Mach 3 Server Writer’s Guide

Open Software Foundation and Carnegie Mellon University

Keith Loepere, Editor

This book is in the **Open Software Foundation Mach 3** series.

Books in the OSF Mach 3 series:

Mach 3 Kernel Principles

Mach 3 Kernel Interfaces

Mach 3 Server Writer’s Guide

Mach 3 Server Writer’s Interfaces

Revision History:

Revision 2	MK67, user11: January 15, 1992	OSF Mach release
Revision 2.2	NORMA-MK12, user15: July 15, 1992	

Change bars indicate changes since MK67, user 11.

This document is partially derived from earlier Mach documents written by Richard P. Draves, Michael B. Jones, Mary R. Thompson, Eric Cooper and Randall Dean.

CHAPTER 4 and CHAPTER 6 are derived from a tutorial given by Richard Draves at the 1991 USENIX Mach symposium.

CHAPTER 7 is derived from a tutorial given by David Black at the 1991 USENIX Mach symposium.

Contents

CHAPTER 1 Introduction

This document describes information of generic interest to anyone who is writing a Mach server, or any Mach task that wishes to communicate with another task. Whereas the *Kernel Interfaces* document describes the “pure kernel” which any task must use (possibly hidden under multiple emulation layers), this document describes facilities that a task does not *need* in an absolute sense, but probably wishes to use to work in the Mach environment.

Mach subscribes to the client–server model of computing. The Mach kernel provides protected access to shared hardware resources sufficient to support server tasks that provide access to the physical and logical resources of the system, and the kernel provides “interprocess” communication (IPC) between tasks so that clients may make service requests upon servers.

Mach IPC is oriented towards sending discrete messages to destinations named by kernel protected capabilities. The facility allows both synchronous and asynchronous communication. The mechanisms are sufficient to construct remote procedure calls (RPC), provide (system-level) object-oriented programming (using capabilities (port rights) to name objects) and streaming of data. Copy-on-write virtual memory optimizations are provided for efficient passing of large data. The mechanisms are network transparent.

The fine details of Mach IPC are described in the *Kernel Principles* document and specified in the *Kernel Interfaces* document. Those documents should be referenced before attempting to read this volume.

The precise details of generating messages suitable for Mach IPC, including marshalling data into and un-marshalling data out of the message, are normally handled by “stub” programs generated by MIG, the Mach Interface Generator. This tool, described in this volume, does not hide IPC features; it provides a more convenient syntax in which to express the use of these features.

MIG does not provide transparent programming language-level remote procedure call support. It is used to specify the operations performed by a server, and the parameters required for the messages between clients and servers, in a way usable by multiple languages, but mostly for C.

A Mach server executes a service loop (provided by library routines) that receives IPC messages from clients, invokes the service routine so requested, then sends a reply message to the client, returning to the service loop. The generation of a server requires:

- The specification of the messages that a client will send to the server and the replies to be returned (normally in the form of RPCs), which MIG will translate into appropriate stubs.
- The implementation of the service routines to be invoked by the MIG stubs.
- Initialization code to start the server, create appropriate initial objects, make them potentially known to clients and invoke the service loop code.

The details of these portions of a server are discussed in this volume at length.

For a server to be known to client tasks, it must register itself in some client visible name space. This name space is provided by a Name Server, the basic interface and support for which are described in this volume.

A server that wishes to process multiple service requests in parallel would use multiple threads to service the requests. Given that most servers are not written in parallel or tasking languages, the C threads package is provided. This package, discussed in this volume, supports multi-threaded programming in the C (or C++) language.

The C threads package provides a friendly interface to kernel-level threads. It provides traditional mutex and condition variable synchronization primitives. Control is provided over the creation and destruction of threads and the multiplexing of user threads onto kernel threads.

Subsequent chapters deal with details involved in server mechanisms and in writing specific types of servers. Issues involved in writing a correctly operating multi-threaded server are also covered in this volume.

This volume describes, with examples, the structure of a basic single-threaded server, the structure of a multi-threaded object-oriented server and the structure of a basic external memory manager.

CHAPTER 2 Name Server

A task starts life with a limited set of port rights. It can ask for its own port (**mach_task_self**), that of its executing thread (**mach_thread_self**), its host (**mach_host_self**) and a reply port (**mach_reply_port**). It has a handful of special ports (**task_get_special_port**) which name an exception port and a bootstrap port. The bootstrap port is typically used to name the principal server controlling the task; in the single-server system, this is the request port for the single-server. The only other ports inherently known to a new task are the *registered* ports (**mach_ports_lookup**). It is these registered ports that are used to locate extra services.

The significant port found in the set of registered ports names a *name service*. Since registered ports are inherited by created tasks, all tasks tend (unless overridden) to share the port to a single name service. The name service provides a common repository of name to server mappings that allow tasks to locate and share access to system servers.

Local Name Space

An application task locates a server with *name_server*→**netname_look_up**. This call takes an argument of a name string naming the desired service. The call returns a send right to the port associated with the service.

A server task makes itself publicly known by calling *name_server*→**netname_check_in**. This call registers a supplied port (send right) with a given service name. The server also supplies a *signature* port of its choosing that the name server requires for subsequent manipulations of this service mapping so that random tasks cannot affect the mapping. Indeed, since any task may use the **netname_check_in** function, it is necessary to provide unique service names for all of the system servers and to arrange for them to initialize and register themselves with the name server prior to running any application tasks.

A server task can de-register itself with *name_server*→**netname_check_out**. This call requires the signature port specified in the check-in call. Destroying the service port has the same result.

The original name server was part of the Net Message Server (the Net Name server), hence the name of these interfaces. The typical name service used in base Mach systems is **snames**, a simple name server. Since any number of servers could be written to provide the name server protocol, an additional call, *name_server*→**netname_version**, is provided which returns a string describing (naming) the name server in use and its version.

Network Name Space

The **snames** name server provides a single local name space. Only those clients who have as their registered name server port the port to the same **snames** server will share the given name space.

The original Net Name server (part of the Net Message server) provides a set of per-node name spaces. Clients on a node have as their registered name server port the port to the local name server. With this port they can look-up and check-in servers on their local node.

As part of system / network initialization, the Net Name server on a given node locates other nodes and announces itself to the name service on the other nodes. With the *host_name* parameter to **netname_look_up**, clients can locate servers on other nodes. It is necessary for clients to establish a convention as to what host will check-in a server (so all clients know what *host_name* value to use to find it) or clients must browse the set of name servers in some undefined manner.

Each name server is registered in its own name space (with the string “NameServer”). As such, a client can obtain the port for the local name server on another node (by supplying that node’s host name to **netname_look_up**). With this port, a client can directly look-up servers on that node, as well as check in servers on that node.

CHAPTER 3 Mach Interface Generator (MIG)

This chapter describes MIG, the Mach Interface Generator.

Introduction

MIG is a program which generates remote procedure call (RPC) code for communication between a client (hereafter referred to as the user side) and a server process (the server side). Mach servers execute as separate tasks and communicate with their users by sending Mach inter-process communication (IPC) messages. The IPC interface is language independent and fairly complex. The MIG program is designed to automatically generate procedures in C to pack and send, or receive and unpack the IPC messages used to communicate between processes.

The user must provide a specification file defining parameters of both the message passing interface and the procedure call interface. MIG then generates (by default) three files:

- User Interface Module: This module is meant to be linked into the user program. It implements and exports procedures and functions to send and receive the appropriate messages to and from the server.
- User Header Module: This module is meant to be included in the user code to define the types and routines needed at compilation time.
- Server Interface Module: This module is linked into the server process. It extracts the input parameters from an IPC message, and calls a server procedure to perform the operation. When the target server procedure or function returns, the generated interface module gathers the output parameters and correctly formats a reply message. Note that this generated module does not perform the action of receiving or sending messages, only the interpretation and generation of messages.

A given MIG specification file defines either some interfaces to a *subsystem*, or the data types of values to be passed to or returned from that subsystem, or both. By default, given a file defining interfaces with names *foo*, *bar*, etc. to subsystem *sys*, the output of MIG will be as follows:

- The user interface module will consist of C functions named *foo*, *bar*, etc. that gather the parameters defined for these operations and generate a Mach IPC message for the desired port.
- The user header module will consist of the necessary data descriptions, and the function prototypes for these C functions. The prototypes for these functions are derived from the MIG specification in a way described at length below.
- The server interface module will consist of a top-level C function whose name is set by the *serverdemux* declaration (or to *sys*_**server** if no name is chosen) appropriate for use by **mach_msg_server**. This generated routine has the following prototype:

  ```
  boolean_t sys_server (mach_msg_header_t *InHeadP, mach_msg_header_t
                *OutHeadP);
  ```

  (The actual parameters are of type *mig_reply_header_t* which are cast from the supplied *mach_msg_header_t*.) This routine examines the given message header to see if the message is one recognized by this subsystem (by examining *InHeadP*→*msgh_id*). If the message is not recognized, FALSE is returned by *sys*_**server**. If the message is recognized, the parameters from the IPC message are unpacked and the corresponding routine (i.e., *foo*, *bar*, etc.) will be called, with TRUE returned by *sys*_**server**. The return code from the called function will be returned as the *RetCode* field in the output message. By default, the routines called in the server have the same names as those called by the user since it is assumed that the user and server live in separate spaces.

  The server interface module also contains a message typing routine named *serverdemux*_**routine** (that is, the same name as the message de-multiplexing routine with **_routine** appended) useful for quickly determining if the given message is acceptable to this MIG subsystem (and which internal stub should be called). This routine is of use when messages received from a given port may be directed to multiple MIG subsystems. This generated routine has the following prototype:

  ```
  typedef void (*mig_routine_t) (mach_msg_header_t *InHeadP,
                mach_msg_header_t *OutHeadP);

  mig_routine_t sys_server_routine (mach_msg_header_t *InHeadP);
  ```

  This routine examines the given message header to see if the message is one recognized by this subsystem (by examining *InHeadP*→*msgh_id*). If the message is recognized, a pointer to a message unpacking routine is returned. The intended use is:

  ```
  mig_reply_setup (&request, &reply);
  if ((routine = sys1_server_routine (&request) != 0) ||
                  (routine = sys2_server_routine (&request) != 0) ||
                  (routine = sys3_server_routine (&request) != 0))
          (*routine) (&request, &reply);
  ```

MIG provides many controls over this generation process, including changing the names of the various generated entities.

MIG Specification File Basics

A MIG specification file contains the following components, some of which may be omitted.

- SubSystem identification
- ServerDemux declaration
- RcsId specification
- Type specifications
- Import declarations
- Operation descriptions
- Options declarations

A MIG specification file may include comments and preprocessor macros such as **#include** or **#define**. For example the statement:

> #include **<std_types.defs>**

can be used to include the type definitions for standard Mach and C types.

The subsystem identification should appear first for clarity. Types must be declared before they are used. Code is generated for the operations and import declarations in the order in which they appear in the definitions files. Options affect the operations that follow them.

SubSystem Identification

The subsystem identification statement is of the form:

> subsystem *sys message-base-id;*

sys is the name of the subsystem. It is used as the prefix for all generated file names. The user file name will be *sys*User.c, the user header file will be *sys*.h, and the server file will be *sys*Server.c. These file names may be overridden by command line switches.

message-base-id is a decimal integer that is used as the IPC message ID (*msgh_id* field) of the first operation in the specification file. Operations are numbered sequentially beginning with this base. The *message-base-id* can be selected arbitrarily by the implementor of the subsystem, but it is recommended for “official” subsystem implementors to request an unused *message-base-id* from the Mach librarian. The only technical requirement is that all requests sent to a single service port should have different IDs, to allow the server to easily distinguish them. The IPC message ID for the reply message for any given user message is 100 (decimal) more than the request ID.

ServerDemux Declaration

By default, the server de-multiplexing routine in the server stub for use by **mach_msg_server** generated by MIG is called *sys***_server**. A different name can be used if set by the *serverdemux* declaration, of the form:

> serverdemux *identifier*;

The message typing routine always has the same name as the server de-multiplexing routine with **_routine** appended.

RcsId Specification

The *rcsid* specification is of the form:

> rcsid “*$Header information$*”;

This specification causes a string variable *Sys_user_rscid* in the user and *Sys_server_rcsid* in the server module to be set to the input string.

Type Declarations

A type declaration is of the form:

> type *user-typename* = **type-desc** [**translation-info**];

(The [] indicate that the **translation-info** is optional.) The purpose of type declarations is to inform MIG about the types to use for the parameters of the various operation routines, and to inform MIG of the storage requirements for parameters and the method by which the parameters are to be “marshalled” to form the data for an IPC message. It is possible to describe the types of the parameters for a routine directly in the declaration of a routine; but, it is often more convenient to define the types and their attributes separately, giving this collection of information a **user-typename**, and then just simply use this **user-typename** when specifying a routine parameter.

The *user-typename* is the name by which MIG will refer to the type. It must eventually be defined in terms of built-in MIG types, although **<mach/std_types.defs>** and **<mach/mach_types.defs>** already make most of the needed definitions. A MIG type definition really defines three related notions:

- The MIG type name itself, used in making new MIG type declarations.
- The underlying Mach IPC type. For example, if a MIG type is defined as an array of structures of the basic type MACH_MSG_TYPE_INTEGER_32, then the Mach IPC type (which is placed in the *msgt_name* field in a data descriptor) will be MACH_MSG_TYPE_INTEGER_32. MIG will set the *msgt_size* field accordingly (in this case, 32 bits), and will compute the value for *msgt_number* on the basis of *msgt_size*, given the array and structure sizes given in the MIG type declaration.
- The data types used in the generated C code. The MIG type name is used, by default, whenever a C type name is used. The C types to be used can be directly specified. Either way, MIG separates the notions of its type name by which it refers to parameters

and the type names used in the generated C code. Regardless of whether the MIG type name is used or a user specified type name is used for the C data types, the C data types must be specified in some header file "imported" into the generated C code, as well as the users of this code.

The **type-desc** defines the properties of the MIG type, such as its derivation from basic MIG types, and the storage attributes for the type. It also specifies attributes that will implicitly or explicitly set various IPC flags in generated type descriptors:

- *msgt_inline* — data follows the type descriptor (as opposed to a pointer to the data)
- *msgt_longform* — type descriptor is the long form (*mach_msg_type_long* as opposed to *mach_msg_type*)
- *msgt_deallocate* — de-allocate related port or storage

The optional **translation-info** defines the C types to use, as well as translation functions to be called to translate data types in the generated server stub.

MIG understands the Mach IPC types as built-in types. It allows the construction of types from these built-in types as structures of, arrays of and "pointers" to these basic or generated types.

Import Declarations

If any of the type names to be used in the generated C code are other than the standard C types (e.g. int, char, etc.), C type definition files must be imported into the user and server interface modules so that they will compile. The *import* declarations specify files which are imported into the modules generated by MIG.

An *import* declaration is of one of the following forms:

import **file-name**;

uimport **file-name**;

simport **file-name**;

where **file-name** is in a form acceptable by **cpp** in **#include** statements, e.g. **<file-name>** or **"file-name"**.

For example:

import "**my_defs.h**";

import "**/usr/mach/include/cthreads.h**";

import **<cthreads.h>**;

import declarations are included in both the user and server side code. *uimport* declarations are included in just the user side. *simport* declarations are included in just the server side.

Standard Operations

There are two types of standard operations that may be specified:

- Routine
- Simple Routine

Also, the keyword *skip* is provided to allowed a procedure to be removed from a subsystem without causing all the subsequent message interfaces to be renumbered. It causes no code to be generated, but uses up a *msgh_id* number.

An operation definition has the syntax:

> *operation-type* **operation-name** (**parameter-list***)*;

The **parameter-list** is a list of parameter names and types separated by “;”. The form of each parameter is:

> **specification** *var-name:* **type-description**

where **specification** may either be omitted or one of the following:

- the direction of the parameter (*in*, *out* or *inout*)
- the explicit specification of which parameter supplies the request port (*requestport*)
- the explicit specification of the reply port to use (*replyport, sreplyport, ureplyport*)
- a property of the generated **mach_msg** call (*waittime, msgseqno, msgoption*)

type-description can be any **user-typename** that was declared in the type definitions section or can be a complete *type description* in the same form as in the type definition section. The **type-description** may be appended with additional IPC flags to override any IPC attributes defined in the **user-typename**.

A *simpleroutine* sends a message to the server but does not expect a reply. The return value of a *simpleroutine* (type *kern_return_t*) is the value returned by the **mach_msg** primitive send operation. A *simpleroutine* is used when asynchronous communication with a server is desired (or to declare the asynchronous response from the server). *simpleroutine*s cannot have *out* or *inout* parameters.

A *routine* operation waits for a reply message from the server. It is a function whose result is of type *kern_return_t*. This result indicates whether the requested operation was successfully completed. If a *routine* returns a value other than KERN_SUCCESS the reply message will not include any of the reply parameters except the error code.

ServerPrefix Specification

The *serverprefix* specification is of the form:

serverprefix *string;*

The word *serverprefix* is followed by an identifier *string* which will be prefixed to the actual names of all the following server side functions implementing the message operations. This is particularly useful when it is necessary for the user and server side functions to have different names, as must be the case when a server is also a client of the same interface. This specification applies to all operations defined after this specification in the definition file.

UserPrefix Specification

The *userprefix* specification is of the form:

userprefix *string;*

The word *userprefix* is followed by an identifier *string* which will be prefixed to the actual names of all the following user side functions calling the message operations. *serverprefix* should usually be used when different names are needed for the user and server functions, but *userprefix* is also available for completeness sake. This specification applies to all operations defined after this specification in the definition file.

Options Declarations

Along with *serverprefix* and *userprefix*, several special-purpose options about the generated code may be specified. Defaults are available for each, and simple interfaces do not usually need to change them. These options may occur more than once in the specification file. Each time an option declaration appears it sets that option for all the following operations.

The *waittime* specification has two forms:

waittime *time;*

nowaittime;

The word *waittime* is followed by an integer or an identifier that specifies the maximum time in milliseconds that the user code will wait for a reply from the server. If an identifier is used, it should be declared as an extern variable by some module in the user code. If the *waittime* option is omitted, or if the *nowaittime* statement is seen, the RPC does not return until a message is received. This value sets the *timeout* argument to the (user-side) **mach_msg** call.

The *msgoption* specification is of the form:

msgoption *manifest-constant;*

where the *manifest-constant* may be one of the option values from the file **<message.h>**. This value is OR'ed into the *option* argument to the (user-side) **mach_msg** call. The value MACH_MSG_OPTION_NONE can be used to reset the *msgoption* value. Most of the option values to **mach_msg** cannot be specified since they would require manipulating the arguments to the **mach_msg** call itself. Two that may be specified are MACH_-SEND_INTERRUPT and MACH_RCV_INTERRUPT, which cause **mach_msg** to return to the caller if the send or receive operations are interrupted instead of retrying the calls.

Type Specifications

All types referenced in a definition file must be defined if they are other than the built-in IPC types.

Simple Types

A simple type declaration declares a *user-typename* which names a MIG type that can be used for parameters of the defined operations. The declaration is of the form:

> type *user-typename* = **type-desc** [**translation-info**];

(the [] indicate that the **translation-info** is optional) where a **type-desc** is either a previously defined *user-typename* or an **ipc-type-desc** which is of the form:

> (*ipc-typename*, *size* [, **IPC-flags**])

or:

> ipc-typename

(The [] indicate that the comma separated **IPC-flags** list is optional.)

The **ipc-type-desc** of simple types is enclosed in parentheses and consists of an IPC typename, decimal integer or integer expression that is the number of bits in the IPC type, and optional IPC option keywords which are: *dealloc*, *dealloc* [], *notdealloc*, *islong*, *isnotlong*, *servercopy* and *countinout*. The user may define additional IPC types.

If the *ipc-typename* is one of the standard ones, except for MACH_MSG_TYPE_S-TRING, MACH_MSG_TYPE_STRING_C, MACH_MSG_TYPE_UNSTRUCTURED or MACH_MSG_TYPE_REAL, just the *ipc-typename* can be used.

IPC Flags

There are three sets of IPC flags: one set pertains to de-allocation of the parameter, one pertains to normal versus long form type descriptors and the third refers to handling variable sized data that may be transmitted in or out of line in the message.

The de-allocation flag is used to describe the treatment of ports and “pointers” (out-of-line data) after the messages with which they are associated have been sent. *dealloc* causes the de-allocation bit in the IPC message to be set on. Otherwise it is always off. *dealloc* can be used only with ports and “pointers”. If it is used with a port, the port will be de-allocated after the message is sent. If *dealloc* is used with a “pointer”, the memory that the pointer references will be de-allocated after the message has been sent. (It can also be used with unbounded arrays (those that can be passed either in-line or out-of-line depending on size). In such a case, the flag only applies when the data is sent out-of-line.) An error will result if *dealloc* is used with any argument other than a port or a “pointer”. The use of *dealloc* with ports is not normally needed, as there are built-in types that dispose of ports properly.

The dynamic de-allocation flag:

```
dealloc []
```

causes the generated stub to accept an additional parameter (of C type *boolean_t*) that allows the caller to specify whether the de-allocation flag should be set in the data descriptor (TRUE meaning that the data should be de-allocated).

Normally, MIG will choose the appropriate size for the type descriptor (**mach_msg_type** or **mach_msg_type_long**). The *islong* keyword forces a long form type descriptor to be used for a parameter of this type. If a long form type descriptor was not needed, a warning is issued, but a long form type descriptor is used anyway. *isnotlong* specifies that a normal form type descriptor is to be used. If a long form type descriptor is needed, a warning is issued, and a long form type descriptor will be used. This keyword is provided for compatibility with previous versions of MIG in which some parameters (variable sized arrays, in particular) always had long form descriptors. It should now be considered obsolete.

The *servercopy* keyword can be used only for *in* parameters and only affects the server side routine. This keyword is used for parameters that can optionally be transmitted either in-line or out-of-line—namely, unbounded “in-line” arrays (data declared as a “pointer” to an unbounded array is always out-of-line). Normally, parameters passed in-line (present in the message body) are destroyed on the server side when the server side routine returns (because the message buffer is recycled) whereas out-of-line parameters (those allocated their own virtual copy) will not be touched on the assumption that the server will de-allocate this space. The server is free, though, to keep this out-of-line copy. For consistency, arguments that may be optionally transmitted either in-line or out-of-line are given the in-line semantics (in as much as that they are declared as in-line); namely, the data will be destroyed when the server routine returns. To achieve the out-of-line semantics, the *servercopy* keyword is specified. When set, the server’s routine will be called with an additional parameter (of type *boolean_t*) specifying whether the corresponding argument was transmitted in-line in the message. If the data was transmitted in-line (the additional *servercopy* argument is TRUE), the data passed to the server routine will be destroyed when the server routine returns (because the message buffer will be recycled) and so the server routine must make a copy if it wishes it to persist. If the data is out-of-line, the *servercopy* parameter will be FALSE, informing the server routine that it does not need to make a copy of the data; the data will not be de-allocated upon return.

The *countinout* keyword applies to variable length *out* parameters. It does what the name implies: the count parameter that the user side must specify to receive the number of returned elements is also considered as an input parameter to be sent to the server routine so that the server does not attempt to return too many values. In the absence of this keyword, the number of elements in the user's buffer is not sent to the server. If the server returns a message with more values than for which the user allowed, the user's MIG stub will discard the data and return MIG_ARRAY_TOO_LARGE.

Simple Types

Some examples of simple type declarations are:

type *int* = MACH_MSG_TYPE_INTEGER_32;

type *my_string* = (MACH_MSG_TYPE_STRING,8*80);

type *kern_return_t* = *int*;

The MIG generated code assumes that the C types *my_string* and *kern_return_t* are defined in a compatible way by a user provided include file. The files **<mach/std_types.-defs>** and **<mach/mach_types.defs>** define the basic C and Mach types.

MIG assumes that any variable of type MACH_MSG_TYPE_STRING or MACH_MS-G_TYPE_STRING_C (which are considered the same) is declared as a C char **foo* or char *foo*[*n*]. Thus it generates code for a parameter passed by reference and uses **strncpy** for assignment statements.

Structured Types

Four kinds of structured types are recognized: arrays, C strings, structures and “pointers”.

Basic Arrays and Structures

Definitions of arrays and structures follow the Pascal-style syntax of:

struct [*size*] of **comp-type-desc**

array [*size*] of **comp-type-desc**

where **comp-type-desc** may be a simple **type-desc** or may be an *array* or *struct* type and *size* may be a decimal integer constant or expression.

If a type is declared as an array the C type must also be an array, since the MIG RPC code will treat the user type as an array (i.e. assuming it is passed by reference and generating special code for array assignments). A variable declared as a *struct* is assumed to be passed by value and treated as a C structure in assignment statements. Other than the difference in argument passing, the fixed size *struct* and *array* declarations are the same. There is no way to specify the fields of a C structure to MIG. That is, all elements of the structure must derive from the same basic C type, as any data conversion that Mach IPC may attempt when crossing machine boundaries will be based on this base type. The *size* and **comp-type-desc** are just used to give the size of the structure.

Variable Sized Arrays

The array form:

array [*: *maxsize*] of **comp-type-desc**

specifies that a variable length array is to be passed in-line in the message. In this form *maxsize* is the maximum length of the item. For variable length arrays an additional count parameter is generated to specify how much of the array is actually being used. Variable-length in-line *inout* arguments are not supported. The **comp-type-desc** must be of fixed size (i.e., not a variable sized array).

Unbounded Arrays

The final array form:

array [] of **comp-type-desc**

specifies an unbounded array whose size is provided at run-time via an additional parameter. If the size of the array does not exceed a certain size (currently 2048 bytes), it will be sent in-line, otherwise out-of-line (appearing allocated as if by **vm_allocate**). This form is not permitted for *inout* parameters.

For an unbounded *in* parameter, the data will be transmitted in-line or out-of-line as appropriate; the server will be passed a pointer to the received data (in accordance with C array rules). By default, the semantics of the data reception at the server will be as if the data is always transmitted in-line; that is, when the server routine returns, the data will be destroyed. If the server wishes to make a copy of the data, and wishes to take advantage of the fact that the data may be transmitted out-of-line (and so a virtual copy had been made), the parameter would be given the *servercopy* keyword. This will prevent the server stub from deleting the array data upon return from the service routine if the data was transmitted out-of-line; also, an additional parameter will be supplied to the server informing it whether it must make a copy (the data was in-line) or whether it can simply use the allocated copy (data was out-of-line).

For an unbounded *out* parameter, the data will be transmitted in-line or out-of-line as appropriate, visible to the user and server sides. The user supplies a pointer to a buffer pointer naming a buffer to receive the data and a pointer to the size of the buffer. Upon return, the size will reflect the number of values returned; if the values were transmitted in-line and fit into the user's buffer, the values will be placed there and the user's buffer pointer will be unchanged; otherwise, **vm_allocate** produced space will be made (either because the data was sent out-of-line or because the MIG stub allocates space) and the user's buffer pointer will be changed to point to the allocated space. For example:

```
[1]  type                                  buffer [MAX];
[2]  type*                                 ptr;
[3]  unsigned int                          count;
[4]  ptr = &buffer[0];
[5]  count = MAX;
```

```
[6]   (void) user_stub (..., &ptr, &count, ...);
 /*
      Reference data
 */
[7]   if (ptr != buffer)
[8]           (void) vm_deallocate (mach_task_self (), (vm_offset_t) ptr, count *
                        sizeof (type));
```

The server operates accordingly. The server is passed a pointer to a buffer pointer and a pointer to the maximum count. (The count will either be the user's value, if the *countinout* keyword is in effect, or the size of the stub's buffer, namely the maximum size that can be transmitted in-line.) If the server's data fits into the supplied buffer, it does so; otherwise it allocates space and changes the buffer pointer to refer to it. The *dealloc* keyword can be used to specify that this allocated memory is to be de-allocated in such cases.

C Strings

C strings have the format:

c_string [*size*]

c_string [*: *size*]

(Variable sized strings cannot be specified as *inout*.) The fixed sized C string is equivalent to an array of characters:

array [*size*] of (MACH_MSG_TYPE_STRING_C, 8)

The variable sized C string presents the semantics of C strings (as presented by the ANSI C library), namely, a null-byte terminated string. A variable sized C string will transmit the (variable sized) valid portion of the string (all characters up to and including the null byte) in the message. This form saves space in the message by not always transmitting the full size of the character array. No additional parameter is generated requesting from the sender nor providing to the receiver the current size of the string; this size is derived from the string itself.

