Major Technique: Small Architecture

How can you manage memory use across a whole system?

- Memory limitations restrict entire systems
- Systems are made up of many components
- Each component can be fabricated by a different team.
- Components' memory requirements can change dynamically.

A system's memory consumption is a global concern. Working well in limited memory isn't a feature that you can incorporate into your program in isolation: you can't ask a separate team of programmers to add code your system hoping to reduce its memory requirements. Rather, memory constraints cross-cut the design of your system, affecting every part of it. This is why designing software for systems for limited memory is difficult. [Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996; Shaw and Garlan 1996; Bass, Paul, and Kazman 1998; Bosch 2000].

For example, the Strap-It-On wrist-top PC's has an email application supporting text in a variety of fonts. Unfortunately in early implementations it cached every font it ever loaded, to improve performance there; but stored every email compressed, threw away attachments and crashed if memory ran out loading a font, giving poor new performance and awful usability. There's no sense in one function limiting its memory use to a few hundred bytes when another part of the program wastes megabytes, and then brings the system down when it fails to receive them.

You could simply design your system as a monolithic single component: a "big ball of mud" [Foote and Yoder 2000]. Tempting though this approach might be, it tends to be unsatisfactory for any but the simplest systems, for several reasons: it's difficult to split the development of such a system between different programmers, or different programming teams; the resulting system will be difficult to understand and maintain, since every part of the system can affect every other part; and you loose any possibility of buying in existing reusable components.

To keep control over your system, you can construct it from components that you can design, build, and test independently. Components can be reused from earlier systems, purchased from external suppliers, or built new; some may need specialised skills to develop; some may even be commercially viable in their own right. Each component can be assigned to a single team, to avoid several teams working on the same code [Szyperski 1999].

These components may be of many different kinds, and interact in many different ways: source libraries to compile into the system; object libraries that must be compiled into an executable; dynamic-linked libraries to load at run-time; run-time objects in separate address-spaces using frameworks like CORBA, Java Beans or ActiveX; or simply separate executables running in their own independent process. All are logically separate components, and communicate, if they communicate at all, through interfaces.

Unfortunately, separating a program into components doesn't reduce its memory use. The whole systems' memory requirements will be the sum of the memory required by each component. Furthermore, the memory requirements for each component, and so for the whole system, will change dynamically as the system runs. Even though memory consumption affects the architecture globally, it is still important that components can be treated separately as much as possible. How can you make the system use memory effectively, and give the best service to its users, in a system is divided into components?

Therefore: *Make every component responsible for its own memory use.*

A system's architecture is more than just the design of its high-level components and their interconnections: it also defines the system's *architectural strategies* — the policies, standards and assumptions common to every component [Bass et al 1998; Brooks 1982]. The architecture for a system for limited memory must describe policies for memory management and ensure that each component's allocations are feasible in the context of the system as a whole.

In a system for limited memory, this means that each individual component must take explicit responsibility for implementing this policy: for managing its own memory use. In particular, you should take care to design **SMALL DATA STRUCTURES** that require the minimum memory to store the information your system needs.

Taking responsibility for memory is quite easy where a component allocates memory statically (**FIXED ALLOCATION**); a component simply owns all the memory that is fixed inside it. Where a component allocates memory dynamically from a heap (**VARIABLE ALLOCATION**) it is more difficult to assign responsibility; the heap is a global resource. A good start is to aim to make every dynamically allocated object or record be owned by one component at all times. [Cargill 1996]. You may need to implement a **MEMORY LIMIT** or allocate objects using **POOLED ALLOCATION** for a component to control its dynamic memory allocation. Where components exchange objects, you can use **SMALL INTERFACES** to ensure some component always takes responsibility for the memory required for the exchange.

A system architecture also needs to set policies for mediating between components' competing memory demands, especially when there is little or no unallocated memory. You should ensure that components suffer only **PARTIAL FAILURE** when their memory demands cannot be met, perhaps by sacrificing memory from low priority components (**CAPTAIN OATES**) so that the system can continue to operate until more memory becomes available.

For example, the software architecture for the Strap-It-On PC defines the Font Manager and the Email Display as separate components. The software architecture also defines a memory budget, constraining reasonable memory use for each component. The designers of the Font Manager implemented a **MEMORY LIMIT** to reduce their font cache to a reasonable size, and the designers of the Email Display component discovered they could get much better performance and functionality than they had thought. When the Email application displays a large email, it uses the **SMALL INTERFACE** of the Font Manager to reduce the size of the Font Cache. Similarly, when the system is running short of memory the font cache discards any unused items (**CAPTAIN OATES**).

Consequences

Handling memory issues explicitly in a program's architecture can reduce the program's *memory requirements,* increase the *predictability* of its memory use, and may make the program more *scalable* and more *usable.*

A consistent approach to handling memory reduces the *programmer effort* required since the memory policies do not have to be re-determined for each component. Individual modules and teams can co-ordinate smoothly to provide a consistent *global* effect, so users can anticipate the final system's behaviour when memory is low, increasing *usability.*

In general, explicitly describing a system's architecture increases its *design quality* improving *maintainability*.

However: designing a small architecture takes *programmer effort*, and then ensuring components are designed according to the architecture's rules takes *programmer discipline.* Making memory an architectural concern moves it from being a *local* issue for individual components and teams to a *global* concern, involving the whole project. For example, developers may try to minimise their components memory requirements at the expense of other components produced by other teams.

Incorporating external components can require large amounts *programmer effort* if they do not meet the standards set by the system architecture — you may have to re-implement components that cannot be adapted.

Designing an architecture to suit limited memory situations can restrict a program's *scalability* by imposing unnecessary restrictions should more memory become available.

❖ ❖ ❖

Implementation

The main ideas behind this pattern are 'consistency' and 'responsibility'. By splitting up your system into separate components you can design and build the system piece by piece; by having a common memory policy you ensure that the resulting pieces work together effectively.

The actual programming mechanism used to represent components is not particularly important. A component may be a class, a package or a namespace, a separate executable or operating system process, a component provided by middleware like COM or CORBA, or an ad-hoc collection of objects, data structures, functions and procedures. In an object-oriented system, a component will generally contain many different objects, often instances of different classes, with one or more objects acting as **FACADES** to provide an interface to the whole component.

Here are two further issues to consider when designing interfaces for components in small systems:

1. Tailorability.

Different clients vary in the memory requirements they place on other components that they use. This is especially the case for components that are designed to be reusable; such components will be used in many different contexts, and those contexts may have quite different memory requirements.

A component can address this by including parameters to tailor its memory use in its interface. Clients can adjust these parameters to adapt the component to fit its context. Components using **FIXED ALLOCATION**, for example, have to provide creation-time parameters to choose the number of items they can store. Similarly, components using **VARIABLE ALLOCATION** can provide parameters to tune their memory use, such as maximum capacity, initial allocation, or even the amount of free space (in a hash table, for example, leaving free space can increase lookup performance). Components can also support operations to control their behaviour directly, such as requesting a database to compact itself, or a cache to empty itself.

For example, the Java vector class has several methods that control its memory use. Vectors can be created with sufficient memory to hold a given number of items (say 10):

```
Vector v = new Vector(10);
```

This capacity can be increased dynamically (say to store twenty items):

```
v.ensureCapacity(20);
```

The capacity can also be reduced to provide only enough memory for the number of elements in the container, in this case one object.

```
v.addElement( new Object() );
v.trimToSize();
```

Allocating correctly sized structures can save a surprisingly large amount of memory and reduce the load a component places on a low level memory allocator or garbage collector. For example, imagine a Vector that will be used to store 520 items inserted one at a time. The vector class initially allocates enough space for 8 elements; when that is exhausted, allocates twice as much space as it is currently using, copies its current elements into the new space, and deallocates the old space. To store 520 elements, the vector will resize itself seven times, finally allocating almost twice the required memory, and having allocated about four times as much memory in total. In contrast, initialising the vector with 520 elements would have required one call to the memory system and allocated only as much memory as required [Soukup 1994].

2. Make clients responsible for components' memory allocation.

Sometimes a component needs to support several radically different policies for allocating memory — some clients might want to use **POOLED ALLOCATION** for each object allocated dynamically within the package; others might prefer a **MEMORY LIMIT** or to use **MEMORY DISCARD**; and still others might want the simplicity of allocating objects directly from the system heap. How can you cater for all of these with a single implementation of the component?

2.1. Callbacks to manage memory. A simple approach is to require the component to call memory management functions provided by the client. In non-OO environments, for example, you can make the component call a function supplied by its client, and linking the client and component together. In C, you might use function pointers, or make the component declare a function prototypes to be implemented by the library environment. For example, the Xt Window System Toolkit for the X Window System supports a callback function, the `XAlloc` function hook; clients may provide a function to do the memory allocation (and of course, another function to do the memory freeing) [Gilly and O'Reilly 1990].

2.2. Memory Strategy. In an object-oriented environment, you can apply the **STRATEGY** pattern: define an interface to a family of allocation algorithms, and then supply the component with the algorithm appropriate for the context of use. For example, in C++, a strategy class can simple provide operations to allocate and free memory:

```
class MemoryStrategy {
    virtual char* Alloc( size_t nBytes ) = 0; // returns null when exhausted.
    virtual void Free( char* anItem; ) = 0;
};
```

Particular implementations of the `MemoryStrategy` class then implement a particular strategy: a `PooledStrategy` implements **POOLED ALLOCATION**, a `LimitStrategy` applies a **MEMORY LIMIT**; a `TemporaryHeapStrategy` implements **MEMORY DISCARD**; and a `HeapStrategy` simply delegates the `Alloc` and `Free` operations straight to the system `malloc` and `free` functions.

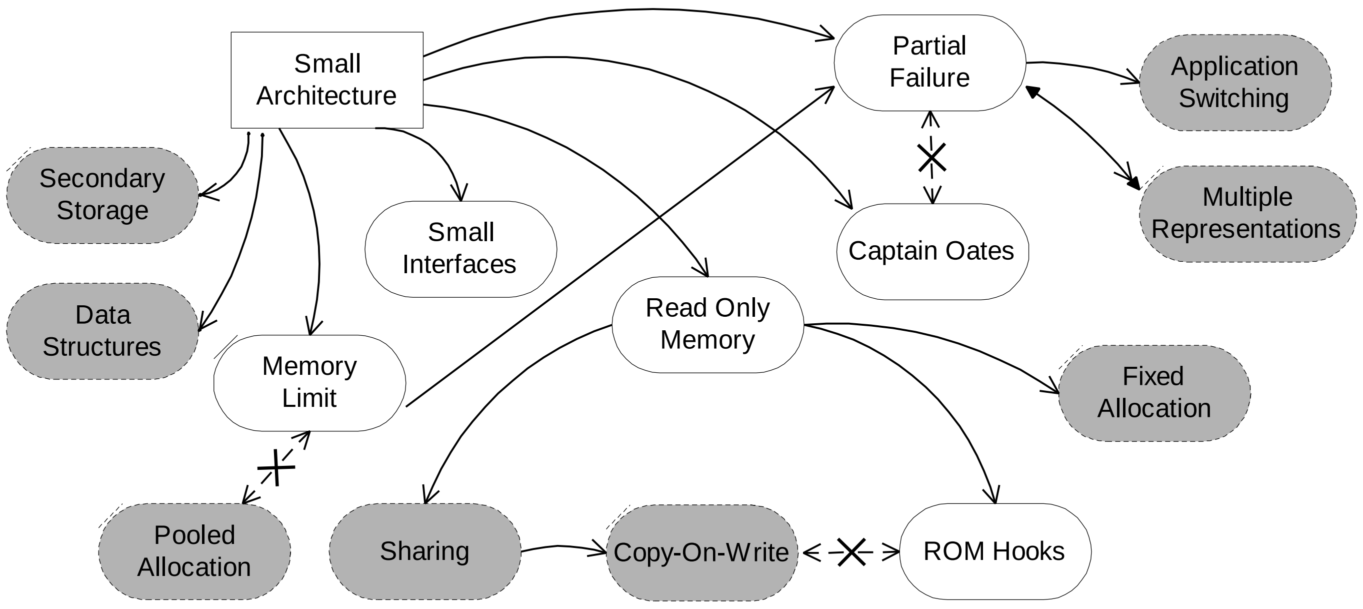
An alternative C++ design uses compile-time template parameters rather than runtime objects. The C++ STL collection and string templates accept a class parameter (called `Allocator`) that provides allocation and freeing functions [Stroustrup 1997]. They also provide a default implementation of the `allocator` that uses normal heap operations. So the definition of the STL 'set' template class is:

