Zone Allocation

The Mach operating system utilizes a paged virtual memory management system. Because of this, it's possible to simultaneously run several applications which may each require more memory than is physically present on the system. When an application references information, the system determines if the memory page containing that information is in main memory. If it's not, a page in main memory is stored to the disk, and the page with the requested information is read in. This swapping of memory pages to the disk is orders of magnitude slower than a direct memory reference. *Zone allocation* is a programming technique used to group related data together to increase the likelyhood that the page containing requested information is already resident in memory. (A zone is memory areaÐtypically a set of memory pagesÐcontaining related data.) Zone allocation also gives the programmer the ability to keep unrelated data from diluting the amount of related data grouped together on a single page. The goal of zone allocation is to minimize the number of virtual memory pages that must be resident in main memory for any given operation, thus minimizing swapping activity and maximizing execution speed.

As an example, consider a text editor that allocates a structure for every line the user types, and frees this structure every time the user deletes the line. Imagine that the user edits three documents, each of which is small enough to fit on a single virtual memory page. If the user switches back and forth between the documents, and makes changes to each document, the application will almost certainly end up with pieces of each document on at least three memory pages (and perhaps many more, if it allocated storage for other things as well). Now, when the user scrolls from one line to another, the application will touch every page, and thus will require all of these pages in main memory. If any one of those virtual memory pages gets swapped out, it will soon need to be swapped back in, regardless of which document the user is editing. Ideally, this text editor should keep all the information for one document together, preferably on one memory page. Then, when the system swaps in the memory page for any given line in the document, it is swapping in the memory page for *every* line in the document. It also is not bringing in a lot of information that it will not soon use.

Strategy

A program must allocate memory from a previously created zone. One zone (known as the default zone) is automatically created for each program. This zone is used by the standard library **malloc()** function, by most Unix functions, and for most class instances created with a **new** method. All NEXTSTEP applications also have a zone that the Objective C run-time system creates for some rarely used data that it maintains.

If you want to use zone allocation to optimize your application's memory page usage, you will need to create additional zones. Your goal should be to partition memory along the lines that it is used in your application. Often a natural division is to dedicate a zone to each open document, or to each window.

Zones are equally useful for keeping related information in main memory, **and for keeping rarely used or unrelated information out of main memory**. Related information that is allocated from a zone will be relatively densely packed onto a set of memory pages, so a reference to one piece of data will probably page in a lot of related data, and further references to the related data are less likely to require paging activity. In addition, by allocating unrelated or infrequently used objects from another zone, you prevent them from sharing a page with frequently used data. For example, every Application Kit panel (such as the PageLayout panel) is allocated from its own zone. If a panel is allocated from the default zone, there is a high probability that it will end up sharing a virtual memory page with at least one data structure that is frequently referenced. Thus, the panel takes up space in main memory even when you have no need for it.

In this way, zones have a sort of "inverse" performance effect: Putting a panel's data structures in its own zone not only speed up that panel, but even more importantly reduce the working set for the rest of the application. Its probably correct to assume that most of the default zone will usually be resident, because it holds such a mixture of different data. Thus one goal might be to make the default zone as small as possible, making sure that any data in the default zone is used during common operation of the application. In this model, the benefit of moving a panel into its own zone is not so much to make the panel faster, but to get its memory out of the general pool of the default zone, so as not to dilute this hot zone with chunks of panel data that are never used once the panel is dismissed.

So what should you keep in mind while designing a zone usage strategy for your application? Ideally, zones should be moderate in size; excessively large zones may fail to group related data onto a small number of pages. In fact, in the case of a single zone which is very large, the zone allocation algorithm reduces to a standard (non zone) allocation algorithm with its inherent problems of fragmentation across many pages. However, there are limits to how much benefit you can derive from zone allocation if your program really requires a large data set. If your application frequently touches all the memory it allocates, your program will require every page to be resident regardless of how you allocate memory. For example, if a large spreadsheet recalculation affects every cell of the spreadsheet, every virtual memory page containing a cell will eventually be paged in, and allocating different cells from different zones will not help you. (This is not always true because of cell dependencies, data size, and recalculation order, but it is an adequate generalization.) Remember, though, that you can minimize the page count for a recalculation if the pages containing cells for one document do not contain cells for unrelated documents or graphics and other data.