"Pointer" (Out-of-Line) Types

In the definition of "pointer" types, the symbol "^" precedes a simple, array or structure definition.

^ **comp-type-desc**

Data types declared as "pointers" are always sent out-of-line in the message. "Pointers" to variable sized arrays (unbounded or otherwise) transmit the actual size as an additional parameter. Since sending out-of-line is considerably more expensive than in-line data, "pointer" types should only be used for large or variable amounts of data. A call that returns an out-of-line item allocates the necessary space in the user's virtual memory (as if with **vm_allocate**). It is up to the application to de-allocate this memory when it is finished with the data.

The C type name is assumed to resolve to a C type of form *... MIG will generate code to indirect through the parameters of these types to access the actual data.

Examples of Complex Types:

```
[1]  type procids =                     array [10] of int;
[2]  type procidinfo =                  struct [5*10] of
               MACH_MSG_TYPE_INTEGER_32;
[3]  type vardata =                     array [*: 1024] of int;
[4]  type array_by_value =              struct [1] of array [20] of
               MACH_MSG_TYPE_CHAR;
[5]  type page_ptr =                    ^ array [4096] of
               MACH_MSG_TYPE_INTEGER_32;
[6]  type var_array =                   ^ array [] of int;
```

Polymorphic Types

MIG supports polymorphic types. For example, using this facility, one may specify an argument which can be either an integer or a port, with the exact type determined at runtime. The type information is passed in an auxiliary argument, similar to the way size information in variable-sized arrays is handled. (If an argument is both variable-sized and polymorphic, the auxiliary type argument comes before the count argument.)

```
[1]  type poly_t = (MACH_MSG_TYPE_POLYMORPHIC,32);
[2]  simpleroutine SendPortOrInt
[3]  (
[4]                                     server: mach_port_t;
[5]                                     poly: poly_t
[6]  );
```

and then in user C code

```
[1]  mach_port_t                        server, port;
[2]  kern_return_t                      kr;
[3]  kr = SendPortOrInt (server, 5, MACH_MSG_TYPE_INTEGER_32);
[4]  kr = SendPortOrInt (server, port, MACH_MSG_TYPE_COPY_SEND);
```

The built-in type *polymorphic* is effectively defined as:

```
type polymorphic = (MACH_MSG_TYPE_POLYMORPHIC,32);
```

Type Translation Information

Optional information describing procedures for translating the type or destroying values may appear after the type definition information. Translation functions allow the type as seen by the user process and the server process to be different. Destructor functions allow the server code to automatically de-allocate input or translated types after they have been used.

For each MIG type, there are up to three corresponding C types. These are:

- The type used in the user stub module (the *UserType*), which is the value that will define the type seen in the prototype to the user side stub module and the type used for inserting data into the outgoing IPC message to the server.
- The type seen by the server stub module (the *ServerType*) when it extracts this value from the incoming IPC message. It is also the type used by the server stub when inserting a value into an outgoing (return) IPC message. No conversion functions are called to convert between the *UserType* and the *ServerType*; these types must be effectively equivalent (at least in storage requirements) and are simply different names given by the different environments to the same storage.
- The translated type used by the server procedure (the *TransType*). This is the value used to define the type seen by the target routine called in the server.

By default these three types are all the same, and have the same name as the MIG type.

The *UserType* and *ServerType* may be specified with the *cusertype* and *cservertype* options, respectively, or both specified together with the *ctype* option. The *TransType* defaults to the *ServerType*; it can not be set explicitly; it is set implicitly by the specification of translation functions.

Examples:

type *mach_port_move_receive_t* = MACH_MSG_TYPE_MOVE_RECEIVE
ctype: *mach_port_t*;

type *funny_int* = *int*
cusertype: *user_int*
cservertype: *server_int*;

If the *ServerType* and the *TransType* are to be different, then translation functions must be defined to perform the conversion. All data type translations (and the destructor function as well) are performed on the server side.

Translation from the *ServerType* to the *TransType* is performed by an *intran* function. The syntax is:

intran: *TransType* **intran-function** (S*erverType)*

Note that specifying an *intran* function implicitly sets both *ServerType* and *TransType*. A call to the *intran* function is inserted automatically by MIG before the call to the target server routine. The actual translation function must be “hand coded” (that is, it is not generated by MIG) and imported, with the following prototype:

TransType **intran-function** (*ServerType x*);

Translation back from the *TransType* to the *ServerType* is performed by an *outtran* function. The syntax is:

outtran: *ServerType* **outtran-function** (T*ransType)*

Note that specifying an *outtran* function implicitly sets both *ServerType* and *TransType*. A call to the *outtran* function is inserted automatically by MIG after the call to the target server routine. The actual translation function must be "hand coded" and imported, with the following prototype:

ServerType **outtran-function** (*TransType x*);

Consider an example for task related kernel primitives:

[1] type *task_t* = *mach_port_t*
[2] intran: *task_t* **convert_port_to_task**(*mach_port_t*)
[3] outtran: *mach_port_t* **convert_task_to_port**(*task_t*)
[4] destructor: **task_deallocate** (*task_t*);

(The destructor function will be explained shortly.) In this example, *mach_port_t*, which is the type seen by the user code, is defined as a port in the message. The type seen by the target server code is *task_t*, which is some data structure used by the server (the kernel, in this case) to store information about each task it is serving. The *intran* function, **convert_port_to_task**, translates values of type *mach_port_t* to *task_t* on receipt by the server process. The *outtran* function, **convert_task_to_port**, translates values of type *task_t* to type *mach_port_t* before return.

For this example to work within the kernel, the kernel must hold some valid reference (and therefore, some lock) upon its *task_t* object. Imagine that the *intran* function will do this. This reference must be released after the target server routine is called. When *intran* functions are called, it is fairly common that the translation results in allocated storage, or perhaps additional reference counts for some objects that must be released. For this purpose, there are *destructor* functions. The syntax is:

destructor: **destructor-function** (T*ransType)*

Note that specifying a *destructor* function implicitly sets *TransType*. A call to the *destructor* function is inserted automatically by MIG after the call to the target server routine. The actual destructor function must be "hand coded" and imported, with the following prototype:

void **destructor-function** (*TransType x*);

In the example, **task_deallocate** is called on the translated input parameter, *task_t*, after the return from the server procedure and can be used to de-allocate any or all parts of the internal variable.

Destructor functions are called only for *in* parameters. They are not called if the parameter is also an *out* argument, because the correct time to de-allocate an *out* parameter is after the reply message has been sent, which is not code that can be generated by MIG. (In these circumstances, *dealloc* flags are needed.)

A *destructor* function can be used even if no translation is specified. For example, if a large out-of-line data segment is passed to the server it could use a *destructor* function to de-allocate the memory after the data was used.

Note that translation functions are called only for scalar values; MIG will not generate code to translate the multiple elements of an array.

Transmission Type Changes

MIG supports types which change during transmission. The syntax is:

> type *type-name* = *sending-IPCType* | *receiving-IPCType*;

This syntax indicates that the *msgt_name* field in the message should be set to the IPC type associated with *sending-IPCType* when the message is transmitted, and that, in some magic way, the *msgt_name* field will be the IPC type associated with *receiving-IPCType* when it is received. Such an IPC type change clearly only occurs when the associated parameter is handled by some intermediary, or when it is handled by the kernel.

In cases other than a *KernelServer* subsystem, when this type is sent (either as an *in* parameter or the return of an *out* or *inout* parameter), the *msgt_name* field is set by the sender of the message to the IPC type associated with *sending-IPCType* and the receiver expects the field to have the IPC type value associated with *receiving-IPCType*. This has the expected behavior for *in* and *out* parameters, namely, that the sender and receiver of the parameter sees the sending and receiving IPC types, respectively. For *inout* parameters, this is probably not what is desired, since the user will see the *inout* parameter return with a different IPC type then that with which it was sent.

Note that this syntax only expresses that the IPC type, as reflected in the *msgt_name* field, will be changed; it does not express an actual type change as seen by the user stub or target server routine, nor does it express how such a conversion occurs. If this feature is used with other than port types, it will probably be used in conjunction with *cusertype* and *cservertype* to express the probable (magic) type conversions that will be occurring.

One use for transmission type changes is in *KernelServer* subsystems where the recipient of a message is the kernel and so the kernel can arrange for IPC type translations. This is most likely to be used in conjunction with request and reply ports, which are translated by necessity in the transmission process. For example, given the following definitions for the Mach IPC port types:

```
#define MACH_MSG_TYPE_MOVE_RECEIVE 16
#define MACH_MSG_TYPE_MOVE_SEND17
#define MACH_MSG_TYPE_MOVE_SEND_ONCE18
#define MACH_MSG_TYPE_COPY_SEND19
#define MACH_MSG_TYPE_MAKE_SEND20
#define MACH_MSG_TYPE_MAKE_SEND_ONCE21
#define MACH_MSG_TYPE_POLYMORPHIC-1
```

the following built-in MIG types can be viewed as defined in the following way from the Mach IPC types:

```
type MACH_MSG_TYPE_MOVE_RECEIVE = 16;
type MACH_MSG_TYPE_MOVE_SEND = 17;
type MACH_MSG_TYPE_MOVE_SEND_ONCE =18;
type MACH_MSG_TYPE_COPY_SEND = 19 | 17;
type MACH_MSG_TYPE_MAKE_SEND =20 | 17;
type MACH_MSG_TYPE_MAKE_SEND_ONCE =21 | 18;
```

The other major use of transmission type changes is to specify a parameter (most likely a port type) that is polymorphic on the sender's side (user side for *in* parameters, server side for *out* parameters, both for *inout* parameters) but of a known type on the receiver's side. That is, the sender will specify a port and a port type, but the result of sending this port will be to generate a known port type. The following built-in MIG types have such a specification:

```
type MACH_MSG_TYPE_PORT_RECEIVE =-1 | 16;
type MACH_MSG_TYPE_PORT_SEND =  -1 | 17;
type MACH_MSG_TYPE_PORT_SEND_ONCE =-1 | 18;
```

As an example, **std_types.defs** makes the following definition:

```
type mach_port_send_t = MACH_MSG_TYPE_PORT_SEND
          ctype: mach_port_t;
```

The (kernel) function **device_set_filter** is defined as follows:

```
routine device_set_filter
(
        device:                 device_t;
        receive_port:           mach_port_send_t;
        priority:               int;
        filter:                 filter_array_t
);
```

On the sending side, *receive_port* is of MIG type *polymorphic*, so it expands into an argument of type *mach_port_t* (the *ctype*) and an argument to specify the MIG type for the message header (*mach_msg_type_name_t*). The sender can supply a receive or send right, specifying MACH_MSG_TYPE_MAKE_SEND or MACH_MSG_TYPE_COPY_-SEND, respectively, for the port type parameter. On the receive side, *receive_port* will automatically become of IPC type MACH_MSG_TYPE_MOVE_SEND, and the receiver will only see a single parameter, of C type *mach_port_t*.

Operation Descriptions

An operation description has the syntax:

operation-type **operation-name** (**parameter-list***)*;

The **parameter-list** is a list of parameter names and types separated by ";". The form of each parameter is:

specification var-name: **type-description** [,**IPC-flags**] where *specification* may either be omitted or be one of the following: *in* | *out* | *inout* | *requestport* | *replyport* | *sreplyport* | *ureplyport* | *waittime* | *msgseqno* | *msgoption*. **type-description** can be any *user-typename* that was declared in the type definitions section or can be a complete **type-description** in the same form as in the type definition section. The **IPC-flags** are either omitted or consist of a comma separated list of *dealloc*, *dealloc* [], *notdealloc*, *islong*, *isnotlong*, *servercopy* and *countinout*. These flags will override any IPC flags associated with the type definition. (Refer to IPC Flags on page 12.)

The *operation-type* is either *routine*, to specify an operation that expects a reply, or *simpleroutine*, for an operation that does not receive a reply.

Request and Reply Ports

The first (not otherwise specified) parameter in any operation statement is assumed be the *requestport* unless a *requestport* parameter is specified. This is the port to which the message will be sent.

By default, MIG user stub routines use a MIG internal reply port for receiving the reply message. The *replyport* specification allows one of the parameters to be an explicitly named reply port. The *sreplyport* and *ureplyport* specifications have a related effect that is related to the generation of the actual function interface prototypes, as explained below.

Parameter Direction

The keywords *in*, *out* and *inout* are optional and indicate the direction of the parameter. If no such keyword is given the default is *in*. The keyword *in* is used with parameters that are to be sent to the server. The keyword *out* is used with parameters to be returned by the server. The keyword *inout* is used with parameters to be both sent and returned.

WaitTime Parameter

The *waittime* keyword indicates that the corresponding argument to the user stub specifies the time-out value to be used. The value is the maximum time in milliseconds that the user code will wait for a reply from the server. If no *waittime* parameter is used, or if no wait time statement has been seen, the RPC does not return until a message is received. This value sets the *timeout* argument to the **mach_msg** call.

MsgOption Parameter

The *msgoption* keyword indicates that the corresponding argument to the user stub specifies message options for the **mach_msg** call. The value is OR’ed into the *option* argument to the **mach_msg** call.

MsgSeqno Parameter

Messages received from a port contain a sequence number indicating the order of reception. This information can be of use to multi-threaded servers to order the activities of multiple threads receiving from the same port. This sequence number can be made available with the *msgseqno* keyword. The associated server-side parameter will receive the message's sequence number.

Function Prototype Generation

The prototype for a user stub or target server function is derived from the MIG operation definition. In a general way, the argument list for the prototype is the same as that for the MIG operation, with the MIG type names replaced by the *UserType* or *TransType* names, when defined.

Parameter Order

A single parameter or possibly a set of parameters will be generated for each parameter listed in the operation definition, in the order specified in the operation definition, in both the user and server sides, with the following exceptions.

- The presence of the *replyport* parameter causes an explicit parameter to appear in the argument list for the user stub routine, as well as the corresponding argument to appear in the call to the target server routine, to pass an explicitly named reply port. If *replyport* is not specified, a reply port private to the MIG interface routines will be used; it will not appear as a parameter in any calls. The *sreplyport* and *ureplyport* parameters can be used if only the user or server side parameter is needed. When a *ureplyport* parameter is present, the user stub routine will have an extra parameter through which it can name an explicit reply port. However, the target server routine will not have this extra parameter; the reply port will be known only to the MIG generated server function. When a *sreplyport* parameter is named, the user stub routine will not have an extra parameter through which an explicit reply port is named. Instead, the MIG local reply port will be used. However, the target server routine will be passed this reply port as an explicit argument (instead of its being known only inside the MIG server stub).
- The *waittime* and *msgoption* parameters appear only on the user side.
- The *msgseqno* parameter appears only on the server side.

The *requestport* parameter, whether implicitly or explicitly specified, is always a parameter to both the user and server side routines. The *requestport* specification merely allows the specification of which parameter in the list is the request port; the default is the first parameter.

Data Parameter Type

The data parameter generated by a given MIG argument definition has a C type chosen from the MIG declared *ctype* as follows:

- A *type-name* that is a built-in *IPCType*, where *IPCType* is other than MACH_MSG_-TYPE_STRING or MACH_MSG_TYPE_STRING_C, has *type-name* as its C data parameter type for *in* parameters, and *type-name* for *out* or *inout* parameters.

- A *type-name* that is a built-in *IPCType* of MACH_MSG_TYPE_STRING or MACH_MSG_TYPE_STRING_C has *type-name* as its C data parameter type, regardless of the direction of the parameter.
- A *type-name* that is an array type or a C string has *type-name* as its C data parameter type, regardless of the direction of the parameter. However, an unbounded *out* array has **type-name* as its C data parameter type. This type is assumed to convert to a C pointer type
- A *type-name* that is a structure type has *type-name* as its C data parameter type for *in* parameters, and **type-name* for *out* or *inout* parameters.
- A *type-name* that is a "pointer" type has *type-name* as its C data parameter type for *in* parameters, and **type-name* for *out* or *inout* parameters.

Pseudo Parameters

A MIG parameter can expand into up to five actual parameters. The first parameter in the prototype is the data parameter derived from the MIG type, as explained above.

The next (pseudo) parameter is added for polymorphic types, to supply the actual IPC type at run-time. Whether such a parameter is added to the user or to the server side, or both, depends on the nature and direction of the polymorphic parameter.

A parameter which is polymorphic to both the sender and receiver of the parameter (for example, derived from MACH_MSG_TYPE_POLYMORPHIC), will add a parameter of type *mach_msg_type_name_t* for *in* parameters and of type **mach_msg_type_name_t* for *out* or *inout* parameters to both the user and server sides.

A parameter which is polymorphic only on the sender's side (for example, MACH_MSG_TYPE_PORT_SEND), will add the extra parameter only to the sender's side. That is, for *in* parameters, an extra parameter of type *mach_msg_type_name_t* will be added to the user side to specify the outgoing type, but not the server side. For an *out* parameter, an extra parameter of type **mach_msg_type_name_t* will be added to the server side to allow the return of the outgoing type, but not the user side. For an *inout* parameter, an extra parameter of type *mach_msg_type_name_t* will be added to the user side to specify the outgoing type, and an extra parameter of type **mach_msg_type_name_t* will be added to the server side to allow the return of the outgoing (return) type.

If the MIG type is or derives in any way from a variable sized (or unbounded) array (but not a C string), an extra parameter is added to contain the actual array size (the number of elements). MIG scales this array element count when calculating the *msgt_number* field in the message type field. This parameter is of type *mach_msg_type_number_t* for *in* parameters, and **mach_msg_type_number_t* for *out* or *inout* parameters. This extra parameter is added to both the user and server sides. (The *countinout* keyword does not affect this; it only affects whether the user side supplied count value is transmitted to the server for unbounded *out* parameters.)

If the MIG type specifies dynamic de-allocation (*dealloc* []), an additional parameter is added to the sending side to permit the specification of dynamic de-allocation. For *in* parameters, an extra parameter of type *boolean_t* will be added to the user side to specify the de-allocation flag, but not the server side. For an *out* parameter, an extra parameter of type **boolean_t* will be added to the server side to allow the return of the de-allocation flag, but not the user side. For an *inout* parameter, an extra parameter of type *boolean_t* will be added to the user side to specify the de-allocation flag, and an extra parameter of type **boolean_t* will be added to the server side to allow the return of the de-allocation flag.

If the MIG type specifies *servercopy*, an additional parameter of type *boolean_t* will be added to the server side, but not the user side, to specify whether the data value was in-line (that is, the server needs to make a copy if desired).

Special Environment Subsystems

MIG generates alternate code for the user or server stubs for various special environments. MIG generates this code when a subsystem modifier is specified in the subsystem specification:

> subsystem *subsystemmod sys message-base-id;*

KernelServer Subsystem

The *KernelServer* environment is that in which a routine which is to be the target server routine associated with a MIG definition is to reside in the Mach kernel. The target server routines have the same prototypes as they would otherwise.

A *KernelServer* subsystem modifies the behavior of MIG type translation in that the server stub expects the types to have been translated on the way to it, but it sets the *msgt_name* field on return to the IPC type associated with *receiving-IPCType*, instead of expecting the type to be changed magically.

MIG hacks port types on the server side of a *KernelServer* subsystem. Any server side C types of *mach_port_t* are automatically converted to their (true) kernel type, *ipc_port_t*. This hack was introduced to avoid the use of explicit type coercions and conditionals in the kernel MIG definitions.

Various name conflicts can arise when the user and server code exist in the same space. One method to resolve this conflict available for *KernelServer* subsystems is to redefine the names of the user stub routines. This is done with an internal header file. This file consists of one line for each operation *foo*, of the form:

> **#define** *foo foo*_external

This file can be **#included** into a file using the user interface, to force it to call the MIG user stubs instead of the target server routines, if they would otherwise have the same name. When an internal header file is generated (via the *-iheader* MIG option), this file is automatically **#include**d in the user stub file.

KernelUser Subsystem

This is the environment in which a routine in the Mach kernel wishes to call what would be the user stub. The user stub for this environment has the same prototype as it would otherwise. It differs only in that it uses a Mach kernel routine to perform the Mach IPC operation; an internal kernel port is used as the reply port.

MIG hacks port types on the user side of a *KernelUser* subsystem. Any user side C types of *mach_port_t* are automatically converted to their (true) kernel type, *ipc_port_t*. This hack was introduced to avoid the use of explicit type coercions and conditionals in the kernel MIG definitions.

Using the Interface Modules

In the following discussion let *random* be the declared subsystem name in the definitions file.

To use the calls exported by the user interface module, a user must first find the port on which to call the server. The service ports for basic system servers are inherited when a task is created and can be found in various globals. One of these ports is the port to the Name Server which can be used to check-in or look-up user supplied ports. The *reply-port* for MIG is a per-thread global that is initialized when the thread is created. The *replyport* can be specified as a parameter to an operation if necessary.

When making specific interface calls the user should be aware if any out-of-line data is being returned to it. If so, it may wish to de-allocate the space with a call to **vm_deallocate**.

The most common system error that a user of MIG interface may encounter is *invalid_-port*. This can mean several things:

- The *requestport* parameter is an invalid port or lacks send rights.
- The *replyport* is invalid or lacks receive rights. If the user is supplying this port as parameter it may be at fault. If the system provided *reply* port is being used this error should not happen.
- A port that is being passed in either the send or reply message is invalid.

timed_out is another system error a user could receive. This results from a RPC with a *timeout* value set, timing out before a reply is received. This usually only happens if the server is on a remote machine from the user. The MIG errors defined in **mig_errors.h** usually only occur if the user is using a different version of the interface than the server. MIG error codes can be interpreted by the routines in **mach_error.**

The subsystem writer must hand code two things in addition to the MIG definition file. First, the actual operations must be declared and imported into the server module, and all normal operations must be coded. Second, code must be written to receive messages, call the server interface module, and then send a reply message when appropriate. This set of functions is provided by the **libmach.a** function **mach_msg_server**. The server module exports one function called *random*__**server**, which accepts as arguments a pointer to the message received, and a pointer to a record for the reply message. The function will return TRUE if the received message id was in the server's range.

In general, a reply should always be returned for any message received unless the return code from the Server was MIG_NO_REPLY or the request message doesn't have a reply port. The boolean function value from the server function may be used to have the same receive loop processing several logically distinct server's requests. Once a server has returned TRUE, or all the servers have returned FALSE the receive-serve-send loop should send a reply (unless of course, the return code was MIG_NO_REPLY or the reply port is MACH_PORT_NULL).

Details of a Mach IPC-based server are discussed at length in CHAPTER 4 and CHAPTER 6.

Compiling Definition Files

MIG is implemented as a cover program that recognizes a few switches, calls **cpp** to process comments and preprocessor macros and then passes the preprocessed output to the program **migcom** which generates the C files.

The switches that MIG recognizes are:

–[r,R] *r*—use **mach_msg** with both the MACH_SEND_MSG and MACH_RCV_MSG options; *R*—use a pair of calls, **mach_msg** with the MACH_SEND_MSG option and **mach_msg** with the MACH_RCV_MSG option. Default is *r.*

–[q,Q] *q*—suppress warning statements; *Q*—print warning statements. Default is *Q*.

–[v,V] *v*—verbose, prints out routines and types as they are processed; *V*—compiles silently. Default is *V.*

–[s,S] *s*—generate symbol table in the *sys*Server.c code. The layout of a symbol table (*mig_symtab_t*) is defined in **<mig_errors.h>**. *S*—suppresses the symbol table. Default is *S*. This is useful for protection systems where access to the server's operations is dynamically specifiable or for providing a server call interface that is re-directed at run-time ala **syscall** (server-to-server calls made on behalf of a user).

–i instead of a single user file, generate individual files for each routine, for ease in building a library. The file name for each file is **operation-name.c**.

–server *name* name the server file *name*.

–user *name* name the user file *name*.

–header *name* name the user header file *name*.

–sheader *name* name the server header file name. This file is generated only if this option is specified. The server header file contains function prototypes for the various functions to be called on the server side by the MIG generated de-multiplexing routine.

–iheader *name* name the user internal header file *name*. This is only used for *kernelserver* subsystems.

Any switches that MIG does not recognize are passed on to **cpp**. MIG also notices if" –MD" is being passed to **cpp**. If it is, MIG fixes up the resulting *sys***.d** file to show the dependencies of the *sys***.h**, *sys***User.c** and *sys***Server.c** on the *sys***.defs** and any **#include**d "**.defs**" files. For this feature to work correctly the name of the subsystem must be the same as the name of the *sys***.defs** file.

To use MIG, give the name of the "**.defs**" file or files and any switch values on a MIG command line, for example:

mig –v *random***.defs**

If *random* is the subsystem name declared in the definitions file, then MIG will produce the files *random***.h**, *random***User.c** and *random***Server.c** as output. If the "–MD" switch was given, a *random***.d** file will also be generated.

CHAPTER 4 Basic IPC–Based Servers

A server has a wide range of concerns in order to correctly function.. This chapter concerns the details involved in writing a basic single-threaded server. Multi-threaded servers are covered in CHAPTER 6.

Basic Request Processing

Unlike most application programs that perform some processing to produce some results and then terminate, a server (almost) never terminates; it continues to process in a loop, receiving requests for service from its clients, performing them, returning results and waiting to receive more requests.

The details of the **mach_msg** call and the format of Mach messages are sufficiently involved that they are best left, and fortunately can be left, to system supplied library routines (and MIG generated stubs). The basic form of a server is as follows:

```
  /*
        Allocate a service port from which the server will receive client request
        messages.
  */
[1]     (void) mach_port_allocate (mach_task_self (),
                    MACH_PORT_RIGHT_RECEIVE, &service_port);
  /*
        “Announce” the request port in some way so that clients can find it.
  */
[2]     result = netname_check_in (name_server_port, “service-name”,
                    mach_task_self (), service_port);
[3]     if (result != KERN_SUCCESS)
[4]             die horribly...
```

```
/*
    Use mach_msg_server to loop over all incoming messages, handling error
    cases, disposition of messages, etc. while calling the server’s message de-
    multiplexing routine (normally generated by MIG).
*/
    result = mach_msg_server (MIG_demux_server, MAX_MSG_SIZE,
                service_port);
    terminate...
```

The **mach_msg_server** routine is the basic server loop. It receives an initial request message, processes it and sends a reply while waiting for the next client request. The server loop routine, beside implementing the basic server driver, hides important details concerning the **mach_msg** call, especially error conditions.

The message de-multiplexing server routine called by **mach_msg_server**,

```
boolean_t demux_server (mach_msg_header_t *request-msg,
            mach_msg_header_t *reply-msg),
```

returns TRUE if this function processed the request. In any case, it must initialize the reply message. This function is normally the *service*_**server** routine generated by MIG. This routine hides the details of the format of Mach messages. It is this routine that calls the actual target server routine.

This basic structure handles all of the details of basic RPC communication with a client. In particular, it handles message formats, handling of request and reply port rights, marshalling and un-marshalling of arguments, method dispatch and error handling.

The use of **mach_msg_server** provides a variety of benefits:

- Uses combined send/receive whenever possible.
- Sends a reply message if there is a reply port.
- De-allocates resources in request message unless server function consumes them.
- De-allocates resources in reply message if it isn't sent.
- Protects against too-large request messages.
- Protects against dead reply ports.
- Protects against “full” reply ports.

Almost all of the remaining details of request processing are concerned with details of the operation of the service routines themselves.

For example, given the simple MIG definition:

```
routine getbalance
(
        server                      : mach_port_t;
        out balance                 : int
)
```

with the corresponding client call:

```
result = getbalance (server, &balance);
```

the server only need provide the following simple service routine:

```
kern_return_t getbalance (mach_port_t server, int *balance)
{
        *balance = server_balance;
        return KERN_SUCCESS;
}
```

Handling Multiple Request Types

The *service*_**server** routine generated by MIG de-multiplexes messages associated with a single MIG definition file. It is sometimes the case that a server needs to handle messages that may be received from a given port that were generated by multiple definition files. In such cases, a custom de-multiplexing routine must be constructed.