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator> class set;
```

Note how the `Allocator` template parameter defaults to `allocator`, the strategy class that uses normal heap allocation.

Specialised Patterns

The following sections describe six specialised patterns that describing ways architectural decisions can reduce RAM memory use. The figure below shows how they interrelate. Two other patterns in this book are closely related to the patterns in this chapter, and these patterns are shown in grey.



This chapter contains the following patterns:

MEMORY LIMIT enforces a fixed upper bound on the amount of memory a component can allocate.

SMALL INTERFACES between components are designed to manage memory explicitly, minimising the memory required for their implementation.

PARTIAL FAILURE ensures a component can continue in a 'degraded mode', without stopping its process or losing existing data, when it cannot allocate memory.

CAPTAIN OATES improves the overall performance of a system, by surrendering memory used by less important components when the system is running low on memory.

READ-ONLY MEMORY can be used to store components that do not need to be modified, in preference to more constrained and expensive main memory.

HOOKS allow information stored in **READ-ONLY MEMORY** (or shared between components) to appear to be changed

Known Uses

Object-oriented APIs designed to support different memory strategies include the C++ standard template library [Stroustrup 1997] and the Booch components [Booch 1987]. These libraries, and to a lesser extent the standard Java and Smalltalk collection classes, also provide parameters that adjust components' strategies, for example, by preallocating the correct amount of memory to hold an entire structure.

See Also

Many small architectures take advantage of **SECONDARY STORAGE** to reduce requirements for main memory. Architectures can also design **SMALL DATA STRUCTURES** to minimise their memory use, and encourage **SHARING** of code and data between components.

Tom Cargill's patterns for *Localized Ownership* [Cargill 1996] describe how you can ensure every object is the responsibility of precisely one component at all times. The **HYPOTH-A-SIZED COLLECTION** pattern [Auer and Beck 1996] describes how collections should be created with sufficient capacity to meet their clients needs without extra allocations.

Software Requirements & Specifications [Jackson 1995] and *Software Architecture* [Shaw and Garlan 1996] describe ways to keep a coherent architecture while dividing an entire system into components. *Software Architecture in Practice* [Bass et al 1998] describes much about software architecture; *Design and Use of Software Architectures* [Bosch 2000] is a newer book that focuses in particular on producing product-lines of similar software systems. *Patterns in Software Architecture* has a number of architecture-level patterns to help design whole systems and is well worth reading [Buschmann et al 1996].

The *Practice of Programming* [Kernighan and Pike, 1999], the *Pragmatic Programmer* [Hunt and Thomas 2000] and the *High-Level and Process Patterns from the Memory Preservation Society* [Noble and Weir 2000] describe techniques for estimating the memory consumption of a system's components, and managing those estimates throughout a development project.

Memory Limit

Also Known As: Fixed-sized Heap, Memory Partitions

How can you share out memory between multiple competing components?

- Your system contains many components, each with its own demands on memory.
- Components' memory requirements change dynamically as the program runs.
- If one component hogs too much memory, it will prevent others from functioning.
- You can define reasonable upper bounds on the memory required for each task.

As part of designing a **SMALL ARCHITECTURE,** you will have divided up your system into architectural components, and made each component responsible for its own memory use. Each components' memory demands will change as the program runs, depending on the overall kind of load being placed on the system. If access to memory is unrestricted, then each component will try to allocate as much memory as it might need, irrespective of the needs of other components. As other components also allocate memory to tackle their work, the system as a whole may end up running out of memory.

For example, the Strap-It-On's Virtual Reality "Stair Wars 1" game has several components: virtual reality display, voice output, music overview, voice recognition, not to mention the artificial intelligence brain co-ordinating the entire game plan. Each of these tasks is capable of using as much memory as it receives, but if every component tries to allocate a large amount of memory there will not be enough to go round. You must apportion the available memory sensibly between each component.

You could consider implementing the Captain Oates pattern, allowing components low on memory to steal it from components with abundant allocations. Captain Oates relies on the goodwill of component programmers to release memory, however, and can be difficult and complex to implement.

You could also consider budgeting components' memory use in advance. Just planning memory consumption is also insufficient; however, unless there is some way to be sure that components will obey the plans. This is trivial for components that use **FIXED ALLOCATION** exclusively, but for others it can be difficult to model their dynamic behaviour to be sure they will not disrupt your plans.

Therefore: *Set limits for each component and fail allocations that exceed the limits.*

There are three steps to applying the memory limit pattern.

1. Keep an account of the memory currently allocated by each component. For example, you might modify a component's memory allocation routine to increase a memory counter when allocating memory, and decrease the counter when deallocating memory.
2. Ensure components cannot allocate more memory then an allotted limit. Allocation operations that would make a component exceed its limit should fail in exactly the same way that they would fail if there were no more memory available in the system. Components should support **PARTIAL FAILURE** so that they can continue running even when they are at the limit.
3. Set the limits for each component, ideally by experimenting with the program and examining the memory use counters for each component. Setting the limits last may seem to be doing things backwards, but, in practice, you will have to revise limits during

development, or alternatively allow users to adjust them to suit their work. So, build the accounting mechanisms first, experiment gathering usage information, and then set the memory use policies that you want enforced.

Should the sum of the limits for each component be equal or greater than the total available? The answer depends on whether all the tasks are likely to be using their maximum memory limit simultaneously. This is unlikely in practice, and the main purpose of the Memory Limit pattern is to prevent a single component from hogging all the memory. It is generally sufficient to ensure that the limit on each task is a reasonable fraction of the total memory available.

Note that it's only worth implementing a limit for components that make variable demands on memory. A memory limit provides little benefits for components where most data structures use **FIXED ALLOCATION** and the memory use doesn't vary significantly with time.

In the 'Stair Wars' program, for example, the artificial intelligence brain component uses memory roughly in proportion to the number of hostile and friendly entities supported. By experimenting with the game, the developers determined a maximum number of such entities, and then adjusted brain component's memory limit to provide enough memory to support the maximum. On the other hand, the screen display component allocates a fixed amount of memory, so Stair Wars doesn't apply an extra memory limit for this component.

Consequences

Because there are guaranteed limits on the memory use of each component, you can *test* each one separately, while remaining sure that it will continue to work the same way in the final system. This increases the *predictability* of the system.

By examining the values of the memory counters, it's easy to identify problem areas, and to see which components are failing due to insufficient memory at run-time, increasing the *localisation* of the system.

Implementing a simple memory counter takes only a small amount of *programmer effort.*

However: Some tasks may fail due to lack of memory while others are still continuing normally; if the tasks interact significantly, this may lead to unusual error situations which are difficult to reproduce and *test.* A component can fail because it's reached its memory limit even when there is plenty of memory in the system; thus the pattern can be wasteful of memory. Most simple memory counters mechanisms don't account for extra wastage due to *fragmentation* (see the **MEMORY ALLOCATION** chapter). On the other hand, more complex operating system mechanisms such as separate heaps for each component tend to increase this same *fragmentation* wastage.

❖ ❖ ❖

Implementation

There are several alternative approaches to implementing memory limits.

1. Intercepting Memory Management Operations.

In many programming languages, you can intercept all operations that allocate and release memory, and modify them to track the amount of memory currently allocated quite simply. When the count reaches the limit, further memory allocations can fail until deallocations return the count below the limit. In C++, for example, you can limit the total memory for a process by overriding the four global `new` and `delete` operators [Stroustrup 1995].

A memory counter doesn't need to be particularly accurate for this pattern to work. It can be sufficient to implement a count only of the major memory allocations: large buffers, for example. If smaller items of allocated memory are allocated in proportion to these larger items, then this limit indirectly governs the total memory used by the task. For example, the different entities in the Stair Wars program each use varying amounts of memory, but the overall memory use is roughly proportional to the total number of entities, so limiting them implemented an effective memory limit.

In C++ you can implement a more localised memory limit by overriding the `new` and `delete` operators for a single class – and thus for its derived classes. This approach also has the advantage that different parts of the same program can have different memory limits, even when memory is allocated from a single heap [Stroupstrup 1997].

2. Separate Heaps.

You can make each component use a separate memory heap, and manage each heap separately, restricting their maximum size. Many operating systems provide support for separate heaps (notable Windows and Windows CE) [Microsoft 97, Boling 1998].

3. Separate processes.

You can make each component an individual process, and use operating system or virtual machine mechanisms to limit each component's memory use. EPOC, and most versions of Unix allow you to specify a memory limit for each process, and the system prevents processes from exceeding these limits. Using these limits requires little *programmer effort*, especially the operating systems also provides tools that can monitor processes memory use so that you can determine appropriate limits for each process. Of course, you have to design or whole system so that separate components can be separate processes — depending on your system, this can be trivial or very complex.

Many operating systems implement heap limits using virtual memory. They allocate the full size heap in the virtual memory address space (see the **PAGING PATTERN**); the memory manager maps this to real memory only when the process chooses to access each memory block. Thus the heap sized is fixed in virtual memory, but until it is used there's no real memory cost at all. The disadvantage of this approach is that very few virtual memory systems can detect free memory in the heap and restore the unused blocks to the system. So in most VM systems a process that uses its full heap will keep the entire heap allocated from then on.

Examples

The following C++ code restricts the total memory used by a `MemoryResrictedClass` and its subclasses. Exceeding the limit triggers the standard C++ out of memory exception, `bad_alloc`. Here the total limit is specified at compile time, as `LIMIT_IN_BYTES`:

```
class MemoryRestrictedClass {
public:
    enum { LIMIT_IN_BYTES = 10000 };
    static size_t totalMemoryCount;

    void* operator new ( size_t aSize );
    void operator delete( void* anItem, size_t aSize );
};

size_t MemoryRestrictedClass::totalMemoryCount = 0;
```

The class must implement an `operator new` that checks the limit and throws an exception:

```
void* MemoryRestrictedClass::operator new ( size_t aSize ) {
    if ( totalMemoryCount + aSize > LIMIT_IN_BYTES )
        throw ( bad_alloc() );
    totalMemoryCount += aSize;
    return malloc( aSize );
}
```

And of course the corresponding `delete` operator must reduce the memory count again:

```
void MemoryRestrictedClass::operator delete( void* anItem, size_t aSize ) {
    totalMemoryCount -= aSize;
    free( (char*)anItem );
}
```

For a complete implementation we'd also need similar implementations for the array versions of the operators [Stroustrup 1995].

In contrast, Java does not provides allocation and deletion operations in the language. It is possible however to limit the number of instances of a given class by keeping a static count of the number of instances created. Java has no simple deallocation call, but we can use finalisation to intercept deallocation. Note that many Java virtual machines do not implement finalisation efficiently (if at all), so you should consider this code as an example of one possible approach, rather than as recommended good practice [Gosling et al 1996].

The following class permits only a limited number of instances. The class counts the number of its instances, increasing the count when a new object is constructed, and decreasing the count when it is finalized by the garbage collector. Now, since objects can only be finalized when the garbage collector runs, at any given time there may be some garbage objects that have not yet been finalised. To ensure we don't fail allocation unnecessarily, the constructor does an explicit garbage collection before throwing an exception if we are close to the limit.

```
class RestrictedClass
{
    static final int maxNumberOfInstances = 5;
    static int numberOfInstances = 0;

    public RestrictedClass() {
        numberOfInstances++;
        if (numberOfInstances > maxNumberOfInstances) {
            System.gc();
        }
        if (numberOfInstances > maxNumberOfInstances) {
            throw new OutOfMemoryError("RestrictedClass can only have " +
                                       maxNumberOfInstances + " instances");
        }
    }
```

There's a slight issue with checking for memory in the constructor: even if we throw an exception, the object is still created. This is not a problem in general, because the object will eventually be finalized unless one of the superclass constructors stores a reference to the object.

The actual finalisation code is trivial:

