Composing Special Memory Allocators in C++

Keith Loepere
Open Software Foundation

Composing Special Memory Allocators in C++

Keith Loepere[1]

Open Group Research Institute
loepere@osf.org

The nature of complex, high performance system software imposes constraints on the nature and operation of memory allocation or requires special properties to be true for the memory. Often multiple constraints or properties must hold at once. It is desirable to express these constraints and properties as classes and methods. This paper presents a technique for expressing and composing special memory allocation mechanisms in C++.

Introduction

The Open Group Research Institute recently completed the initial version of MK++ [5], an operating system microkernel implemented in C++, based on the concepts that underlie the Mach operating system. The long term objective of the MK++ project is to produce a high performance, scalable microkernel technology that supports real-time computing and provides a high level of assurance; indeed, the current implementation of MK++ is being used as the basis for a highly secure operating system environment, while providing excellent performance.

The nature of a microkernel's software imposes considerable constraints on the use of memory — how it is laid out, how usage is accounted, the conditions under which allocation is performed, and so on. In the past, OS's provided and satisfied these constraints through code scattered throughout the system, at the points where such storage is allocated, used and de-allocated. Since MK++ was being written in C++, with extensive object orientation, it seemed natural to encapsulate the implementation of these special memory mechanisms within classes and methods, especially the C++ language defined memory allocation operators.

Most examples in the literature of the use of the C++ ability to define special memory allocation ([6] being the best tutorial) concerns using type specific knowledge to optimize allocation, e.g. pool allocation. There are many other reasons why one would want to have special memory allocators:

- support for variable sized objects
- data caching
- storage recycling
- allocation from special (physical) memory areas
- dynamic grouping
- real-time memory management
- memory resource control
- debugging

This paper starts with a discussion of some of these special allocation needs as motivation. The body of the paper presents a technique by which these allocation needs can be expressed and a technique for composing special memory allocation mechanisms in C++.

Special Allocation Needs

Complex system and application software has many needs for special memory allocators. Some are considered here.

Buffer Objects

The first example concerns an object that has an associated variable sized buffer. The C trick:

```
struct foo {...; char buf[1];};
```

(in which additional space is allocated beyond the declared end of the structure as an "extension" of `buf`, and in which the user will explicitly override the subscript of `buf`) does not work in C++. The compiler cannot be relied upon not to place fields past the declared end of a class, especially in the face of derivation. Without the use of special allocation, an object with a variable sized buffer would have to be allocated in two pieces — the object and the buffer —

1. This research was supported in part by the Advanced Research Projects Agency (ARPA) and the Air Force Materiel Command (AFMC).

requiring two calls to the underlying storage allocator. With the use of a special allocator, these pieces can be allocated as one unit.

Debugging

An obvious place to place memory allocation debugging logic is in special allocators — e.g., allocating extra storage as guard areas, keeping redundant size information and providing memory tracing.

Pool Allocation

Pool allocation refers to a range of specialized allocation strategies in which separate logical memory areas are established for allocation of some subset of the objects in the system. A common property of these strategies is that some class-specific knowledge is used to optimize allocation. The most common pool allocation strategy divides a memory pool into fixed sized chunks, where the chunk size is the size of some class of objects. Allocation of one those objects becomes a simple matter of finding a free chunk.

Pool allocation can also be used for allocation of objects of disjoint classes for which some common storage property must hold. In real-time processing, dedicated areas may be set aside for allocations that must not fail, or that must complete in bounded time [7]. Another example is the allocation of objects in shared memory [3] where the particular region needs to be specified. Unlike the earlier examples above, in which the identity of the pool can be implicit in the class of objects involved, in these latter examples the identity of the pool must be explicitly passed to the special allocator (by overloading the new operator).

A related idea is recyclable storage, in which there are *recycling center* objects that hold onto some number of storage units that were previously allocated so as to reduce the cost of subsequent allocations. The idea is similar to pool allocation, except that no special area need be set aside up front for the objects' storage, and only a limited amount of storage is allowed to be idle.

Per-Thread Caching

In a multi-threaded environment, a shared memory pool or recycling center requires some form of concurrency control (such as locking) to provide safety of the allocation and de-allocation. When high performance is needed, one can avoid this synchronization overhead by caching some storage against each thread, via a special allocator.