You should also not create a large number of frequently used zones that have very little information in them. The free space from a memory page of one zone will not be available for allocation from other zones. An exception to this is child zones, mentioned later in this document. Child zones obtain memory from the parent zone, and thus they may share memory pages.

If your application frequently both allocates and frees a certain type of object, there are a few strategies you might consider. First, consider a separate zone for these objects. Freeing tends to fragment memory, and it may be best to prevent such objects from fragmenting other zones. Second, freeing takes a bit of time, because the freed memory must be coalesced into the free memory pool. Rather than issue a large number of **free** calls for a lot of small objects, you might create a temporary zone that can't free memory. (The ability to free allocated memory is an option when you create a zone with **NXCreateZone()**.) Such a zone will allocate memory very quickly, but can only grow in size, so you should only use this technique in specific circumstances where you will soon destroy the zone. When you are done with the objects that occupy the zone, you simply destroy it with a single call to **NXDestroyZone()**. **Be careful when pursuing such a strategy that you know which objects will go into the zone to be destroyed.** Many Application Kit objects depend on having their **free** methods called so they can deallocate resources held in other servers, such as the Window Server or the Pasteboard Server. Application Kit objects should never be reaped by destroying the zone they inhabit. If you

use **NXDestroyZone()** be sure you know that the objects in that zone do not depend on having their **free** methods called for proper deallocation.

Child Zones

A child zone is a zone within a normal zone. Child zones allow you to further control the grouping of data within a zone. For example, consider once again a text editor which creates a zone for each open document. A child zone can be used to group dynamically allocated data that is heavily used within a single document. If you dynamically allocate elements of a list to keep track of data for each line in the text editor, you can allocate the list elements from a child zone to keep them from being spread across many pages belonging to the document. You can then access every element in the list without touching a large number of the document's memory pages.

Child zones can provide the benefit of zone allocation without requiring a full memory page per zone. A normal zone is always an integral number of pages in size, and unused space in the zone is unavailable to other zones. However, several child zones can reside on a single memory page belonging to a common parent zone. You can choose a granularity value other than a multiple of the virtual memory page size when creating a child zone. However, choosing a very small value will allow the child zone to become excessively fragmented within its parent zone. Child zones can only be created from normal zones; a child zone can't have children.

A child zone can be merged back into its parent zone using **NXMergeZone()**. When you merge a child zone back into its parent, the data allocated from the child remains unchanged, but the memory then belongs to the parent zone.

Usage

The default zone is returned by **NXDefaultMallocZone()**. You will need to create additional zones to optimize

your application's memory page usage. When you create a new zone using **NXCreateZone()**, you need to specify a starting size for the zone, an allocation granularity, and whether the zone can free the memory allocated from it. Granularity is the amount by which the zone grows and shrinks; a good value for a normal (non child) zone is the size of a virtual memory page, defined by the global variable **vm_page_size** (from **<mach.h>**). Code to create a new zone for your own objects would look something like this:

```
#import <zone.h>
#import <mach.h>
typedef struct {
    float foo1, foo2, foo3;
    } Thing;

NXZone *newZone;
Thing *newThing;        /* Thing is your own structure */

newZone = NXCreateZone(vm_page_size, vm_page_size, YES);
newThing = NXZoneMalloc(newZone, sizeof(Thing));
```

Note that if you use the standard C library **malloc()**, the returned storage is allocated from the default zone. Thus, the following two lines have essentially the same functionality:

```
newThing = malloc(sizeof(Thing));
newThing = NXZoneMalloc(NXDefaultMallocZone(), sizeof(Thing));
```

When you free memory that you allocated from a specific zone, you normally use the **NXZoneFree()** function for the best performance. However, you can also use the standard **free()** function for any allocated memory, and it will free the memory for the correct zone.