```
    public void finalize() {
        --numberOfInstances;
    }
};
```

❖ ❖ ❖

Known Uses

By default, UNIX operating systems put a memory limit on each user process [Card et al 1998]. This limit prevents any one process from hogging all the system memory as only processes with system privileges can override this limit. The most common reason for a process to reach the limit is a continuous memory leak: after a process has run for a long time a memory request will fail, and the process will terminate and be restarted.

EPOC associates a heap with each thread, and defines a maximum size or each heap. There is a default, very large, limit for applications, but server threads (daemons) are typically created with rather smaller limits using an overloaded version of the thread creation function `RRhread::Create` [Symbian 1999]. The EPOC culture places great importance on avoiding memory leaks, so the limit serves to limit the resources used by a particular part of the system. EPOC servers are often invisible to users of the system, so it is important to prevent them from growing too large. If a server does reach the memory limit it will do a **PARTIAL FAILURE**, abandoning the particular request or client session that discovered the problem rather than crashing the whole server [Tasker et al 2000].

Microsoft Windows CE and Acorn Archimedes RICS OS allow users to adjust the memory limits of system components at runtime. Windows CE imposes a limit on the amount of memory used for programs, as against data, and RISC OS imposes individual limits on every component of the operating system [Boling 1998, RISC OS 2000].

Java virtual machines typically provide run-time flags to limit the total heap size, so you can restrict the size of a Java process [Lindholm and Yellin 1999]. The Real-Time Specification for Java will support limits on the allocation of memory within the heap [Bollella et al 2000].

See Also

Since it's reasonably likely a typical process will reach the limit, it's better to suffer a **PARTIAL FAILURE** rather than failing the whole process. Using only **FIXED ALLOCATION** (or **POOLED ALLOCATION**) is a simpler, but less flexible, technique to apportion memory among competing components.

Small Interfaces

How can you reduce the memory overheads of component interfaces?

- You are designing a **SMALL ARCHITECTURE** where every component takes responsibility for its own memory use.
- Your system has several components, which communicate via explicit interfaces.
- Interface designs can force components or their clients to allocate extra memory, solely for inter-component communication.
- Reusable components require generic interfaces, which risk needing more memory than would be necessary for a specific example.

You are designing a **SMALL ARCHITECTURE,** and have divided your system into components with each component responsible for its own memory use. The components collaborate via their interfaces. Unfortunately the interfaces themselves require temporary memory to store arguments and results. Sending a large amount of information between components can require a correspondingly large amount of memory.

For example, the Strap-It-On 'Spookivity' ghost hunter's support application uses a compressed database in ROM with details of every known ghost matching given specifications. Early versions of the database component were designed for much smaller RAM databases, so they implemented a 'search' operation that simply returned a variable sized array of structures containing copies of full details of all the matching ghosts. Though functionally correct, this interface design meant that Spookivity required a temporary memory allocation of several Mbytes to answer common queries – such as "find ghosts that are transparent, whitish, floating and dead"- an amount of memory simply not available on the Strap-It-On.

Interfaces can also cause problems for a **SMALL ARCHITECTURE** by removing the control each component has over memory allocation. If an object is allocated in one component, used by another and finally deleted by a third, then no single component can be responsible for the memory occupied. In the Spookivity application, although the array of ghost structures was allocated by the database component it somehow became the responsibility of the client.

Reusable components can make it even more difficult to control memory use. The designer of a reusable component often faces questions about the trade-offs between memory use and other factors, such as execution speed or failure modes. For example, a component might pre-allocate some memory buffers to support fast response during normal processing: how much memory should it allocate? The answers to such questions depend critically on the system environment; they may also depend on which client is using the component, or even depend on what the client happens to be doing at the time. The common approach – for the designer to use some idea of an 'average' application to answer such questions – is unlikely to give satisfactory results in a memory limited system.

Therefore: *Design interfaces so that clients control data transfer.*

There are two main steps to designing component interfaces:

1. *Minimise the amount of data transferred across interfaces.* The principles of 'small interfaces' [Meyer 1997] and 'strong design' [Coplien 1994] say that an interface should present only the minimum data and behaviour to its client. A small interface should not transmit spurious information that most components or their clients will not

need. You can reduce the amount of memory overhead imposed by interfaces by reducing the amount of data that you need to be transfer across them.

2. *Determine how best to transfer the data.* Once you have identified the data you need to pass between components, you can determine how best to transfer it. There are many different mechanisms for passing data across interfaces, and we discuss the most important of them in the Implementation section.

For example, later versions of the Spookivity Database 'search' method returned a database **ITERATOR** object [Gamma et al 1995]. The iterator's '`getNext'` function returned a reference to a '`GhostDetails'` result object, which provided methods to return the data of each ghost in turn. This also allowed the implementers of the database component to reuse the same `GhostDetails` object each time; their implementation contained only a database ID, which they changed on each call. The `GhostDetails` methods accessed their data directly from the high-speed database. The revised interface required only a few bytes of RAM to support, and since the database is itself designed to use iterators there was no cost in performance.

Consequences

By considering the memory requirements for each component's interface explicitly, you can reduce the *memory requirements* for exchanging information across interfaces, and thus for the system as a whole. Because much of the memory used to pass information across interfaces is transient, eliminating or reducing interface's memory overheads can make your program's memory use more *predictable*, and support better *real-time* behaviour. Reducing inter-component interface memory requirements reduces the overheads of using more components in a design, increasing *locality* and *design quality* and *maintainability*.

However: Designing small interfaces requires *programmer discipline*, and increases team co-ordination overheads. A memory-efficient interface can be more *complex, and so* require more code and *programmer effort* and increase *testing costs*. As with all designs that save memory, designing small interfaces may increase *time performance*.

❖ ❖ ❖

Implementation

There are a number of issues and alternatives to consider when using designing interfaces between components in small systems. The same techniques can be used whether information is passing 'inward' from a client to a server, in the same direction as control flow (an *efferent flow* [Yourdon and Constantine 1979]), or 'outward' from component to client (an *afferent flow)*.

1. Passing data by value vs. by reference.

Data can be passed and returned either by value (copying the data) or by reference (passing a pointer to the data). Passing data by reference usually requires less memory than by value, and saves copying time. Java and Smalltalk programs usually pass objects by reference. Passing references does means that the components are now **SHARING** the data, so the two components need to co-operate somehow to manage the responsibility for its memory. On the other hand, in pass-by-value the receiving components must manage the responsibility for the temporary memory receiving the value. as well. Pass-by-value is common in C++, which can **DISCARD** stack memory.

2. Exchanging memory across interfaces.

There are three common strategies for a client to transfer memory across a component interface:

- **Lending** —some client's memory is lent to the supplier component for the duration of the clients call to the supplier (or longer).
- **Borrowing** —the client gets access to an object owned by the supplier component.
- **Stealing** — the client receives an object allocated by the supplier, and is responsible for its deallocation.

When information is passed inward the client can often *lend* memory to the component for the duration of the call. Returning information 'outward' from component to is more difficult. Although clients can *lend* memory to a supplier, it is often easier for the client to *borrow* a result object from the server, and easier still for the client to *steal* a result object and use it without constraint.

The following section describes and contrasts each of these three approaches. For convenience, we describe a component that returns a single result object; but the same sub-patterns apply when a number of objects are returned.

2.1. Lending: The client passes an object into the component method, and the component uses methods on the object to access its data. If the client keeps a reference to the result object, it can access the data directly, or the component can pass it back to the client. For example, the following Java code sketches how an object using a word processor component could create a new document properties object, and pass it to the word processor, which initialises it to describe the properties of the current document.

```
DocumentProperties d = new DocumentProperties();
wordProcessor.getCurrentDocumentProperties( d );
```

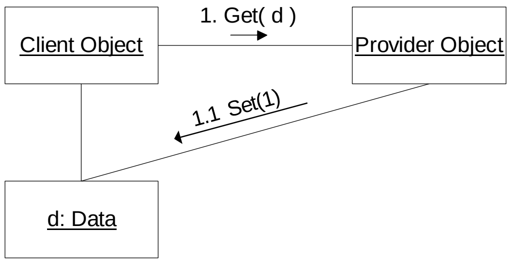
The client can then manipulate the document properties object:

```
long docsize = d.getSize();
long doctime = d.getEditTime();
```

The client must also release the document properties object when it is no longer useful:

```
d = null;
```

because it has kept the responsibility for the document properties object's memory.



When lending memory to a component, the client manages the allocation and lifetime of the data object (the document properties in this case), which may be allocated statically, or on the heap or the stack.

Consider using lending to pass arguments across interfaces when you expect the client to have already allocated all the argument objects, and when you are sure they will need all the results returned. Making the client own a result object obviously gives a fair amount of power and flexibility to the client. It does requires the client to allocate a new object to accept the results, and take care to delete the object when it is not longer needed, requiring *programmer discipline.* The component must calculate all the result properties, whether the client needs them or not.

In C++ libraries a common form of this technique is to return the result by value, copying from temporary stack memory in the component to memory lent by the client.

Another example of lending is where the client passes in a buffer for the component to use. For example in the **BUFFER SWAP** pattern, a component needs to record a collection of objects (e.g. sound samples) in real-time and return them to the client. The client begins by providing a single buffer to the main component, and then provides a new empty buffer every time it receives a filled one back. [Sane and Campbell 1996].

2.2. Borrowing: The component owns a simple or composite object, and returns a reference to that object to the client. The client uses methods on the object to access its data, then signals to the component when it no longer needs the object. For example, the word processor component could let its client borrow an object representing the properties of the current document:

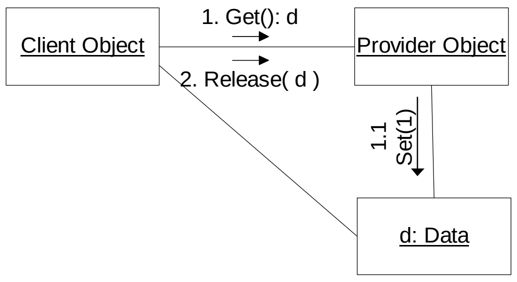
```
DocumentProperties d = wordProcessor.getDocumentProperties();
```

The client can then manipulate the document properties object:

```
long docsize = d.getSize();
long doctime = d.getEditTime();
```

but must tell the word processor when the properties object is no longer required.

```
wordProcessor.releaseDocumentProperties(d);
```



Like lending, borrowing can be used to transfer data both in to and out of a component. Having the component own the result object gives maximum flexibility to the component returning the result. The component can allocate a new data object each time (**VARIABLE DATA STRUCTURE**), or it can hold one or more instances permanently (**FIXED DATA STRUCTURE**), or some combination of the two.

On the other hand, the component now has to manage the lifetime of the result object, which is difficult if there are several clients or several data objects needed at a time. Alternatively, you can allocate only one result object statically, and recycle it for invocation. This requires the client to copy the information immediately it is returned (effectively similar to an ownership transfer). A static result object also cannot handle concurrent accesses, but this is fine as long as you are sure there will only ever be one client at a time.

Alternatively, the component interface can provide an explicit 'release' method to delete the result object. This is rarer in Java and Smalltalk, as these languages make it clumsy to ensure that the release method is called when an exception is thrown. This is quite common in C++ interfaces, as it allows the component to implement **REFERENCE COUNTING** on the object, or just to do `delete this` in the implementation of the `Release` function. For example, the EPOC coding style [Tasker et al 2000] is that all interfaces ('R classes') must provide a `Release` function rather than a destructor. Consider using borrowing when components need to create or to provide large objects for their clients, and clients are unlikely to retain the objects for long periods of time.

2.3. Stealing: The component allocates a simple or composite object, and transfers responsibility for it to the client. The client uses methods on the object to get data, then frees it (C++) or relies on garbage collection to release the memory. For example, the wordprocessor can let its client steal a document properties object:

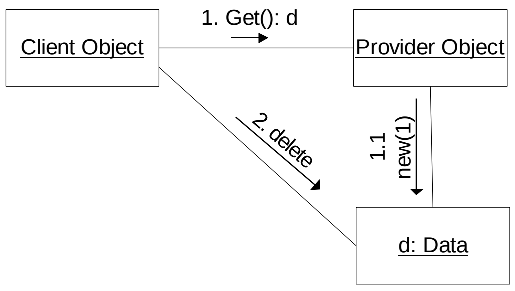
```
DocumentProperties d = wordProcessor.getDocumentProperties();
```

allowing the client to use it as necessary,

```
long docsize = d.getSize();
long doctime = d.getEditlong();
```

but the client now has the responsibility for managing or deleting the object

```
d = null;
```



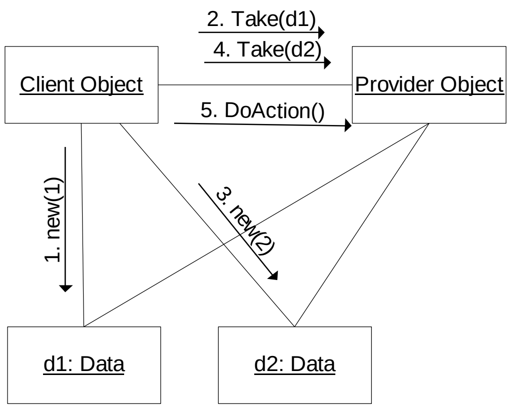
This example shows a client stealing an object originally belonging to a component, however, components can also steal objects belonging to their clients when data is flowing from clients to components. Transferring responsibility for objects (or ownership of objects) is simple to program, and is particularly common in languages such as Java and Smalltalk that support garbage collection and don't need an explicit `delete` operation. In C++ it's most suitable for variable size structures, such as unbounded strings. However in systems without garbage collection, this technique can cause memory leaks unless great *programmer discipline* is used to delete every single returned objects. Ownership transfer forces the server to allocate a new object to return, and this object needs memory. The server must calculate all the properties of the returned object, whether the client needs them or not, wasting *processing time* and memory. Consider using stealing when components need to provide large objects that their clients will retain for some time after receiving them.

3. Incremental Interfaces.

It is particularly difficult to pass a sequence or collection of data items across an interface. In systems with limited memory, or where memory is often fragmented, there may not be enough memory available to store the entire collection. In these cases, the interface needs to be made *incremental* — that is, information is transferred using more than one call from the client to a component, each call transferring only a small amount of information. Incremental interfaces can be used for both inward and outward data transfer. Clients can either make multiple calls directly to a component, or an **ITERATOR** can be used as an intermediate object. Consider using Iterator Interfaces when large objects need to be transferred across interfaces.

3.1. Client Makes Multiple Calls: The client makes several method calls to the component, each call *loaning* a single object for the duration of the call. When all the objects are passed, the client makes a further call to indicate to the component that it's got the entire collection, so it can get on with processing. For example, a client can insert a number of paragraphs into a word processor, calling `addParagraph` to ask the word processor to take each paragraph, and then `processAddedParagraphs` to process and format all the new paragraphs.