Dynamic Grouping

Traditional virtual memory systems often perform poorly when supporting large object systems because of the typically small object size and potentially far flung allocation of related objects. For a consideration of the issues, [8] discusses the idea of dynamic grouping of objects, [4] discusses the development of application-specific paging mechanisms, and [1] can be thought of as a combination.

Special allocators have two roles to play in this area. First of all, they can encapsulate the mechanisms that would make allocation time decisions as to the placement of objects based on some understanding of the usage patterns. Secondly, special allocators can accumulate allocation pattern information, perhaps for making dynamic placement decisions, or possibly for off-line analysis to be fed into developing pre-optimized placement strategies.

Memory Resource Control

A server maintaining state on behalf of multiple clients may face problems managing its memory resources. As part of making fairness decisions, or considering load distribution, or generally assessing the burden of the various clients, it can be desirable to keep some knowledge of the memory resources consumed by various activities. Implementing such accounting in special memory allocators makes the accounting automatic and centralizes the logic involved in maintaining the information.

Language Defined Memory Allocators

This section reviews C++'s support for dynamic memory allocation; an understanding of the language's more subtle aspects with respect to this functionality will simplify the discussion that follows. (Refer to [2] for a complete exposition.)

In C++, one dynamically (heap) allocates an object of class T as follows:

```
T * Tp = new(Nargs) T(Cargs);
```

This statement causes two functions to be invoked (in this order):

```
T::operator new(sizeof(T), Nargs)
T::T(Cargs)
```

That is, an operator new function is called, that defined for (or inherited by) class T[2], given the size of the object

2. Or the global operator if no class operators are defined.

to allocate[3] along with additional arguments. This operator allocates the “raw” storage to hold the object. An appropriate constructor is then called to initialize the storage.

Conversely, the de-allocation of a dynamic (heap) object, given a pointer to it:

```
delete Xp;
```

causes the following two functions to be called (in this order):

```
T::~T()
T::operator delete(void *)
```

That is, the destructor for type T is called (with no arguments) followed by a call to an operator delete, given a pointer to the storage, which is expected to deal with the storage. Assuming the destructor is virtual[4], the operator delete called is that defined for (or inherited by) the most derived class with a destructor.

The rules of C++ significantly restrict the way in which special allocators can be implemented.

The pointer returned by operator new references the raw storage for the object; given the permission the language gives to the compiler to lay out that storage, only the most derived class has the property that the raw storage pointer points to itself. Given that one can derive new classes from a class T that defines an operator new, `T::operator new` and `T::T` do not inherently know the relationship between the raw storage and the base class storage. That is, these two operators cannot pass information between them through storage without extra help.[5]

Likewise, the pointer passed to any given destructor is not necessarily that passed to operator delete.

Although one can define multiple operator new’s to define multiple memory allocation behaviors, the impracticality of having the single operator delete coping with all of those behaviors leads one to encapsulating different behavior via different operator new / operator delete pairs. In any case, since one cannot pass additional arguments to the destructor or operator delete, they need extra help to have all the information they need lying around.

The next section introduces a stylized and composable technique for providing this “extra help.”

Generalizing Allocation Information

One of the key goals of the structured special memory allocators is to encapsulate the special memory behavior in memory operators. To be able to define a new operator new / operator delete pair, one must declare a new class, so it is really a class that encapsulates the special memory behavior. However, given a class that defines special memory behavior, it is desirable to be able to derive other classes from it, without those classes needing to be aware of the special memory allocators. Since each of the operator new, constructor, destructor and operator delete methods for the special memory class will be provided with different information such that they do not know, or have consistent knowledge, of the layout of the overall object being allocated, the object itself does not serve to hold the special information that operator new and operator delete will use. Thus, this extra information must be kept outside of the object proper.

For performance, the allocation of a special memory object should not involve two calls to the underlying heap allocator; the object storage and the allocation information want to be kept together.

This special allocation information is here called the *allocation record*. As described in more detail below, allocation records derive from the base class `allocrec` and objects allocated with allocation records derive from the base class `allocation`. The job of `allocation::operator new` is to arrange for the allocation of sufficient storage for the object proper and its allocation record, get the allocation record initialized, and generally arrange the storage so that subsequent constructors, methods and finally operator delete can find the components. Figure 1 shows the basic allocation layout as arranged by `allocrec` and `allocation`. The reason for this layout is coupled to the mechanics of how it is established and used.