The following example uses the *service*_**server_routine**s generated by MIG. (The construct below is faster than calling the various *service*_**server** routines in succession.)

```
boolean_t multiple_service_demux (mach_msg_header_t *inmsg,
            mach_msg_header_t *outmsg)
{
        mig_routine_t                    routine;
        mig_reply_setup (&inmsg, &outmsg);
        if ((routine = sys1_server_routine (&inmsg) != 0) ||
                (routine = sys2_server_routine (&inmsg) != 0) ||
                (routine = sys3_server_routine (&inmsg) != 0))
        {
                (*routine) (&inmsg, &outmsg);
                return TRUE;
        }
        else
                return FALSE;
}
```

Synchronous, Asynchronous and Deferred Processing

Most client–server interactions are synchronous (RPC): the client makes a request of the server and waits for a response; the server receives the request and directly replies. This mode of operation is declared as a MIG *routine*. This mode of operation is recommended because of general simplicity.

The problem with synchronous processing is the fact that clients must wait. This can either lead to loss of parallelism (the client thread can't be doing anything else), or a potential source of denial-of-service if the server is not trusted (as is the case for external memory managers). For these circumstances, asynchronous processing can be desirable. With asynchronous processing, the client does not wait for a response; at some later time the client may receive a reply, but need not. Asynchrony on the server side can involve deferring client requests, processing them out of order (by priority order, for example). Asynchronous processing is discouraged for general use for a variety of reasons:

- Asynchronous programming is difficult. It is also difficult to maintain; both the client and server code looks like an RPC; the asynchrony is not immediately evident.
- The server must be very careful about the order in which it executes client requests.
- The server's request port is no longer the sole source of control flow; the server also contributes in the way that it defers or processes these requests.
- Since it is the most common use of Mach IPC, Mach is optimized for synchronous processing. In particular, combined send/receive calls are the most efficient (and the type used by the MIG RPC stubs and **mach_msg_server**).

Asynchronous processing is necessary at times. There are two ways in which this can occur.

First, the asynchrony can be explicitly declared in MIG as a *simpleroutine*.

```
simpleroutine asetbalance
(
        server                          : mach_port_t;
        in balance                      : int
)
```

with the corresponding client call:

```
result = asetbalance (server, balance);
```

has the following (non-replying) service routine:

```
kern_return_t asetbalance (mach_port_t server, int balance)
{
        server_balance = balance;
        return KERN_SUCCESS;
}
```

Notice that both the client and server routines appear the same as their synchronous counterparts. The client call still returns a result; this is the success of the message queuing. The server routine also returns a result value to indicate to the server loop that it has accepted the request.

In typical asynchronous use, the server would eventually return a reply to the client. This reply would also have to be declared as a *simpleroutine* to MIG. For example, given what appears to a client as a synchronous request:

```
subsystem client_example                362700;
routine getbalance
(
        server                          : mach_port_t;
        out balance                     : int
)
```

with the corresponding client call:

```
result = getbalance (server, &balance);
```

The server could process this asynchronously with the following pair of MIG definitions:

```
subsystem server_example                    362700;
simpleroutine getbalance
(
            server                          : mach_port_t;
            sreplyport reply_port           : mach_port_make_send_once_t;
            out balance                     : int
)
```

Notice that the server’s view of this interface has the same message ID but differs in that it is declared as a *simpleroutine*. An important addition is the *sreplyport* declaration. Since a reply must be generated explicitly by the server, the server must know the reply port itself, instead of this being known merely by the server loop routine.

Since the server is so declared, the MIG stub does not expect a reply. Not only must the server generate a reply, but the reply message must have its own MIG definition. A reply message consists of:

- The client’s reply port.
- The client’s result code.
- Any return values.

```
type reply_port_t = MACH_MSG_TYPE_MOVE_SEND_ONCE ctype:
                    mach_port_t;
msgoption MACH_SEND_TIMEOUT;
subsystem server_example_reply              363700;
simpleroutine getbalance_reply
(
            reply_port                      : reply_port_t;
            in result                       : kern_return_t;
            in balance                      : int
)
```

Note that the reply message ID is 100 more than the request message ID. Note that what were output values in the *routine* are now input values to the reply stub. This pair of messages could have the following (effectively synchronous) routine:

```
[1]   kern_return_t getbalance (mach_port_t server, mach_port_t reply_port, int
                *balance)
[2]   {
[3]           result = getbalance_reply (reply_port, KERN_SUCCESS,
                        server_balance);
[4]           return KERN_SUCCESS;
[5]   }
```

In line [3], the KERN_SUCCESS value is the result code to be returned to the client. The KERN_SUCCESS value in line [4] indicates to the MIG stub that the client request was accepted.

The *msgoption* of MACH_SEND_TIMEOUT is supplied to provide protection against a full reply port. This will cause the send to time-out (after zero time since the MIG stubs provide a zero value for the time-out argument to **mach_msg**) if the message cannot be sent.

In some cases, a server may provide a synchronous interface for which some requests for service are indeed handled synchronously but for which some requests must be deferred (perhaps processed out of order) and therefore processed asynchronously. In this case, the client would still see the synchronous (*routine*) MIG definition. The server also wants to use this definition, since it wants to use synchronous processing when possible. A server routine can have it both ways, that is, to be able to synchronously or asynchronously handle different requests as they come in. To do this, the server declares its interface as a *routine*. It also needs to declare a *simpleroutine* reply message as above to handle the cases when it will defer processing and send its own response. When the server routine does handle a request synchronously, it returns KERN_SUCCESS. When the routine wishes to defer processing, it returns MIG_NO_REPLY. This tells the server loop not to return a reply, but otherwise indicates the server routine did accept the client’s request.

Consume-On-Success

The server loop and message de-multiplexing routines handle the details of message formats and transmittal. The in-coming request and out-going reply message headers, as well as the in-line data and descriptors, are considered the property of **mach_msg_server**. The server routine allocates space for the messages. It recycles these spaces as it receives additional messages. For *in* arguments to the service function, this is not a problem; all of these values are passed by value. For *out* arguments, which are passed by pointer reference (according to C requirements), this can be a potential source of error; the space to which these pointers refer will be recycled when the service routine returns so it must not attempt to hold these pointers once it returns.

Data not contained in the message body (out-of-line, declared as pointer types (“^”) in MIG) or port rights (remember that the value in the message is a port name, the actual right is kernel protected and not actually in the message) need special processing. These resources (port rights and out-of-line memory) must be counted in some way and de-allocated when no longer needed. The responsibility for dealing with these resources varies with circumstances.

The basic rule is referred to as *consume-on-success*. Any resources present in the request message are the responsibility of the service function, if it is successful in responding to the request. Success is defined in two ways:

- Returning KERN_SUCCESS. This says that the user’s request was processed and all in-coming resources are recorded or de-allocated.
- Returning MIG_NO_REPLY. This says that the user’s request has been accepted and all in-coming resources are recorded by the server routine.

If the server routine returns any other value, it is assumed that the server routine has rejected the request. As such, **mach_msg_server** will destroy all resources present in the in-coming message. This behavior occurs whether the server routine was a *routine* or a *simpleroutine*. Therefore, the return code from asynchronous server functions is important.

For example:

```
type memory_t = ^array [] of MACH_MSG_TYPE_INT ctype: vm_address_t;
routine setvalues
(
            server                                  : mach_port_t;
            in values                               : memory_t
)
```

with the corresponding client call:

```
address = 0;
(void) vm_allocate (mach_task_self (), &address, sizeof (int) * size, TRUE);
fill in values...
result = setvalues (server, address, size);
vm_deallocate (mach_task_self (), address, sizeof (int) * size);
```

(Of course, the *values* parameter could have the de-allocate flag set.) The corresponding server routine would be as follows:

```
kern_return_t setvalues (mach_port_t server, vm_address_t address, vm_size_t
                  size)
{
            if (size > MAX_SIZE)
                        return SERVICE_TOO_BIG;
            copy values...
            vm_deallocate (mach_task_self (), address, sizeof (int) * size);
            return KERN_SUCCESS;
}
```

Notice that the memory is de-allocated only in the success path.

Likewise,

```
routine giveport
(
            server                                  : mach_port_t;
            in port                                 : mach_port_t =
                              MACH_MSG_TYPE_MOVE_RECEIVE
)
```

with the corresponding client call:

```
(void) mach_port_allocate (mach_task_self (),
                  MACH_PORT_RIGHT_RECEIVE, &port);
result = giveport (server, port);
```

The corresponding server routine would be as follows:

```
kern_return_t giveport (mach_port_t server, mach_port_t port)
{
            if (MACH_PORT_VALID (port))
                        mach_port_mod_refs (mach_task_self (), port,
                                    MACH_PORT_RIGHT_RECEIVE, -1);
```

```
        return KERN_SUCCESS;
}
```

What about resources in reply messages? Since the server routine loses control after returning to the MIG stub, it can do nothing. Therefore, **mach_msg_server** has the responsibility. The only way to dispose of the resources in the reply message is to use the appropriate port types that cause movement of the rights, or the appropriate de-allocate flag so that out-of-line memory is deleted when the reply message is queued. If the message cannot be delivered, **mach_msg_server** takes the responsibility of destroying all resources that would have been destroyed (or moved) if the message had been sent.

Port Reference Maintenance

As mentioned above, server routines are responsible for the maintenance of the port rights they receive in messages. Also, the interface definitions must specify the correct disposition of port rights in outgoing (reply) messages.

The request port in the incoming message is not a concern. This port right is a special case in message transmission; the act of receiving the message does not cause the reception of the send right (or now dead send-once right) used to send the message. The port name in the received message header names the receive right over which the message was received; the send right count for this name is unaffected.

MIG generated stubs produce send-once rights for reply ports. As such, when the server produces a reply, this right will vanish. If a reply is explicitly generated, as in the example above, the reply port (which appears as the request port for the reply generating stub) must specify MACH_MSG_TYPE_MOVE_SEND_ONCE. (Standard type *mach_port_t* defaults to MACH_MSG_TYPE_COPY_SEND.)

The difficulties arise when other port rights are received in a message. These port rights do accumulate in the server. For instance, every message from the kernel to an external memory manager receives a new send right for the memory cache control port to name the kernel making the request. The server can remove each user right reference each time it receives one, but this can be costly. Instead, it is typical practice to keep a count of all such accumulated rights and to periodically remove them en masse.

Single-Threaded Server Example

The following source is a simplified version of the core of the **machid** server. This server maintains a mapping between port (send) rights and MachIDs (integers) naming the ports. The server provides requests to return a send right given a MachID or to return the MachID given a send right. All send rights to the same port will have the same MachID. Whenever a new send right is presented to the server (one it hasn’t seen before), a new MachID is assigned for it. The <MachID, send right> pair is added to a pair of hash tables. This entry will continue to exist until the port is destroyed. The server becomes aware of this via a dead-name notification for the port.

Type Definitions

The definition of the server's interface includes a custom C type *mach_id_t* which defines the client name for a send right and the MIG equivalent thereof.

```
#ifndef _MACHID_TYPES_H_
#define _MACHID_TYPES_H_
/*
File: machid_types.h
Author: Richard P. Draves
External definitions for machid.
*/
typedef unsigned int                    mach_id_t;
#endif _MACHID_TYPES_H_
```

MIG types. It is good practice to place type definitions in separate MIG files from the interface definitions. Notice the importation of the corresponding C header file.

```
#ifndef _MACHID_TYPES_DEFS_
#define _MACHID_TYPES_DEFS_
/*
File: machid_types.defs
Author: Richard P. Draves
MIG definitions for machid.
*/
#include <mach/std_types.defs>
type mach_id_t = unsigned;
import "machid_types.h";
#endif _MACHID_TYPES_DEFS_
```

Interface Definitions

Notice the use of *userprefix* and *serverprefix* declarations. In this way, client and server side routines will have distinct names. This can be very useful if both client and server functions are bound into the same program, perhaps during testing.

The **register** function returns a MachID given a send right. It is named **register** because it registers the new port right if not previously known. The MACH_MSG_TYPE_PORT_SEND indicates that the server routine will receive a send right; whether that right was copied, moved or made by the client is not known (or relevant).

The **lookup** function returns a port right given a MachID. Notice that the output is declared as MACH_MSG_TYPE_MOVE_SEND, which will remove a user reference from the server's send right. This is done to help keep down the number of user references that are accumulating in the server.

```
/*
File: machid.defs
Author: Richard P. Draves
MIG interface for machid.
*/
subsystem machid 4375300;
```

```
/*
A simple interface to establish and query mappings between send rights and
MachIDs.
*/
userprefix machid_;
serverprefix do_;
#include <mach/std_types.defs>
#include “machid_types.defs”
routine register
(
        server                      : mach_port_t;
        port                        : mach_port_t =
                  MACH_MSG_TYPE_PORT_SEND;
        out name                    : mach_id_t
);
routine lookup
(
        server                      : mach_port_t;
        name                        : mach_id_t;
        out port                    : mach_port_t =
                  MACH_MSG_TYPE_MOVE_SEND
);
```

Server Program Definitions

The server source itself follows. This basic translation service is derived from the Mach 3 **machid** server.

port_record_t defines the relationship between a port right and a MachID. Notice the maintenance of the count of current user references (send right count) for the port right. This example also shows the desired hash function for port names given the current implementation of Mach IPC.

```
/*
File: machid.c
Author: Richard P. Draves
MachID to name translation server.
*/
#include <stdio.h>
#include <mach.h>
#include <mach/message.h>
#include <mach/notify.h>
#include <mach_error.h>
#include “machid_types.h”
/*
Port–name records
*/
static mach_id_t next_machid_name = 1;
/*
                                        MachID to be assigned to next new
                                        port
*/
typedef struct port_record
```

```
{
            struct port_record          *pr_pnnext;
/*
                                        link for port→name hash table
*/
            struct port_record          *pr_npnext;
/*
                                        link for name→port hash table
*/
            mach_port_t                 pr_port;
/*
                                        a send right for the port
*/
            mach_port_urefs_t           pr_urefs;
/*
                                        number of user references we hold
*/
            mach_id_t                   pr_name;
/*
                                        the MachID for the port
*/
} port_record_t;
#define HASH_TABLE_SIZE                 256
/*
                                        good hash function for port names
                                        combine the low 8 bits and the
                                        high 24 bits
*/
#define PORT_HASH (port)                ((((port) & 0xff) + ((port) >> 8))
                  % HASH_TABLE_SIZE)
#define NAME_HASH (name)                ((name) % HASH_TABLE_SIZE)
/*
Hash table headers
*/
static port_record_t                    *port_to_name
                  [HASH_TABLE_SIZE];
static port_record_t                    *name_to_port
                  [HASH_TABLE_SIZE];
```

Hash Table Search Routines

```
/*
Look for a port in the port→name hash table
*/
static port_record_t * find_port (mach_port_t port)
{
            port_record_t               *this;
            for (this = port_to_name[PORT_HASH(port)]; this != NULL; this =
                        this→pr_pnnext)
                        if (this→pr_port == port)
                                    return this;
            return NULL;
}
```

```
/*
Look for a name in the name→port hash table
*/
static port_record_t * find_name (mach_id_t name)
{
        port_record_t                     *this;
        for (this = name_to_port[NAME_HASH(name)]; this != NULL; this =
                    this→pr_npnext)
                if (this→pr_name == name)
                        return this;
        return NULL;
}
```

User Reference Count

The **add_reference** routine is called when a send right is received. Most of the time this routine merely counts the new send right. When this count exceeds a reasonable bounds, the extra rights are removed en masse.

sub_reference (subtract reference) is called when a port right is being returned to a client. This routine mostly subtracts from the user reference count. If that would remove the last reference (thereby deleting our send right), a number of user references are added (prior to returning this right to the client).

```
/*
User reference count manipulation.
*/
#define MAX_UREFS                         1000
/*
                                          max urefs we will hold for a port
*/
#define MORE_UREFS                        100
/*
                                          when we need urefs, how many to
                                          make
*/
/*
Add a user-reference to the port record
*/
static void add_reference(port_record_t *pr)
{
        kern_return_t                     kr;
        if (++pr→pr_urefs > MAX_UREFS)
        {
/*
                Remove excess user references held by this module
*/
                kr = mach_port_mod_refs(mach_task_self(), pr→pr_port,
                            MACH_PORT_RIGHT_SEND, 1 –
                            pr→pr_urefs);
                if (kr == KERN_INVALID_RIGHT)
                        kr = mach_port_mod_refs(mach_task_self(),
                                    pr→pr_port,
                  MACH_PORT_RIGHT_DEAD_NAME
                  , 1 – pr→pr_urefs);
        if (kr != KERN_SUCCESS)
            quit(1, “machid: add_reference:
                  mach_port_mod_refs: %s\n”,
                  mach_error_string(kr));
        pr→pr_urefs = 1;
    }
}
/*
Take a user-reference from the port record
*/
static void sub_reference(port_record_t *pr)
{
    kern_return_t       kr;
    if (—pr→pr_urefs == 0)
    {
/*
        Create more user references for this module
*/
        kr = mach_port_mod_refs(mach_task_self(), pr→pr_port,
              MACH_PORT_RIGHT_SEND,
              MORE_UREFS);
        if (kr == KERN_INVALID_RIGHT)
            kr = mach_port_mod_refs(mach_task_self(),
                  pr→pr_port,
                  MACH_PORT_RIGHT_DEAD_NAME
                  , MORE_UREFS);
        if (kr != KERN_SUCCESS)
            quit(1, “machid: sub_reference:
                  mach_port_mod_refs: %s\n”,
                  mach_error_string(kr));
        pr→pr_urefs += MORE_UREFS;
    }
}
```

Service Routines

The following **name_lookup** and **port_lookup** routines do the actual work of locating names, creating new entries and maintaining the reference counts. They form the body of the actual top level server routines.

Note that **name_lookup** accounts for the received send right whether it is previously known or not. It does this once it finds or creates the corresponding port record. **port_lookup**, on the other hand, subtracts a reference because of the use of MACH_MSG_TYPE_MOVE_SEND when returning this right to the client.

```
/*
Convert a send right to our exported name, consuming a user-reference for the
send right
*/
static mach_id_t name_lookup (mach_port_t port)
{
        port_record_t                   *this, **bucket;
        mach_port_t                     previous;
        kern_return_t                   kr;
        if (!MACH_PORT_VALID (port))
                return 0;
        this = find_port (port);
        if (this != NULL)
        {
/*
                Found an existing port record
*/
                add_reference (this);
                return this→pr_name;
        }
/*
        No port record, so create a new one
*/
        this = (port_record_t *) malloc (sizeof *this);
        if (this == NULL)
                quit(1, "machid: malloc failed\n");
        this→pr_port = port;
        this→pr_name = next_machid_name++;
        this→pr_urefs = 1;
/*
                                        consume a user-reference
*/
/*
        Link record into port→name hash table
*/
        bucket = &port_to_name[PORT_HASH(port)];
        this→pr_pnnext = *bucket;
        *bucket = this;
/*
        Link record into name→port hash table
*/
        bucket = &name_to_port[NAME_HASH(this→pr_name)];
        this→pr_npnext = *bucket;
        *bucket = this;
/*
        Request a dead-name notification
*/
        kr = mach_port_request_notification(mach_task_self(), port,
                    MACH_NOTIFY_DEAD_NAME, TRUE, notify,
                    MACH_MSG_TYPE_MAKE_SEND_ONCE,
                    &previous);
        if ((kr != KERN_SUCCESS) || (previous != MACH_PORT_NULL))
                quit(1, "machid: mach_port_request_notification: %s\n",
                        mach_error_string(kr));
        return this→pr_name;
}
/*
Convert an exported name to a send right, returning a user-reference for the
send right
*/
```

```
static kern_return_t port_lookup (mach_id_t name, mach_port_t *portp)
{
        port_record_t                   *this;
        this = find_name (name);
        if (this == NULL)
                return KERN_INVALID_NAME;
        sub_reference (this);
        *portp = this→pr_port;
        return KERN_SUCCESS;
}
```

A port record is destroyed when the port is. This function, **port_destroy**, is called when the server receives a dead name notification. Remember that receiving a dead name notification adds a user reference to the dead right.

```
/*
Destroy the port record of a dead port, consuming an extra user-reference for
the dead-name notification
*/
static void port_destroy (mach_port_t port)
{
        port_record_t                   *this, **prev;
        kern_return_t                   kr;
/*
        Remove the port record from port→name hash table
*/
        for (prev = &port_to_name[PORT_HASH(port)]; (this = *prev) !=
                    NULL; prev = &this→pr_pnnext)
                if (this→pr_port == port)
                        break;
        if (this == NULL)
                quit(1, "machid: port_destroy: didn't find port\n");
        *prev = this→pr_pnnext;
/*
        Remove the port record from name→port hash table
*/
        for (prev = &name_to_port[NAME_HASH(this→pr_name)]; (this =
                    *prev) != NULL; prev = &this→pr_npnext)
                if (this→pr_port == port)
                        break;
        if (this == NULL)
                quit(1, "machid: port_destroy: didn't find port\n");
        *prev = this→pr_npnext;
/*
        De-allocate the dead name
*/
        kr = mach_port_mod_refs(mach_task_self(), port,
                    MACH_PORT_RIGHT_DEAD_NAME, –
                    (this→pr_urefs + 1));
        if (kr != KERN_SUCCESS)
                quit(1, "machid: port_destroy: mach_port_mod_refs:
                            %s\n", mach_error_string(kr));
        free ((char *) this);
[125]   }
```

Top-Level Server Routines

```
  /*
        Server functions for the machid interface.
  */
[126]   kern_return_t do_register (mach_port_t server, mach_port_t port, mach_id_t
                *namep)
[127]   {
[128]           mach_id_t                       name;
  /*
                We have a port right that must be consumed. name_lookup does this
                for us.
  */
[129]           name = name_lookup(port);
[130]           *namep = name;
[131]           return KERN_SUCCESS;
[132]   }
[133]   kern_return_t do_lookup (mach_port_t server, mach_id_t name, mach_port_t
                *portp)
[134]   {
[135]           mach_port_t                     port;
[136]           kern_return_t                   kr;
  /*
                We must return a port right reference to put into the reply message.
                port_lookup returns a reference for us.
  */
[137]           kr = port_lookup (name, &port);
[138]           if (kr != KERN_SUCCESS)
[139]                   return kr;
[140]           *portp = port;
[141]           return KERN_SUCCESS;
[142]   }
```

Notification Routines

```
  /*
        Server functions for the notification interface. We should only get dead-name
        notifications.
  */
[143]   kern_return_t do_mach_notify_dead_name(mach_port_t notify, mach_port_t
                name)
[144]   {
  /*
                The dead-name notification generated an extra reference for the dead
                name. port_destroy consumes it for us.
  */
[145]           port_destroy(name);
[146]           return KERN_SUCCESS;
[147]   }
[148]   kern_return_t do_mach_notify_port_deleted(mach_port_t notify, mach_port_t
                name)
[149]   {
        quit(1, “machid: do_mach_notify_port_deleted\n”);
        return KERN_FAILURE;
}
kern_return_t do_mach_notify_msg_accepted(mach_port_t notify,
            mach_port_t name)
{
        quit(1, “machid: do_mach_notify_msg_accepted\n”);
        return KERN_FAILURE;
}
kern_return_t do_mach_notify_port_destroyed(mach_port_t notify,
            mach_port_t port)
{
        quit(1, “machid: do_mach_notify_port_destroyed\n”);
        return KERN_FAILURE;
}
kern_return_t do_mach_notify_no_senders(mach_port_t notify,
            mach_port_mscount_t mscount)
{
        quit(1, “machid: do_mach_notify_no_senders\n”);
        return KERN_FAILURE;
}
kern_return_t do_mach_notify_send_once(mach_port_t port)
{
        quit(1, “machid: do_mach_notify_send_once\n”);
        return KERN_FAILURE;
}
```

Server Loop and Main Program

A custom message de-multiplexing routine (**machid_demux**) is provided so that a single server loop may service both notification and client request messages. The de-multiplexing routine merely differentiates the type on the basis of the destination port, then calls the MIG generated de-multiplexing routine for messages associated with that port.

```
/*
Main server loop support.
*/
static mach_port_t                      service;
/*
                                        our own service port
*/
static mach_port_t                      notify;
/*
                                        our notification port
*/
static boolean_t machid_demux (mach_msg_header_t *request,
            mach_msg_header_t *reply)
{
/*
        Handle a request message based on the port from which it was received.
*/
        if (request→msgh_local_port == service)
                return machid_server (request, reply);
        else if (request→msgh_local_port == notify)
                return notify_server (request, reply);
        else
                quit(1, “machid: machid_demux: bad local port %x\n”,
                            request→msgh_local_port);
}
/*
Initialization and main server loop
*/
#define MAX_MSG_SIZE                    512
/*
                                        An upper bound on messages that
                                        we handle
*/
int main (int argc, char *argv[])
{
        mach_port_t                     pset;
        kern_return_t                   kr;
/*
        Allocate our service port and check it into the name service.
*/
        (void) mach_port_allocate (mach_task_self(),
                    MACH_PORT_RIGHT_RECEIVE, &service);
        kr = netname_check_in (name_server_port, “demoMachID”,
                    mach_task_self(), service);
        if (kr != KERN_SUCCESS)
                quit (1, “machid: netname_check_in: %s\n”,
                            mach_error_string (kr));
/*
        Allocate a port for receiving notifications and a port set, and put the
        service port and the notify port into the port set.
*/
        (void) mach_port_allocate(mach_task_self(),
                    MACH_PORT_RIGHT_RECEIVE, &notify);
        (void) mach_port_allocate(mach_task_self(),
                    MACH_PORT_RIGHT_PORT_SET, &pset);
        (void) mach_port_move_member(mach_task_self(), service, pset);
        (void) mach_port_move_member(mach_task_self(), notify, pset);
        kr = mach_msg_server(machid_demux, MAX_MSG_SIZE, pset);
        quit(1, “machid: mach_msg_server: %s\n”, mach_error_string(kr));
        return 0;
}
```

Sample Client

```
/*
File: mclient.c
Author: Richard P. Draves
Sample client for machid.
*/
#include <stdio.h>
#include <mach.h>
#include <mach/message.h>
#include <mach_error.h>
```

```
#include <servers/netname.h>
#include “machid.h”
main (int argc, char *argv)
{
        mach_port_t                     server;
        kern_return_t                   kr;
/*
        Lookup the port for the machid service
*/
        kr = netname_look_up(name_server_port, “”, “demoMachID”,
                    &server);
        if (kr != KERN_SUCCESS)
                quit(1, “mclient: netname_look_up (demoMachID): %s\n”,
                            mach_error_string(kr));
/*
        Loop reading and executing commands until eof
*/
        for (;;)
        {
                char                    command[2];
                mach_port_t             port;
                mach_id_t               name;
                printf (“mclient>”);
                if (scanf(“%1s”, command) < 1)
                {
                        printf(“\n”);
                        break;
                }
                switch (command[0])
                {
                case '?':
                case 'h':
                        printf(“Commands:\n”);
                        printf (“a              – allocate a port\n”);
                        printf (“d <port>       – de-allocate a port\n”);
                        printf (“r <port>       – register a port to get a name\n”);
                        printf (“l <name>– lookup a name to get a port\n”);
                        break;
                case 'a':
/*
                        Allocate a port
*/
                        kr = mach_port_allocate(mach_task_self(),
                                    MACH_PORT_RIGHT_RECEIVE,
                                    &port);
                        if (kr == KERN_SUCCESS)
                                printf (“allocated port %x\n”, port);
                        else
                                printf (“mclient: mach_port_allocate:
                                            %s\n”, mach_error_string(kr));
                        break;
                case 'd':
```

```
                if (scanf (“%x”, &port) < 1)
                {
                        printf (“syntax error\n”);
                        break;
                }
/*
                Destroy all rights for the port
*/
                kr = mach_port_destroy(mach_task_self(), port);
                if (kr == KERN_SUCCESS)
                        printf (“de-allocated port %x\n”, port);
                else
                        printf (“mclient: mach_port_destroy:
                                    %s\n”, mach_error_string(kr));
                break;
        case 'r':
                if (scanf (“%x”, &port) < 1)
                {
                        printf (“syntax error\n”);
                        break;
                }
/*
                Get a name for the port, using a new send right
*/
                kr = machid_register (server, port,
                            MACH_MSG_TYPE_MAKE_SEND,
                            &name);
                if (kr == KERN_SUCCESS)
                        printf (“register port %x => name %d\n”,
                                    port, name);
                else
                        printf (“mclient: machid_register: %s\n”,
                                    mach_error_string(kr));
                break;
        case 'l':
                if (scanf (“%d”, &name) < 1)
                {
                        printf (“syntax error\n”);
                        break;
                }
/*
                Convert the name to a send right, which we don't de-
                allocate
*/
                kr = machid_lookup (server, name, &port);
                if (kr == KERN_SUCCESS)
                        printf (“lookup name %d => port %x\n”,
                                    name, port);
                else
                        printf (“mclient: machid_lookup: %s\n”,
                                    mach_error_string(kr));
                break;
                }
        }
        exit(0);
}
```

Makefile

```
/*
File: Makefile
Author: Richard P. Draves
Makefile for machid.
*/
all : machid mclient
machid : machid.o machidServer.o
        $(CC) –o $@ machid.o machidServer.o –lthreads –lmach
mclient : mclient.o machidUser.o
        $(CC) –o $@ mclient.o machidUser.o –lmach
mclient.o : machid.h
machid.h machidUser.c machidServer.c : machid.defs
        mig machid.defs
```

CHAPTER 5 C Threads

Mach provides a set of low-level, language-independent primitives for manipulating threads of control in support of multi-threaded programming. The C Threads package is a run-time library that provides a C language interface to these facilities. The constructs provided are similar to those found in Mesa and Modula-2+: forking and joining of threads, protection of critical regions with mutex variables and synchronization by means of condition variables.