```
for (int i = 0; i < num_new_paras; i++) {
    wordProcessor.addParagraph(paras[i]);
};
wordProcessor.processAddedParagraphs();
```



The client making multiple calls is easy to understand, and so is often the approach chosen by novice programmers or used in non-OO languages. However, it forces the component either to find a way of processing the data incrementally (see **DATA CHAINING**), or to create its own collection of the objects passed in, requiring further allocated memory. Alternatively the client can *loan* the objects for the duration of the processing rather than for each call, but this forces the client to keep all the data allocated until the `DoAction` operation completes.

To return information from a component incrementally, the client again makes multiple calls, but the component signals the end of the using a return value or similar.

```
spookivity.findGhosts("transparent|dead");
while (spookivity.moreGhostsToProcess()) {
    ghostScreen.addDisplay(spookivity.getNextGhost());
};
```

3.2 Passing Data via an Iterator: Rather than make multiple calls, the client may *lend* an iterator to the component. The component then accesses further *loaned* objects via the iterator. For example, the client can pass an iterator to one of its internal collections:

```
ghostScreen.displayAllGhosts(vectorOfGhosts.iterator());
```

and the component can use this iterator to access the information from the client:

```
void displayAllGhosts(Iterator it) {
    while (it.hasNext()) {
        displayGhost((Ghost) it.next());
    }
}
```

Passing in an iterator reverses the control flow, so that the component is now invoking messages on the client.

Using an iterator is generally more flexible than making multiple calls to a special interface. The component doesn't have to store its own collection of objects, since it can access them through the iterator. It's important that the interface uses an abstract iterator or abstract collection class, however; a common interface design error is to use a specific collection class instead, which constrains the implementation of the client.

3.3. Returning Data with a Writeable Iterator. A *writable iterator* is an iterator that insert elements into a collection, rather than simply traverse a collection. A writeable iterator produced by the client can be used to implement outward flows from component to client, in just the same way that a normal iterator implements inward flows.

```
Vector retrievedGhosts = new Vector();
spookivity.findGhosts("transparent|dead");
spookivity.returnAllGhosts(retrievedGhosts.writeableIterator());
```

Note that at the time of writing, the Java library does not include writeable iterators.

3.4. Returning data by returning an iterator. Alternatively the client may *borrow* or *steal* an iterator object from the component, and access returned values through that:

```
Iterator it = spookivity.findGhostsIterator("transparent|dead");
while (it.hasNext()) {
    ghostScreen.displayGhost((Ghost) it.next());
}
```

Returning an iterator keeps the control flow from the client to the component, allowing the iterator to be manipulated by client code, or passed to other client components.

❖ ❖ ❖

Known Uses

Interfaces are everywhere. For good examples of interfaces suitable for limited memory systems, look at the API documentation for the EPOC or PalmOs operating systems [Symbian 1999, Palm 2000].

Operating system file IO calls have to pass large amounts of information between the system and user applications. Typically, they require buffer memory to be allocated by the client, and then read or write directly into their client side buffers. For example, the classic Unix [Ritchie and Thompson 1978] file system call:

```
read(int fid, char *buf, int nchars);
```

reads up to `nchars` characters from file `fid` into the buffer starting at `buf`. The buffer is simply a chunk of raw memory.

EPOC client-server interfaces always use lending, since the server is in a different memory space to its client, and can only return output by copying it into memory set aside for it within the client. This ensures that memory demand is typically small, and that the client 's memory requirements can be fixed at the start of the project.

Many standard interfaces use iterators. For example, the C++ iostreams library uses them almost exclusively for access to container classes [Stroustrup 1997], and Java'z `Zlib` compression library uses iterators (streams) for both input and output.

See Also

Interfaces have to support the overall memory strategy of the system, and therefore many other memory patterns may be reflected in the interfaces between components.

Interfaces can supply methods to set up simulating a memory failure in the component to allow **EXHAUSTION TESTING** of both client and component. Interfaces that return references to objects owned by the component may **SHARE** these objects, and may use **REFERENCE COUNTING** or **COPY ON WRITE**.

Interfaces, particularly in C++, can enforce constant parameters that refer to **READ ONLY MEMORY** and thus may not be changed. In other languages, such enforcement is part of the interface documentation. Where components use **RESOURCE FILES**, interfaces often specify strings or resources as resource IDs rather than structures. As well as reducing the amount of information passing across the interface, the memory costs of the resource can be charged to the component that actually instantiates and uses it.

If the component (or the programming environment) supports **MEMORY COMPACTION** using handles, then the interface may use handles rather than object references to specify objects in the component.

The patterns for *Arguments and Results* [Noble 2000] and *Type-Safe Session* [Pryce 2000] describe how objects can be introduced to help design interfaces between. Meyers' *Effective C++* [1998] and Sutter's *Exceptional C++* [2000] describe good C++ interface design. Tony Simons has described some options using borrowing, copying and stealing for designing C++ classes [Simons 1998].

Partial Failure

Also known as: Graceful degradation; Feast and Famine.

How can you deal with unpredictable demands for memory?

- No matter how much you reduce a program's *memory requirements,* you can still run out of memory.
- It is better to fail at a trivial task than to rashly abandon a critical task.
- It is more important to keep running that to run perfectly all the time…
- … And much more important to keep running than to crash.
- The amount of memory available to a system varies wildly over time.

No matter how much you do to reduce the *memory requirements* of your program, it can always run out of memory. You can silently discard data you do not have room to store, terminate processing with a rude error message, or continue as if you had received the memory you requested so that your program crashes in unpredictable ways, but you can't avoid the problem. Implicitly or explicitly, you have to deal with running out of memory. In a 'traditional' system, low memory conditions are sufficiently rare that it is not really worth spending programmer effort dealing with the situation of running out of memory. The default, letting the program crash, is usually acceptable. After all, there are lots of other reasons why programs may crash, and users will hardly notice one or two more! However in a memory-limited system, low memory situations happen sufficiently often that this approach would seriously affect the usability of the system, or even makes it unusable.

For example, the Word-O-Matic word processor provides voice output for each paragraph; adds flashing colours on the screen to highlight errors in spelling, grammar and political correctness; and provides a floating window that continuously suggests sentence endings and possible rephrasing. All this takes a great deal of memory, and frequently uses up all the available RAM memory in the system.

There is some good news, however. First, some system requirements are more important than others — so if you have to fail something, some things are better to fail at than others. Second, provided your system can keep running failing to meet one requirement does not have to mean that you will fail subsequent ones. Finally, you are unlikely to remain short of memory indefinitely. When a system is idle, its demands on memory will be less than when it is heavily loaded.

In the Strap-It-On PC, for example, it's more important that the system keeps running, and keeps its watch and alarm timers up to date, than that any fancy editing function actually works. Within Word-O-Matic, retaining the text users have laboriously entered with the two-finger keypad is more important even than displaying that text, and much more important than spelling or grammar checking and suggesting rephrasing.

Therefore:. *Ensure that running out of memory always leaves the system in a safe state.*

Ensure that for every memory allocation there is a strategy for dealing with failure before it propagates through your program.

When a program detects that its memory allocation has failed, its first priority must be to get back to a safe, stable state as soon as possible, and clean up any inconsistencies caused by the failure. As far as possible this should happen without losing any data. Depending on what was being allocated when memory ran out, it may be enough to back out of the action that required the extra memory. Alternatively you might reduce the functionality provided by one or more components; or even shut down the component where the error occurred.

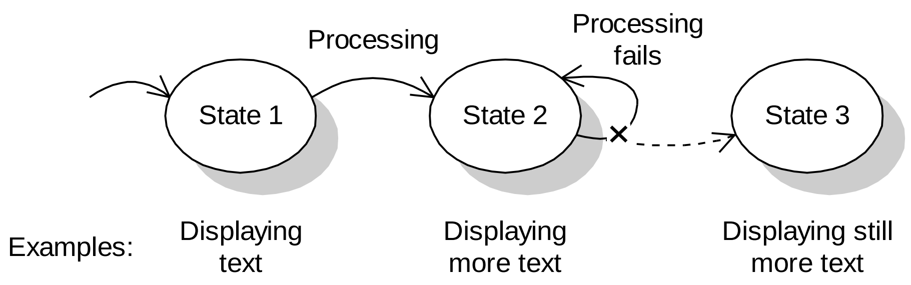


Figure 1: Failing the action that required the extra memory

What is vitally important, however, is to ensure that from the user's point of view, an action succeeds completely or fails completely, leaving the system in a stable state in either case. User interfaces, and component interfaces should make it clear when an important activity that affects the user has failed: if some data has been deleted, or an computation has not been performed.

Once a program has reached a stable state, it should continue as best it can. Ideally it should continue in a 'degraded mode', providing as much functionality as possible, but omitting less important memory-hungry features. You may be able to provide a series of increasingly degraded modes, to cater for increasing shortages of memory. Components can implement a degraded mode by hiding their memory exhaustion from their clients, perhaps accepting requests and queuing them for later processing, or otherwise offering a lower quality service. For example the Word-O-Matic's voice output module accepts but ignores commands from its clients in its 'out of memory' state, which makes programming its clients much simpler.

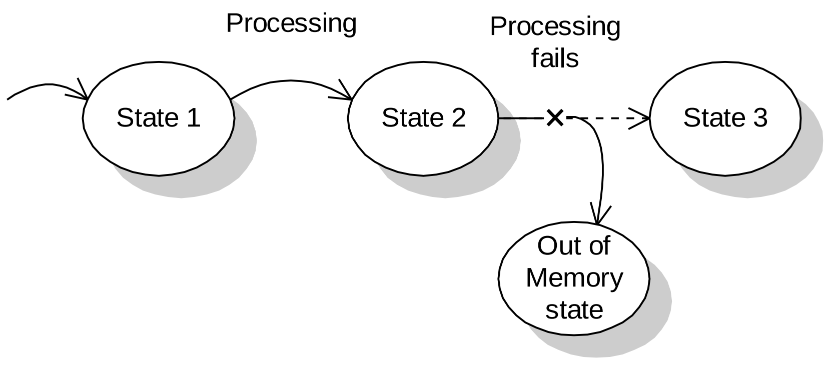


Figure 2: Failing to an Out of Memory state.

Finally, a system should return to full operation when more memory becomes available. Memory is often in short supply while a system copes with high external loads, once the load has passed its memory requirements will decrease. Users directly determine the load on multiprocessing environments like MS Windows and EPOC, so they can choose to free up memory by closing some system applications. A component running in a degraded mode should attempt to return to full operation periodically, to take advantage of any increase in available memory.

For example, when the Word-O-Matic fails to allocate the memory required for voice output of a document, its display screen continues to operate as normal. If the text checker fails, Word-O-Matic doesn't highlight any problems; if the floating window fails it doesn't appear, and the rest of the program carries on regardless. None of these fancier features are essential; and most users will be quite happy with just a text display and the means to enter more text.

Consequences

Supporting partial failure significantly improves a program's *usability*. With careful design, even the degraded modes can provide enough essential functionality that users can complete their work. By ensuring the program can continue to operate within a given amount of memory, partial failure decreases the program's minimum *memory requirements* and increases the *predictability* of those requirements.

Supporting partial failure increases the program's *design quality* – if you support partial failure for memory exhaustion, it's easy to support partial failure (and other forms of failure handling) for other things, like network faults and exhaustion of other resources. Systems that support partial failure properly can be almost totally *reliable*.

However: Partial Failure is hard work to program, requiring *programmer discipline* to apply consistently and considerable *programmer effort* to implement.

Language mechanisms that support partial failure – exceptions and similar – considerably increase the implementation complexity of the system, since programmers must cater for alternative control paths, and for releasing resources on failure.

Partial Failure tends to increase the *global* complexity of the systems, because *local* events – running out of memory – tend to have global consequences by affecting other modules in the system.

Supporting partial failure significantly increases the complexity of each module, increasing the *testing cost* because you must try to test all the failure modes.

❖ ❖ ❖

Implementation

Consider the following issues when implementing Partial Failure:

1 Detecting Memory Exhaustion.

How you detect exhaustion depends on the type of **MEMORY ALLOCATION** you are using. For example, if you are allocating memory from a heap, the operation that creates objects will have some mechanism for detecting allocation failure. If you are managing memory allocation yourself, such as using **FIXED ALLOCATION** or allocating objects dynamically from a pool, then you need to ensure the program checks to determine when the fixed structure or the memory pool is full. The **MEMORY ALLOCATION** chapter discusses this in more detail.

How you communicate memory exhaustion within the program depends on the facilities offered by your programming language. In many languages, including early implementations of C and C++, the only way to signal such an error was to return an error code (rather than the allocated memory). Unfortunately, checking the value returned by every allocation requires a very high level of programmer discipline. More modern languages support variants of exceptions, explicitly allowing functions to return abnormally. In most environments an out-of-memory exception terminates the application by default, so components that implement **PARTIAL FAILURE** need to handle these exceptions.

2 Getting to a Safe State.

Once you have detected that you have run out of memory, you have to determine how to reach a safe state, that is, how much of the system cannot continue because it absolutely required the additional memory being available. Typically you will fail only the function that made the request; in other situations the component may need a degraded mode, or, if a separate executable, may terminate completely.

To determine how much of the system cannot be made safe, you need to examine each component in turn, and consider their invariants, that is, what conditions must be maintained for them to operate successfully [Hoare 1981, Meyer 1997]. If a components' invariants are unaffected by running out of memory, then the component should be able to continue running as is. If the invariants are affected by the memory failure, you may be able to restore a consistent state by deleting or changing other information within the component. If you cannot restore a component to a safe state, you have to shut it down.

If you have to fail entire applications, you may be able to use **APPLICATION SWITCHING** to get to a safe state.

3 Releasing Resources.

A component that has failed to allocate the memory must tidy up after to ensure it has not left any side effects. Any resources it allocated but can no longer use (particularly memory) must be released, and its state (and that of any other affected components) must be restored to values that preserve its invariants.

In C++, exceptions 'unwind' the stack between a `throw` statement and a `catch` statement [Stroustrup 1997]. By default, all stack-based pointers between them are lost, and any resources they own are orphaned. C++ exceptions guarantee to invoke the destructor on any stack-based object, however, so any object on the stack can clean up in their destructors so that they will be tidied up correctly during an exception. The standard template class `auto_ptr` wraps a pointer and deletes it when the stack is unwound.