Since the special allocation properties are associated with the allocation record information, that record is a class in its own right (`allocrec`) to which class `allocation` delegates to provide the mechanics. `allocation::operator new` delegates to the static method `allocrec::alloc`, which allocates space for the allocation record, a “back pointer” and

3. This is the size of the most derived class, even if the most derived class does not define these special operators.
4. Otherwise the operator delete is chosen by the static type of Xp.
5. With `dynamic_cast<void *>`, a base class constructor can locate the beginning of the most derived class (the raw storage), but the converse is not true — operator new, knowing the location of the raw storage, cannot reasonably locate where any particular base class will lie within that, nor can safely store data there.

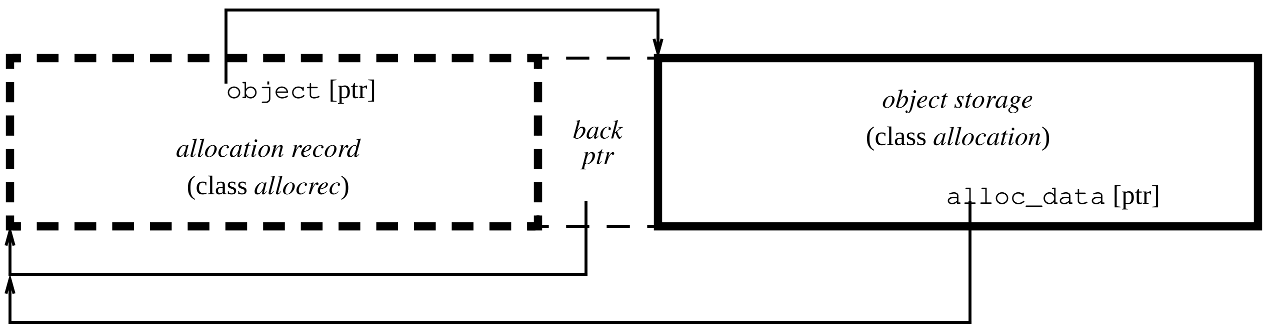

Figure 1 Object Allocation Layout

then space for the object proper. The back pointer enables `allocation::allocation` and `allocation::operator delete` to find the allocation record. `allocation::allocation` can find the back pointer (in the presence of derivation from class `allocation`) by locating the beginning of the most derived class (with `dynamic_cast<void *>`) and passing that pointer to `allocrec::record`. `allocation::allocation` saves the pointer to the allocation record (in `alloc_data`) so that methods may efficiently access the allocation information. Definition 1 defines class `allocation`.

```
class allocation {
public:
   allocation(): alloc_data(allocrec::record(dynamic_cast<void *>(this))) {}
   void * operator new(size_t obj_size) {
      return allocrec::alloc(obj_size);
   }
   void operator delete(void * item_base) {
      allocrec::record(item_base)->discard();
   }
   virtual ~allocation() {}
protected:
   allocrec * const alloc_data;                  // allocation record base
};
```

Definition 1 Class allocation

`allocation::operator delete` delegates to class `allocrec` via the virtual `discard` method to free or otherwise process the storage. When `allocation::operator delete` is called, the allocation object will have been destructed, and the pointer passed to operator delete will locate the raw storage, as opposed to the `allocation` class (in the presence of derivation). As such, operator delete needs to reference through the back pointer (by delegating to class `allocrec`).

Definition 2 defines class `allocrec`. Note the overloaded operator new so as to receive the size of the object itself (since the compiler would only pass the size of the allocation record). Note that the back pointer points to the `allocrec` base class (as opposed to the most derived allocation record class). As such, `allocrec::operator new` (which doesn't know the layout of the most derived allocation record) can't set the back pointer; only `allocrec::allocrec` can. `allocrec::allocrec` saves a pointer to the raw storage for the object proper in field `object`.