Naming Conventions

An attempt has been made to use a consistent style of naming for the abstractions implemented by the C Threads package. All types, macros, and functions implementing a given abstract data type are prefixed with the type name and an underscore. The name of the type itself is suffixed with *_t* and is defined via a C typedef. Documentation of the form:

typedef struct *mutex* {...} **mutex_t*;

indicates that the *mutex_t* type is defined as a pointer to a *referent type* struct *mutex* which may itself be useful to the programmer. (In cases where the referent type should be considered *opaque*, documentation such as:

typedef... *cthread_t*;

is used instead.)

Continuing the example of the *mutex_t* type, the functions **mutex_alloc** and **mutex_free** provide dynamic storage allocation and de-allocation. The functions **mutex_init** and **mutex_clear** provide initialization and finalization of the referent type. These functions are useful if the programmer wishes to include the referent type itself (rather than a pointer) in a larger structure, for more efficient storage allocation. They should not be called on objects that are dynamically allocated via **mutex_alloc**. Type-specific functions such as **mutex_lock** and **mutex_unlock** are also defined, of course.

Initializing the C Threads Package

Initialization of the C threads package is fairly automatic, given the inclusion of **libthreads.a** when linking. The **libthreads** library must be included when linking before **libmach** because **libthreads** redefines certain functions to operate correctly given the presence of multiple threads.

The header file **cthreads.h** defines the C threads interface. All programs using C threads must include this file.

The **cthread_init** function initializes the C threads implementation. This call is automatically made by **_start** if the C threads package was linked with the task.

Threads of Control

The C threads package allows for multiple threads of control in a C application.

Creation

When a C program starts, it contains a single thread of control, the one executing **main**. The thread of control is an active entity, moving from statement to statement, calling and returning from procedures. New threads are created by *fork* operations.

Forking a new thread of control is similar to calling a procedure, except that the caller does not wait for the procedure to return. Instead, the caller continues to execute in parallel with the execution of the procedure in the newly forked thread. At some later time, the caller may rendezvous with the thread and retrieve its result (if any) by means of a *join* operation, or the caller may *detach* the newly created thread to assert that no thread will ever be interested in joining it.

Termination

A thread *t* terminates when it returns from the top-level procedure it was executing.[1] If *t* has not been detached, it remains in limbo until another thread either joins it or detaches it; if *t* has been detached, no rendezvous is necessary.

[1] This is also effectively true of the initial thread (executing **main**). After effectively terminating itself, the initial thread waits for all other threads to terminate. It then terminates the task.

Stack Management

The C threads package automatically allocates space for the stack for each C thread. Each C thread will have a stack of the same size. All stacks are page aligned and have a length which is an integral multiple of the page size. The only control over this process is control over the size of stacks to be allocated. Such control can only be affected at compile time by declaring (at file scope):

vm_size_t **cthread_stack_size** = *N*;

The stack size in effect can be examined at run-time by reading **cthread_stack_size**.

Thread Management

The sole means of creating new threads is via the **cthread_fork** function. This function returns a value of type *cthread_t*. The **cthread_self** function returns the *cthread_t* value that was returned by **cthread_fork** to the creator of the thread. This value forms a thread identifier that uniquely identifies the thread, and hence may be used as a key in data structures that associate user data with individual threads. Since thread identifiers may be reused by the underlying implementation, the programmer should be careful to clean up such associations when threads exit.

Although a thread automatically terminates when it returns from its top-level function, it may also terminate itself explicitly with **cthread_exit**.

The **cthread_detach** function indicates that the named thread will not rendezvous with any other thread. **cthread_join** is used to rendezvous with a thread's termination. When a rendezvous occurs, the joining thread proceeds, given the return value from the joined thread's top-level function, or the value supplied to **cthread_exit**, if that is the means of termination for the joined thread.

It follows that attempting to join one's own thread will result in deadlock.

The C threads package allows for a single value (of type *any_t*) to be associated with each thread, providing a simple form thread-specific "global" variable. This value is set by **cthread_set_data** and retrieved with **cthread_data**. More elaborate mechanisms, such as per-thread property lists or hash tables, can then be built with these primitives.

The current number of C threads can be obtained from **cthread_count**.

Execution of Threads

A set of C threads may execute in parallel on multiple processors within a system. Various factors, even beyond those involve kernel scheduling, affect the extent of parallelism achieved by a set of C threads.

Underlying C threads is the notion of *cproc*'s, and under *cproc*'s fall Mach threads. It is Mach threads that are actually managed by the kernel and therefore schedulable on processors. A C thread executes only when:

- A *cproc* has been assigned to the C thread
- A Mach thread has been assigned to the *cproc*.
- The kernel has decided to schedule the Mach thread.

(The issues involving the scheduling of Mach threads involves processor sets and task and thread assignments, policies and priorities and is not covered here.)

A *cproc* is assigned to a C thread when one is free. By default, *cproc*'s are created so that each C thread has its own. However, the number of *cproc*'s can be limited (with **cthread_set_limit**) so that the number of C threads that can be making progress is limited. A *cproc* becomes free when the C thread assigned to it terminates.

A Mach thread is assigned to a *cproc* when one is free. By default, Mach threads are created so that each *cproc* has its own. However, the number of Mach threads can be limited (with **cthread_set_kernel_limit**). Making such a limit has a variety of effects:

- The maximum degree of true parallelism is set.
- The kernel overhead associated with the number of Mach threads is controlled.
- Allowing multiple C threads the chance to share a Mach thread reduces the overhead of synchronization between them by allowing context switches without kernel involvement.

It is possible to force a binding between a C thread and a Mach thread with **cthread_wire** (**cthread_unwire** undoes this). Wiring a C thread to a Mach thread keeps that C thread from being serviced by any other Mach thread; it also prevents that Mach thread from servicing any other C thread. In this way, a Mach thread will always be available to service this C thread, in the case where the number of Mach threads is being limited to less than the number of C threads.

The **cthread_yield** procedure is a hint to the scheduler, suggesting that this would be a convenient point to schedule another thread to run on the current processor. This may be required to avoid starvation when the number of Mach threads is limited.

When a C thread executes a blocking kernel function (**mach_msg** perhaps being the most extreme example), its corresponding Mach thread is blocked as well. To avoid blocking an excessive number of Mach threads, a special version of **mach_msg**, **cthread_mach_msg** is provided. This function performs just like **mach_msg**, but it limits the number of C threads that can be “active” listeners for a port. A C thread is declared an active listener for a port when it calls **cthread_msg_active** or **cthread_mach_msg** specifying a port. It ceases to be an active listener when it calls **cthread_msg_busy** or **cthread_mach_msg** for a different port. If a C thread is not an active listener for a port when it calls **cthread_mach_msg** (intending to perform a message receive), and making it a listener would exceed the limit set for that port, the C thread itself will block (without blocking its underlying Mach thread), thereby providing more free (un-blocked) Mach threads. The C thread will get to execute its message receive operation when some other C thread stops being a listener for that port.

Synchronization

This section describes mutual exclusion and synchronization primitives, called mutexes and condition variables. In general, these primitives are used to constrain the possible interleaving of threads' execution streams. They separate the two most common uses of Dijkstra's *P* and *V* operations into distinct facilities. This approach basically implements monitors, but without the syntactic sugar.

Mutex Variables

Mutually exclusive access to mutable data is necessary to prevent corruption of data. As simple example, consider concurrent attempts to update a simple counter. If two threads fetch the current value into a (thread-local) register, increment, and write the value back in some order, the counter will only be incremented once, losing one thread's operation. A mutex solves this problem by making the fetch-increment-deposit action atomic. Before fetching a counter, a thread locks the associated mutex. After depositing a new value, the thread unlocks the mutex. If any other thread tries to use the counter in the meantime, it will block when it tries to lock the mutex. If more than one thread tries to lock the mutex at the same time, the C threads package guarantees that only one will succeed; the rest will block.

While a mutex is locked, all other attempts to lock the mutex will block until the mutex is unlocked. When the mutex is unlocked, one of the blocked threads will then succeed in locking the mutex. Unlocking a mutex does not guarantee that one of the blocked threads will run immediately; normally the unlocking thread will continue to run and other threads will have to wait for assignment to kernel threads.

A mutex variable is locked with the **mutex_lock** function. This function will block until the mutex can be (and is) locked. It is fairly common that a thread wishes to lock a mutex, but has other things it can do until it can succeed in locking the mutex. For these purposes, there is the **mutex_try_lock** function. If this function cannot lock the mutex, it simply returns instead of blocking (returning a status indicating this occurrence). The thread can proceed to accomplish other work and try its lock attempt later. A mutex is unlocked with **mutex_unlock**.

Condition Variables

Condition variables are used when one thread wants to wait until another thread has finished doing something. Every condition variable should be protected by a mutex. Conceptually, the condition is a boolean function of the shared data that the mutex protects. Commonly, a thread locks the mutex and inspects the shared data. If it doesn't like what it finds, it waits using a condition variable. This operation also temporarily unlocks the mutex, to give other threads a chance to get in and modify the shared data. Eventually, one of them should signal the condition (which wakes up the blocked thread) before it unlocks the mutex. At that point, the original thread will regain its lock and can look at the shared data to see if things have improved. It can't assume that it will like what it sees, because some other thread may have slipped in and mucked with the data after the condition was signaled.

The **condition_signal** function should be called when one thread wishes to indicate that the condition represented by the condition variable is now true. If any other threads are waiting (via **condition_wait**), then at least one of them will be awakened. If no threads are waiting, then nothing happens. The **condition_broadcast** function is similar to **condition_signal**, except that it awakens *all* threads waiting for the condition, not just one of them.

The **condition_wait** function takes as arguments a mutex *m* and a condition variable *c*. It unlocks *m*, suspends the calling thread until the specified condition is *likely* to be true, and locks *m* again when the thread resumes. Since there is no guarantee that the condition will be true when the thread resumes, use of this procedure should always be of the form:

```
mutex_lock(m);
while (/* condition is not true */)
        condition_wait (c, m);
/* do work requiring condition */
mutex_unlock (m);
```

Shared variables should be inspected on each iteration to determine whether the condition is true.

Because of the use of a mutex in the **condition_wait** call, that mutex must be held by any caller of **condition_signal** or **condition_broadcast** for the associated condition variable, or the results are unspecified.

Precautions

One must take special care with data structures that are dynamically allocated and de-allocated. In particular, if the mutex that is controlling access to a dynamically allocated record is part of the record, one must be sure that no thread is waiting for the mutex before freeing the record.

Attempting to lock a mutex that one already holds is another common error. The offending thread will block waiting for itself. This can happen when a thread is traversing a complicated data structure, locking as it goes, and reaches the same data by different paths. Another instance of this is when a thread is locking elements in an array, say to swap them, and it doesn’t check for the special case that the elements are the same.

In general, one must be very careful to avoid deadlock. Deadlock is defined as the condition in which one or more threads are permanently blocked waiting for each other. The above scenarios are a special case of deadlock. The easiest way to avoid deadlock with mutexes is to impose a total ordering on the mutexes, and then ensure that threads only lock mutexes in increasing order.

One important issue the programmer must decide is what kind of granularity to use in protecting shared data with mutexes. The two extremes are to have one mutex protecting all shared memory, and to have one mutex for every byte of shared memory. Finer granularity normally increases the possible parallelism, because less data is locked at any one time. However, it also increases the overhead lost to locking and unlocking mutexes and increases the possibility of deadlock.

Management of Synchronization Variables

A mutex or condition variable can be allocated dynamically from the heap, or the programmer can take an object of the referent type, initialize it appropriately, and then use its address.

The functions **mutex_alloc** and **condition_alloc** provide dynamic allocation of mutex and condition objects.

mutex_free and **condition_free** allow the programmer to de-allocate mutex and condition objects that were allocated dynamically. Before de-allocating such an object, the programmer must guarantee that no other thread will reference it. In particular, a thread blocked in **mutex_lock** or **condition_wait** should be viewed as referencing the object continually, so freeing the object "out from under" such a thread is erroneous, and can result in bugs that are extremely difficult to track down.

A programmer can initialize an object of the struct *mutex* or struct *condition* referent type with **mutex_init** or **condition_init**, so that its address can be used wherever an object of type *mutex_t* or *condition_t* is expected. For example, the **mutex_alloc** function could be written in terms of **mutex_init** as follows:

```
 mutex_t mutex_alloc()
{
        register mutex_t                m;
         m = (mutex_t) malloc (sizeof (struct mutex));
         mutex_init(m);
         return m;
}
```

Initialization of the referent type is most often used when the programmer has included the referent type itself (rather than a pointer) in a larger structure, for more efficient storage allocation. For instance, a data structure might contain a component of type struct *mutex* to allow each instance of that structure to be locked independently. During initialization of the instance, the programmer would call **mutex_init** on the struct *mutex* component. The alternative of using a *mutex_t* component and initializing it using **mutex_alloc** would be less efficient.

mutex_clear and **condition_clear** should be called before freeing the storage associated with an object of the struct *mutex* or struct *condition* referent type. (Clearing a mutex currently does nothing; clearing a condition variable broadcasts its condition.) For example, the **mutex_free** procedure could be written in terms of **mutex_clear** as follows:

```
void mutex_free (mutex_t m)
{
        mutex_clear(m);
        free ((char *) m);
}
```

Shared Variables

All global and static variables are shared among all threads: if one thread modifies such a variable, all other threads will observe the new value. In addition, a variable reachable from a pointer is shared among all threads that can de-reference that pointer. This includes objects pointed to by shared variables of pointer type, as well as arguments passed by reference in **cthread_fork**.

When pointers are shared, some care is required to avoid dangling reference problems. The programmer must ensure that the lifetime of the object pointed to is long enough to allow the other threads to de-reference the pointer. Since there is no bound on the relative execution speed of threads, the simplest solution is to share pointers to global or heap-allocated objects only. If a pointer to a local variable is shared, the procedure in which that variable is defined must remain active until it can be guaranteed that the pointer will no longer be de-referenced by other threads. The synchronization functions can be used to ensure this.

The programmer must remember that unless a library, like the standard C library, has been designed to work in the presence of re-entrancy, the operations provided by the library must be presumed to make unprotected use of shared data. Hence, the programmer must protect against this through the use of a mutex that is locked before every library call (or sequence of library calls) and unlocked afterwards.

Dynamic allocation and freeing of user-defined data structures is typically accomplished with the standard C functions **malloc** and **free**. The C threads package provides versions of these functions that work correctly in the presence of multiple threads.

Using the C Threads Package

To compile a program that uses C threads, the user must include the file **cthreads.h**. To link a program that uses C threads, the user must specify on the **cc** command line the **-lthreads** library before the **-lmach** library. Linking a program in this way is sufficient to initialize and begin use of the C threads package.

Debugging

Debugging with multiple C threads can be a problem for a variety of reasons:

- The order of thread context switching is not repeatable in successive executions of the program, so obvious synchronization bugs may be difficult to find.
- Since the program consists of multiple execution points, existing debuggers may not be usable.[1]
- The user may need to worry about concurrent calls to C library routines.

[1] A C thread version of **gdb** is in the works.

One way to avoid these problems is to set the number of kernel threads to 1 (**cthread_set_kernel_limit**). In this way, the various C threads act as co-routines and may be debugged more as a sequential program.

Associating Names with C Thread Objects

A name (actually, a pointer to a character string) can be associated with a C thread, a mutex or a condition variable. These names are not currently used for anything, but they can be of use to debugging. Such names are set via **cthread_set_name**, **mutex_set_name** and **condition_set_name**, and retrieved via **cthread_name**, **mutex_name** and **condition_name**.

Pitfalls of Preemptively Scheduled Threads

The C run-time library needs a substantial amount of modification in order to be used with preemptively scheduled threads. Currently the user must ensure that calls to the standard I/O library are serialized, through the use of one or more mutex variables. (The storage allocation functions **malloc** and **free** do not require any special precautions.)

The debuggers currently available under Mach cannot be used on programs that run with preemptively scheduled threads. Furthermore, the very act of turning on tracing or adding print statements may perturb programs that incorrectly depend on thread execution speed. One technique that is useful in such cases is to vary the granularity of locking and synchronization used in the program, making sure that the program works with coarse-grained synchronization before refining it further.

Examples

The following examples illustrate the use of the C threads facilities.

Hoare Monitors

This program demonstrates the use of these facilities to program Hoare monitors.

```
/*
    Producer/consumer with bounded buffer.
    The producer reads characters from stdin and puts them into the buffer. The
    consumer gets characters from the buffer and writes them to stdout. The two
    threads execute concurrently except when synchronized by the buffer.
*/
#include <stdio.h>
#include <cthreads.h>
typedef struct buffer
{
            struct mutex                lock;
            char *                      chars;
/*
                                        chars[0...size–1]
*/
            int                         size;
```

```
        int                         px, cx;
/*      producer and consumer indices */
        int                         count;
/*      number of un-consumed chars in buffer */
        struct condition            non_empty, non_full;
} *buffer_t;
void buffer_put (char ch, buffer_t b)
{
        mutex_lock (&b→lock);
        while (b→count == b→size)
                condition_wait (&b→non_full, &b→lock);
        ASSERT(0 <= b→count && b→count < b→size);
        b→chars [b→px] = ch;
        b→px = (b→px + 1)% b→size;
        b→count += 1;
        condition_signal (&b→non_empty);
        mutex_unlock (&b→lock);
}
char buffer_get (buffer_t b)
{
        char                        ch;
        mutex_lock (&b→lock);
        while (b→count == 0)
                condition_wait (&b→non_empty, &b→lock);
        ASSERT(0 < b→count && b→count <= b→size);
        ch = b→chars [b→cx];
        b→cx = (b→cx + 1)% b→size;
        b→count –= 1;
        condition_signal (&b→non_full);
        mutex_unlock (&b→lock);
        return ch;
}
void producer (buffer_t b)
{
        int                         ch;
        do
                buffer_put ((ch = getchar()), b);
        while (ch!= EOF);
}
void consumer (buffer_t b)
{
        int                         ch;
        while ((ch = buffer_get (b))!= EOF)
                printf (“%c”, ch);
}
buffer_t buffer_alloc (int size)
```

```
{
        buffer_t                        b;
        extern char *malloc();
        b = (buffer_t) malloc (sizeof (struct buffer));
        mutex_init (&b→lock);
        b→size = size;
        b→chars = malloc ((unsigned) size);
        b→px = b→cx = b→count = 0;
        condition_init (&b→non_empty);
        condition_init (&b→non_full);
        return b;
}
#define BUFFER_SIZE                     10
int main()
{
        buffer_t                        b = buffer_alloc(BUFFER_SIZE);
        cthread_detach(cthread_fork(producer, b));
        cthread_detach(cthread_fork(consumer, b));
        return 0;
}
```

Single Master / Multiple Slaves

The following example shows how to structure a program in which a single master thread spawns a number of concurrent slaves and then waits until they all finish.

```
/*
Master/slave program structure.
*/
#include <stdio.h>
#include <cthreads.h>
int                                     count;
/*
                                        number of slaves active
*/
mutex_t                                 lock;
/*
                                        mutual exclusion for count
*/
condition_t                             done;
/*
                                        signalled each time a slave finishes
*/
/*
Each slave just counts up to its argument, yielding the processor on each
iteration. When it is finished, it decrements the global count and signals that it is
done.
*/
void slave (int n)
{
        int                             i;
        for (i = 0; i < n; i += 1)
                cthread_yield();
        mutex_lock(lock);
        count –= 1;
        printf (“Slave finished %d cycles.\n”, n);
        condition_signal (done);
        mutex_unlock (lock);
}
/*
The master spawns a given number of slaves and then waits for them all to
finish.
*/
void master (int nslaves)
{
        int                                 i;
        for (i = 1; i <= nslaves; i += 1)
        {
                mutex_lock(lock);
/*
                Fork a slave and detach it, since the master never joins it
                individually.
*/
                count += 1;
                cthread_detach(cthread_fork(slave, random()% 1000));
                mutex_unlock (lock);
        }
        mutex_lock(lock);
        while (count!= 0)
                condition_wait (done, lock);
        mutex_unlock (lock);
        printf (“All %d slaves have finished.\n”, nslaves);
        cthread_exit(0);
}
int main()
{
        count = 0;
        lock = mutex_alloc();
        done = condition_alloc();
        srandom (time ((int *) 0));
/*
                                            initialize random number generator
*/
        master ((int) random()% 16);
/*
                                            create up to 15 slaves
*/
        return 0;
}
```

CHAPTER 6 Multi-Threaded IPC–Based Servers

A server has a wide range of concerns in order to correctly function. This is especially true if the server is multi-threaded; that is, if it can process multiple simultaneous requests. This chapter concerns the details involved in writing a correctly functioning multi-threaded server. (The basic issues involved in an IPC-based server are covered in CHAPTER 4.)

Basic Multi-Threaded Server Structure

A multi-threaded server can process multiple requests in parallel. The most obvious reason to write a multi-threaded server is for parallelism. Multi-threading allows a server to make use of multiple processors. Also, for server operations that can block waiting for I/O completion, multiple threads can support multiple outstanding I/O operations for greater I/O parallelism.

This latter case, overlapping I/O operations, is a special case of the more general issue of starvation. If a server has some operations that can take a long time, multiple threads allow other client requests to be processed in the interim, assuming no interdependencies.

Not all servers should be multi-threaded, though. A server whose operations are invoked infrequently (there tends to be little parallelism) or whose operations complete quickly or always prevent other concurrent operations from proceeding would be best served being single-threaded.

The issues involved in writing a multi-threaded server—server organization, locking, operation sequencing, consistency—are the basic issues involved in all system programming, clearly beyond the scope of a text such as this. These issues are driven by the nature and extent of parallelism desired and the consistency that must be presented to clients. The remainder of this text covers such issues at top level in the context of Mach IPC-based servers.

Server Organization

With a single-threaded (synchronous) server, it does not matter which service requests act upon which server maintained objects. There are no conflicts; each request is processed in order; order of requests is maintained; the single thread handles whatever object a request names, and when it is done, can safely work on a different object for the next request.

When a server is multi-threaded, the relationship between active threads and the objects they are manipulating is much more complex. Unless something is done, it is possible that multiple threads are simultaneously trying to manipulate the same object (one even possibly trying to delete it).

There are various ways to handle this class of problem. The simplest approach is to have a single thread that receives all incoming messages and hands them off, portioning them to service threads. The single thread keeps track of which service threads are doing what so that it can guarantee order and consistency. This approach has the advantage that order is easily handled. Request priorities, if provided, are also easily handled. This approach is not very efficient, though; the single thread is a bottleneck and all requests require additional context switches for processing.

Most multi-threaded servers utilize multiple threads each directly receiving requests for service. Given the multiple possible receivers, it is possible to have different mappings of service threads to objects they service.

At one extreme, each thread can receive from its own (unique) port (set) which names a set of objects maintained by that thread which is disjoint from the set maintained by all other threads. In this way, each thread is basically its own single-threaded server. This was the approach taken by some early Mach servers. The disadvantage of this approach is that given the static allocation of objects to threads, the server does not adjust to the dynamic distribution of requests from clients—it is difficult to make a reasonable assignment of a new object to a thread. Also, this approach does not simplify locking concerns as much as might be suspected; the various threads must still be concerned with the creation and deletion of objects.

At the other extreme, each service thread receives requests from a single port (set) representing all objects maintained by the server. This approach provides the greatest dynamics, but requires the greatest care for operation ordering and consistency.

Either of these multiple thread approaches makes priority management, if provided, difficult. Different ports (and sets) are typically required to represent different priority of service.

How Many Threads?

The number of threads a server should have depends on the possible or expected parallelism the server will provide.

The number and nature of threads allocated depends on the server organization. If individual threads service individual objects or clients, then the allocation of threads is driven by this concern. In a more dynamic server, the question becomes more difficult.

First of all, assuming that the possible usage could be that high, a server would normally allocate a number of Mach kernel threads at least as large as the number of processors that could execute them.

The number of C-threads multiplexed onto those kernel threads depends on the possible resource or locking conflicts that can occur while servicing a client request. For example, if no client request will ever cause a C-thread to block waiting for another, there is no reason why a one-to-one mapping of C-threads to kernel threads is not possible.

If client requests can block (especially indefinitely), there are various other concerns. First of all, there should always be a thread waiting for new client requests (such as the one that may un-block some waiting threads). One possibility is to create a new thread when the last thread becomes busy. A more deferred approach is to create a new thread when a thread (perhaps the last one) blocks.

Locking, Operation Sequencing and Consistency

Unless one allocates a different thread for each object (or set) and has no issue with the dynamic creation and deletion of objects, then one needs to lock server data structures to guarantee the proper operation sequencing and consistency. This section will briefly touch on these issues.

Lock Granularity

The basic mechanism used to maintain consistency when manipulating an object is locking. A lock (called a *mutex* in C-threads parlance) protects data (not code) by preventing other threads not holding the lock from manipulating the data protected by the lock while a thread holding the lock is manipulating the data. A lock *protects* a data element if a thread must *hold* the lock protecting the data element in order to make any reference to the data element. Since only one thread holds the lock at a time, the data element cannot be undergoing modification by another thread while being examined or modified by the lock holding thread.

The granularity of locks is concerned with the extent of data protected by that lock and the extent of the operations performed upon that data while holding the lock; coarse-grained locking is in effect when a small number of locks is used, each protecting a wide range of data (the limit being a single global lock that protects all data); fine-grained locking is in effect when each lock protects a small amount of data (possibly one lock per object or even finer-grained than that).

The locking granularity needed for a server is directly related to the reason why the server is multi-threaded in the first place: if the server wishes to perform some number of operations concurrently, the data must have sufficiently fine-grained locks to permit those multiple threads to execute without needing the locks held by the other threads. If threads are assigned per object or client, locks should be chosen accordingly; if threads are dynamically assigned, locks are chosen to permit operations to proceed with as few lock conflicts as possible given the expected distribution of requests.

It is often reasonable to use a small number of coarse-grained locks. For example, if each service request is very short, a single global lock may be reasonable. The existence of a single lock doesn’t mean that every operation in the server is serialized; there is still parallel reception and validation of service requests, the execution time of which may very much dwarf the request execution time. Such coarse-grained locks also have the advantage that they can be statically allocated.

When there are more and finer-grained locks, the complexity rises dramatically. Also, it is possible that the overhead in manipulating the locks can exceed the benefit achieved from maintaining them.

A typical form of fine-grained locks is a lock per structure. As such, the allocation of a new structure involves the allocation of a protecting lock as well, most likely in the structure itself. This brings up the topic of the existence of objects and the protection needed.

Object Existence and Naming

With a single-threaded (synchronous) server, the server has no constraint on its object naming scheme. Each service request is received, with some object name; the server determines if the named object exists, performs the operation and returns a result.

With a multi-threaded server, the issue of whether an object exists when a request comes in and locating the object is not trivial. It follows that while the server is receiving and beginning to start a service request for some object that another thread may be trying to delete that object.

A simple approach to this problem is to maintain a table that maps client object names into data structure references. This table would have a single global lock protecting the existence of elements in the table. However, this global lock forms a major bottleneck. A global lock may be necessary to protect the master data structures that govern the existence of an object (and therefore this global lock must be held to create or delete an object), but it is undesirable to have to hold the global lock merely to locate an object for each and every service request.