```
auto_ptr<NetworkInterfaceClass> p(new NetworkInterfaceClass);
p->doSomethingWhichCallsAnException();  // the instance is deleted
```

Although Java has garbage collection, you still have to free objects (by removing all references to them) and release external resources as the stack unwinds. Rather than using destructors, the Java '`try..finally`' construct will execute the '`finally`' block whenever the '`try`' block exits, either normally or abnormally. This example registers an instance of a **COMMAND** [Gamma et al 1995] subclass into a set, and then removes it from the set when an exception is thrown or the command's `execute` method returns normally.

```
Command cmd = new LongWindedCommand();
setOfActiveCommands.add(cmd);

try {
    cmd.execute();
}
finally {
    setOfActiveCommands.remove(cmd);
}
```

EPOC, as an operating system for limited memory systems, has Partial Failure as one of its most fundamental architectural principles [Tasker et al 2000]. Virtually every operation can to fail due to memory exhaustion; but such failure is limited as much as possible and never permitted to cause a memory leak. EPOC's C++ environment does not use C++ exceptions, rather an operating system TRAP construct. Basically, a call to the `leave` method unwinds the stack (using the C longjmp function), until it reach a TRAP harness call. . Client code adds and removes items explicitly from a 'cleanup stack', and then `leave` method automatically invokes a cleanup operation for any objects stored on the cleanup stack. The top-level EPOC system scheduler provides a `TRAP` harness for all normal user code. By default that puts up an error dialog box to warn the user the operation has failed, then continues processing.

Here's an example of safe object construction in EPOC. [Tasker et al 2000]. A **FACTORY METHOD** [Gamma et al 1995], `NewL`, allocates a zero-filled (i.e. safe) object using `new(Eleave)`, then calls a second function, `ConstructL`, to do any operations that may fail. By pushing the uninitialised object onto the cleanup stack, if `ConstructL` fails then it will be deleted automatically. Once the new object is fully constructed it can be removed from the cleanup stack.

```
SafeObject* SafeObject::NewL( CEikonEnv* aEnv )
{
  SafeObject* obj = new (ELeave) SafeObject( aEnv );
  CleanupStack::PushL( obj );
  obj->ConstructL();
  CleanupStack::Pop(); // obj is now OK, so remove it
  return obj;
}
```

The **CAPTAIN OATES** pattern includes another example of the EPOC cleanup stack.

4. Degraded Modes.

Once you've cleaned up the mess after your memory allocation has failed, your program should carry on running in a stable state, even though its performance will be degraded. For example:

- Loading a font may fail; in this case you can use a standard system font.
- Displaying images may fail; you can leave them blank or display a message.
- Cached values may be unavailable; you can get the originals at some time cost.
- A detailed calculation may fail; you can use an approximation.
- Undo information may not be saved (usually after warning the user).

Wherever possible components should conceal their partial failure from their clients. Such encapsulation makes the components easier to design and *localises* the effect of the failure to the components that detect it. Component interfaces should not force clients to know about these failure modes, although they can provide additional methods to allow interested clients to learn about such failure.

You can often use **MULTIPLE REPRESENTATIONS** to help implement partial failure.

5. Rainy Day Fund.

Just as you have to spend money to make money, handling memory exhaustion can itself *require* memory. C++ and Java signal memory exhaustion by throwing an exception, which requires memory to store the exception object; displaying a dialog box to warn the use about memory problems requires memory to store the dialog box object. To avoid this problem, set aside some memory for a rainy day. The C++ runtime system, for example, is required to preallocate enough memory to store the `bad_alloc` exception thrown when it runs out of memory [Stroustrup 1997]. Windows CE similarly sets aside enough memory to display an out-of-memory dialog box [Boling 1998]. The Prograph visual programming language takes a more sophisticated approach — it supplies a rainy day fund class that manages a memory reserve that is automatically released immediately after the main memory is exhausted [MacNeil and Proudfoot 1985].

Example

The following Java code illustrates a simple technique for handling errors with partial failure. The method `StrapFont.font` attempts to find a font and ensure it is loaded into main memory. From the client's point of view, it must always succeed.

We implement a safe state by ensuring that there is always a font available to return. Here, the class creates a default font when it first initialises. If that failed, it would be a failure of process initialisation – implemented by `new` throwing an uncaught `OutOfMemoryError` – preventing the user entering any data in the first place.

```
class StrapFont {

    static Font myDefaultFont =  new Font("Dialog",Font.PLAIN,12);

    public static Font defaultFont() {
        return myDefaultFont;
    }
```

The `StrapFont.font` method tries to create a new Font object based on the description `priavteGetFont` method, which can run out of memory and throw and `OutOfMemoryError`. If a new font object cannot be created then we return the default font. This mechanism also allows safe handling of a different problem, such as when the font does not exist:

```
    public static Font font(String name, int style, int size) {
        Font f;
        try {
            f = privateGetFont(name, style, size);
        }
        catch (BadFontException e) {
            return defaultFont();
        }
        catch (OutOfMemoryError e) {
            return defaultFont();
        }
        return f;
    }

