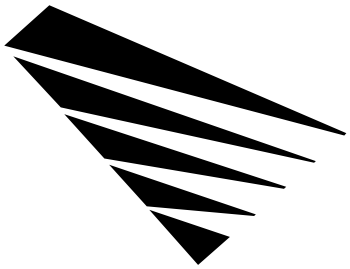

Chapter 4
Memory Pool Services

Memory Pool Introduction 32

Memory Pool Overview 32

Memory Pool Details 35
- Configuring the Memory Pool 36
- Data Structures of the Memory Pool 36
  - MemoryPoolHandle 36
  - Other Definitions 37
- Functions Calls of the Memory Pool 37

Memory Pool Introduction

This chapter describes a memory cache interface and its intended use in the NIOS Client-32 project. The interface is designed after the VCache management facility in Windows 95.

The memory cache described here is a pool of memory with an associated set of functions which are available for cached data objects, as desired by different components of the NIOS Client. While the actual implementation of the memory pool may differ between operating environments, the API and its use remain the same in order to provide a layer of independence to the consumers of this API.

The memory pool and its associated functions are referred to in this chapter as the *memory pool*, or the *system*. The users of the memory pool are referred to as the *applications*.

Memory Pool Overview

Multiple modules may wish to allocate a chunk of memory for a short period of time. Rather than returning the memory immediately to the system, the application may prefer holding on to the memory, in case it is needed again. Since memory is a limited resource, the solution is either to have the module wait a period of time before returning the memory to the system, or to have the system ask for the memory back when it is needed.

The first solution is limited because the module is unaware of the memory needs of other modules. The memory pool represents an implementation of the latter solution.

A module that wishes to access memory in a memory pool must take several steps. First, the memory must be allocated from the pool. When this occurs, the application provides a callback function which can be called by the system. The system uses this callback to request the memory be flushed, returned, and so on.

However, allocating the memory block is not enough in some systems. Consequently, the application must hold the memory before accessing it. The hold serves at least two purposes. First, it makes sure that the memory address is physically available and will not be moved. It also prevents the system from requesting that

block of memory be returned until the memory is unlocked. During the time the hold is active, the memory is available for read or write purposes.

When the application finishes accessing the memory, it removes the hold by performing an unhold request. The unhold does not mean that the system will immediately request the memory be returned, but rather it returns the block to the system queue of memory blocks. Thus, when another application makes a memory allocation request, the system requests the block which has not been accessed for the longest period of time.

The system keeps track of memory block access through an LRU (least recently used) list. This means that on one end of this list is the block that has been accessed most recently, while on the other end of the list is the block that has not been accessed for the greatest period of time. The block on the LRU end of the list will be the first one requested when an allocation request is made which requires memory to be returned to the system. (See Figure 4.1.)

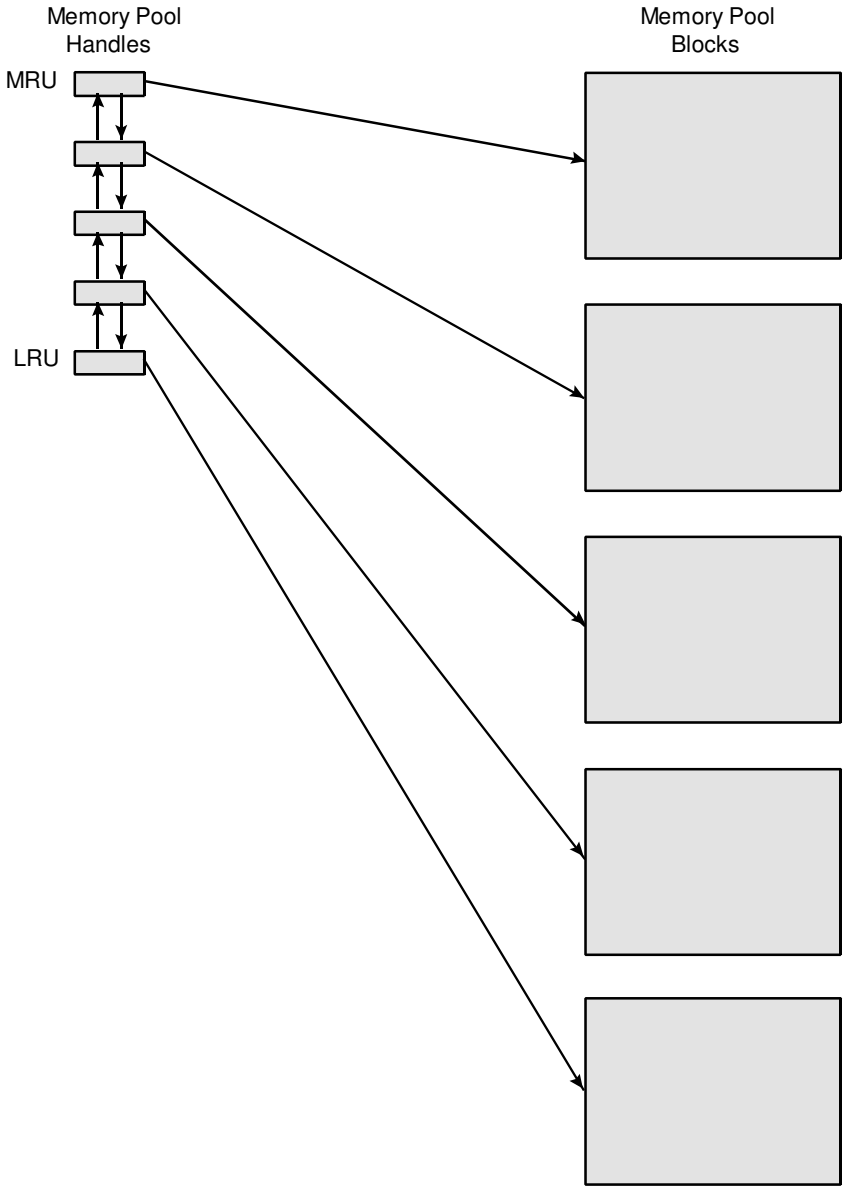

Figure 4.1: Mempool handles point to memory blocks.