Zones and the Application Kit

You can allocate Application Kit objects from specific zones. In NEXTSTEP 1.0, object allocation and

initialization were grouped together in variations of the **new** method. With NEXTSTEP 2.0, allocation and initialization were separated to give you the ability to specify the zone from which an object is allocated. The **new** method still works, but it is considered obsolete, and its use is discouraged. (Each Application Kit class inherits a **new** method to create a new object. This method typically allocates space for the new object from the default zone, and then initializes the new object using the **init** instance method.) To create a new object in your own zone, send the class object an **allocFromZone:** message (which is an Object class method), specifying your own zone. Then send the new object an **init** (or similar) message. For example, to create a new Text object in your own zone:

```
myText = [[Text allocFromZone:myZone] init];
```

Very often you may not want to specify an explicit zone for an object; rather, you may want it to be allocated from the zone of a related object. For example, let's say you have a subclass of View called a FooView. Your FooView may need a subview, and it makes most sense to put the FooView's subview in the same zone with the FooView. You can get the zone for an object by invoking Object's **zone** method. In this case, your FooView designated initializer **initFrame:** method would look something like this:

```
- initFrame:(const NXRect *)frameRect
{
    [super initFrame:frameRect];
    fooSubview = [[View allocFromZone:[self zone]] initFrame:NULL];
    [self addSubview:fooSubview];
    return self;
}
```

The fact that you can determine the zone of any object with the **zone** method, or the zone of any malloced data with **NXZoneFromPointer()** makes it very easy to propagate zone allocation through a tree of objects. These features eliminate the need to add a "zone" argument to every method and function to pass along the right zone to use for any allocation that may happen along the way.

Measurement

The **MallocDebug** application allows you to view the various objects and other data allocated in your applications zones. Examining this data will often reveal wasteful allocation, or suggest ways to group the memory into zones. In addition to listing these nodes, **MallocDebug** can also let you dynamically track exactly which malloced objects are touched by particular operations in your program. This is a powerful feature for understanding what data can data is used together.

The success of the zone strategy is somewhat "all or nothing". To illustrate, let's say you move a particular panel into its own zone. You intend that when the panel is not visible, the memory in this zone should not be used, and its pages should be free to migrate out of physical memory. However, even if 99% of the data in that zone is not used, it only takes an access to one small data item to keep its whole page in RAM. Once common pitfall is to have all the data for a panel in its own zone, but fail to ever page out those pages because the panel is updated to reflect changes in the selection even when it is not visible.

MallocDebug is very helpful in verifying whether your zones are really working, because it allows you to see a list of nodes accessed in a particular zone. Thus you can dismiss a panel and then monitor whether any data in that Panel's zone is subsequently touched. **MallocDebug** identifies a piece of data in the zone by the stack backtrace at the time of allocation, and the stack backtrace at the time it was touched. This shows you who allocated the data (and therefore what it is) and why it is being touched.

Synopsis

void ***NXZoneMalloc**(NXZone **zonep*, size_t *size*)
void ***NXZoneCalloc**(NXZone **zonep*, size_t *numElems*, size_t *byteSize*)
void ***NXZoneRealloc**(NXZone **zonep*, void **ptr*, size_t *size*)
void **NXZoneFree**(NXZone **zonep*, void **ptr*)
NXZone ***NXDefaultMallocZone**(void)
NXZone ***NXCreateZone**(size_t *startSize*, size_t *granularity*, int *canFree*)

NXZone *__NXCreateChildZone__(NXZone *_parentZone_, size_t _startSize_,
    size_t _granularity_, int _canFree_)
void __NXMergeZone__(NXZone *_zonep_)
void __NXDestroyZone__(NXZone *_zonep_)
NXZone *__NXZoneFromPtr__(void *_ptr_)
void __NXZonePtrInfo__(void *_ptr_)
int __NXMallocCheck__(void)
void __NXNameZone__(NXZone *_zonep_, char *_name_)