The base alternative is to provide clients with some identifier that more directly locates the object data structures in the server.

One such identifier is an array index. If objects are allocated from an array, clients can directly supply an array index to name an object’s data structure. The index that a client supplies may not name an entry in use, but the static location of the entry corresponding to that index allows the server to locate and lock the structure safely. With the structure locked, the server can determine whether the indexed element is still valid.

An alternative is to use the request port name itself. With **mach_port_allocate_name**, it is possible to arrange for the request port names to match the address of the object structure so named. However, since a task does not have complete control over its port name space (the kernel creates port names for rights received in messages), an arbitrary virtual address may not be usable as a port name because of name collision. Thus, some care in choosing port names is necessary

For this latter approach to be valid, the server must arrange for the object address to remain valid as long as any client can send a service request, that is, the server cannot free the object structure until it processes a no-more-senders notification. (A no-more-senders notification indicates the absence of send rights at the time the message was generated. It does not, however, indicate the absence of send-once rights. If send-once rights are given to clients as well, the server must wait until the number the number of send-once generated messages (and send-once notifications) received matches the number of send-once rights it generated.) Since the no-more-senders processing must synchronize with other threads possibly executing (older) requests upon the object, the no-more-senders processing would typically require obtaining the object's own lock.

The no-more-senders notification races with the production of send rights in the server. That is, between the time the kernel generates a no-more-senders notification and the time the server receives it, the server may have generated another send right. Thus, the server must synchronize with its own send right generation via the make-send count in the no-more-senders notification message.

The no-more-senders notification message does not include the name of the port (receive right) losing its senders. Thus, each port for which notifications are desired needs a separate port from which the no-more-senders notification can be received so as to differentiate the notifications. To avoid a proliferation of ports, a convenient choice for this port is to use the object's port itself (that is, use the same port for both client service requests and kernel notifications). The server can safely detect kernel notifications (differentiate them from fake client generated messages) by the IPC right type included in the message. Notification messages are sent via send-once rights. If only the kernel is provided send-once rights (clients need send rights so as to send multiple service requests anyway), then the server can tell messages that must have originated in the kernel.

Blocking Operations and Deadlock

It is typically the case that the data in a server will be protected by multiple locks. For example, there may be a global lock that protects the list of objects and a per-object lock protecting the state of each object. The order in which these multiple locks can be locked must be chosen carefully to avoid deadlock.

Deadlock is a situation in which multiple threads are each blocked waiting for locks that the other threads hold. For example, if thread A holds lock X and is waiting for lock Y, and thread B holds lock Y and is waiting for lock X, neither thread will ever proceed.

The typical way to avoid these situations is to choose a (partial) order in which locks must be acquired. So, given any two locks X and Y, there would be a rule stating whether X must be obtained first, or Y. If all code locks these locks in the same order, no deadlock can occur. It is not necessary that all locks be acquired by all code paths, though; if locks X, Y and Z must be acquired in that order to prevent deadlock, but a particular code path needs only locks X and Z, it need not acquire Y. Also, code is free to release the locks in any order, as long as it does not try to reacquire any locks it released.

It must be remembered that a server's own locks are not the only source of deadlock. If a thread makes a request of another thread (or another server) that then makes a (nested) request of this server/thread, this loop of calls can deadlock if the nested call requires the same lock (or thread, if the request requires service by a thread dedicated to some function).

Request Ordering

If a single client makes only synchronous requests of a server, then no matter how the server is organized, the client's requests will be processed in order. When there are multiple (asynchronously executing) clients, or clients making asynchronous service requests (such as the kernel's external memory management and notification interfaces), the issue of the order in which these requests should be executed arises.

On the one hand, it would seem as though requests should be executed in strict order according to the order in which they were received from the request port. Even this rule, though, is difficult to implement. If multiple threads can receive messages from a given port, then service thread A can receive a message, be preempted, be followed by service thread B which receives the next message and processes it before thread A even had a chance to record the fact that it received a message.

This problem can be avoided by using the kernel generated sequence numbers associated with each port. The kernel increments its sequence number whenever a message is received. The server would maintain its own count of the last message processed. When a thread receives a message, it would compare its message's sequence number with the server's count (under protection of some lock, of course); if there is a gap, then some other threads have received messages that they have not yet processed, so this thread must defer to them. Note that the set of threads holding messages waiting to be processed forms the deferred processing queue.

It is not always possible, or desirable, to execute client requests in strict first-in, first-out order, nor is it always necessary. For example, consider the data read logic in a POSIX file system. A given client request may require reading more than one disk block. The read atomicity rules state that the data read must be an atomic view of the file—the read must not show partial results of not-yet-completed write calls. Thus, all blocks affected by the data read must be locked while the data is being copied from the blocks. However, it is not necessary to keep the blocks already in memory locked while waiting for others to be transferred from disk; only when all blocks are in memory is it necessary to hold them locked. As such, a request to write one of the blocks that strictly speaking arrived at the server after the read request could be processed first.

In general, it is not necessary to execute client requests in strict client order; it is only necessary to execute them in the order that the clients can detect that they were executed. That is, if a client cannot tell that a request was executed until it gets its reply or via some other request of the server, then the server is free to (implicitly) re-order requests until it does send such a reply or respond to some other request that exposes the completeness or non-completeness of the previous request.

For increased parallelism, a server may not want to hold an object locked during the entire execution of a service request. In such a situation, then, the server needs some other defense besides the object lock to prevent no-more-senders processing from deleting the object. A typical approach is to maintain a count of the number of in-progress operations on the object (the count being protected by the object lock). The object can be deleted when the no-more-senders notification is received and the in-progress count hits zero.

Documentation

The locking rules are perhaps the most important property of a server to document. The server should document:

- The locking hierarchy.
- For each data field, the lock protecting it.
- For each module entry, what locks must be held on entry, what locks can be held and what locks might be taken or released.

Multi-Threaded Object-Oriented Example

The following C++ example shows a simplified but typical object-oriented server. In this server, a send right held by a client represents a reference to an object managed by the server. The server's port name for the corresponding receive right locates the data structure representing the object. No-more-senders notification is used to determine when an object should be deleted. The example maintains a single linked list of objects protected by a global lock; each object has its own object lock that protects manipulation of the object's visible state.

Interface Definitions

There are two interface definition files. Operations upon objects, sent to the object's port, are defined in **object**.defs. The server's lookup operation, though, is not sent to an object port, it is sent to a specific server port that responds to such requests. This operation is defined in **ooserver**.defs.

The interesting detail in the lookup operation is the use of MACH_MSG_TYPE_-MOVE_SEND. Refer to the definition of this operation (*object_list*::**find_or_make**) for an explanation.

```
/*
    File: ooserver.defs
    Author: Richard P. Draves
    Interface definitions for ooserver.
*/
subsystem ooserver                        2783600;
serverprefix do_;
#include <mach/std_types.defs>
/*
    Get an object port.
*/
routine ooserver_lookup
(
        server                            : mach_port_t;
        name                              : int;
        out object                        : mach_port_t =
                    MACH_MSG_TYPE_MOVE_SEND
);
```

This file defines a simple change and query operation for the object’s integer value.

```
/*
    File: object.defs
    Author: Richard P. Draves
    Interface definitions for ooserver objects.
*/
subsystem object                          2783700;
serverprefix do_;
#include <mach/std_types.defs>
/*
    Operations on an object port.
*/
routine object_change
(
        object                            : mach_port_t;
        value                             : int
);
routine object_query
(
        object                            : mach_port_t;
        out value                         : int
);
```

Server Program Definitions

The server code follows. It begins with the pre-requisite external definitions (all defined as *extern “C”* because they were written for general C use). Included are definitions of the MIG generated server routines. Notice that some internal routines are also declared here as *extern “C”* so that appropriate linkage is generated for them so that they can be called by MIG generated stubs.

```
/*
    File: ooserver.c
    Original C version by Richard P. Draves
    Sample object-oriented server. Each object provides a single value that can be
    set and examined.
*/
extern "C"
{
#include <stdio.h>
#include <mach.h>
#include <mach/message.h>
#include <mach/notify.h>
#include <mach_error.h>
#include <mig_errors.h>
#include "cthreads.h"
        extern kern_return_t mach_msg_server (boolean_t (*)
                        (mach_msg_header_t*, mach_msg_header_t*),
                        mach_msg_size_t, mach_port_t);
        extern kern_return_t netname_check_in (mach_port_t, char*,
                        mach_port_t, mach_port_t);
        extern void quit (int, ...);
        extern int atoi (char*);
        extern boolean_t ooserver_server(mach_msg_header_t*,
                        mach_msg_header_t*);
        extern boolean_t object_server(mach_msg_header_t*,
                        mach_msg_header_t*);
        typedef long int size_t;
        extern void *malloc(size_t);
        extern void free (void*);
        extern kern_return_t do_object_change(mach_port_t port, int value);
        extern kern_return_t do_object_query(mach_port_t port, int *valuep);
        extern kern_return_t do_ooserver_lookup(mach_port_t server, int
                        name, mach_port_t *portp);
}
```

Space and Matching Port Allocator

Since the objects maintained by this server will use their object data structure virtual address as their receiving port right name, memory space and receive rights are generated together. To handle the case where a virtual address accidentally collides with a port name, the general memory allocator (global ::**operator new**) is redefined to ensure this match. The allocator continues to allocate space until it can allocate a matching port name, then frees any extraneous blocks (*badobjects*). Any space freed will probably be usable in the future, as the conflicting rights in client messages come and go.

```
/*
    Allocate a structure and a port together, so that the structure's address and the
    port's name are the same
*/
static void* ::operator new (size_t size)
{
        char *                      badobjects = NULL;
        char *                      object_p;
        kern_return_t               kr;
        for (;;)
    {
        object_p = (char*) malloc(size);
        if (object_p == NULL)
            quit(1, “ooserver: malloc failed\n”);
/*
        Try to allocate a port with this name
*/
        kr = mach_port_allocate_name(mach_task_self(),
                MACH_PORT_RIGHT_RECEIVE,
                (mach_port_t) object_p);
        if (kr == KERN_SUCCESS)
            break;
        if (kr != KERN_NAME_EXISTS)
            quit(1, “ooserver: mach_port_allocate_name:
                        %s\n”, mach_error_string(kr));
/*
        Save this structure and try again
*/
        * (char **) object_p = badobjects;
        badobjects = object_p;
    }
/*
    Free the unused objects
*/
    while (badobjects != NULL)
    {
        char *              next = * (char **) badobjects;
        free(badobjects);
        badobjects = next;
    }
    return (void*) object_p;
}
```

Object Definition

The *object* class defines the client visible object. It’s definition is divided among three classes (via inheritance) that define data protected by different locks supporting different levels of mechanism.

The lowest level class, *object_link*, defines the fields that link an object into the object list (friend class *object_list*). These fields concern the basic existence and naming of an object. (A class parameterized by type of name would probably be used in a more general example.) This class defines only a constructor. All manipulations of these fields are made by the *object_list* class and so they are private to the *object_link* class. Since all of these fields concern existence and naming, they are protected by the global lock (in the *object_list* class), another good reason to make them private so derived classes can’t manipulate them (since they don’t reference the global lock).

The *object_base* class defines fields concerned with controlling and synchronizing operations upon objects. These fields are all protected by the object lock, one of the fields.

The *object* class, then, only defines the fields of interest to the client visible object (the object’s value). These fields are also protected by the (inherited as protected) object lock.

```
class object_list;
/*
            forward reference
*/
```

The *object_link* class defines the name–port relationship and the linking of the object in the global list so that a name lookup operation can be done. The name–port relationship includes the make-send count used to determine when the object may be unlinked (no more send rights mean that no one continues to care about the object’s value).

```
class object_link
{
            friend class object_list;
public:
            object_link (int name, object_link* next_object);
private:
/*
All fields are protected by the global lock. They are all private since only friend
class object_list manipulates them.
*/
            object_link*                    o_next;
/*
                                            next in list of objects
*/
            mach_port_t                     o_port;
/*
                                            port representing the object
*/
            mach_port_mscount_t             o_mscount;
/*
                                            make-send count for the port
*/
            int                             o_name;
/*
                                            client ID
*/
};
object_link::object_link (int name, object_link* next_object)
{
            o_port = (mach_port_t) this;
/*
                                            operator new allocated a port with
                                            this name
*/
            o_mscount = 0;
            o_next = next_object;
            o_name = name;
}
```

The *object_base* class encapsulates the operation-in-progress and object synchronization mechanisms. Included is a lock on all of these fields (and fields in derived classes, which is why the lock is protected instead of private). The *wait* and *seqno* fields are used to synchronize the beginning of operations; in particular, to defer the beginning of no-more-senders notifications until all preceding operations have obtained a *reference*. The *alive_-count* field is a count of references to the object that prevent the object from being deleted. (The object can be unlinked whenever the send count is zero; the *object_list* class will do this without obtaining the object lock. The object will not actually be deleted, though, until the global list reference is removed (when the object is unlinked from the global list) and all operations in progress have completed. The alive count records all of these references.)

The methods defined for the *object_base* class (aside from the constructor) add a reference and remove a reference. **synchronize_and_reference** adds a reference; while it is at it, it synchronizes (under protection of the object lock and sequence number fields) with other operations. **is_alive_and_dereference** removes a reference; it also indicates whether there are any remaining references. If there are none, no new references can be added (the object is not in the global list so no client can find it and no operations are in progress) so the object can be deleted.

Although the object is locked for every manipulation thereof (and one of the benefits of the structure here is that each method in *object_base* and its derived class *object* take and release the object lock), the object is not held locked from the time that the message is received and synchronized with others to the time that the reply is generated. This is done here to show the mechanisms when the object is not to be locked for the entire duration. The alternative would be to have a method that performs object synchronization, returning with the object locked, and another method that unlocks the object. (Additional methods would be necessary to handle the global list reference.) Having such external knowledge as to whether the object is locked differs little from the explicit reference manipulation here. One must be careful holding the lock for the operation duration, though; the no-more-senders logic would then need to hold both the global and object locks. Some order would be chosen for these locks (either obtain the global lock before starting no-more-senders processing, or after the synchronization and object locking has been done); this order must be the same for all other such manipulations (of which there are currently none).

```
class object_base : public object_link
{
public:
        object_base (int name, object_link* next_object);
        void synchronize_and_reference (int seqno);
        int is_alive_and_dereference ();
protected:
/*
All fields are protected by the object lock.
*/
        struct mutex                    o_lock;
/*
                                        lock for the object
*/
private:
        struct condition                o_wait;
/*
                                        wait for sequence number
*/
            mach_port_seqno_t           o_seqno;
/*
                                        sequence number for the port
*/
            unsigned int                o_alive_count;
/*
                                        internal refs—one for being in the
                                        object list, one for each
                                        outstanding operation
*/
    };
    object_base::object_base (int name, object_link* next_object)
    : object_link (name, next_object)
    {
            o_alive_count = 1;
/*
                                        One because the object will have a
                                        send right assigned
*/
            o_seqno = 0;
            mutex_init (&o_lock);
            condition_init (&o_wait);
    }
```

The **condition_wait** call below releases the lock (and re-obtains it); this is okay, since before that we increment the alive count to indicate our operation in progress, and we haven't yet incremented the sequence number indicating that we have synchronized with other operations in progress.

```
    void object_base::synchronize_and_reference (int seqno)
    {
            mutex_lock(&o_lock);
            o_alive_count++;
/*
            We wait for any previous requests to get this far. In particular, we can't
            process a no-more-senders notification for a port while some previous
            requests on the port haven't yet translated the port into an object and
            taken a reference for the object, incrementing the alive_count count.
*/
            while (o_seqno < seqno)
                    condition_wait (&o_wait, &o_lock);
            o_seqno++;
            mutex_unlock(&o_lock);
            condition_broadcast(&o_wait);
    }
    int object_base::is_alive_and_dereference ()
    {
            int                         referenced;
            mutex_lock(&o_lock);
            referenced = (0 != ––o_alive_count);
```

```
        mutex_unlock(&o_lock);
        return referenced;
}
```

The *object* class has only the object’s value to define. The only methods are the **query** and **change** operations. The object lock is held while referencing the object value.

```
class object : public object_base
{
public:
        object (int name, object_link* next_object);
        void change (int value);
        int query ();
private:
/*
The following fields are protected by the object lock.
*/
        int                             o_value;
/*
                                        the object’s value
*/
};
object::object (int name, object_link* next_object)
: object_base (name, next_object)
{
        o_value = 0;
}
void object::change (int value)
{
        mutex_lock(&o_lock);
        o_value = value;
        mutex_unlock(&o_lock);
}
int object::query ()
{
        int value;
        mutex_lock(&o_lock);
        value = o_value;
        mutex_unlock(&o_lock);
        return value;
}
```

Object List Definition

The *object_list* class defines the list of object known to the server. Its methods concern the existence of the objects (class *object*).

The **find_or_make** method locates or creates an object. It can do this using just the global lock. If the object exists, the method returns a new send right for the object (using only fields in the *object_link* base class). If the object doesn’t exist, it is created. Note that the object lock isn't necessary in such a case because there can be no references to the object and no other thread can find it (given the holding of the global list lock).

The **unreferenced_object** method is effectively the reverse of **find_or_make**. If successful, it unlinks the object. (Actual deletion of the object cannot be done with only the global lock held—there may be operations in progress.) This method contains the logic to synchronize with the kernel's make-send count logic, so the method will not necessarily unlink the object.

The **port_set** method is used simply to return the value of the port set held by the list naming all object ports.

```
class object_list
{
public:
        object_list ();
        mach_port_t find_or_make (int name);
        int unreferenced_object (object_link* object_p,
                        mach_port_mscount_t mscount);
        mach_port_t port_set ();
private:
/*
The global lock protects the object list, which implies the existence and names
of objects, including the mscount. Each object has its own lock for its private
fields.
*/
        struct mutex                    g_lock;
        object_link*                    g_head;
/*
                                        pointer to first object in list
*/
        mach_port_t                     g_object_ports;
/*
                                        port set to hold all object ports
*/
};
object_list::object_list ()
{
        mutex_init (&g_lock);
        g_head = NULL;
/*
        Allocate a port set from which server threads will receive
*/
        (void) mach_port_allocate(mach_task_self(),
                        MACH_PORT_RIGHT_PORT_SET, &g_object_ports);
}
```

The structure of the objects and the list allows this method to operate purely with the global lock, simply searching the object list for the client supplied name. As mentioned above, it is safe to create a new object here if the desired one is not found, adding its receive right to the port set and starting the no-more-senders notification (race).

The interesting detail here is the use of **mach_port_insert_right** (MACH_PORT_TYPE_MAKE_SEND) and the use of MACH_PORT_TYPE_MOVE_SEND in the MIG definition for the method (**ooserver_lookup**). It would seem that this could be replaced with the use of MACH_PORT_TYPE_MAKE_SEND in the MIG interface definition. This replacement fails, however, in error cases. To see this, consider the race between the time that the global lock is released and the time that the send right would be created via the send of the reply message. During this time a no-more-senders notification could arrive which could delete the object (and the port) thereby causing our reply to use an invalid port (possibly a port which could be reassigned during that time). This is why our make-send count must be incremented here (under protection of the global lock). However, if the reply message (which would have made a send right) cannot be sent (such as because of an invalid reply destination), **mach_msg_server** will destroy the message without a send right having been made (thereby not incrementing the kernel's make-send count, which must be done so that it matches our count). As such, we need a way to guarantee the generation of the send right, by doing it explicitly. The use of MACH_MSG_TYPE_MOVE_SEND in the reply message not only arranges for the kernel to give away the right, but it also cues in **mach_msg_server** to deallocate the right if the message cannot be sent.

```
mach_port_t object_list::find_or_make (int name)
{
        object_link*                    object_p;
        mach_port_t                     port;
        kern_return_t                   kr;
/*
        We reference no fields here protected by the object lock.
*/
        mutex_lock(&g_lock);
        for (object_p = g_head; object_p != NULL; object_p =
                    object_p→o_next)
                if (object_p→o_name == name)
                        break;
        if (object_p == NULL)
        {
                mach_port_t             previous;
/*
                Create a new object and port for it
*/
                object_p = new object (name, g_head);
                g_head = object_p;
                port = (mach_port_t) object_p;
/*
                Put the object port into the port set
*/
                kr = mach_port_move_member(mach_task_self(), port,
                            g_object_ports);
                if (kr != KERN_SUCCESS)
                        quit(1, “ooserver: mach_port_move_member:
                                    %s\n”, mach_error_string(kr));
/*
                Request the initial no-more-senders notification
*/
```

```
                    kr = mach_port_request_notification(mach_task_self(),
                                object_p→o_port,
                                MACH_NOTIFY_NO_SENDERS, 1,
                                object_p→o_port,
                                MACH_MSG_TYPE_MAKE_SEND_ONCE,
                                &previous);
                    if ((kr != KERN_SUCCESS) || (previous !=
                                MACH_PORT_NULL))
                            quit(1, “ooserver:
                                    mach_port_request_notification:
                                    %s\n”, mach_error_string(kr));
            }
            object_p→o_mscount++;
            port = object_p→o_port;
            kr = mach_port_insert_right (mach_task_self(), port, port,
                        MACH_MSG_TYPE_MAKE_SEND);
            if (kr != KERN_SUCCESS)
                    quit(1, “ooserver: mach_port_insert_right: %s\n”,
                                mach_error_string(kr));
            mutex_unlock(&g_lock);
    /*
            It is safe to unlock everything now, with the port value remaining valid,
            even though we have not incremented the alive_count count, because
            no new no-more-senders notification will be generated given our new
            send right, and any pending no-more-senders notification will find our
            make-send count not matching the kernel’s since we have incremented
            our mscount.
    */
            return port;
    }
```

The following method unlinks the object if there really are no more send rights. Object deletion doesn’t occur until **object_demux** finds the last reference removed. If the object is unlinked, TRUE is returned so the caller knows to remove the global list reference (alive count) corresponding to being un-linked. The caller of this method faces no race in removing the global link reference (which requires the object lock) with other threads trying to locate the object (which requires the global lock), even though the caller will not be holding the global lock when it does, because no new messages will be arriving for this object (no send rights exist) and the object is unlinked. An interesting property of the program’s structure is that there is no place in the program where both the global and object locks are held at the same time.

```
    /*
    Possibly delete (unlink) an object.
    */
    int object_list::unreferenced_object (object_link* object_p,
                mach_port_mscount_t mscount)
    {
            int                                 result = FALSE;
            mutex_lock(&g_lock);
            if (object_p→o_mscount == mscount)
            {
```

```
/*
        Since we hold the global lock, no new send rights are being
        generated (that are not reflected in our make-send count).
        Thus, we are safe to mark the object dead and unlink it. There
        may still be other outstanding operations in progress; when
        the last one ends (probably this one), the object will be
        deleted (by object_demux).
*/
        object_link*            prev_object_p;
        if (object_p == g_head)
            g_head = object_p→o_next;
        else
        {
            for (prev_object_p = g_head; prev_object_p !=
                  NULL; prev_object_p =
                  prev_object_p→o_next)
                if (prev_object_p→o_next == object_p)
                    break;
            if (prev_object_p == NULL)
                quit(1, “ooserver: object not in object
                      list\n”);
            prev_object_p→o_next = object_p→o_next;
        }
        (void) mach_port_mod_refs (mach_task_self (),
              object_p→o_port,
              MACH_PORT_RIGHT_RECEIVE, –1);
        object_p→o_port = MACH_PORT_NULL;
        result = TRUE;
/*
                                means object no longer known
*/
    }
    else
    {
        mach_port_t             previous;
        kern_return_t           kr;
/*
        This means a do_ooserver_lookup slipped in and created
        another send right for the object, so we shouldn’t destroy it
        yet. We request another no-more-senders notification instead.
*/
        kr = mach_port_request_notification(mach_task_self(),
              object_p→o_port,
              MACH_NOTIFY_NO_SENDERS,
              object_p→o_mscount, object_p→o_port,
              MACH_MSG_TYPE_MAKE_SEND_ONCE,
              &previous);
        if ((kr != KERN_SUCCESS) || (previous !=
              MACH_PORT_NULL))
            quit(1, “ooserver:
                  mach_port_request_notification:
                  %s\n”, mach_error_string(kr));
    }
    mutex_unlock(&g_lock);
```

```
        return result;
}
/*
Return the port set into which all object ports are placed
*/
mach_port_t object_list::port_set ()
{
        return g_object_ports;
}
```

MIG Target Routines

The following routines are the actual target routines of the MIG generated stubs, after initial object reference processing by **object_demux**. Virtually all work is done by object methods.

```
static object_list*                         all_objects;
/*
        pointer to the global object list
*/
kern_return_t do_object_change (mach_port_t port, int value)
{
        object*                             object_p = (object*) port;
        object_p→change (value);
        return KERN_SUCCESS;
}
kern_return_t do_object_query(mach_port_t port, int *valuep)
{
        object*                             object_p = (object*) port;
        *valuep = object_p→query();
        return KERN_SUCCESS;
}
kern_return_t do_ooserver_lookup(mach_port_t server, int name, mach_port_t
            *portp)
{
        *portp = all_objects→find_or_make (name);
        return KERN_SUCCESS;
}
```

Object Reference Processing

The **object_demux** routine brackets all incoming operations on an object so as to properly maintain the *alive_count* count. Note that a no-more-senders notification, which can be viewed as an operation on the global list, is still object specific and must synchronize with all other operations on an object (to prevent premature unlinking of the object).

```
static boolean_t object_demux(mach_msg_header_t *request,
            mach_msg_header_t *reply)
{
        object*                             object_p;
/*
        Translate the port into an object
*/
        object_p = (object*) request→msgh_local_port;
/*
        It is not necessary to hold the global lock here to maintain the
        existence of the object while we try to lock it. We can’t possibly be
        racing with some other thread trying to delete the object (that is,
        processing a no-more-senders notification) because the strict sequence
        number ordering of these messages (the same port is used for no-more-
        senders as well as client requests) means that no-more-senders will be
        deferred (not started) until the other requests (such as this one) have at
        least started (thereby obtaining a reference).
*/
        object_p→synchronize_and_reference (request→msgh_seqno);
/*
        If this message was sent to a send-once right, then it must be a
        legitimate no-more-senders notification. (We give send-once rights
        only to the kernel.) Otherwise use object_server, passing the object.
*/
        if (MACH_MSGH_BITS_LOCAL(request→msgh_bits) ==
                        MACH_MSG_TYPE_PORT_SEND_ONCE)
        {
                mach_no_senders_notification_t *n =
                                (mach_no_senders_notification_t *) request;
                mig_reply_header_t *        r = (mig_reply_header_t *) reply;
/*
                Handle the notification
*/
                if (all_objects→unreferenced_object (object_p,
                                n→not_count))
                        (void) object_p→is_alive_and_dereference ();
/*
                                        remove global reference
*/
/*
                Tell mach_msg_server not to send a reply. we need this
                because we didn’t use notify_server()
*/
                r→Head.msgh_bits = 0;
                r→Head.msgh_remote_port = MACH_PORT_NULL;
                r→RetCode = KERN_SUCCESS;
        }
        else
                (void) object_server (request, reply);
/*
        Decrement the alive_count count. If we find no references, the object
        has already been unlinked from the object chain, there are no extant
        send rights for it, therefore no one can find it, so we can delete the
        object here (without the global lock).
*/
        if (! object_p→is_alive_and_dereference ())
                delete object_p;
        return TRUE;
}
```

Server Loop and Initialization

The following is basic multi-threaded server loop start-up processing.

```
#define MAX_MSG_SIZE                        512
/*
                                            an upper bound on messages that
                                            we handle
*/
static any_t server_thread (any_t arg)
{
        kern_return_t                       kr;
        kr = mach_msg_server(object_demux, MAX_MSG_SIZE,
                    all_objects→port_set());
        quit(1, “ooserver: mach_msg_server: %s\n”, mach_error_string(kr));
        return 0;
}
int main (int argc, char *argv[])
{
        int                                 i, num_threads;
        mach_port_t                         service;
        kern_return_t                       kr;
        switch (argc)
        {
        case 1:
                num_threads = 1;
                break;
        case 2:
                num_threads = atoi(argv[1]);
                if (num_threads >= 0)
                        break;
        default:
                quit(1, “usage: ooserver [num-threads]\n”);
        }
/*
        Start object list.
*/
        all_objects = new object_list;
/*
        Service port created when object_list was allocated.
*/
        service = (mach_port_t) all_objects;
        kr = netname_check_in(name_server_port, “OOServer”,
                    mach_task_self(), service);
        if (kr != KERN_SUCCESS)
                quit(1, “ooserver: netname_check_in: %s\n”,
                            mach_error_string(kr));
        for (i = 0; i < num_threads; i++)
                cthread_detach(cthread_fork(server_thread, 0));
/*
        We service messages sent to the service port.
*/
        kr = mach_msg_server(ooserver_server, MAX_MSG_SIZE, service);
        quit(1, "ooserver: mach_msg_server: %s\n", mach_error_string(kr));
        return 0;
}
```