}
```

The client must reload the font using `StrapFont.font` every time it redraws the screen, rather than caching the returned value; this ensures that when memory becomes available the correct font will be loaded.

❖ ❖ ❖

Known Uses

Partial Failure is an important architectural principle. If a system is to support Partial Failure, it must do so consistently. A recent project evaluated a third-party database library for porting to EPOC as an operating system service. Everything looked fine: the code was elegant; the port would be trivial. Unfortunately the library, designed for a memory-rich system, provided no support for partial failure; all memory allocations were assumed either to succeed or to terminate the process. In a service for simultaneous use by many EPOC applications that strategy was unacceptable; memory exhaustion is common in EPOC systems, and the designers couldn't allow a situation where it would cause many applications to fail simultaneously. The library was unsuitable because it did not support Partial Failure.

Degraded Modes are common in GUI applications. If Netscape fails to load a font due to insufficient memory, it continues with standard fonts. Microsoft PowerPoint will use standard fonts and omit images. PhotoShop warns the user and then stops saving undo information.

At a lower level, if the Microsoft Foundation Class framework detects an exception while painting a window, its default behaviour is to mark the window as fully painted. This allows the application to continue although the window display may be incorrect; the window will be repainted when it is subsequently changed by the application.

EPOC's Word Processor makes its largest use of memory when formatting part of a page for display. If this fails, it enters an out-of-memory mode where it displays as much of the text as has been formatted successfully. Whenever a user event occurs, (scrolling, or a redisplay command) Word attempts to reformat the page, leaves its degraded mode if it is successful. EPOC's architecture also has an interesting policy about safe states. The EPOC application framework is event-driven; every application runs by receiving repeated function calls from a central scheduler. Every application is in a safe state when it is not currently executing from the scheduler, so any EPOC application can fail independently of any other [Tasker et al 2000].

See Also

APPLICATION SWITCHING can fail an entire application and begin running another application, rather than terminating an entire system of multiple applications. **MULTIPLE REPRESENTATIONS** can also support partial failure, by replacing standard representations with more memory efficient designs.

An alternative to failing the component that needed the memory is to use the **CAPTAIN OATES** pattern and fail a different and less important component. The **MEMORY ALLOCATION** chapter describes a number of strategies for dealing with allocation failures, such as deferring requests, discarding information, and signalling errors.

Ward Cunningham's **CHECKS** pattern language discusses several ways of communicating partial failure to the user. [Cunningham 1995]. *Professional Symbian Programming* [Tasker et al 2000], *More Effective C++* [Meyers 1996] and *Exceptional C++* [Sutter 2000] describe in detail programming techniques and idioms for implementing Partial Failure with C++ exceptions.

Captain Oates

Also known as: Cache Release.

How can you fulfil the most important demands for memory?

- Many systems have components that run in the background.
- Many applications cache data to improve performance
- User's care more about what they are working on than background activities the system is doing for its own sake.

To the operating system all memory requirements appear equal. To the user, however, some requirements are more equal than others [Orwell 1945].

For example, when someone is using the Strap-It-On PC's word processor to edit a document, they don't care what the fractal screen background looks like. You can increase a system's *usability* by spending scarce resources doing what users actually wants.

Many systems include background components, such as screen savers, chat programs, cryptoanalysis engines [Hayes 1998], or Fourier analyses to search for extraterrestrial intelligence [Sullivan et al 1997]. Systems also use memory to make users' activities quicker or more enjoyable, by downloading music, caching web pages, or indexing file systems. Though important in the longer term, these activities do not help the user while they are happening, and take scarce resources from the urgent, vital, demands of the user.

Therefore: *Sacrifice memory used by less vital components rather than fail more important tasks.*

Warn every component in the system when memory is running out, but while there is still some space left. When a component receives this warning it should release its inessential memory, or in more extreme situations, terminate activities.

If there is no support for signalling memory conditions, processes can keep track of the free memory situation by regular polling, and free inessential resources (or close down) when memory becomes short.

For example when the Word-O-Matic is about to run out of memory the IP networking stack empties its cache of IP address maps and the web browser empties its page cache. Background service processes like the 'Fizzy^{TM}' fractal generator automatically closes down. Consequently, the word processor's *memory requirements* can be met. Figure XXX illustrates a system-wide implementation of the Captain Oates pattern:

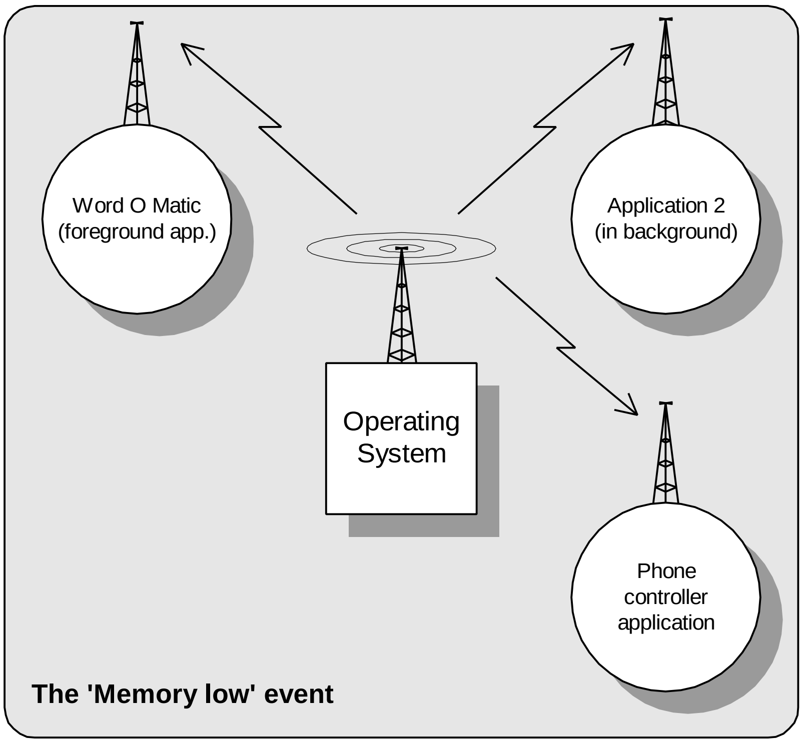


Figure 3: The Memory Low Event

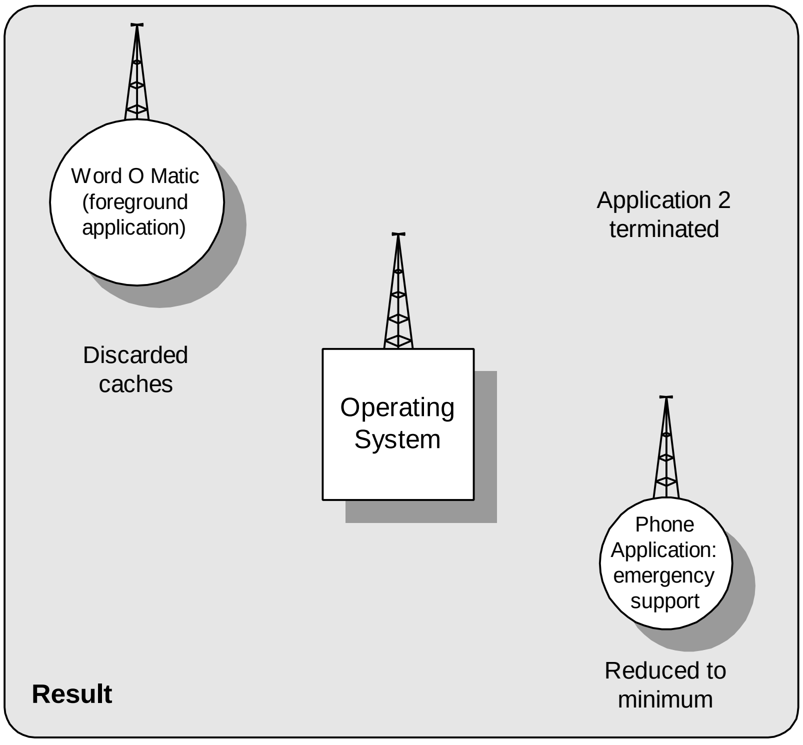


Figure 4: Result of the Memory Low Event

The name of this pattern celebrates a famous Victorian explorer, Captain Lawrence ‘Titus’ Oates. Oates was part of the British team led by Robert Falcon Scott, who reached the South

Pole only to discover that Roald Amundsen's Norwegian team had got there first. Scott's team ran short of supplies on the way back, and a depressed and frostbitten Oates sacrificed himself to give the rest of his team a chance of survival, walking out into the blizzard leaving a famous diary entry: "I may be some time". Oates' sacrifice was not enough to save the rest of the team, whose remains were found in their frozen camp the next year. Thirty-five kilograms of rock samples, carried laboriously back from the Pole, were among their remains [Limb and Cordingley 1982; Scott 1913].

Consequences

By allocating memory where it is most needed this pattern increases the systems *usability*, and reduces its *memory requirements.* Programs releasing their temporary memory also increase the *predictability* of the system's memory use.

However: Captain Oates requires *programmer discipline* to consider voluntarily releasing resources. Captain Oates doesn't usually benefit the application that implements it directly, so the motivation for a development team to implement it isn't high –there needs to be strong cultural or architectural forces to make them do so. The pattern also requires *programmer effort* to implement and test.

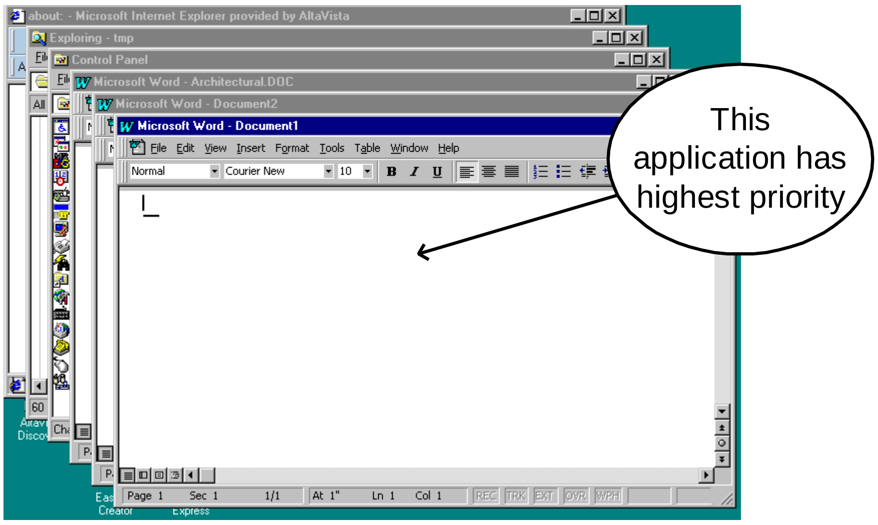
Captain Oates introduces coupling between otherwise unrelated components, which decreases the *predictability* of the system. Releasing resources can reduce the program's *time performance*. Programs need to be *tested* to see that they do release resources, and that they continue to perform successfully afterwards. Because many programs must handle the memory low signal, Captain Oates is easier with *operating system support*. This is another *global mechanism* that introduces *local complexity* to handle the signal.

❖ ❖ ❖

Implementation

The main point of the Captain Oates pattern is that it releases memory from low priority activities so that high priority activities can proceed. It is inappropriate for a component to release memory if it is supporting high-priority activities. Yet mechanisms that detect low memory conditions are indiscriminate and notify all components equally. So how can you work out what components to sacrifice?

A user interface application can usually determine whether is the current application, i.e. whether it has the input focus so users can interact with it. If so, it should not sacrifice itself when it receives low memory warnings.


This application has highest priority
Microsoft Word - Document1

A background process, though, cannot usually ask the system how important they are. In MS Windows, for example, high priority threads block waiting for some events — the Task manager has a high priority when waiting for Ctrl+Alt+Del key strokes. When the Task Manager detects an event, however, it changes its priority down to a normal. So, calling `GetThreadPriority` cannot give a true indication of how important the task is and whether it's being used.

Most processes, though, can determine how important they are from other information. A component managing network connections, for example, could check whether it had any active connections. Other background processes may not even have that information; a web page cache, for example, may have no direct information about the applications that it supports. Such processes, however must not be directly interacting with the user (otherwise they would have more information about users' activities) and so can usually quite safely release inessential resources when required.

1. Detecting Low Memory Conditions.

Many operating systems provide events that warn applications when memory is low. MS Windows and MS Windows CE send `WM_COMPACTING` and `WM_HIBERNATE` messages to all windows (though not, therefore, to background processes) to warn them that the system memory is getting low [Boling 1998,Microsoft 1997]. Rather than send events, some operating systems or language runtimes call back to system components when memory is low — one example, C++'s `new_handler,` is discussed in the **PARTIAL FAILURE** pattern.

As an alternative, if the system provides functions to show how much memory is in use, then each component can poll to see if memory is low, and release memory when it is. Polling can be unsatisfactory in battery-powered machines, however, since the processor activity uses battery power.

2. Handling low memory events.

When a low memory even occurs, it's useful if each component can determine how short of memory the system is. In the Java JDK 1.2 environment, the runtime object's `getMemoryAdvice()` call answers one of four modes: 'green' meaning there's no shortage, 'yellow' then 'orange' meaning memory is getting low, and 'red' meaning memory is critically low. MS Windows' event, WM_COMPACTING, sends an indication of the proportion of time spent paging memory: 1/8 is equivalent to 'yellow', and is when the message is first sent; anything over 1/4 is critically low [Microsoft 1997].

3. Good Citizenship.

Perhaps the simplest, and often the easiest, approach is for each process to voluntarily give up inessential resources they are not really using. By observing a simple timer, you can release latent resources after a specific time, regardless of the memory status of the rest of the system. For example, the EPOC Web browser loads dynamic DLLs to handle specific types of Web data. If a particular type of data occurs once, it may recur almost immediately, so the Browser DLL loader caches each DLL. If the DLL isn't reused within a few seconds, however, the loader releases it.

Example

This C++ example implements a piece of operating system infrastructure to support a simple Captain Oates mechanism for the EPOC operating system. The Captain Oates application runs in the background and closes applications not currently in use when memory becomes low. Since closing an EPOC application automatically saves its state (a requirement of the PC-synchronisation mechanisms), this does not lose any data. Transient editing state, such as the cursor position in a document or the current item displayed in a file browser, is not maintained, however.

The functionality is in class `COatesTerminator`, which is as follows (omitting function declarations):

```
class COatesTerminator : public CBase {
private:
    RNotifier  iPopupDialogNotifier;    // Provides user screen output
    CPeriodic* iTimer;                  // Timer mechanism
    CEikonEnv* iApplicationEnvironment; // User I/O Handler for this app.

    enum {
        EPollPeriodInSeconds = 10,      // How often to check memory
        EDangerPercentage = 5 };        // Close applications when less free
                                        // memory than this.
};
```

There are various construction and initialisation functions (not included here) to set up the periodic timer and dialog notifier.

The core of the application, however, is the `TimerTickL` function that polls the current memory status and closes applications when memory is low. The free memory reading can be deceptively low if other applications have allocated more memory then they are using. If free memory appears to be low on a first reading, we compress all the memory heaps; this claws back any free pages of memory at the end of each heap. Then a second reading will measure all free memory accurately. If the second reading is also low, we call `CloseAnApplicationL` to close an application.

```
void COatesTerminator::TimerTickL() {
    if ( GetMemoryPercentFree() <= EDangerPercentage ||
         (User::CompressAllHeaps(),
          GetMemoryPercentFree() <= EDangerPercentage ))  {
        CloseAnApplicationL();
    }
}
```

`CloseAnApplicationL` must first select a suitable application to terminate — we do not want to close the current foreground application, the system shell, or this process. Of the other candidates, we'll just close the one lowest in the Z order. Applications are identified to the system as 'window groups' (WG). To find the right window, we first get the identifiers of the window groups we don't want to close (`focusWg`, `defaultWg`, `thisWg`), get the

`WindowGroupList`, then work backwards through the list, and close the first suitable application we find.

Note also the use of the `CleanupStack`, as described in **PARTIAL FAILURE**. We push the array holding the `WindowGroupList` onto the stack when it is allocated, and then remove and destroy it as the function finishes. If the call to get the window group suffers an error, we immediately `leave` the `CloseAnApplicationL` function, automatically destroying the array as it is on the cleanup stack.

```
void COatesTerminator::CloseAnApplicationL() {
    RWsSession& windowServerSession = iApplicationEnvironment->WsSession();

    TInt foregroundApplicationWG =  windowServerSession.GetFocusWindowGroup();
    TInt systemShellApplicationWG = windowServerSession.GetDefaultOwningWindow();
    TInt thisApplicationWG =        iApplicationEnvironment->RootWin().Identifier();

    TInt nApplications=windowServerSession.NumWindowGroups(0);
    CArrayFixFlat<TInt>* applicationList=
                                    new (ELeave) CArrayFixFlat<TInt>(nApplications);
    CleanupStack::PushL( applicationList );
    User::LeaveIfError( windowServerSession.WindowGroupList(0,applicationList) );
    TInt applicationWG=0;
    TInt i= applicationList->Count();
    for (i--; i>=0; i--)  {
        applicationWG = applicationList->At( i );
        if (applicationWG != thisApplicationWG &&
            applicationWG != systemShellApplicationWG &&
            applicationWG != foregroundApplicationWG)
            break;
    }
```

If we find a suitable candidate, we use a standard mechanism to terminate it cleanly. Note that `_LIT` defines a string literal that can be stored in ROM – see the **READ ONLY MEMORY** pattern.