The `allocrec` class accepts three size parameters: the total size of the allocation record, the total size of the object proper, and an "extra" size. The purpose of this third size is for extra space allocated between the allocation record and the (back pointer before the) object. The code assumes that the caller ensured that the "extra" size preserves pointer alignment by being a multiple of the size of a pointer. (The code shown assumes that the alignment requirements for pointers are the strictest of that for any fundamental type.)

```
class allocrec {
public:
   allocrec(size_t rec_size, size_t extra_size):
      object(((char *)dynamic_cast<void *>(this))+ rec_size+ extra_size+
         sizeof(allocrec *)) {
      *(((allocrec **)object)-1) = this;        // set back pointer
   }
   static void * alloc(size_t obj_size) {       // “allocate”
      return (new(obj_size, 0) allocrec())->object;
   }
   virtual void discard() {                     // “de-allocate”
      delete this;
   }
   void * operator new(size_t rec_size, size_t obj_size, size_t extra_size) {
      return malloc(rec_size+ obj_size+ extra_size+ sizeof(allocrec *));
   }
   void operator delete(void * alloc_base) {
      free(alloc_base);
   }
   static allocrec * record(void * item_base) {
      return *(allocrec **)(((char *)item_base)- sizeof(allocrec *));
   }
   virtual ~allocrec() {}
   void * const object;                         // most derived object class
protected:
   allocrec(): // virtual bases require this definition
      object(((char *)dynamic_cast<void *>(this))+ sizeof(allocrec)+
         sizeof(allocrec *)) {
      *(((allocrec **)object)-1) = this;
   }
};
```

Definition 2   Class allocrec

Composing Special Allocation Properties

This section demonstrates by example how to encapsulate and compose special allocation properties by deriving from the classes `allocrec` and `allocation`.[6]

Buffer Object

With the use of a special allocator, an object with a variable sized buffer can have its two pieces — the object and the buffer — allocated as one unit. Figure 2 shows the memory layout. The buffer is kept in the “extra” space allotted by the `allocrec` class.

6. The examples are representative of special allocators in MK++. Various details of the actual MK++ microkernel environment are omitted; as such, these examples are a simplification and yet a generalization of the MK++ allocators.

Definition 3 defines an allocation record supporting an arbitrary sized buffer. The `alloc` method shows the typical form — an overloaded new of that type’s allocation record with extra space set aside for the object and buffer. The constructor assumes that this class is the only one that requests and uses the “extra” space. A more general buffer allocator could subdivide the extra area as indicated by derived allocators.

Definition 4 shows the object definition. The operator new has the typical form — simple delegation to the allocation record type. For efficiency of later access, the constructor fetches and saves a pointer to the buffer area. Note, as shown in Definition 5, how derivation is possible from class `buffer` with no special concern.

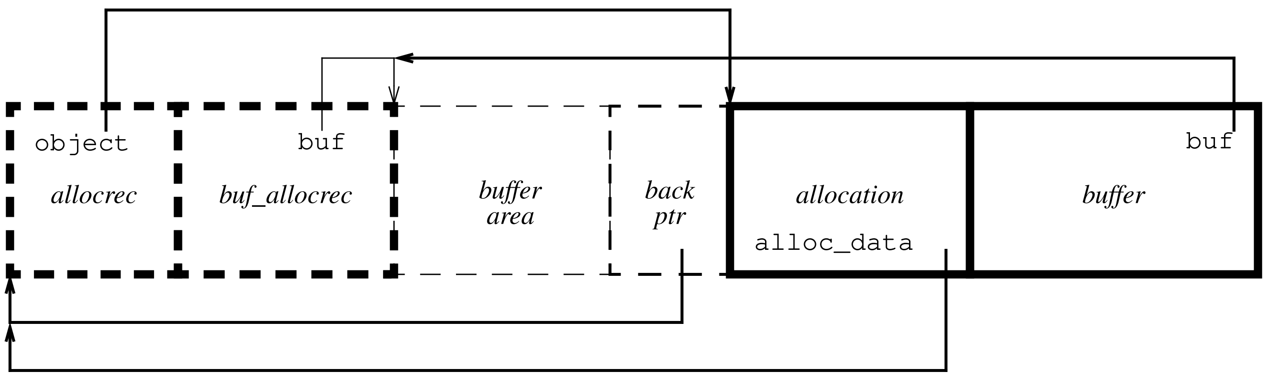

Figure 2 Buffer Object Allocation Layout

```
class buf_allocrec: public virtual allocrec {
public:
   buf_allocrec(size_t rec_size, size_t buf_size): allocrec(rec_size,
         buf_size), buf(((char *)dynamic_cast<void *>(this))+rec_size) {}
   static void * alloc(size_t obj_size, size_t buf_size) {
      return (new(obj_size, buf_size)
         buf_allocrec(sizeof(buf_allocrec), buf_size))->object;
   }
   void * const buf;
};
```