Sample Client

This sample client program can directly invoke all defined methods. It does not itself keep track of the rights it accumulates, but it does detect when it releases the last right for the object last located with the *l* request.

```
/*
File: ooclient.c
Author: Richard P. Draves
Sample client for ooserver.
*/
#include <stdio.h>
#include <mach.h>
#include <mach/message.h>
#include <mach_error.h>
#include <servers/netname.h>
#include "ooserver.h"
#include "object.h"
main (int argc, char *argv)
{
        mach_port_t                     server;
        mach_port_t                     object;
        kern_return_t                   kr;
/*
        Lookup the port for the ooservice
*/
        kr = netname_look_up(name_server_port, "", "OOServer", &server);
        if (kr != KERN_SUCCESS)
                quit(1, "ooclient: netname_look_up(OOServer): %s\n",
                            mach_error_string(kr));
/*
        Loop reading and executing commands until eof
*/
        object = MACH_PORT_NULL;
        for (;;)
        {
                char                    command[2];
                int                     value;
                mach_port_urefs_t       object_refs;
                printf ("ooclient> ");
                if (scanf("%1s", command) < 1)
                {
                        printf("\n");
                        break;
                }
                kr = KERN_SUCCESS;
                switch (command[0])
                {
```

```
        case '?':
        case 'h':
            printf (“Commands:\n”);
            printf (“l <num> – lookup object port reference\n”);
            printf (“d – de-allocate object port reference\n”);
            printf (“c <num> – change the value of the
                  object\n”);
            printf (“q – query the value of the object\n”);
            break;
        case 'l':
            if (scanf (“%d”, &value) < 1)
            {
                printf (“syntax error\n”);
                break;
            }
/*
            Lookup a send right for the object port
 */
            kr = ooserver_lookup (server, value, &object);
            if (kr == KERN_SUCCESS)
                printf (“Received a send right for the object
                      port.\n”);
            else
                printf (“ooclient: ooserver_lookup: %s\n”,
                      mach_error_string(kr));
            break;
        case 'd':
/*
            De-allocate a send right for the object port
 */
            kr = mach_port_get_refs (mach_task_self (),
                  object, MACH_PORT_RIGHT_SEND,
                  &object_refs);
            if (kr != KERN_SUCCESS)
            {
                printf (“ooclient: mach_port_get_refs:
                      %s\n”, mach_error_string(kr));
                break;
            }
            kr = mach_port_deallocate(mach_task_self(),
                  object);
            if (kr == KERN_SUCCESS)
            {
                if (object_refs == 1)
                {
                    printf (“De-allocated the last send
                          right for the object
                          port.\n”);
                    object = MACH_PORT_NULL;
                }
                else
```

```
                    printf (“De-allocated a send right
                          for the object
                          port.\n”);
            }
            else
                printf (“ooclient: mach_port_deallocate:
                      %s\n”, mach_error_string(kr));
            break;
        case 'c':
            if (scanf (“%d”, &value) < 1)
            {
                printf (“syntax error\n”);
                break;
            }
/*
            Change the value of the object
*/
            kr = object_change (object, value);
            if (kr == KERN_SUCCESS)
                printf (“Changed the object's value to
                      %d.\n”, value);
            else
                printf (“ooclient: object_change: %s\n”,
                      mach_error_string(kr));
            break;
        case 'q':
/*
            Query the value of the object
*/
            kr = object_query (object, &value);
            if (kr == KERN_SUCCESS)
                printf (“The object's current value is
                      %d.\n”, value);
            else
                printf (“ooclient: object_query: %s\n”,
                      mach_error_string(kr));
            break;
        }
        if (kr != KERN_SUCCESS)
            object = MACH_PORT_NULL;
    }
    exit(0);
}
```

Makefile

```
/*
File: Makefile
Author: Richard P. Draves
Makefile for ooserver.
*/
all : ooserver ooclient
ooserver : ooserver.o ooserverServer.o objectServer.o
```

```
 [3]            $(CC) –o $@ ooserver.o ooserverServer.o objectServer.o –lthreads –lmach –lc++
 [4]    ooserver.o : ooserver.h object.h ooserver.c
 [5]            $(CC_PLUS_PLUS) –c ooserver.c
 [6]    ooclient : ooclient.o ooserverUser.o objectUser.o
 [7]            $(CC) –o $@ ooclient.o ooserverUser.o objectUser.o –lmach
 [8]    ooclient.o : ooserver.h object.h
 [9]    ooserver.h ooserverUser.c ooserverServer.c : ooserver.defs
[10]            mig ooserver.defs
[11]    object.h objectUser.c objectServer.c : object.defs
[12]            mig object.defs
```

CHAPTER 7 External Memory Managers

Perhaps the most novel of Mach's features is its support for external (to the kernel, that is) memory managers. The basic dialog between the memory manager, which maintains the "backing store" representing an abstract memory object, and the kernel, which maintains the physical memory cache (memory cache object) of data from the abstract memory object, is described in the *Kernel Principles* document. This chapter looks at that dialog in more detail, examining a simple example of a memory manager and considering other details that a memory manager must cover.

Overview

Because of the infinite variety of external memory managers that could be written, it is difficult in a document such as this to provide sufficiently general information for those wishing to write an arbitrary manager. In principle, all of the information that one needs appears in the *Kernel Principles* and *Kernel Interfaces* documents. Practice, however, has shown that producing a correct external memory manager is a difficult task.

This chapter discusses in detail the mechanics of basic memory managers, such as might appear in the default memory manager. The initial discussion concerns the basics of the page-in / page-out path, reviewing the kernel interactions and considering the interactions with backing storage that a simple manager would encounter. Of course, not all managers have separate backing store; some might back storage in their own address space (such as a shared memory server) or not back storage at all, if they reconstruct pages when needed.

Eventually the discussion leads to an example manager that supports both a paging and a message interface to a memory object. This simple example illustrates the use of much of the kernel mechanisms and interfaces. In particular, the example raises the issue of maintaining consistency between the page images held by the kernel in its memory object cache versus the page image a manager might hold (or that other kernels might hold). The discussion leading to the chapter's example considers the simple case of maintaining strict read–write consistency between a single manager and a single kernel.

After the example, more advanced topics are discussed, but in less detail. Some ideas are provided for managers that are multi-threaded, although much of these concerns are covered in CHAPTER 4.

The chapter also includes a brief discussion of consistency management between kernels sharing a memory object. A full discussion of this topic (distributed shared memory) is beyond the scope of this text.

Role of a Memory Manager

A memory manager has two responsibilities:

- "Paging" the memory object (supplying data to the kernel to satisfy page-in requests, accepting the (modified) data back for storage to satisfy page-out requests).
- Providing consistency between backing store and the kernel's memory object cache or the memory object caches of multiple kernels.

One would write a memory manager if one had specific requirements for:

- Backing storage management (such as the source of storage or its format, such as compressed or encrypted).
- Consistency management of the backing store versus the memory cache. Examples include mapped file support, transaction-based virtual memory or distributed shared memory.
- Paging mechanisms. The memory manager has direct involvement with the paging mechanisms for the objects it manages. It has indirect influence on overall paging policy by its actions.

This chapter concentrates on the issues of mechanism and consistency. The issue of the management of backing store proper is out of scope for this volume.

Memory Object State

Most of the services of a memory manager concern individual pages or ranges of pages. Certain operations, mostly initialization and termination, concern the memory object as a whole.

Initialization

There are two broad phases of memory object initialization.

A port representing the abstract memory object must be created and made available to clients that wish to **vm_map** the object. There is nothing special about this port relative to port management. The memory manager is free to destroy it when it chooses. Normally, this would not be done until the object is known to not be in use (until all **memory_object_terminate** messages have been received).

The abstract memory object is not known to the kernel until some client performs a **vm_map** operation against the abstract memory object port. This results in a **memory_object_init** message being sent to the memory manager. The manager responds with **memory_object_ready** (or **memory_object_set_attributes** if an old form manager) when ready.

The **memory_object_init** message carries a send right to the memory cache name port. This port serves a single purpose: it is the port returned by **vm_region** in an attempt to provide the identity of virtual memory regions. This port is normally discarded once received.

The memory cache control port is the important value in the **memory_object_init** message. The manager saves this port as the identification of the object being manipulated by future kernel requests. All kernel–-to–manager messages (except **memory_object_terminate**) provide a send right to this memory cache control port. These extra send rights accumulate in the manager which must handle and dispose of them appropriately.

It is common practice to merely count these accumulated rights and to de-allocate them in bunches, thereby avoiding a (common) extra kernel interaction.

Termination

At some future time, unknown to the manager, all clients will **vm_deallocate** their accesses to the object. This will cause a **memory_object_terminate** message to be sent to the manager.

Alternately, the manager can invoke **memory_object_destroy** to destroy the object; the kernel’s response is the same (**memory_object_terminate**). If the manager de-allocates the abstract memory object port (as would happen if it died), the result is the same as **memory_object_destroy**. Destroying the object discards its pages. An explicit lock request (flush) sequence should be used if the pages are desired.

There is a race between the delivery of the **memory_object_terminate** message and manager invoked operations upon the object. That is, the kernel can send the terminate message (and disavow any knowledge of the object) before the manager realizes this and is therefore still trying to manipulate the object. For this reason, the **memory_object_terminate** message includes the receive right to the memory cache control port (as well as the memory cache name port). When the manager receives the memory cache control port, it should drain any messages from this port, if they carry valuable information. This would be the case if *precious* pages could have been supplied to the kernel during this race.

It does not follow that when a memory object is terminated that no clients hold rights to the abstract memory object port. Indeed, the **memory_object_terminate** message may be followed by a subsequent **memory_object_init** message. (Actually, it is currently the case that the subsequent initialization message can precede the termination message, but this is scheduled to be corrected in the future.) Of course, when multiple kernels are involved, multiple initialization messages can be received from the various kernels.

Page-in and Page-out

The basic kernel page-in request and kernel page-out eviction path for a page is fairly simple. What complicates the situation is the asynchronous nature of this path and its relationship to the (typically asynchronous) backing store update path.

In-Transit Pages

The various states of a page (as visible to the memory manager) for the basic page-in / page-out path are shown in FIGURE 1.

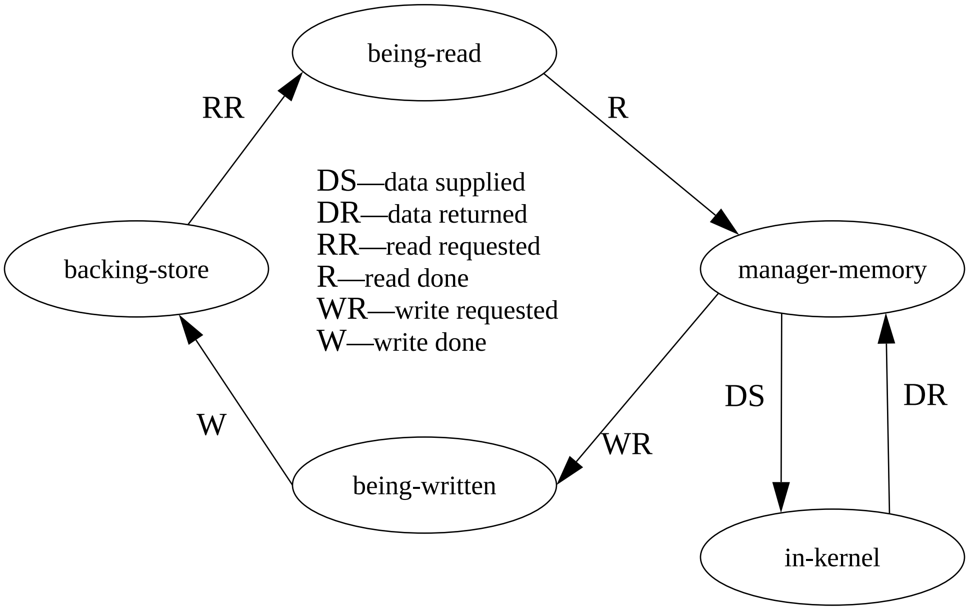


FIGURE 1 **Basic Page-in / Page-out Path**

The kernel makes a variety of guarantees involving its protocol:

- No page-in requests (**memory_object_data_request**) will be made for a resident page.
- No page-in request will be made for a page with an outstanding (unsatisfied) page-in request.

- Page-out requests (**memory_object_data_return** (new form) or **memory_object_data_write** (old form)) will precede a subsequent page-in request for the same page.

The page-in path by itself presents no special difficulty. The kernel will request a page (**memory_object_data_request**). The manager will request the page from its backing store; when the page is received it can be supplied to the kernel (**memory_object_data_supply** (new form) or **memory_object_data_provided** (old form)). When a page is evicted, it will be supplied by the kernel to the manager (**memory_object_data_return** (new form) or **memory_object_data_write** (old form)). The manager will write the page out to its backing store.

The difficulty presented is caused by the relationship between the page-out and page-in paths. The kernel may request a page that it has just evicted. That is, since the kernel has no knowledge or understanding of the manager’s interaction with its backing store, the kernel may request a page that is in transit to the manager’s backing store. This means that the manager cannot just blindly send pages (asynchronously) to its backing store and forget about them; the manager must keep track of outstanding backing store activities so that, if the kernel should request a page on transit to the backing store, the manager can either supply a copy it is still holding or the manager will wait for the page to be known to have to reached the backing store (a reply was received) before requesting the page again from backing store.

Page “Stealing”

Whenever a page is sent to the kernel in a **memory_object_data_supply** or **memory_object_data_provided** message or sent to the manager in a **memory_object_data_return** or **memory_object_data_write** message (or in interactions with backing store, most likely), it is logically copied. Performance is clearly improved if the page is not physically copied. A page can be *stolen* (that is, directly moved (mapped) from one place to another) if three conditions hold: the page is in memory, it is not shared and if it is de-allocated from its source as it is sent in the message (de-allocate flag set). The kernel transfers pages to the manager in this way. **memory_object_data_supply** allows the manager to specify the de-allocation of the pages it is sending.

Unfortunately, Mach does not track uses of individual pages, and therefore cannot tell whether an individual page is shared or not. The page stealing code relies on the object reference count that records all mappings or other uses of the object. If the count is one, then the object is not shared, and if the object is shared, then the count will be at least two. False page sharing will be seen when disjoint ranges of an object are mapped. Managers must therefore avoid creating such mappings (e.g., by de-allocating memory in the middle of the object). Two possible techniques are:

- Creating a separate (buffer) region of memory and using it sequentially
- Passing memory obtained directly from the Mach device interface (which returns data in newly created memory objects).

Strict Kernel and Manager Page Consistency

The external memory management interactions may involve multiple pages, but they always deal with whole pages. Because these interactions are asynchronous, it is not, in general, possible for the memory manager to know precisely the state of the pages at any given time. That is, a page may be in-transit (in a message) or in the process of being manipulated by the kernel (or the hardware).

A simple memory manager such as the default manager is not concerned with this class of difficulty. This manager does not synchronize with kernel manipulations (actually client manipulations) of the pages. The manager deals with one kernel; if that kernel requests a page, the manager knows its contents are the current contents; if that kernel returns the page, the result must be the latest contents. No other kernel interactions are meaningful.

Some memory managers, however, need to manipulate their pages at times other than when the kernel voluntarily gives up access. Possible reasons for this are:

- The manager supports a message interface to access the memory object.
- Like the network shared memory server, the manager needs to provide the latest contents of the page to another node.
- The manager is providing transaction semantics for the pages and needs their value when a client “commits” them.

These memory managers need to synchronize with kernel manipulations of the pages.

There are a variety of consistency policies that a memory manager may provide with respect to the view that it and its multiple clients (or kernels) have to a given memory object. Given sufficient higher level information about client accesses to memory, advanced consistency protocols are possible.

The most typical consistency policy, that supplied by the default memory manager and the XMM library, is full read–write consistency. This policy mimics standard hardware’s shared memory policy—once a client modifies a memory object, those changes are (effectively) immediately visible to all clients referencing the memory object.

Clients on any given node that share a memory object (**vm_map** the same abstract memory object port) automatically receive full read–write consistency. It may seem that a memory manager could enjoy this same level of consistency with its clients simply by **vm_map**-ping the object itself. A memory manager does not typically do this, though, for two reasons:

- A manager can only share (physical) pages with the kernel on which it is executing, which is not necessarily the ones on which its clients are executing.
- Direct manager reference to its own memory objects often leads to deadlock. Any reference at all to the contents of a memory object must be viewed as a potential interaction with its manager, since the kernel can evict pages at any time. Therefore, if the manager were busy processing some request when it went to touch an object it maintains, the result is a circular dependency. If the pager has only one thread handling re-

quests for this memory object, it will deadlock. Even if it has multiple threads to support this object, the pager is most likely holding internal locks on the page in question when it tries to reference it, again leading to deadlock.

When multiple kernels are involved, the situation becomes even more involved.

Since the various kernels referencing a shared memory object, and the manager managing it, cannot directly share the memory, it is necessary for these components to interact to implement their consistency policy.

One way to think about these interactions, in the specific case of full read–write consistency, is to consider the various states of *ownership* of a page—which entities hold the current (valid) page contents—and the transitions between these states.

The following sections discuss maintaining full read–write consistency between a manager and a single kernel. Issues involved when there are multiple kernels are discussed at the end of this chapter.

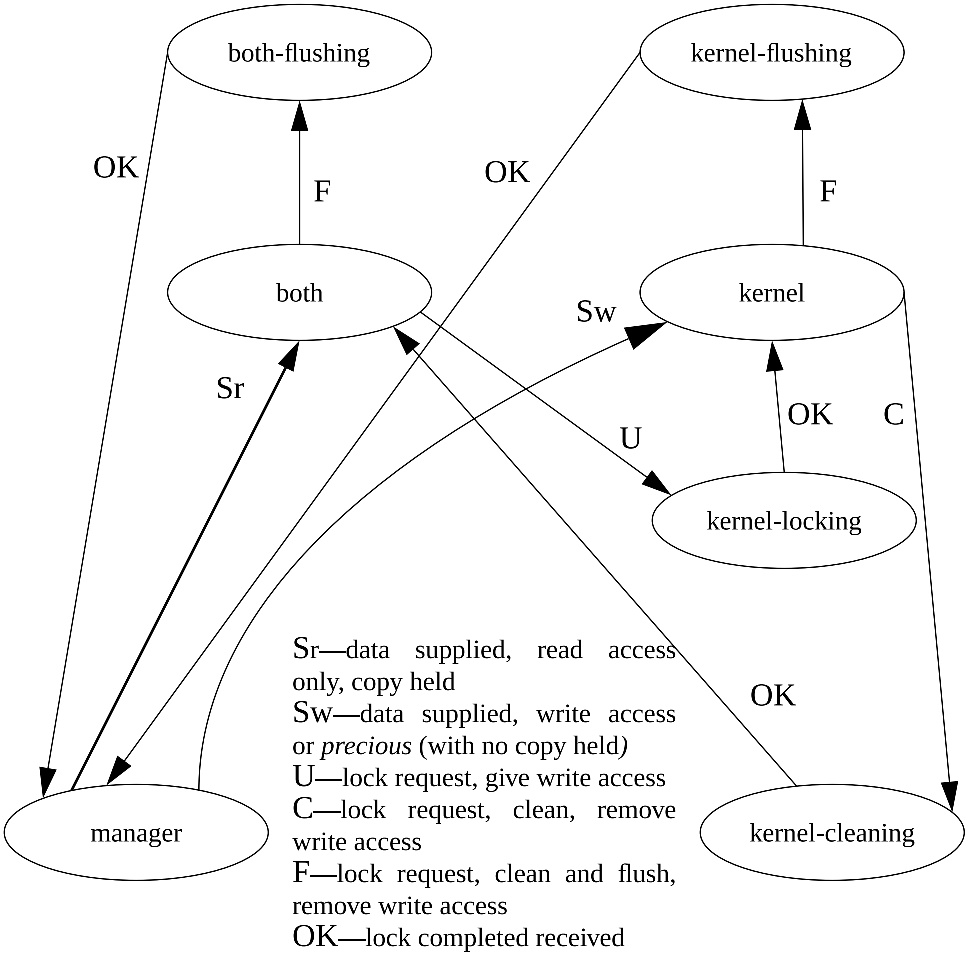
Page “Ownership”

The memory manager must deal with the state of a page in broad terms having to do with the potential *ownership* of the page (that is, who holds a valid copy) and any operation that the memory manager has requested upon the page (but is not yet known to have completed). (Actually, the situation can be even more complicated if the memory manager has sent multiple requests to the kernel to manipulate a single page without waiting for the kernel’s responses. This section takes the simplifying assumption that only one transition has been requested at a time.) These *ownership* states are shown in FIGURE 2 from the point of view of the memory manager.

There are three general classes of ownership:

- The memory manager holds the only valid copy of the page. This copy may reside in the address space of the manager, or be placed on some backing storage. The memory manager is free to modify the page’s contents as it sees fit.
- Both the kernel and the memory manager hold valid copies of the page’s contents. Neither one of them is allowed to modify the page.
- The kernel holds the only valid copy of the page because it (actually, clients referencing the virtual page) has the ability to modify the page. In these states it is not known whether or not the kernel holds modified page contents, or whether the kernel holds the page at all. (It may have discarded the page without modification.) What is known is that the manager’s copy of the page may be stale. In these states, if the current contents of the page are desired, it is necessary to interact with the kernel so that it can return its copy, or return the fact that it does not hold a copy (and therefore the manager’s copy is valid).

  This is also the state for *precious* pages that have been supplied to the kernel and for which the manager has not kept a copy. In this case, it is known that the kernel holds a copy, which may or may not be undergoing modification depending on the access set. Interaction with the kernel is necessary for the manager to hold a current copy.



Sr—data supplied, read access only, copy held
Sw—data supplied, write access or *precious* (with no copy held*)*
U—lock request, give write access
C—lock request, clean, remove write access
F—lock request, clean and flush, remove write access
OK—lock completed received

FIGURE 2 **Strict Consistency Page “Ownership” Transitions**

Ownership Transitions

There are two operations a manager can invoke to affect the ownership of a page: **memory_object_data_supply / memory_object_data_provided** (or **memory_object_data_unavailable**) and **memory_object_lock_request**.

- **memory_object_data_supply / memory_object_data_provided** must be considered to affect the ownership of the page as soon as the message is sent since it cannot be known when the kernel will receive (and possibly allow modifications to) the page.
- **memory_object_lock_request** can cause a variety of ownership changes to occur. This operation produces a reply for which the manager must be prepared. Whether or not the ownership is viewed as changed when the message is sent or when the reply is received depends on the operation being performed.

These operations affect merely the validity of the manager's or the kernel's page contents. Other operations (such as controlling execute access) are omitted for simplicity since they do not affect page ownership.

The various transitions in FIGURE 2 are detailed below. Pages start in the *manager* state; that is, the manager holds the (only) valid copy. (If the page starts out with no data, **memory_object_data_unavailable** can be used instead to cause the kernel to generate an empty page instead of the manager having to create an empty page to send in the **memory_object_data_provided** message, but this can be viewed as an optimization.)

- *manager→kernel*: This transition can be caused by two events:
    - The manager sends a *precious* page via a **memory_object_data_supply** message (in response to a previous **memory_object_data_request** message) and the manager does not keep a copy of the page (which is the basic reason why the page is supplied as *precious*). The kernel is viewed as holding the only copy of the page as soon as it is sent. It does not matter whether the page is given write access; the kernel will return this page to the manager, modified or not, and interaction with the kernel is necessary if the manager needs to see the page.
    - The manager sends a **memory_object_data_supply** message (in response to a previous **memory_object_data_request** message) that specifies write access. The kernel must be viewed as holding the only current copy of the page as soon as it is sent. The manager must always assume that the kernel holds the current page contents, even though the kernel may discard them at any time. Only via lock request interactions with the kernel can the manager know whether the copy it saves is the current page contents.
- *manager→both*: The manager sends a **memory_object_data_supply** message (in response to a previous **memory_object_data_request** message) that specifies only read access. The manager keeps a copy of the page. The kernel must be viewed as holding a copy of the page as soon as this message is sent. Both the manager and the kernel hold current page contents, but neither is allowed to modify them since the changes would not be reflected in the other's copy.
- *kernel→kernel-flushing→manager*: The manager sends a **memory_object_lock_request** (flush) to the kernel. When (but not until) the kernel responds (possibly with **memory_object_data_return** and ending with **memory_object_lock_completed**), it is known that the manager has the sole copy of the page.
- *kernel→kernel-cleaning→both*: The manager sends a **memory_object_lock_request** (clean) to the kernel. When (but not until) the kernel responds (possibly with **memory_object_data_return** and ending with **memory_object_lock_completed**), both the manager and the kernel will have valid read-only copies. (Of course, the kernel may discard its copy at any time.) Notice that this sequence races with the basic kernel eviction mechanisms. That is, the kernel may have already sent the (modified) page via **memory_object_data_return** before the manager even started this sequence. As such, when **memory_object_lock_request** are being used, it is not possibly to distinguish between a **memory_object_data_return** that means that the kernel is providing the current page contents but still keeping a copy versus giving up the page all together. (**memory_object_data_return** does return that information, but the kernel can discard its copy immediately afterward, unless the page is *precious*.) This

is why **memory_object_data_return** is not shown as a meaningful ownership transition in FIGURE 2. It does follow, though, that the data returned via **memory_object_-data_return** is more current than any data the manager holds. If no **memory_object_lock_request** operations are ever performed by the manager, then receiving a **memory_object_data_return** message does indeed signify transition from *kernel*-owned to *manager*-owned. This observation doesn't help much, though, since it only applies to pages that are indeed modified by the kernel. Non-modified (and non-*precious*) pages are simply discarded by the page eviction mechanisms, and so **memory_object_lock_request** is the only way to be sure that the kernel does not hold a copy of the page. Of course, if the manager doesn't care what the ownership is (the manager never examines the data itself), this is not a concern.

- *both→both-flushing→manager*: The manager sends a **memory_object_lock_request** (flush) to the kernel. When (but not until) the kernel responds (possibly with **memory_object_data_return** and ending with **memory_object_lock_completed**) it is known that the manager has the sole copy of the page.
- *both→kernel-locking→kernel*: The manager sends a **memory_object_lock_request** (unlock) to the kernel. As soon as this is sent, the kernel must be viewed as being able to modify the page, and therefore owns it. The **memory_object_lock_completed** message merely confirms the action.

Preserving Operation Order During Lock Sequences

Some of these transitions require receiving a **memory_object_lock_completed** message. If the memory manager must serialize its operations upon a given page (which it typically must do), then it has to wait for this message, and defer processing other operations upon this page until it does receive the message. Once a **memory_object_lock_-request** message has been sent to the kernel for a page, the following messages may be subsequently received for that page:

- Direct client service requests. Of course, clients can always make direct requests of the memory manager at any time.
- **memory_object_data_request**. It is possible that the kernel does not hold the page in question (it discarded it previously) and now, by coincidence, it is requesting the page to satisfy some client's page fault. The kernel is clearly prepared to wait for this page until the manager completes its lock processing. The manager must not forget that the kernel is waiting for the page; the kernel never multiply issues page requests.
- **memory_object_data_unlock**. The kernel is requesting more access for the page. This will have to wait until lock processing is completed. If the manager's lock request involves cleaning, then the kernel will still need to be granted greater access when the lock sequence is completed. If the manager's lock request is a flush operation, though, the kernel will cease to hold the page and so granting greater access will have no effect. Indeed, the kernel, after flushing the page, will proceed to request the page back (with the greater access).
- **memory_object_data_return**. Either the kernel is supplying the requested page as a result of our lock processing or it is randomly evicting the page. Either way, this sent page is more current than any copy the manager holds.
- **memory_object_terminate**. The manager lost the race with the un-mapping of the object by clients. It follows that any data to be returned from the kernel has been al-

ready, so the manager has the latest page copy. Any lock processing is aborted since the kernel clearly did not see the manager's lock request.