```
    if (i >= 0) {
        TApaTask task(windowServerSession);
        task.SetWgId(applicationWG);
        task.EndTask();
        _LIT( KMessage, "Application terminated" );
        iPopupDialogNotifier.InfoPrint( KMessage );
    }
    CleanupStack::PopAndDestroy(); // applicationList
}
```

This implementation has the disadvantage that it requires polling, consuming unnecessary CPU time and wasting battery power. A better implementation could poll only after writes to the RAM-based file system (straightforward), after user input (difficult), or could vary the polling frequency according to the available memory.

❖ ❖ ❖

Known Uses

The MS Windows application 'ObjectPLUS', a hypercard application by ObjectPLUS of Boston, responds to the WM_COMPACTING message. As the memory shortage becomes increasingly critical, it:

- Stops playing sounds
- Compresses images
- Removes cached bitmaps taken from a database

Though this behaviour benefits other applications in the system, it also benefits the HyperCard application itself by releasing memory for other more important activities. By implementing the behaviour in the Windows event handler, the designers have kept that behaviour architecturally separate from other processing in the application.

The Apple Macintosh memory manager (discussed in **COMPACTION**) supports "purgeable memory blocks" — that the memory manager reclaims when memory is low [Apple 1985]. They are used for **RESOURCE FILES**, and file system caches, and dynamically allocated program memory.

MS Windows CE Shell takes a two phase approach to managing memory [Microsoft 1998, Boling 1998]. When memory becomes low, it sends a `WM_HIBERNATE` message to every application. A CE application should respond to this message by releasing as many system resources as possible. When memory becomes even lower, it sends the message `WM_CLOSE` to the lowest priority applications, asking those applications to close — like EPOC, Windows CE requires applications to save their state on `WM_CLOSE` without prompting the user. Alternatively, if more resources become available, applications can receive the WM`_ACTIVATE` message, requesting them to rebuild the internal state they discarded for `WM_HIBERNATE`.

A number of distribued internet projects take advantage of Captain Oates by running as screensavers. When a machine is in use, the screensavers do not run, but after a machine is idle for a few minutes the screensaver uses the idle processor to search for messages from aliens [Hayes 1998] or crack encrypted messages [Sullivan et al 1997].

See Also

Where **CAPTAIN OATES** describes what a program should do when another process in the system runs out of memory, **PARTIAL FAILURE** describes what a process should do when it runs out of memory itself. Many of the techniques for **PARTIAL FAILURE** (such as **MULTIPLE REPRESENTATIONS** and **PROGRAM CHAINING**) are also appropriate for **CAPTAIN OATES**.

FIXED ALLOCATION describes a simple way to implement a form of **CAPTAIN OATES**, where each activity is merely a data structure – simply make new activities overwrite the old ones.

Scott and his team are popular heroes of British and New Zealand culture. See '*Captain Oates: Soldier and Explorer*' [Limb and Cordingley 1982], and '*Scott's Last Expedition: The Personal Journals of Captain R. F. Scott, R.N., C.V.O., on his Journey to the South Pole.*' [Scott 1913].

Read-Only Memory

Also known as: Use the ROM

What can you do with read-only code and data?

- Many systems provide read-only memory as well as writable memory
- Read-only memory is cheaper than writable memory
- Programs do not usually modify executable code.
- Programs do not modify resource files, lookup tables, and other pre-initialised data.

Programs often have lots of *read-only* code and data. For example, the Word-O-Matic word-processor has a large amount of executable code, and large master dictionary files for its spelling checker, which it never changes. Storing this static information in main memory will take memory from data that does need to change, increasing the *memory requirements* of the program as a whole.

Many hardware devices — particularly small ones —support read-only memory as well as writable main memory. The read-only memory may be primary storage, directly accessible from the processor, or indirectly accessible secondary storage. A wide range of technologies can provide read-only memory, from semiconductor ROMs and PROMS of various kinds, through flash ROMs, to read-only compact discs and even paper tape. Most forms of read-only memory are better in many ways than corresponding writable memory — simpler to build, less expensive to purchase, more reliable, more economical of power, dissipating less heat, and more resistant to stray cosmic radiation.

Therefore: S*tore read-only code and data in read-only memory.*

Divide your system code and data into those portions that can change and those that never change. Store the immutable portions in read-only memory and arrange to re-associate them with the changeable portions at run-time.

Word-O-Matic's program's code, for example, is contained in ROM memory in the Strap-It-On portable PC. Word-O-Matic's master dictionary and other resource files are stored in in read-only secondary storage (flash ROM); only user documents and configuration files are stored in writeable memory.

Consequences

This pattern trades off writable main storage for *read-only* storage, reducing the *memory requirements* for main storage and making the it easier to test.. Read-only storage is cheaper than writable storage, in terms of financial cost, power consumption and reliability. If the system can execute programs directly from read-only memory, then using read-only memory can decrease the system's *start-up time.*

Although you may need to copy code and data from *read-only secondary storage* to main memory, you can delete read-only information from main memory without having to save it back to secondary storage. Because they cannot be modified, read-only code and data can be shared easily between programs or components, further reducing the *memory requirements* of the system as a whole.

However: *programmer effort* is needed to divide up the program into read-only and writable portions, and then *programmer discipline* to stick to the division. The disctinction between read-only and writeable inforamtion is fundamentally a *global* concern, although it must be made *locally* for every component in the program.

Code or data in read-only memory is more difficult to *maintain* than information in writable secondary storage. Often, the only way to replace code or data stored in read-only memory is to physically replace the hardware component storing the information. Updating flash memory, which can be erased and rewritten, usually requires a complicated procedure particuarly if the operating system is stored in the memory being updated.

❖ ❖ ❖

Implementation

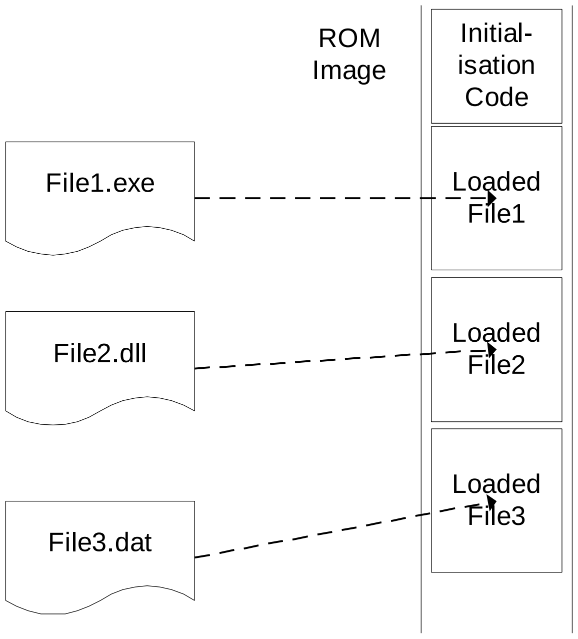
Creating a 'ROM Image' (a copy of the final code and data to be stored into read-only memory)is invariably a magical process, requiring major incantations and bizarre software ingredients that are specific to your environment. Across most environments, however, there are common issues to consider when using read-only memory.

1. Storing Executable Code.

If you can run programs directly from read only memory, then you can use it to store executable code. This generally poses two problems: how should the code be represented in read-only memory, and how can it get access to any data it needs?

Most environments store programs as object files — such as executables and dynamic linked libraries — that do not refer to any absolute addresses in memory, instead containing symbolic references to other files or libraries. Before object files can be executed, the operating system's run-time loader must bind the symbolic references to create completely executable machine code.

To store this executable code in read-only memory you need an extra tool, the 'ROM builder', that does the job of the run-time loader, reading in object files and producing a ROM Image. A ROM Builder assigns each object file a base address in memory and copies it into the corresponding position in the ROM image, binding symbolic references and assigning writable memory for heap memory, static memory, and static data. For example, the EPOC system includes a Java ROM builder takes the 'jar' or 'class' files, and loads them into a ROM image, mimicking the actions of the Java run-time class loader.



If the system starts up by executing code in read-only memory, then the ROM image will also need to contain initialisation code to allocate main memory data structures and to bootstrap the whole system. The ROM Builder can know about this bootstrap code and install it in the correct place in the image.

2. Including Data within Code.

Most programs and programming languages include constant data as well as executable code — if the code is being stored in read-only memory, then this data should accompany it. To do this you need to persuade the compiler or assembler that the data is truly unchangeable.

The C++ standard [Ellis and Stroustrup 1990], for example, defines that instances of objects can be placed in the code segment — and thus in read-only memory — if:

- The instance is defined to be `const`, and
- It has no constructor or destructor.

Thus

```
const char myString[] = "Hello";   // In ROM
char* myString = "Hello";          // Not in ROM according to the Standard.
const String myString( "Hello" );  // Not in ROM, since it has a constructor
```

In particular you can create C++ data tables that can be compiled into ROM:

```
const int myTable[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  // In ROM
```

Note that non-`const` C++ strings are generally not placed in the code segment, since they can be modified, but some compilers support flags or `#pragma` declarations to change this behaviour.

The EPOC system uses a combination of C++ macros and template classes to create instances of strings in read-only memory, containing both a length and the text, as follows:

```
template <TInt S> class TLitC {
   // Various operators...
public:
   int iTypeLength;  // This is the structure of a standard EPOC string
   char iBuf[S];
   };

#define _LIT(name,s) const static TLitC<sizeof(s)> name={sizeof(s)-1,s}
```

This allows EPOC code to define strings in ROM using the `_LIT` macro:

```
_LIT( MyString, "Hello World" );
User::InfoPrint( MyString ); // Displays a message on the screen.
```

The linker filters out duplicate constant definitions, so you can even put `_LIT` definitions in header files.

2.1. Read-only objects in C++. C++ compilers enforce `const` as far as the bitwise state of the object is concerned: `const` member functions may not change any data member, nor may a client delete an object through a `const` pointer [Stroupstrup 1997]. A well-designed class will provide logical `const`-ness by ensuring that any public function is `const` if it doesn't change the externally visible state of the object. For example, the simple String class below provides both a 'logically `const`' access operator, and a non-`const` one. A client given using a `const String&` variable can use only the former.

```
class String {
public:
  // Constructors etc. not shown...
  char operator[]( int i ) const { return rep[i]; }
  char& operator[]( int i ) { return rep[i]; }
private:
  char* rep;
};
```

C++ supports passing parameters by value, which creates a copy of the shared object on the stack. If the object is large and if the function does not modify it, it's common C++ style to **SHARE** the representation by passing the object as a `const` reference. Thus:

```
void function( const String& p );
```

is usually preferable to

```
void function( String p );
```

because it will use less stack space.

2.2. Read-only objects in Java. Java lacks `const`, and so is more restrictive on what data can be stored with the code in read-only memory — only strings and single primitive values are stored in Java constant tables. For example, the following code

```
final int myTable[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  // Don't do this!
```

compiles to a very large function that constructs `myTable`, assigning values to an array in main memory element by element. Storing data for Java programs in read-only memory is thus quite complex. You can encode the data as two-byte integers and store in a string; use C++ to manage the data and access it via the Java Native Interface; or keep the data in a resource file and use file access calls [Lindholm and Yellin 1999].

3. Static Data Structures. Some programs require relatively large constant data structures, for example:

- Encryption algorithms, such as the US Data Encryption Standard (DES).
- Mathematical algorithms, such as log, sine and cosine functions.

- State transition tables, such as those generated by tools to support the Shlaer-Mellor object-oriented methodology [1991].

These tables can be quite large, so its usually not a good idea to store them in main memory, but since they are constant, they can be moved to read-only memory. Managing the development of these data structures can be quite a large task, however.

If the table data changes often during the development process, the best approach is to use a tool to generate the table as a separate file that is incorporated by the ROM Image builder. If the data changes very rarely, then it's usually easiest to copy the table manually into the code, and modify it or the surrounding code to ensure the compiler will place it into read-only memory.

4. Read-Only File Systems.

Some environments can treat read-only memory as if it were a file system. This has the advantage that file system structures can organise the read-only data, and that applications can read it through the normal file operations, although they cannot modify it. For example, EPOC supports a logical file system (Z:), normally invisible to users, which is stored in read-only memory and constructed by the EPOC ROM Builder. All the Resource Files for ROM-based applications are stored in this file system.

File system access is usually slower than direct memory access. If read-only memory can be mapped into applications' address spaces, the data in a ROM filing system can be made available directly, as an optimisation. For example, the EPOC Bitmap Server uses the function `User::IsFileInROM` to access bitmap data directly from ROM.

5. Version Control

Different versions of ROM images will place the same code or data at different addresses. You need to provide some kind of index so that other software in the system can operate with different ROM versions. For example, ROM images often begin with a table of pointers to the beginning of every routine and data structure: external software can find the correct address to call by indirection through this table [Smith 1985].

The **HOOKS** pattern describes how you can store the table in writable memory, so that routines can be extended or replaced with versions stored in writable memory.

Example

The following example uses a read-only lookup table to calculate the mathematical sine function for a number expressed in radians. Because the example is in Java, we must encode the table as a string (using hexadecimal values) because numeric arrays cannot be stored in Java constant tables. The following code runs on our development machine and calculates 256 values of the sine function as sixteen bit integers.