Definition 3 Class buf_allocrec

```
class buffer: public virtual allocation {
public:
   buffer(): allocation(),
      buf(dynamic_cast<buf_allocrec *>(alloc_data)->buf) {}
   void * operator new(size_t obj_size, size_t buf_size) {
      return buf_allocrec::alloc(obj_size, buf_size);
   }
   void * const buf;
};
```

Definition 4 Class buffer

```
class message: public buffer {
public:
   message(destination & a_target):
      allocation(), buffer(), target(a_target) {}
protected:
   destination & target;
};
```

Definition 5 Class message

Recyclable Storage

In this example, a *recycling center* (class `recycler`) is assumed to have a `get` method, which returns an unconstructed (raw) storage unit, if one is available, and a `put` method, which accepts a destructed (raw) storage unit (unless doing so would exceed some recycling policy, in which case the storage will need to be released).

The design decision here was to make recycling an allocation property rather than an object property. If recycling were an object property, than it would be done against a constructed object, most likely leaving it constructed. By making it an allocation property, recycling is done by operator new and operator delete, against raw storage either never constructed or subsequently destructed.

The reason for this decision (aside from purity) is that it allows recycling to occur without the object's user's knowledge; the user can simply "delete" the object when done, and operator magic recycles the storage for a subsequent "new". Note, though, that when the user "deletes" the object, it is destructed, but the allocation record is not. The allocation record is destructed only when the storage is to be de-allocated.

Figure 3 shows the allocation layout. Definition 6 defines the allocation record, which contains a reference to the target recycling center. The `alloc` method differs from the typical form in that it first asks the recycling center for storage before allocating some. Likewise, the `discard` method is redefined to try recycling before de-allocating the storage. Definition 7 defines the `recyclable` class.

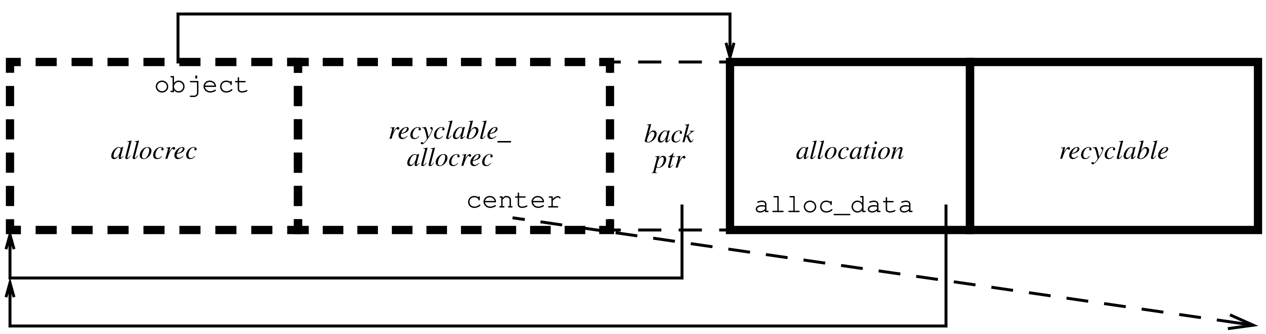

Figure 3 Recyclable Object Allocation Layout

```
class recyclable_allocrec: public virtual allocrec {
public:
   recyclable_allocrec(size_t rec_size, size_t extra_size, recycler &
         owner): allocrec(rec_size, extra_size), center(owner) {}
   void discard() {
      if (! center.put(object))
         delete this;
   }
   static void * alloc(size_t obj_size, recycler & owner) {
      void * target = owner.get();              // see if something recycled
      return target? target: (new(obj_size, 0)
         recyclable_allocrec(sizeof(recyclable_allocrec), 0, owner))->object;
   }
   recycler & center;
};
```

Definition 6 Class recyclable_allocrec

```
class recyclable: public virtual allocation {
public:
   recyclable(): allocation() {}
   void * operator new(size_t obj_size, recycler & owner) {
      return recyclable_allocrec::alloc(obj_size, owner);
   }
};
```

Definition 7 Class recyclable

Combining Allocators

The above allocators can be combined to provide a recyclable message object through multiple (virtual) inheritance.