Simple Memory Manager Example

The management of page consistency will be shown with a simple example of a memory manager. This memory manager can be viewed as the basis for a network shared memory manager: it supports paging operations to a kernel but it also supports direct requests upon the memory requiring the use of direct cache management primitives.

Various simplifications are made for this manager:

- The manager supports a single memory object consisting of a single page.
- At most one kernel has the object mapped.
- The manager is single threaded.
- The message-based memory references access only one integer.

Overview

The **netmem** manager consists of the following files:

- **Makefile**—Build commands
- **netmem_defs**.h—Definitions exported to clients
- **netmem_msg**.defs—MIG definitions of client–server messages
- **netmem_msg_reply**.defs—MIG definitions of manager reply to client messages (discussed below)
- **netmem_obj**.h—Internal definitions
- **netmem_pager**.c—Memory manager code itself

There is also a simple test client program (**netmem_client**.c).

Most of the structure of the memory manager can be derived from the sections above. The only real complication in this manager results from the direct client message interface, resulting in the page consistency protocol described earlier. This protocol requires that the pager be able to defer processing of some incoming requests to serialize all requests for the page. A queue of pending operations is maintained.

External Definitions

```
/*
    File: netmem_defs.h
    Author: David L. Black
    External definitions for netmem_pager.
*/
#include <mach/error.h>
#define NETMEM_NAME                    “netmem_pager”
```

```
 /*
                                    Name of the pager for the
                                    nameservice.
 */
 /*
      Failure codes. Use user-reserved errors.
 */
 [3]  #define NETMEM_BAD_OBJECT     (err_local | err_sub (13)|1)
 [4]  #define NETMEM_BAD_OFFSET     (err_local | err_sub (13)|2)
 [5]  #define NETMEM_FAILURE        (err_local | err_sub (13)|3)
```

Client Request Definitions

Special MIG definitions are required for client messages because there can be incoming requests that will not be processed and therefore will not have a reply before returning to the server loop to obtain the next message.

The request definitions must explicitly specify *sreplyport* options for the reply port for the message (lines [7] and [14]) so that the reply port becomes an explicit argument for the call to the server routines. The server will need to save the reply port if it needs to defer the request.

The requests consist of two simple requests: **netmem_read** and **netmem_write** which each take an object offset and get or set, respectively, the integer at that offset.

```
  /*
       File: netmem_msg.defs
       Author: David L. Black
       MIG interface for messages to simple netmemory server
  */
  [1]  subsystem netmem_msg                417200;
  [2]  #include <mach/std_types.defs>
  [3]  #include <mach/mach_types.defs>
  /*
       Read data from the object
  */
  [4]  routine netmem_read
  [5]  (
  [6]          object                      : mach_port_t;
  [7]          sreplyport reply_to         : mach_port_make_send_once_t;
  [8]          offset                      : vm_address_t;
  [9]          out value                   : int
 [10]  );
  /*
       Write data to the object
  */
 [11]  routine netmem_write
 [12]  (
 [13]          object                      : mach_port_t;
 [14]          sreplyport reply_to         : mach_port_make_send_once_t;
 [15]          offset                      : vm_address_t;
 [16]          value                       : int
 [17]  );
```

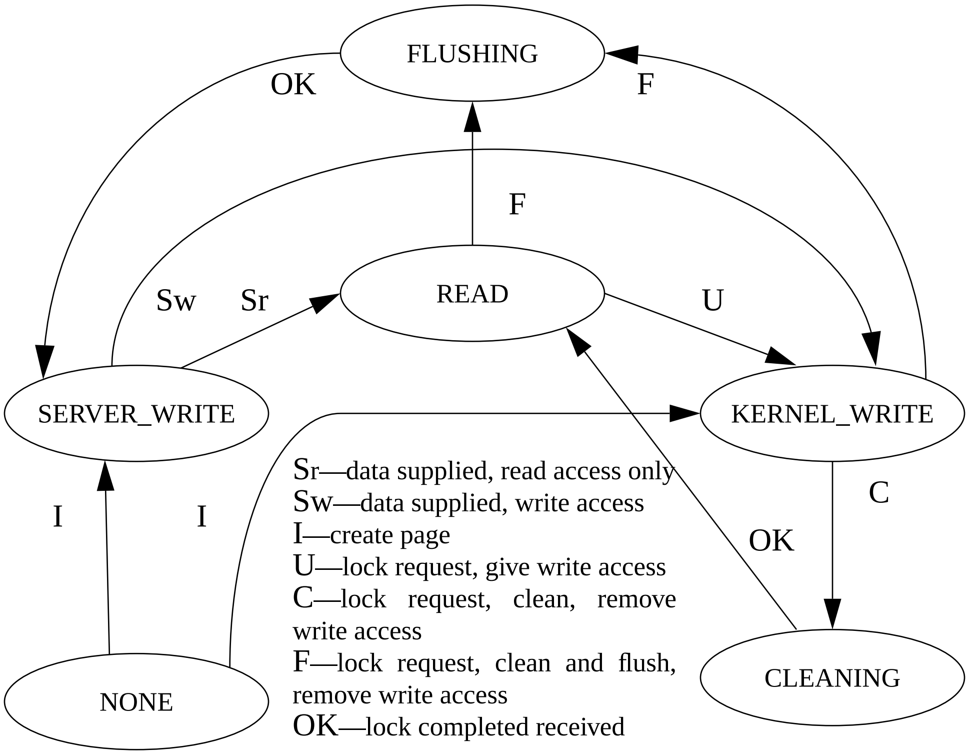
It is necessary to separately define the reply messages that will be explicitly generated when these deferred requests are processed (lines [6] and [12]). Note that the subsystem ID is 100 more than that in the request definitions which is Mach convention. In general, most processing routines will perform their processing when called, returning an appropriate success or failure return to the MIG generated server stubs. However, when a request is to be deferred, MIG_NO_REPLY is returned to the stub which prevents a reply from being sent. Replies are explicitly sent via these MIG generated reply stubs (lines [73] and [81] in **pending_op_execute**).

```
  /*
       File: netmem_msg_reply.defs
       Author: David L. Black
       MIG interface for reply messages from simple netmemory server
  */
 [1]   subsystem netmem_msg_reply                    417300;
 [2]   msgoption MACH_SEND_TIMEOUT;
  /*
               Protect against full reply ports.
  */
 [3]   #include <mach/std_types.defs>
 [4]   #include <mach/mach_types.defs>
 [5]   type reply_port_t = MACH_MSG_TYPE_MOVE_SEND_ONCE ctype:
                      mach_port_t;
  /*
       Reply routine to complete a read operation. Return the result of the operation
       and the value read.
  */
 [6]   simpleroutine netmem_read_reply
 [7]   (
 [8]           reply_port                            : reply_port_t;
 [9]           result                                : kern_return_t;
[10]           value                                 : int
[11]   );
  /*
       Reply routine to compete a write operation.
  */
[12]   simpleroutine netmem_write_reply
[13]   (
[14]           reply_port                            : reply_port_t;
[15]           result                                : kern_return_t
[16]   );
```

Internal Definitions

This file defines the internal memory object structure (line [21]). The object state consists of the memory object cache port reference, a right count (discussed later) and a reference to the page data. Since the object consists of only one page, the page information can be kept in the same structure (lines [26] and on). The page information consists of the “backing store” for the page (space obtained from **vm_allocate**), the *ownership* state and the queue of pending operations.

There are six states defined for the page as shown in FIGURE 3. Various simplifications from FIGURE 2 are possible because of the nature of the manager.



Sr—data supplied, read access only
Sw—data supplied, write access
I—create page
U—lock request, give write access
C—lock request, clean, remove write access
F—lock request, clean and flush, remove write access
OK—lock completed received

FIGURE 3 **Example Program Page State Transitions**

- NETMEM_OBJ_NONE—No memory exists for the object. This is the initial state. As soon as any request for the page is made, a page of memory in the server will be allocated. This extra state is added simply to allow the use of **memory_object_data_unavailable** when the page is first requested (line [214]).
- NETMEM_OBJ_READ—This corresponds to the *both-reading* state in FIGURE 2.
- NETMEM_OBJ_SERVER_WRITE—This corresponds to the *manager-writing* state in FIGURE 2.
- NETMEM_OBJ_KERNEL_WRITE—This corresponds to the *kernel-writing* state in FIGURE 2.
- NETMEM_OBJ_CLEANING—This corresponds to the *kernel-cleaning* state in FIGURE 2.
- NETMEM_OBJ_FLUSHING—This corresponds to either the *both-flushing* or the *kernel-flushing* states in FIGURE 2. It is not necessary to differentiate between these states in this example because no other operations will be performed upon the page while it is in either state.

The *kernel-locking* state is not needed because the manager does not wait for (nor request) a reply from that lock request.

The only possible pending kernel operations are data unlock and data request. A data unlock will never follow a data request, so the structure need only remember one operation (the last), line [30] (and the access so requested, line [29]). Client requests, however, can stack up arbitrarily, so a queue is needed for them (line [28]). Client pending operations (line [10]) must remember all of the parameters from the client's message, including the reply port.

```
  /*
       File: netmem_obj.h
       Author: David L. Black
       Internal definitions for netmem_msg server.
  */
 [1]   #include <mach.h>
  /*
       Possible object states
  */
 [2]   #define NETMEM_OBJ_NONE                1
  /*
                                              No memory exists for the object
  */
 [3]   #define NETMEM_OBJ_READ                2
  /*
                                              Server has read only copy, kernel
                                              may have read only copy
  */
 [4]   #define NETMEM_OBJ_SERVER_WRITE        3
  /*
                                              Server has writable copy, no kernel
                                              copy
  */
 [5]   #define NETMEM_OBJ_KERNEL_WRITE        4
  /*
                                              Kernel may have writable copy,
                                              server has no-access copy
  */
 [6]   #define NETMEM_OBJ_CLEANING            5
  /*
                                              Pending request to kernel to clean
                                              page and set read-only
  */
 [7]   #define NETMEM_OBJ_FLUSHING            6
  /*
                                              Pending request to kernel to clean
                                              and flush page
  */
  /*
       Structure for a pending message operation
  */
 [8]   #define PENDING_READ                   1
 [9]   #define PENDING_WRITE                  2
[10]   struct pending_op
[11]   {
[12]             struct pending_op            *next;
[13]             int                          op;
  /*
                                              read or write
  */
[14]             vm_address_t                 offset;
```

```
            int                         value;
            mach_port_t                 reply_port;
};
typedef struct pending_op               *pending_op_t;
typedef struct pending_op               pending_op_data_t;
#define PENDING_OP_NULL                 ((pending_op_t) 0)
/*
Object data structure
*/
struct netmem_obj
{
            mach_port_t                 object_port;
/*
                                        represents object
*/
            mach_port_t                 control_port;
/*
                                        kernel control port
*/
            int                         send_rights;
/*
                                        control send right count
*/
            int                         state;
/*
                                        who can access how
*/
            vm_address_t                data;
/*
                                        pointer to contents
*/
/*
            Next three fields used only in cleaning and flushing states.
*/
            pending_op_t                pending_op_queue;
/*
                                        pending msg ops
*/
            vm_prot_t                   pending_kernel_access;
/*
                                        pending kernel op
*/
            int                         pending_kernel_type;
/*
                                        which kernel op
*/
#define PENDING_TYPE_LOCK               1
#define PENDING_TYPE_DATA               2
};
typedef struct netmem_obj               *netmem_obj_t;
typedef struct netmem_obj               netmem_obj_data_t;
#define NETMEM_OBJ_NULL                 ((netmem_obj_t) 0)
```

External Memory Manager

The source for the manager itself follows.

The one and only memory object is declared in line [11].

```
    /*
         File: netmem_pager.c
         Author: David L. Black
         The actual netmem_msg pager. This is a simple pager that provides one page of
         memory, accessible either via mapping or a message based interface. It only
         works with one kernel.
    */
 [1] #include <mach.h>
 [2] #include <mach_init.h>
 [3] #include <mig_errors.h>
 [4] #include <servers/netname.h>
 [5] #include <stdio.h>
 [6] #include <mach/message.h>
 [7] #include “netmem_defs.h”
 [8] #include “netmem_obj.h”
 [9] #include “netmem_msg.h”
[10] #include “netmem_msg_reply.h”
[11] netmem_obj_data_t                    master_obj;
```

Initialization

The immediately following **init** routine puts the object in the obvious initial state. Most of the routine is concerned with logically creating the memory object and making its presence known to the name server. Of course, the object isn’t really known to the kernel until the **memory_object_init** sequence. Note in the **memory_object_ready** call (line [175]) that the MEMORY_OBJECT_COPY_NONE strategy is used since the manager does, at times, modify the object itself. (Refer to the discussion in the *Kernel Principles* document.)

```
    /*
         Initialization routine for the server.
    */
[12] void init ()
[13] {
[14]         kern_return_t                r;
    /*
             Initialize the object. This had better work.
    */
[15]         r = mach_port_allocate (mach_task_self (),
                         MACH_PORT_RIGHT_RECEIVE,
                         &master_obj.object_port);
[16]         if (r != KERN_SUCCESS)
[17]         {
[18]                 printf (“Can’t allocate object/service port\n”);
[19]                 exit (–1);
[20]         }
[21]         master_obj.control_port = MACH_PORT_NULL;
```

```
        master_obj.state = NETMEM_OBJ_NONE;
        master_obj.data = (vm_address_t) 0;
        master_obj.pending_op_queue = PENDING_OP_NULL;
        master_obj.pending_kernel_access = VM_PROT_NONE;
/*
        Now check it into the name service. No signature port means this is
        unprotected.
*/
        if (r != NETNAME_SUCCESS)
        {
                mach_error ("Check In:", r);
                exit (–1);
        }
}
```

Port Manipulation

netmem_lookup is a typical routine needed to translate between the abstract memory object port over which messages come into the memory object control structure that describes that object. With only one object, this conversion is trivial here.

The routine **netmem_control_cleanup** is called whenever a message is received from the kernel. Since all such messages carry a send right to the memory cache control port, these accumulate. Instead of de-allocating them one at a time as they come in, they are left to add up, and (almost all) deleted periodically.

```
/*
Conversion routine from port to object. Very simple because there's only one
object.
*/
netmem_obj_t netmem_lookup (memory_object_t port)
{
        if (port != master_obj.object_port)
        {
                printf ("object mismatch\n");
                return (NETMEM_OBJ_NULL);
/*
                                        Can't happen
*/
        }
        return (&master_obj);
}
/*
Routine for lazy discarding of send rights to control port. Add one more right to
the number we have. If over limit, get rid of all but one. We retain one right to
ensure that the port's name doesn't change.
*/
#define SEND_RIGHT_MAX                  10000
void netmem_control_cleanup (netmem_obj_t obj)
{
        obj→send_rights++;
        if (obj→send_rights > SEND_RIGHT_MAX)
        {
                (void) mach_port_mod_refs (mach_task_self (),
                        obj→control_port,
                        MACH_PORT_RIGHT_SEND, –
                        (SEND_RIGHT_MAX));
                obj→send_rights –= SEND_RIGHT_MAX;
        }
}
```

Deferred Operation Processing

```
/*
Routines for handling pending operations.
*/
void pending_op_queue (netmem_obj_t obj, int op, vm_address_t offset, int
            value, mach_port_t reply_to)
{
        pending_op_t                    pend, *pend_ptr;
/*
        Using malloc () here is not particularly efficient.
*/
        pend = (pending_op_t) malloc (sizeof (pending_op_data_t));
/*
        Initialize pending op and add to queue. This results in LIFO pending
        operation order.
*/
        pend→op = op;
        pend→offset = offset;
        pend→value = value;
        pend→reply_port = reply_to;
        pend→next = obj→pending_op_queue;
        obj→pending_op_queue = pend;
}
```

The routine **pending_op_execute** performs the processing deferred by **pending_op_queue**. It is called only when a **memory_object_lock_completed** message is received from the kernel. As such, the page is either in the NETMEM_OBJ_READ or NETMEM_OBJ_SERVER_WRITE states, namely, the manager has a valid copy. This routine can then execute all pending read requests. It can also execute all pending write requests if in the NETMEM_OBJ_SERVER_WRITE state. If not in this state, the manager must start a new lock request sequence to enter this state, so the writes (and subsequent reads) must continue to be deferred.

```
/*
Execute operations from pending op queue. This routine checks the object state
to find out what is allowed, so any state transition must occur BEFORE calling
this routine. Processing stops when the queue is empty or a write is found that
can't be executed because the server only has read permission. Caller is
responsible for handling pending kernel access.
*/
void pending_op_execute (netmem_obj_t obj)
{
        pending_op_t                    pend, new_pend;
        kern_return_t                   result;
        pend = obj→pending_op_queue;
[67]        while (pend != PENDING_OP_NULL)
[68]        {
[69]                if (pend→op == PENDING_READ)
[70]                {
  /*
                        Can always do read. Must send reply ourselves.
  */
[71]                        result = netmem_read (obj→object_port,
                                        MACH_PORT_NULL, pend→offset,
                                        &pend→value);
[72]                        printf (“read_reply\n”);
[73]                        result = netmem_read_reply (pend→reply_port,
                                        result, pend→value);
[74]                }
[75]                else
[76]                {
  /*
                        Write case. Stop if read only.
  */
[77]                        if (obj→state == NETMEM_OBJ_READ)
[78]                                break;
[79]                        result = netmem_write (obj→object_port,
                                        MACH_PORT_NULL, pend→offset,
                                        pend→value);
[80]                        printf (“write_reply\n”);
[81]                        result = netmem_write_reply (pend→reply_port,
                                        result);
[82]                }
  /*
                Reply might fail if client died. INVALID_DEST is the only
                possible error.
  */
[83]                if (result == MACH_SEND_INVALID_DEST)
[84]                        mach_port_deallocate (mach_task_self (),
                                        pend→reply_port);
  /*
                Onward to next pending operation.
  */
[85]                new_pend = pend→next;
[86]                free (pend);
[87]                pend = new_pend;
[88]        }
  /*
        Reset pending op queue. This sets it to null if we ran it completely, else
        to the first write that stopped us.
  */
[89]        obj→pending_op_queue = pend;
[90]    }
```

Client Request Processing

The **netmem_read** and **netmem_write** routines handle read and write requests from the clients. The page state determines whether the operation can be done directly (lines [106] or [135]) or must be deferred (lines [114] or [144]). If the operation is to be deferred but there is no outstanding request for the page (no lock request is pending), then a lock request is made.

```
/*
Routines for handling messages from clients
*/
mach_error_t netmem_read (mach_port_t object, mach_port_t reply_to,
                vm_address_t offset, int *value)
{
        netmem_obj_t                    my_obj;
        printf (“read\n”);
/*
        Check arguments.
*/
        if ((my_obj = netmem_lookup (object)) == NETMEM_OBJ_NULL)
                return (NETMEM_BAD_OBJECT);
        if (offset < 0 || offset >= (vm_page_size / sizeof (int)))
                return (NETMEM_BAD_OFFSET);
/*
        Execute the read.
*/
        switch (my_obj→state)
        {
        case NETMEM_OBJ_NONE:
/*
                Need to initialize. Server has exclusive access (write) when
                we’re done.
*/
                (void) vm_allocate (mach_task_self (), &my_obj→data,
                            vm_page_size, TRUE);
                my_obj→state = NETMEM_OBJ_SERVER_WRITE;
/*
                                        Fall through...
*/
        case NETMEM_OBJ_READ:
        case NETMEM_OBJ_SERVER_WRITE:
/*
                Read the value.
*/
                *value = * (((int *) (my_obj→data)) + offset);
                return (ERR_SUCCESS);
        case NETMEM_OBJ_KERNEL_WRITE:
/*
                Request kernel to give up write access to page. Ask for a
                reply when it’s finished.
*/
                printf (“lock_request: clean, lock %d\n”,
                            VM_PROT_WRITE);
                (void) memory_object_lock_request (my_obj→
                            control_port, (vm_address_t) 0, vm_page_size,
                            TRUE, FALSE, VM_PROT_WRITE,
                            my_obj→object_port);
                my_obj→state = NETMEM_OBJ_CLEANING;
```

```
/*
                                        Fall through...
*/
        case NETMEM_OBJ_CLEANING:
        case NETMEM_OBJ_FLUSHING:
/*
                Wait for this change to complete. Enqueue this operation to be
                handled later.
*/
                pending_op_queue (my_obj, PENDING_READ, offset, 0,
                        reply_to);
/*
                Reply will happen when lock_request completes.
*/
                return (MIG_NO_REPLY);
        default:
                printf ("Bad state %d\n", my_obj→state);
                return (NETMEM_FAILURE);
        }
/*
                                        NOTREACHED
*/
}
/*
Write request.
*/
mach_error_t netmem_write (mach_port_t object, mach_port_t reply_to,
            vm_address_t offset, int value)
{
        netmem_obj_t                    my_obj;
        printf ("write\n");
/*
        Check arguments.
*/
        if ((my_obj = netmem_lookup (object)) == NETMEM_OBJ_NULL)
                return (NETMEM_BAD_OBJECT);
        if (offset < 0 || offset >= (vm_page_size / sizeof (int)))
                return (NETMEM_BAD_OFFSET);
/*
        Execute the write.
*/
        switch (my_obj→state)
        {
        case NETMEM_OBJ_NONE:
/*
                Need to initialize. Server has exclusive access (write) when
                we're done.
*/
                (void) vm_allocate (mach_task_self (), &my_obj→data,
                        vm_page_size, TRUE);
                my_obj→state = NETMEM_OBJ_SERVER_WRITE;
/*
                                        Fall through...
*/
        case NETMEM_OBJ_SERVER_WRITE:
```

```
/*
                Write the value.
*/
                * ((int *) (my_obj→data) + offset) = value;
                return (ERR_SUCCESS);
        case NETMEM_OBJ_READ:
        case NETMEM_OBJ_KERNEL_WRITE:
/*
                Request kernel to give up all access to page. Ask for a reply
                when it's finished.
*/
                printf ("lock_request: clean, flush, lock %d\n",
                            VM_PROT_ALL);
                (void) memory_object_lock_request (my_obj→
                            control_port, (vm_address_t) 0, vm_page_size,
                            TRUE, TRUE, VM_PROT_ALL,
                            my_obj→object_port);
                my_obj→state = NETMEM_OBJ_FLUSHING;
/*
                                        Fall through...
*/
        case NETMEM_OBJ_CLEANING:
        case NETMEM_OBJ_FLUSHING:
/*
                Wait for this change to complete. Enqueue this operation to be
                handled later.
*/
                pending_op_queue (my_obj, PENDING_WRITE, offset,
                            value, reply_to);
/*
                Reply will happen when lock_request completes.
*/
                return (MIG_NO_REPLY);
        default:
                printf ("Bad state %d\n", my_obj→state);
                return (NETMEM_FAILURE);
        }
/*
                                        NOTREACHED
*/
}
```

External Memory Management Protocol

The routines that follow handle the external memory management protocol itself.

```
/*
Routines for handling paging messages from the kernel.
*/
/*
Kernel request to initialize object.
*/
kern_return_t memory_object_init (memory_object_t object,
            memory_object_control_t control, memory_object_name_t name,
            vm_size_t obj_page_size)
```

```
{
    netmem_obj_t                    my_obj;
    printf (“init\n”);
/*
    Check the object’s page size.
*/
    if (obj_page_size != vm_page_size)
    {
        printf (“page size mismatch\n”);
        return (KERN_FAILURE);
    }
/*
    Check that it’s our object.
*/
    if ((my_obj = netmem_lookup (object)) == NETMEM_OBJ_NULL)
        return (KERN_FAILURE);
/*
    Set up control port. If it’s not NULL, we’ve just won an init/terminate
    race. This code does not cope with that race. The right thing to do is
    allocate a new data structure to represent the object. This race will be
    removed from future Mach kernels.
*/
    if (my_obj→control_port != MACH_PORT_NULL)
    {
        printf (“init before terminate\n”);
        if (my_obj→control_port == control)
        {
            printf (“control port match: I’m confused\n”);
            return (KERN_FAILURE);
        }
/*
        Get rid of all the send rights on that port.
*/
        (void) mach_port_mod_refs (mach_task_self (),
                    my_obj→control_port,
                    MACH_PORT_RIGHT_SEND, –
                    (my_obj→send_rights));
    }
    my_obj→control_port = control;
    my_obj→send_rights = 1;
/*
    Reply: the object is ready. Not cacheable, no special copy strategy.
*/
    printf (“ready\n”);
    (void) memory_object_ready (control, FALSE,
                MEMORY_OBJECT_COPY_NONE);
/*
    Send right on control port is held by object data structure until
    termination. Get rid of name port right. mach_port_deallocate could
    also be used here.
*/
    (void) mach_port_mod_refs (mach_task_self (), name,
                MACH_PORT_RIGHT_SEND, –1);
    return (KERN_SUCCESS);
}
/*
Kernel object termination.
*/
kern_return_t memory_object_terminate (memory_object_t object,
            memory_object_control_t control, memory_object_name_t name)
{
        netmem_obj_t                    my_obj;
        printf ("terminate\n");
/*
        Check that it's our object.
*/
        if ((my_obj = netmem_lookup (object)) == NETMEM_OBJ_NULL)
                return (KERN_FAILURE);
/*
        Clear control port, and remove any send rights we have.
*/
        if (my_obj→control_port == control)
        {
                my_obj→control_port = MACH_PORT_NULL;
                (void) mach_port_mod_refs (mach_task_self (), control,
                            MACH_PORT_RIGHT_SEND, –
                            (my_obj→send_rights));
                my_obj→send_rights = 0;
        }
/*
        Reset object state: no kernel copy, so server write ok.
*/
        my_obj→state = NETMEM_OBJ_SERVER_WRITE;
/*
        Now get rid of the receive rights that came in this message.
*/
        (void) mach_port_mod_refs (mach_task_self (), control,
                    MACH_PORT_RIGHT_RECEIVE,–1);
        (void) mach_port_mod_refs (mach_task_self (), name,
                    MACH_PORT_RIGHT_RECEIVE,–1);
        return (KERN_SUCCESS);
}
```

memory_object_data_request is fairly simple. The mere fact that we received this message means that the kernel does not hold the page and wants it. We must clearly be holding the most recent copy and can supply it to the kernel directly, unless we are in the midst of a lock request sequence, in which case the page request is deferred.

```
/*
Page-in
*/
kern_return_t memory_object_data_request (memory_object_t object,
            memory_object_control_t control, vm_address_t offset, vm_size_t
            length, vm_prot_t access)
{
        netmem_obj_t                    my_obj;
        vm_prot_t                       lock_value;
        printf ("data_request %d\n", access);
```

```
/*
        Check that it’s our object.
*/
        if ((my_obj = netmem_lookup (object)) == NETMEM_OBJ_NULL)
                return (KERN_FAILURE);
/*
        Check offset and length arguments.
*/
        if (offset > 0 || offset + length > vm_page_size)
        {
/*
                Accessing this region of the object is an error. NOTE: This
                code assumes that there will not be multi-page requests that
                include both valid and invalid regions. Current kernels do not
                make such requests. A request like that would be split into its
                components here.
*/
                printf (“data_error\n”);
                (void) memory_object_data_error (control, offset, length,
                                NETMEM_BAD_OFFSET);
                return (KERN_FAILURE);
        }
/*
        Now do something about it.
*/
        switch (my_obj→state)
        {
        case NETMEM_OBJ_NONE:
/*
                Need to initialize. Initialize kernel to zeros with
                data_unavailable. Kernel gets write access.
*/
                (void) vm_allocate (mach_task_self (), &my_obj→data,
                                vm_page_size, TRUE);
                printf (“data_unavailable\n”);
                (void) memory_object_data_unavailable (control, offset,
                                length);
                my_obj→state = NETMEM_OBJ_KERNEL_WRITE;
                break;
        case NETMEM_OBJ_READ:
        case NETMEM_OBJ_SERVER_WRITE:
        case NETMEM_OBJ_KERNEL_WRITE:
/*
                Kernel does not have a copy. Give it one. If write permission
                wasn’t requested, don’t grant it.
*/
                if (access & VM_PROT_WRITE)
                {
                        lock_value = VM_PROT_NONE;
                        my_obj→state =
                                        NETMEM_OBJ_KERNEL_WRITE;
                }
                else
                {
```

```
[227]                       lock_value = VM_PROT_WRITE;
[228]                       my_obj→state = NETMEM_OBJ_READ;
[229]                   }
[230]                   printf (“data_supply: lock %d\n”, lock_value);
[231]                   (void) memory_object_data_supply (control, offset,
                              my_obj→data, length, FALSE, lock_value,
                              FALSE, MACH_PORT_NULL);
[232]                   break;
[233]           case NETMEM_OBJ_CLEANING:
[234]           case NETMEM_OBJ_FLUSHING:
    /*
                        This will be handled when the cleaning or flushing completes.
    */
[235]                   my_obj→pending_kernel_access |= access;
[236]                   my_obj→pending_kernel_type = PENDING_TYPE_DATA;
[237]                   break;
[238]           default:
[239]                   printf (“Bad state %d\n”, my_obj→state);
[240]                   break;
[241]           }
    /*
                Get rid of send right on control port.
    */
[242]           netmem_control_cleanup (my_obj);
[243]           return (KERN_SUCCESS);
[244]   }
```