An application that desires fair treatment by the memory pool must make a call to the system to make a block the most recently used (MRU) when accessing it. This call is made in addition to the hold and unhold request required when accessing the memory.

The function of making a block MRU or LRU and holding or unholding it is maintained separately to allow applications to browse through their blocks without causing them to become MRU or LRU.

Memory Pool Details

The memory pool uses an LRU algorithm to implement a linked list of memory blocks controled by memory handles. See Figure 4.2. While the application is given pointers to the memory handles, only certain portions of the handles are defined for the application's use. Applications should leave the unexposed fields alone.

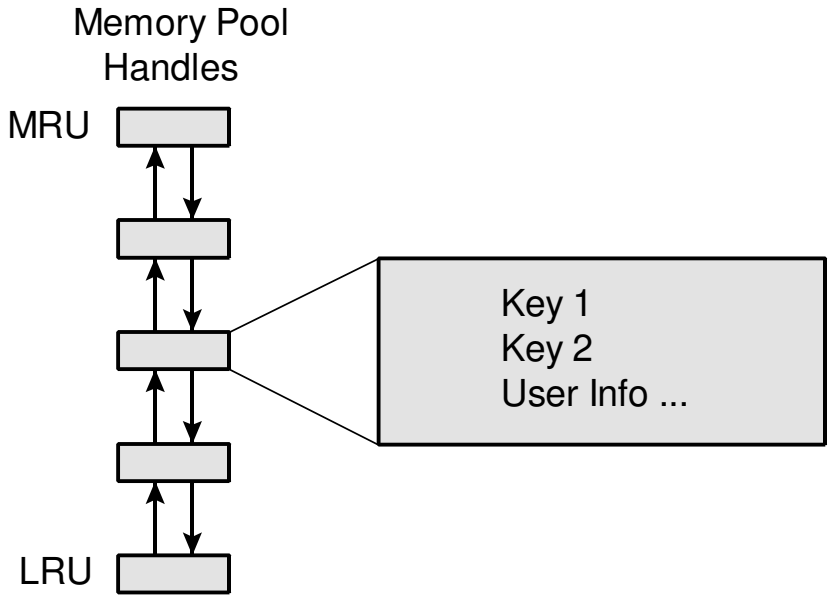

Figure 4.2: Mempool Handles

Much of the memory pool's design is modeled after Microsoft's VCache system. Thus the memory pool passes calls from applications directly through to the VCache when Windows is active, achieving system-wide sharing of memory resources. In a DOS only configuration, memory sharing is limited to NIOS Client modules.

Configuring the Memory Pool

While the memory pool is designed to snap directly into Microsoft's VCache in Windows 95 and MS Windows 3.11, under DOS it is necessary to allocate a block of system memory which acts as the source for the memory pool. To control this allocation you can add a configuration parameter to the configuration database (typically the NET.CFG file).

Memory Pool Services use one configuration parameter to allow the user to specify the amount of memory (in kilobytes) to allocate for the pool. If no parameter is specified, memory pool uses 10% of the available memory when the client registers. The configuration parameter is specified as follows:

NIOS
    MEM POOL SIZE = <value>

<value> specifies the amount of memory in kilobytes desired for the memory pool. Each block allocated in the memory pool uses 4K plus 74 bytes.

Data Structures of the Memory Pool

The following data structures and definitions are used to define the memory pool.

MemoryPoolHandle

Although much of the memory block handle is not defined for the application, some areas are made available in order to give the application some convenience in working with the memory block. Following are the fields which are defined in the memory pool handle:

```
UINT32 Key1;         offset 0x08 (08) // Set by block owner when it is allcated.
UINT32 Key2;         offset 0x0C (12) // Set by block owner when it is allcated.
void *buffPtr;       offset 0x10 (16)
UINT32 appData[7];   offset 0x14 (20) // Can be used freely by the memory block owner.
UINT8 dirty;         offset 0x32 (50) // Memory block owner should set this to non-zero
                                      when block is dirty.  Otherwise, set to zero.
```

Other Definitions

BLOCK_SIZE	0x4096
MP_CREATE	0x01
MP_HOLD	0x02
MP_MAKE_MRU	0x04
MP_LOW_PRIORITY	0x08
MP_MUST_CREATE	0x10

When a block is allocated, the truth table below explains what may happen according to the status of a given memory block.

Memory block flags			Flags needed to reuse this block
Alloc	Dirty	Held	
0	0	0	MP_LOW_PRIORITY, MP_CREATE or MP_MUST_CREATE
0	0	1	invalid state
0	1	0	invalid state
0	1	1	invalid state
1	0	0	MP_CREATE or MP_MUST_CREATE
1	0	1	Unusable
1	1	0	MP_MUST_CREATE
1	1	1	Unusable

Functions Calls of the Memory Pool

The following functions are provided as part of the memory pool system. For detailed information, see Chapter 6.

NiosMemPoolGetVersion
NiosMemPoolRegister
NiosMemPoolGetSize
NiosMemPoolCheckAvail
NiosMemPoolFindBlock
NiosMemPoolFreeBlock
NiosMemPoolMakeMRU
NiosMemPoolHold
NiosMemPoolTestHold
NiosMemPoolUnhold
NiosMemPoolEnum
NiosMemPoolMakeLRU