
The example shown in Figure 7-18a provides a simple illustration of how the pattern works. *Client 1* uses three *Shared Resources: SR1* (ID=0), *SR3* (ID=2), and *SR4* (ID=3). *Client 2* uses three *Shared Resources: SR2* (ID=1), *SR3* (ID=2), and *SR4* (ID=3). They both, therefore, share SR2, SR3, and SR4.

Note that in the absence of some scheme to prevent deadlock (such as the use of the Ordered Locking Pattern), deadlock is easily possible in this configuration. Suppose *Client 1* ran and locked SR2, and when it was just about to lock SR3, *Client 2* (running in a higher-priority thread) preempts *Client 1. Client 2* now locks SR3 and tries to lock SR2. It cannot, of course, because it is already locked (by *Client 1*), and so it must block and allow *Client 1* to run until it releases the resource. However, now *Client 1* cannot successfully run because it needs SR3, and it is locked by *Client 2.* A classic deadlock situation. This particular scenario is not allowed with the Ordered Locking Pattern. Figure 7-18b shows what happens when this scenario is attempted.



a. Ordered Locking Pattern Example

Figure 7-18: *Ordered Locking Pattern*



b. Scenario

Figure 7-18: *Ordered Locking Pattern (continued)*

7.8 References

[1] *Response to the OMG RFP for Schedulability, Performance, and Time, Revised Submission,* Boston, MA: Object Management Group OMG Document Number: ad/2001-06-14, 2001.

[2] Klein, M., T. Ralya, B. Pollak, R. Obenza, and M. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems,* Norwell, MA: Kluwer Academic Press, 1993.

[3] Douglass, Bruce Powel. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns,* Reading, MA: Addison-Wesley, 1999.

[4] Douglass, Bruce Powel. *Real-Time UML 2nd Edition: Developing Efficient Objects for Embedded Systems,* Boston, MA: Addison-Wesley, 2000.