```
final int nPoints = 256;
for (int i = 0; i<nPoints; i++) {
    double radians = i * Math.PI / nPoints;
    int tableValue = (int)(Math.sin(radians) * 65535);
    System.out.print("\\u"+Integer.toHexString(tableValue));
}
```

This code doesn't produce quite correct Java: a few of the escape codes at the start and end lack of the table leading zeros, but it's easier to correct this by hand than to spend more time on a program that's only ever run once.

The `sin` function itself does linear interpolation between the two points found in the table. For brevity, we've not shown the whole table:

```
        static final String sinValues = "\u0000\u0324\u0648. . .\u0000";

    public static float sin(float radians) {
        float point = (radians / (float)Math.PI) * sinValues.length();
        int lowVal = (int) point;
        int hiVal = lowVal + 1;
        float lowValSin = (float)sinValues.charAt(lowVal) / 65535;
        float hiValSin = (float)sinValues.charAt(hiVal) / 65535;
        float result = ((float)hiVal - point) * lowValSin
            + (point - (float)lowVal) * hiValSin;
        return result;
    }
```

On a fast machine with a maths co-processor this `sin` function runs orders of magnitude more slowly than the native `Math.sin()` function! Nevertheless this program provides an accuracy of better than 1 in 20,000, and illustrates the lookup table technique. Lookup tables are widely used in environments that don't support mathematics libraries and in situations where you prefer to use integer rather than floating point arithmetic, such as graphics compression and decompression on low-power processors.

❖ ❖ ❖

Known Uses

Most embedded systems — from digital watches and washing machine controllers to mobile telephones and weapons systems — keep their code and some of their data in read-only memory, such as PROMs or EPROMs. Only run-time data is stored in writable main memory. Palmtops and Smartphones usually keep their operating system code in ROM, along with applications supplied with the phone. In contrast, third party applications live in secondary storage (battery backed-up RAM) and must be loaded into main memory to execute. Similarly, many 1980's home computers, such as the BBC Micro, had complex ROM architectures [Smith 1985].

Even systems that load almost all their code from secondary storage still need some 'bootstrap' initialisation code in ROM to load the first set of instructions from disk when the system starts up. PCs extend this bootstrap to be ROM-based Basic Input Output System (BIOS), which provides generic access to hardware, making it easy to support many different kinds of hardware with one (DOS or Windows) operating system [Chappel 1994].

See Also

Data in read-only storage can be changed using **COPY-ON-WRITE** and **HOOKS. COPY-ON-WRITE** and **HOOKS** also allow some kinds of infrequently changing (but not constant) data to be moved to read only storage.

Anything in read-only storage is suitable for **SHARING** between various programs and different components or for moving to **SECONDARY STORAGE.**

PAGING systems often distinguish between read-only pages and writable pages, and ignore or prevent attempts to write to read-only pages. Several processes can safely share a read-only page, and the paging system can discard it without the cost of writing it back to disk.

Hooks

Also known as: Vector table, Jump table, Patch table, Interrupt table.

How can you change information read-only storage?

- You are using read-only memory
- It is difficult or impossible to change read-only memory once created.
- Code or data in read-only memory needs to be maintained and upgraded.
- You need to make additions and relatively small changes to the information stored in read-only memory.

The main disadvantage of read-only storage is that it is *read-only*. The contents of read-only memory are set at manufacturing time, or possibly upgrade, time; whereupon they are fixed for eternity. Unfortunately, there are always bugs that need to be fixed, or functionality to be upgraded. For example, the released version of the Word-O-Matic code in the Strap-It-On's ROM is rather buggy, and fixes for these bugs need to be included into existing systems. In addition, Strap-It-On's marketing department has decreed that it needs an additional predictive input feature, to automatically complete users' input and so reduce the number of input keystrokes [Darragh, Witten, and James 1990].

If the information is stored in partly writable storage, such as EPROMs, your could issue a completely new ROM image and somehow persuade all the customers to invest the time and risk of upgrading it. Upgrading ROMs is painful for your customers, and often commercially impractical if you don't have control over the whole system. Moreover, a released ROM is unlikely to be so badly flawed as to demand a complete re-release. Often the amount of information that needs to be changed is small, even for significant changes to the system as a whole.

You could ignore the existing read-only memory, and store a new copy of the information in writable main memory. Even if there is enough writable memory in the system to hold a full copy of the contents of the read-only memory, you generally cannot afford to dedicate large amounts of main memory to storing copies of the ROM.

Therefore: *Access read-only information through hooks in writable storage and change the hooks to give the illusion of changing the information..*

The key to making read-only storage extensible is to link your system together through writeable memory, rather than read-only memory. When designing a system that uses read-only storage, do not access that storage directly. Allocate a 'hook' in writable memory for each entry point (to a function, component, data structure, or resource) that is stored in read-only memory, and initialise each hook to refer to its corresponding entry point. Ensure that every access to the entry point is via the writable hook — all accesses, whether from read-only memory or writable memory should use the hook.

To update the read-only memory you copy just that part of the memory you need to modify, and then make the required changes to the copy. Then, you can store the modified copy in writable store, and set the hooks to point to the modified portion. The modified portion can call other parts of the program, if necessary again by indirection through the hooks.

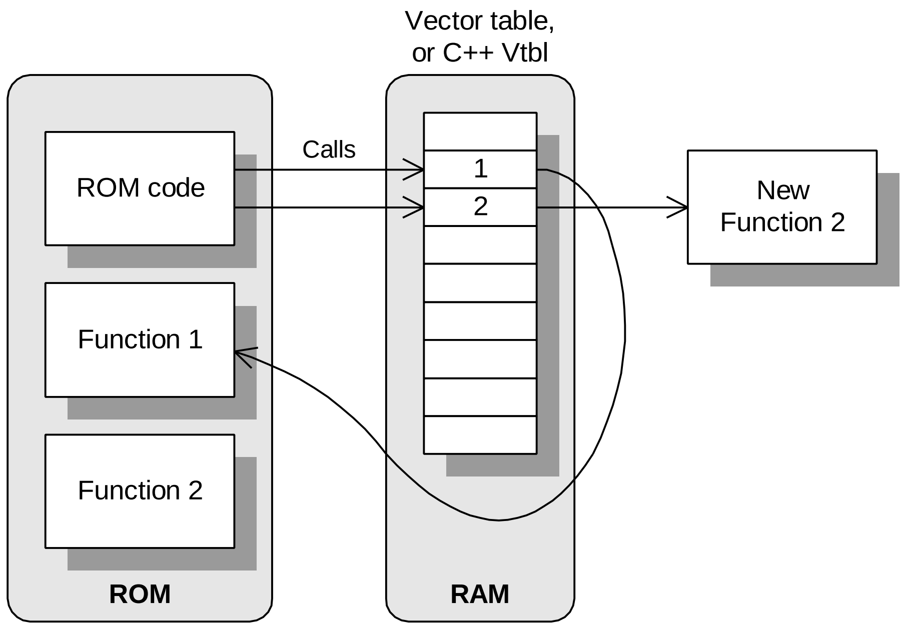


Figure 5: Code Hooks

For example, the Strap-It-On was carefully designed so that every major function is called indirectly through a table of hooks that are stored in RAM and initialised when the system is booted. Bug fixes, extensions, and third party code can be loaded into the system's main memory and the hooks changed to point to them. When an application uses a system function, the hooks ensures it finds the correct piece of code — either the original code in ROM, or the new code in RAM.

Consequences

Hooks let you extend read-only storage, and by making read-only storage easier to use, can reduce the program's writable *memory requirements*.

Providing good hooks increases the *quality* of the *program's design*, making it easier to *maintain* and extend in future. A ROM-based operating system that provides good hooks can enormously reduce the *programmer effort* required to implement any specific functionality.

However: Hooks require *programmer discipline* to design into programs and then to ensure they are used. They also increases the *testing* cost of the program, because the hooks have to be tested to see if they are called at the right times.

Indirect access via hooks is slower than direct access, reducing *time performance*; and the hook vectors take up valuable writable storage, slightly increasing *memory requirements*. Hook vectors are great places to attack system integrity, as any virus writer will tell you, so using hooks can make the system less reliable.

❖ ❖ ❖

Implementation

Consider the following issues when implementing the Hooks pattern:

1. Calling Writable Memory from Read-Only Memory. You can't predict the addresses or entry points of code and data stored in main memory — indeed, because the memory is writable memory addresses can change between versions of programs (or even as a program is running). This makes it difficult for code in ROM to call code or rely on data that is stored in writable memory.

You can address this by using additional hooks that are stored at known addresses in main memory — hooks that point to code and data in main memory, rather than into read-only memory. Code in ROM can follow these hooks to find the addresses of the main memory components that it needs to use.

2. Extending Objects in Read-Only Memory.

Object-oriented environments associate operations with the objects they operate upon – called 'dynamic dispatch', 'message sending' or 'ad-hoc polymorphism'. You can use this to implement rather more flexible hooks. For example, both EPOC and Windows CE support C++ derived classes stored in RAM that inherit from base classes stored in ROM. When the system calls a C++ virtual function, the code executed may be ROM or in RAM depending on the class of the object that the function belongs to. The compiler and runtime system ensures that the C++ virtual function tables (`vtbls`) have the correct entry for each function, so the `vtbls` behave like tables of hooks [Ellis and Stroustrup 1990,]. ROM programmers can use many object-oriented design patterns (such as **FACTORY METHOD** and **ABSTRACT FACTORY**) to implement extensible code [Gamma et al 1995] because the inheritance mechanism does not really distinguish because ROM and RAM classes.

This works equally well in a Java implementation. Java's dynamic binding permits ROM-based code to call methods that may be in ROM or RAM according to the object's class.

3. Extending Data in Read-Only Memory.

Replacing ROM-based data is simplest when the data exists as files in a ROM filing system. In this case, it is sufficient to ensure that application code looks for files in other file systems before the ROM one. EPOC, for example, scans for resource files in the same directory on each drive in turn, taking the drive letters in alphabetic order. Drive Z, the ROM drive, is therefore scanned last.

You can also use accessor functions to use data structures stored in read-only memory. Provided these functions are called through hooks, you can modify the data the rest of the system retrieves from read-only memory by modifying these accessor functions.

If you access read-only memory directly, then you need *programmer discipline* to write code that can use both ROM and RAM simultaneously. When reading data, you should generally search the RAM first, then the ROM; when writing data, you can only write into the RAM. This ensures that if you replace the ROM data by writing to RAM, the updated version in RAM will be found before the original in ROM.

Example

The Strap-It-On's operating system is mostly stored in ROM, and accessed via a table of hooks. The operating system can be updated by changing the hooks. This example describes C code implementing the creation of the hook table and intercepting the operating system function `memalloc`, that allocates memory.

The basic data type in the Strap-It-On operating system is called a `sysobj` — it may be a pointer to a block of memory, a single four-byte integer, two two-byte short integers and so on. Every system call takes and returns a single `sysobj`, so the hook table is essentially a table of pointers to functions taking and returning `sysobjs`.

```
typedef void* sysobj;
const int SIO_HOOK_TABLE_SIZE = 100;
typedef sysobj (*sio_hook_function) (sysobj) ;

sio_hook_function sio_hook_table[SIO_HOOK_TABLE_SIZE];
```

As the system begins running, it stores a pointer to the function that implements `memalloc` in the appropriate place in the hook table.

```
extern sysobj sio_memalloc( sysobj );
const int SIO_MEMALLOC = 0;
sio_hook_table[SIO_MEMALLOC] = sio_memalloc;
```

Strap-It-On applications make system calls, such as the function `memalloc`, by calling 'trampoline functions' that indirect through the correct entry in the hook table.

```
void *memalloc(size_t bytesToAllocate) {
    return (void*)sio_hook_table[SIO_MEMALLOC]((sysobj)bytesToAllocate);
}
```

1. Changing a function using a hook

To change the behaviour of the system, say to implement a memory counter, we first allocate a variable to remember the address (in read-only memory) of the original implementation of the `memalloc` call. We need to preserve the original implementation because our memory counter will just count the number of bytes requested, but then needs to call the original function to actually allocate the memory.

```
static sio_hook_function original_memalloc  = 0;

static size_t mem_counter = 0;
```

We can then write a replacement function that counts the memory requested and calls the original version:

```
sysobj mem_counter_memalloc(sysobj size) {
    mem_counter += (size_t)size;
    return original_memalloc( size );
}
```

Finally, we can install the memory counter by copying the address of the existing system `memalloc` from the hook table into our variable, and install our new routine into the hook table.

```
original_memalloc = sio_hook_table[SIO_MEMALLOC];
sio_hook_table[SIO_MEMALLOC] = mem_counter_memalloc;
```

Now, any calls to `memalloc` (in client code and in the operating system, as ROM also uses the hook table) will first be processed by the memory counter code.

❖ ❖ ❖

Known Uses

The Mac, BBC Micro, and IBM PC ROMs are all reached through hook vectors in RAM, and can be updated by changing the hooks. Emacs makes great use of hooks to extend its executable-only code — this way, many users can share a copy of the Emacs binary, but each one have their own, customised environment [Stallman 1984]. NewtonScript allows objects to inherit from read-only objects, using both hooks and copy-on-write so that they can be modified [Smith 1999].

The EPOC 'Time World' application has a large ROM-based database of world cities and associated time zones, dialling codes and locations. It also permits the user to add to the list; it stores new cities in a RAM database similar to the pre-defined ROM one, and searches both whenever the user looks for a city.

EPOC takes an alternative approach to updating its ROM. Patches to ROMs are supplied as device drivers that modify the virtual memory map of the system, to map one or more new pages of code in place of the existing ROM memory. This is awkward to manage as the new code must occupy exactly the same space as the code, and exactly the same entry points at exactly the same memory addresses.

See Also

COPY-ON-WRITE is a complementary technique for changing information in **READ-ONLY** STORAGE, and **COPY-ON-WRITE** and **HOOKS** can often be used together.

Using **HOOKS** in conjunction with **READ-ONLY** storage is a special instance of the general use of hooks to extend systems one cannot change directly. Many of the Object-Oriented Design Patterns [Gamma et al 1995] patterns are also concerned with making systems extensible without direct changes.

HOOKS form an important part of the hot-spot approach to systems design [Pree 1995].