Figure 4 shows the storage layout. This example emphasizes the reason why the pointers are established the way they are. Note that the `alloc_data` pointer does not point to the start of the allocation record, and so it is not possible to locate the buffer area if its location had not been recorded. Likewise, class `allocation` is not at the start of the most derived object, so the extra space cannot be located via its this pointer.

Definition 8 defines the allocation record class. The methods are a simple composition of the base classes. Note that `alloc` must be redefined, so as to allocate the appropriate allocation record, but `discard` need not be redefined — that from `recyclable_allocrec` is fine.

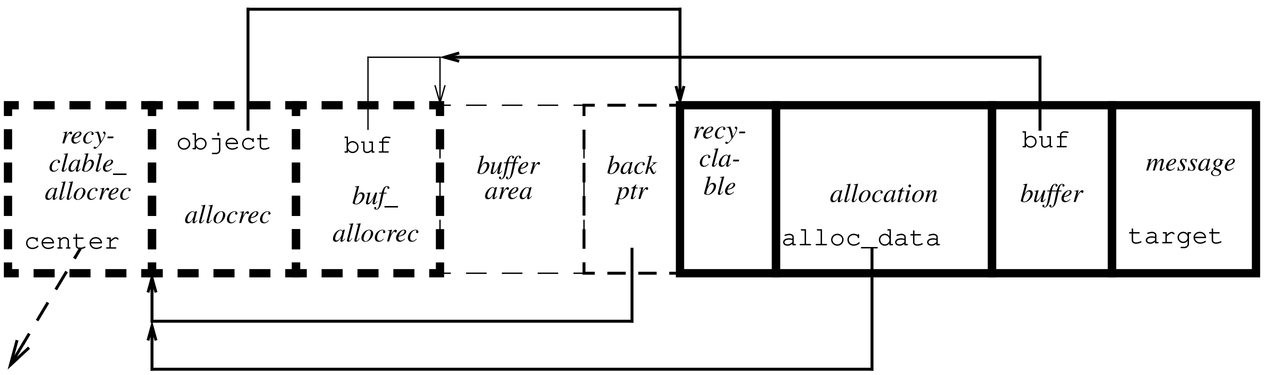

Figure 4 Recyclable message storage layout

```
class recyclablebuf_allocrec: public recyclable_allocrec, public buf_allocrec {
public:
   recyclablebuf_allocrec(size_t rec_size, size_t buf_size, recycler & owner):
         allocrec(rec_size, buf_size), buf_allocrec(rec_size, buf_size),
         recyclable_allocrec(rec_size, buf_size, owner) {}
   static void * alloc(size_t obj_size, size_t buf_size, recycler & owner) {
      void * target = owner.get();              // see if something recycled
      return target? target: (new(obj_size, buf_size)
         recyclablebuf_allocrec(sizeof(recyclablebuf_allocrec), buf_size,
         owner))->object;
   }
};
```

Definition 8 Class recyclablebuf_allocrec

```
class recyclable_message: public recyclable, public message {
public:
   recyclable_message(destination & a_target): allocation(),
         message(a_target), recyclable() {}
   void * operator new(size_t obj_size, size_t buf_size, recycler & owner) {
      return recyclablebuf_allocrec::alloc(obj_size, buf_size, owner);
   }
};
```

Definition 9 Class recyclable_message

Definition 9 defines the `recyclable_message` class, which has the typical form. Note that each derivation of the allocation record had a corresponding derivation of the object class, but the converse is not true. Specially allocated objects can inherit without affecting the storage allocators.

MK++ Evaluation

The technique presented in this paper is used extensively within the MK++ microkernel.

Uses

--- Buffer Objects

Aside from their use for messages, MK++ uses buffer objects for physical memory descriptors. The object proper provides the threading of the descriptor into an address space or message, and the buffer area holds an arbitrary sized array of physical page locators.

--- Storage Recycling

MK++ uses recyclable storage to support network input packet processing. When a network packet arrives, a packet object is selected to hold the data. Since the general system memory allocator can block waiting for availability of memory, and network input processing takes place in interrupt context where blocking is not permitted, this allocation must take place from a non-blocking, pre-allocated pool. The purpose of using recyclable storage has to do with the fact that the thread that subsequently processes the packet object does not know it is a packet object, only that it is some form of message. By using recyclable storage, the "deletion" of the message object arranges for the storage to be returned to the network input packet pool, not only replenishing the pool, but also self-throttling the maximum delivery rate of packets to the rate of packet processing. A memory pool with special resource management policies is also a candidate for storage recycling.

--- Memory Resource Control

An optional feature of MK++ is space accounting — resource management mechanisms that keep an accounting of memory utilization and take action when memory usage by a resource account exceeds reasonable limits. This feature is implemented in one of the most base classes in the allocator derivation hierarchy, providing for automatic accounting of storage. By design, each data structure in the kernel is considered to be "owned" by a specified high level system object (a "space source"). All allocations require a space source to be named (to operator new) which tracks the total memory consumption.

Space accounting is normally only enabled for environments in which uncontrolled memory usage by untrusted tasks may occur. However, it has found considerable use as a memory leak detector, asserting system failure when a space source is deleted without having de-allocated all its associated sub-structures.

Performance

The code shown in this paper was purposely coded using virtual base classes and virtual methods to show generality, and to show that the techniques work in the presence of those mechanisms. The uses within MK++, however, do not use virtual base classes or methods[7] and the code is extensively inlined.

An object allocated with an allocation record consumes additional space. It is assumed, for objects for which these techniques are needed, that the extra data in the allocation record (buffer information, for example) would be in memory somewhere anyway, so the real overhead is the various pointers linking the information. In the absence of this technique, the allocation information and the object proper would be allocated as two separate pieces with some sort of linkage, and the extra allocation unit is likely to have allocation overhead from the underlying system allocator, so the memory consumption balances out.

The ultimate test of the suitability of this technique for MK++ was the ability of the compiler (gcc in our case) to optimize it. For MK++' most important case, `new cached_message()` (determine the current thread, locate its cached storage and construct a message object in it) takes 11 instructions on an Intel 486, and the cost of the corresponding delete (to destruct the object, determine the current thread and return the cached storage to it) takes 10 instructions, each of which is barely more than the cost of the calling sequence to the system's allocator, yet alone the cost of executing any code within it.

Summary

The use of special allocators provides the structure by which complex memory allocation properties and mechanisms can be encapsulated and composed. The technique described in this paper extends the C++ inheritance mechanisms to the area of memory allocators. These composed allocators allow complex allocation properties to be encapsulated, not only hiding their details and removing their concerns from the users of the objects, but also allowing high performance memory mechanisms to be easily implemented.

References

[1] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska and Miche Baker-Harvey,

7. Virtual base classes and methods are used within MK++, but not currently within allocation record classes.

Lightweight Shared Objects in a 64-Bit Operating System, Proceedings OOPSLA '92, ACM SIGPLAN Notices, 1992.

[2] Margaret A. Ellis and Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison Wesley, 1990.

[3] David Jordan, *Instantiation of C++ Objects in Shared Memory*, Journal of Object-Oriented Programming, Vol 4, No 1, 1991.

[4] Keith Krueger, David Loftesness, Amin Vahdat and Thomas Anderson, *Tools for the Development of Application-Specific Virtual Memory Management*, Proceedings OOPSLA '93, ACM SIGPLAN Notices, 1993.

[5] MK++ Project, MK++ Kernel High Level Design, Open Group Research Institute, 1996.

[6] Robert B. Murray, C++ Strategies and Tactics, Addison Wesley, 1993.

[7] Franco Travostino, Franklin Reynolds and Tim Martin, *Real-Time Local and Remote MACH IPC: Architecture and Design*, Operating Systems Collected Papers, Vol. 3, Open Software Foundation, 1994.

[8] Ifor Williams, Mario Wolczko and Trevor Hopkins, *Dynamic Grouping in an Object-Oriented Virtual Memory Hierarchy*, Proceedings ECOOP '87, Springer-Verlag, 1987.