For **memory_object_data_unlock**, it follows that the kernel holds the page, but wishes more access. If not in the middle of a lock request sequence, we can grant it. Note (line [257]) that no reply port is specified for the lock request that grants the access. Since the page must be viewed as entering the NETMEM_OBJ_KERNEL_WRITE state as soon as we send this request, there is no sense waiting for a reply.

```
    /*
        Unlock request. The kernel has a read only page and wants to write.
    */
[245]   kern_return_t memory_object_data_unlock (memory_object_t object,
                    memory_object_control_t control, vm_address_t offset, vm_size_t
                    length, vm_prot_t access)
[246]   {
[247]           netmem_obj_t                    my_obj;
[248]           vm_prot_t                       lock_value;
[249]           printf (“data_unlock %d\n”, access);
    /*
                Check that it’s our object.
    */
[250]           if ((my_obj = netmem_lookup (object)) == NETMEM_OBJ_NULL)
[251]                   return (KERN_FAILURE);
    /*
                Check offset and length arguments.
    */
[252]           if (offset > 0 || offset + length > vm_page_size)
[253]                   return (KERN_FAILURE);
```

```
 /*
               Can handle in read state, otherwise mark as pending.
 */
[254]          if (my_obj→state == NETMEM_OBJ_READ)
[255]          {
 /*
                      Drop the read page lock.
 */
[256]                 printf ("lock_request: lock %d\n", VM_PROT_NONE);
[257]                 (void) memory_object_lock_request (control, offset, length,
                                 FALSE, FALSE, VM_PROT_NONE,
                                 MACH_PORT_NULL);
[258]                 my_obj→state = NETMEM_OBJ_KERNEL_WRITE;
[259]          }
[260]          else if (my_obj→state == NETMEM_OBJ_CLEANING ||
                          my_obj→state == NETMEM_OBJ_FLUSHING)
[261]          {
[262]                 my_obj→pending_kernel_access |= access;
[263]                 my_obj→pending_kernel_type = PENDING_TYPE_LOCK;
[264]          }
[265]          else
[266]                 printf ("Unexpected state %d\n", my_obj→state);
 /*
               Get rid of send right on control port.
 */
[267]          netmem_control_cleanup (my_obj);
[268]          return (KERN_SUCCESS);
[269]   }
```

memory_object_data_return gives the manager a fresh copy of the page. Since the page was virtually copied as part of the data message, the manager uses this page, freeing the old (line [281]). If the manager knows that it is not in the middle of a lock request sequence, it follows that this page-out means that the kernel is giving up its copy; otherwise the manager cannot be sure.

```
 /*
        Page-out
 */
[270]   kern_return_t memory_object_data_return (memory_object_t object,
                  memory_object_control_t control, vm_address_t offset, pointer_t
                  data, vm_size_t count, boolean_t dirty, boolean_t kernel_copy)
[271]   {
[272]          netmem_obj_t                    my_obj;
[273]          printf ("data_return\n");
 /*
               Check that it's our object.
 */
[274]          if ((my_obj = netmem_lookup (object)) == NETMEM_OBJ_NULL)
[275]                 return (KERN_FAILURE);
 /*
               Check offset and length arguments. Failure return causes
               mach_msg_server to de-allocate data for us.
 */
[276]          if (offset > 0 || offset + count > vm_page_size)
[277]           {
[278]                   printf (“wrong data\n”);
[279]                   return (KERN_FAILURE);
[280]           }
   /*
                Handle the write. Swap the data for our old copy.
   */
[281]           (void) vm_deallocate (mach_task_self (), my_obj→data,
                            vm_page_size);
[282]           my_obj→data = data;
   /*
                Either in kernel write state, or middle of a clean or flush. In latter case,
                state changes when that operation completes.
   */
[283]           if (my_obj→state == NETMEM_OBJ_KERNEL_WRITE)
[284]                   my_obj→state = NETMEM_OBJ_SERVER_WRITE;
[285]           else if (my_obj→state != NETMEM_OBJ_CLEANING &&
                            my_obj→state != NETMEM_OBJ_FLUSHING)
[286]                   printf (“Unexpected state %d\n”, my_obj→state);
   /*
                Get rid of send right on control port.
   */
[287]           netmem_control_cleanup (my_obj);
[288]           return (KERN_SUCCESS);
[289]   }
```

The **memory_object_lock_completed** routine exists mostly to perform deferred processing. If the lock request was a clean operation, it follows that the kernel no longer has write access (and all modified data has been received); if the lock request was a flush operation, the kernel no longer holds the page at all. The interesting thing about this latter condition (line [304]) is that, if there were a pending kernel request for more access to the page, the kernel no longer simply wants more access, it wants the page back. The kernel will send a page request (following its flushing of the page), so the manager discards the access request. The lock completed routine uses **pending_op_execute** to execute all pending client requests it can given the page state. If there is a pending write request and the kernel still holds a copy, a new lock request sequence must be started. Once all pending requests are processed, any kernel data requests can be honored.

```
   /*
        Completion of a clean or flush operation.
   */
[290]   kern_return_t memory_object_lock_completed (memory_object_t object,
                      memory_object_control_t control, vm_address_t offset, vm_size_t
                      length)
[291]   {
[292]           netmem_obj_t                    my_obj;
[293]           vm_prot_t                       lock_value;
[294]           printf (“lock_completed\n”);
   /*
                Check that it’s our object.
   */
[295]           if ((my_obj = netmem_lookup (object)) == NETMEM_OBJ_NULL)
[296]                   return (KERN_FAILURE);
```

```
/*
        Check offset and length arguments.
*/
        if (offset > 0 || offset + length > vm_page_size)
                return (KERN_FAILURE);
/*
        Action depends on what state we were in, but cleaning and flushing are
        similar.
*/
        if (my_obj→state == NETMEM_OBJ_CLEANING)
/*
                Kernel may have read only copy of page.
*/
                my_obj→state = NETMEM_OBJ_READ;
        else if (my_obj→state == NETMEM_OBJ_FLUSHING)
        {
/*
                Kernel no longer has page.
*/
                my_obj→state = NETMEM_OBJ_SERVER_WRITE;
                if (my_obj→pending_kernel_type =
                            PENDING_TYPE_LOCK)
                        my_obj→pending_kernel_access =
                                    VM_PROT_NONE;
        }
        else
        {
                printf (“unexpected state: %d\n”, my_obj→state);
                return (KERN_FAILURE);
        }
/*
        Handle pending client message operations.
*/
        pending_op_execute (my_obj);
/*
        Pending message write takes priority (cleaning state only).
*/
        if (my_obj→pending_op_queue != PENDING_OP_NULL)
        {
/*
                Request kernel to give up all access to page. Ask for a reply
                when it’s finished.
*/
                printf (“lock_request: clean, flush, lock %d\n”,
                            VM_PROT_ALL);
                (void) memory_object_lock_request (control,
                            (vm_address_t) 0, vm_page_size, TRUE,
                            TRUE, VM_PROT_ALL, my_obj→object_port);
                my_obj→state = NETMEM_OBJ_FLUSHING;
        }
/*
        Does the kernel want the page?
*/
        else if (my_obj→pending_kernel_access != VM_PROT_NONE)
        {
```

```
[321]                   if (my_obj→pending_kernel_access &VM_PROT_WRITE)
[322]                   {
[323]                           lock_value = VM_PROT_NONE;
[324]                           my_obj→state =
                                        NETMEM_OBJ_KERNEL_WRITE;
[325]                   }
[326]                   else
[327]                   {
[328]                           lock_value = VM_PROT_WRITE;
[329]                           my_obj→state = NETMEM_OBJ_READ;
[330]                   }
[331]                   my_obj→pending_kernel_access = VM_PROT_NONE;
[332]                   if (my_obj→pending_kernel_type ==
                                    PENDING_TYPE_LOCK)
[333]                   {
[334]                           printf (“lock_request: lock %d\n”, lock_value);
[335]                           (void) memory_object_lock_request (control,
                                        (vm_address_t) 0, vm_page_size,
                                        FALSE, FALSE, lock_value,
                                        MACH_PORT_NULL);
[336]                   }
[337]                   else
[338]                   {
[339]                           printf (“data_supply: lock %d\n”, lock_value);
[340]                           (void) memory_object_data_supply (control,
                                        (vm_address_t) 0, my_obj→data,
                                        vm_page_size, FALSE, lock_value,
                                        FALSE, MACH_PORT_NULL);
[341]                   }
[342]           }
   /*
                Get rid of send right on control port.
   */
[343]           netmem_control_cleanup (my_obj);
[344]           return (KERN_SUCCESS);
[345]   }
```

Non-Used Stubs

```
   /*
        Not used by this pager
   */
[346]   kern_return_t memory_object_copy (memory_object_t old_object,
                    memory_object_control_t old_control, vm_address_t offset,
                    vm_size_t length, memory_object_t new_object)
[347]   {
[348]           printf (“object_copy\n”);
[349]           return (KERN_FAILURE);
[350]   }
[351]   kern_return_t memory_object_supply_completed (memory_object_t object,
                    memory_object_control_t control, vm_address_t offset, vm_size_t
                    length, kern_return_t result, vm_offset_t error_offset)
```

```
{
        printf ("supply_completed\n");
        return (KERN_FAILURE);
}
kern_return_t memory_object_data_write (memory_object_t object,
            memory_object_control_t control, vm_address_t offset, pointer_t
            data)
{
        printf ("data_write\n");
        return (KERN_FAILURE);
}
kern_return_t memory_object_change_completed (memory_object_t object,
            boolean_t may_cache, boolean_t copy_strategy)
{
        printf ("change_completed");
        return (KERN_FAILURE);
}
```

Server Loop and Main Program

```
/*
De-mux routine for mach_msg_server. This allows us to splice two interfaces
into mach_msg_server ().
*/
boolean_t netmem_demux (mach_msg_header_t *inmsg, mach_msg_header_t
            *outmsg)
{
        return (netmem_msg_server (inmsg, outmsg) ||
                memory_object_server (inmsg, outmsg));
}
/*
Size of max message we're willing to receive or send. Have to allow enough
space for memory object messages from the kernel. This number is on the high
side.
*/
#define NETMEM_MAX_MSG_SIZE     512
/*
Main routine: initialize and loop forever handling messages. Ignore error returns
from mach_msg_server.
*/
main ()
{
        (void) init ();
        while (1)
        {
                (void) mach_msg_server (netmem_demux,
                            NETMEM_MAX_MSG_SIZE,
                            master_obj.object_port);
        }
}
```

Sample Client

```
/*
    File: netmem_client.c
    Author: David L. Black
    Simple client to demonstrate netmem server.
*/
#include <mach.h>
#include <mach_init.h>
#include <servers/netname.h>
#include <stdio.h>
#include "netmem_defs.h"
main ()
{
        kern_return_t                   r;
        mach_port_t                     server_port;
        vm_address_t                    netmem_address;
        r = netname_look_up (name_server_port, "", NETMEM_NAME,
                    &server_port);
        if (r != NETNAME_SUCCESS)
        {
                mach_error ("Server Look Up", r);
                exit (–1);
        }
/*
        Deliberately map in an extra page to exercise error path.
*/
        netmem_address = 0;
        r = vm_map (mach_task_self (), &netmem_address, 2 *
                    vm_page_size, 0, TRUE, server_port, 0, FALSE,
                    VM_PROT_ALL, VM_PROT_ALL,
                    VM_INHERIT_NONE);
        if (r != KERN_SUCCESS)
        {
                mach_error ("Object vm_map", r);
                exit (–1);
        }
        printf ("Mapped at 0x%x\n", netmem_address);
/*
        Simple command format:
        Letter offset data
        Letter: p[ut], g[et], r[ead], w[rite]
        put and get are messages, read and write are memory.
*/
        printf ("Server contacted. pgrw = put/get/read/write x = exit\n");
        for (;;)
        {
                char                    cmd;
                int                     c, offset, data;
                printf ("pgrw> ");
                scanf ("%c %d", &cmd, &offset);
                switch (cmd)
                {
```

```
                case ‘p’:
                        scanf (“%d”, &data);
                        printf (“put %d = %d”, offset, data);
                        r = netmem_write (server_port, offset, data);
                        printf (“ result %d\n”, r);
                        break;
                case ‘g’:
                        printf (“get %d”, offset);
                        r = netmem_read (server_port, offset, &data);
                        printf (“ = %d result %d\n”, data, r);
                        break;
                case ‘r’:
                        data = * (((int *) netmem_address) + offset);
                        printf (“read %d = %d\n”, offset, data);
                        break;
                case ‘w’:
                        scanf (“%d”, &data);
                        * (((int *)netmem_address) + offset) = data;
                        printf (“write %d = %d\n”, offset, data);
                        break;
                case ‘x’:
                        exit (1);
                default:
                        printf (“???\n”);
                }
                while ((c = getc (stdin)) != ‘\n’);
        }
/*
                                        NOTREACHED
*/
}
```

Makefile

```
/*
File: Makefile
Author: David L. Black
Makefile for netmem example. Avoids use of make features for clarity. Make
knows how to make foo.o from foo.c (cc –c).
*/
all: netmem_client netmem_pager
netmem_pager: netmem_msg_server.o netmem_pager.o
            netmem_msg_reply_user.o
        cc –o netmem_pager netmem_msg_server.o
                    netmem_msg_reply_user.o netmem_pager.o –lmach
netmem_pager.o: netmem_msg.h netmem_defs.h netmem_obj.h
            netmem_pager.c netmem_msg_reply.h
        cc –c netmem_pager.c
netmem_client: netmem_msg_user.o netmem_client.o
        cc –o netmem_client netmem_msg_user.o netmem_client.o –lmach
netmem_client.o: netmem_msg.h netmem_defs.h netmem_client.c
        cc –c netmem_client.c
```

```
[10]  netmem_msg_server.c netmem_msg_user.c netmem_msg.h:
                  netmem_msg.defs
[11]          mig –server netmem_msg_server.c –user netmem_msg_user.c
                          netmem_msg.defs
[12]  netmem_msg_reply.h netmem_msg_reply_user.c: netmem_msg_reply.defs
[13]          mig –user netmem_msg_reply_user.c –server /dev/null
                          netmem_msg_reply.defs
```

Sample Output

Start the pager and the client. (The pager must be given time to register itself with the name server, so it is started first.)

```
 [1]  > netmem_client                          > netmem_pager
 [2]  Mapped at 0x1d000                        init
 [3]                                           ready
 [4]  Server contacted. pgrw = put/get/read/write x = exit
```

netmem_write. Page exists only in the manager.

```
 [5]  pgrw> p 0 10                             write
 [6]  put 0 = 10 result 0
```

netmem_read. Page still only in the manager.

```
 [7]  pgrw> g 0                                read
 [8]  get 0 = 10 result 0
```

Touch page. Kernel requests read-only copy of page. Manager provides page. Both kernel and manager have current copy.

```
 [9]  pgrw> r 0                                data_request 1
[10]                                           data_supply: lock 2
[11]  read 0 = 10
```

netmem_read. Manager can directly reply since it has a current copy.

```
[12]  pgrw> g 0                                read
[13]  get 0 = 10 result 0
```

Modify page. Kernel asks manager for write access. Write access granted. Now only the kernel has a current copy.

```
[14]  pgrw> w 1 20                             data_unlock 3
[15]                                           lock_request: lock 0
[16]  write 1 = 20
```

Touch page.

```
[17]  pgrw> r 1
[18]  read 1 = 20
```

netmem_read. Manager must ask kernel for current copy. Since the manager is only going to read the page, it does not ask the kernel to flush its copy, merely provide it and remove write access.

```
[19]  pgrw> g 1                          read
[20]                                     lock_request: clean, lock 2
[21]                                     data_return
[22]                                     lock_completed
```

Pending read done and reply generated.

```
[23]                                     read
[24]                                     read_reply
[25]  get 1 = 20 result 0
```

netmem_write. Since the manager will modify the page, it must ask the kernel to no longer have a copy.

```
[26]  pgrw> p 2 30                       write
[27]                                     lock_request: clean, flush, lock 7
[28]                                     lock_completed
```

Pending write done here and reply generated.

```
[29]                                     write
[30]                                     write_reply
[31]  put 2 = 30 result 0
```

Modify page. Kernel asks manager for page with write access.

```
[32]  pgrw> w 3 40                       data_request 3
[33]                                     data_supply: lock 0
[34]  write 3 = 40
```

netmem_write. Since the manager will modify the page, it must ask the kernel to no longer have a copy. Here a full flush is necessary.

```
[35]  pgrw> p 4 50                       write
[36]                                     lock_request: clean, flush, lock 7
[37]                                     data_return
[38]                                     lock_completed
```

Pending write done here.

```
[39]                                     write
[40]                                     write_reply
[41]  put 4 = 50 result 0
```

Exit client. This will also send **memory_object_terminate** to the manager.

```
[42]  pgrw> x 0                          terminate
```

Start client again.

```
[43]  > netmem_client
[44]  Mapped at 0x1d000                  init
[45]                                     ready
[46]  Server contacted. pgrw = put/get/...
```

netmem_read. Notice that the server preserved the page.

```
[47]  pgrw> g 3                         read
[48]  get 3 = 40 result 0
```

Touch page. Kernel requests read-only copy.

```
[49]  pgrw> r 4                         data_request 1
[50]                                    data_supply: lock 2
[51]  read 4 = 50
```

Modify page. Kernel asks manager for write access.

```
[52]  pgrw> w 5 60                      data_unlock 3
[53]                                    lock_request: lock 0
[54]  write 5 = 60
```

Touch data within mapped window size but beyond memory object size. The kernel does not know that this is an error. The manager replies with **memory_object_data_error** to say that the page doesn't exist.

```
[55]  pgrw> r 1200                      data_request 1
[56]                                    data_error
```

The client takes a memory error as a result. Its termination un-maps the (only) window, which sends a **memory_object_terminate** message to the manager.

```
[57]  Bus error (core dumped)           data_return
[58]                                    terminate
```

Object Cache

The Mach kernel, through its memory cache object, maintains a physical memory cache of pages for a given memory object. The contents of this cache varies over time, of course, but the cache exists as long as does the memory object. This is normally until the last user of the object un-maps it.

Various objects (such as executable files and files that are read as soon as they are written) would benefit from being known to the kernel even though they (temporarily) are not mapped by any task. This behavior can be achieved with the cacheability attribute set via **memory_object_change_attributes**.

Objects marked as cacheable in this way are not affected during normal operation. However, when the last task un-maps then, they are kept in a (small) kernel cache of objects, thereby preserving their physical cache contents (to the extent that their pages are not evicted via normal system paging activity). When a task does map them again, no **memory_object_init** message will be sent to the manager, since they were never terminated. If not mapped, though, at some future time the object will be terminated, with the appropriate message being sent to the manager.

If an object is cacheable but not mapped, and the cacheable attribute is disabled (**memory_object_change_attributes**), the object will be terminated.

Precious Pages

Precious pages (described in the *Kernel Principles* document) create a variety of difficulties in the page management protocols:

- The memory manager can send multiple *precious* pages with a single **memory_object_data_supply** call. The kernel, however, may not accept them all for various reasons (most likely because the kernel already has one or the kernel does not want one). As such, the kernel may be faced with some pages it must return to the manager. It will do so with **memory_object_data_return** calls. If the memory manager wants to synchronize with this data return, it can do so by requesting a reply message from the data supply call. This reply, **memory_object_supply_completed** indicates which pages were accepted. The reply will follow all data return messages resulting from the data supply, so the manager can use this reply message as the delimiter of the sequence.
- When the kernel evicts pages of the memory object, it must return to the manager both modified pages (*precious* or otherwise) as well as *precious* pages (modified or not). The **memory_object_data_return** message has information to allow the memory manager to be able to tell the difference between these cases.
- Given the asynchronous nature of the memory management messages, it is possible that the memory object is terminated while the memory manager is trying to supply (*precious*) data to it (by data supply messages to the memory cache control port). To avoid discarding these *precious* pages, the kernel gives the receive right to the memory cache ports to the memory manager in the **memory_object_terminate** message. The memory manager must drain this port to re-obtain all of its *precious* pages.

Synchronization with Multiple Objects, Kernels and Threads

The external memory management protocol consists of asynchronous interactions so that the maximum degree of parallelism can be achieved. This places considerable burden upon the memory manager to correctly synchronize activities upon its objects. Many of the concerns involved in writing a multi-threaded pager are the same as those for writing any other multi-threaded server.

Multiple Pages

First of all, the manager must track the state of each page. For each page, the manager must track its ownership and location (which kernels hold a copy, whether the page is in transit to backing store or other kernels) and any outstanding operations upon the page. Also, the manager must not attempt multiple simultaneous page-in or access changes on the same page.

The manager is allowed to operate on a set of pages in a given interaction with the kernel. However, multiple ranges must not overlap. If a request supplies pages 2, 3 and 4, for example, a second request must not supply pages 4, 5 and 6.

It is an error to supply a page more than once, and the kernel does not guarantee that the first request will be completely processed before the second. If a page is supplied that the kernel still holds, the data supply is ignored. However, the following scenario will result in page inconsistency:

- The manager supplies a writable page to the kernel which is subsequently modified.
- The kernel decides to evict the modified page.
- Not yet seeing the evicted page, the manager re-supplies the (old contents of the) page.
- The kernel accepts this old page thereby effectively discarding the recent changes.

Multiple Objects

When the manager manages multiple objects, it is typical to create a port set that contains the receive rights for the various abstract memory object ports to be used for the server loop (**mach_msg_server**). The manager needs some translation table to translate from port to memory object control structure. With care the manager can actually use the memory object control structure address as the port name for the corresponding abstract memory object port as discussed in CHAPTER 6.

Multiple Threads

To achieve greater parallelism in the memory manager, aside from the use of asynchronous interactions, the manager would typically use multiple threads. A simple minded use of multiple threads is to assign each memory object to one and only one thread (but different objects or sets of objects to different threads) thereby preserving the operation ordering described in the previous sections. However, this limits parallelism for any given object; the kernel is fully capable of supporting operations on multiple pages of a single object at the same time. The use of multiple threads to handle paging operations for a single object requires special care.

As an example of the synchronization problem, consider the following sequence of events:

- The kernel decides to evict a modified page.
- The kernel subsequently decides to read the page again.
- Manager thread X receives the **memory_object_data_return** message and is preempted before it can process the request.
- Manager thread Y receives the **memory_object_data_request** message. Not realizing that the page had been modified, it returns the old copy of backing store.
- Thread X gets to run again, updating backing store.

In this case, the kernel does not receive the correct (latest) page contents.

Mach ports guarantee that the order that messages are en-queued onto a port (from a single source) is the same order in which they will be de-queued. However, this is not necessarily the same order in which a server will process them; that order is determined by scheduling concerns. For a memory manager to correctly maintain any given page, though, it must process these operations in order.

To maintain this order, Mach IPC assigns sequence numbers to messages. These numbers provide the order in which messages are de-queued from a source. For a multi-threaded manager to maintain operation order, it must synchronize against the memory object structure. That is, the memory object structure maintains an operation sequence number that is incremented whenever an operation is processed. If a thread (thread Y in the example sequence above) finds that the sequence number in its message is not one more than the sequence number reflected in the memory object structure, it knows that some other thread has the intervening operation. The thread would then put its operation onto the pending operation queue for that object.

Multiple Kernels

A manager that supports distributed memory spanning multiple kernels “simply” has to track the state of each page on each kernel, revoking access and passing pages from machine to machine. Much of the mechanisms to support this have already been discussed.

Various additions would be necessary to FIGURE 2 to handle the multiple kernels. The manager may well need to track which kernels hold the current page contents, not just whether the (single) kernel does. Only a single kernel would hold the page if it has write access. A page may be present with read access on multiple kernels, changing the *both* state into a *many* state.

Additional transitions are meaningful:

- If the manager itself needs the page, it must clean or flush all kernels holding the page.
- A data request message from one of the kernels may find the page in any state:
    - If in the *manager* state, the page would be provided to the requesting kernel with appropriate access (read implying a *many* state, write implying the *kernel* state).
    - If in a *kernel* state (current contents on some kernel), it is necessary to clean or flush that kernel as appropriate to get current contents and to permit the access requested of this additional kernel.
    - If in the *many* state (no kernel has write permission) the page can be supplied to the new kernel; the other kernels need to be flushed only if the additional kernel requests write access.

  Note that a data request message from a kernel only means that particular kernel no longer holds the page—the other kernels may well still hold it.
- A data unlock request can only find the page in the *many* state. If a kernel has write access to the page, it follows that no other kernel holds the page and so no unlock request is possible. However, multiple kernels may be holding a read-only copy when a kernel requests write access. It is necessary to flush the page from the other kernels before granting the write access.

- It is meaningful to supply a read-only *precious* page to the kernel even for a memory object that permits modification. This might be done if the page existed on other nodes.
- It would be meaningful to clean a *precious* page. The manager might do this so as to receive the current contents of the page to be provided to another kernel.

As it so happens it is more efficient to place a memory manager on each node that communicate with each other at a higher level than the external memory management protocol. For example, if kernel A holds a page that kernel B wants (and the central manager is on neither A nor B), it is more efficient for the page to travel from A directly to B than from A to the manager and then to B. For a discussion of this, see:

> Forin, A., et al., “The Shared Memory Server,” in *Proceedings of the Winter 1989 USENIX Conference*.

Loose Consistency

This chapter has discussed the use of kernel mechanisms to maintain strict read–write consistency for shared memory. Not all memory managers need to maintain such strict consistency.

The consistency so far described is referred to as *strict* consistency because there is absolutely no way that a client could detect any inconsistency. The manager revokes all access to all other entities (clients and kernels) whenever any entity needs to write so that these other entities will fault on their memory and re-fetch it, thereby receiving the absolute latest changes. All accesses to a page are serialized; at any given time their are multiple readers or a single writer. This brute force approach, however, is often overkill. Its strict serialization reduces the parallelism with which the shared memory can be referenced. Also, the use of the external memory management protocols to ensure consistency requires the movement of whole pages where much smaller data movement may do (if managed directly by the application).

The issue in loosening consistency constraints (that is, allowing multiple readers and one or more writers) is whether, given a set of clients, one of them can see inconsistent memory values. That is, consider data values A and B kept in separate shared memory pages where client X modifies A and then B. If client Y fetches B and sees the new value, it follows that Y should also see the new value for A since it was modified first. Without some synchronization, it would be possible for client Y to see the old value of A. Of course, this consistency is only meaningful to Y if it knows (or cares) that A is modified before B.

One way to loosen the consistency constraints is for an application to take explicit control over when page images are updated on other nodes given its own knowledge of the data dependencies. Another way is for the memory manager to track data references by the various client (nodes). In this way, if the manager sends a new value of a page to a node, it also sends new values (or invalidates old values) of any pages that it knows were modified prior to the modification of the page it is sending. It is possible for a manager to track the order of modification of pages via kernel generated **memory_object_unlock_-request** messages. For a more detailed discussion of this “causal ordered” memory, see:

> Boyer, F., “A Causal Distributed Shared Memory Based on External Pagers,” in *Proceedings of the 1991 USENIX Mach Symposium*.

A sample use of loosened consistency appears in servers that maintain multiple copies of data that need only be recent. Note that strict read–write consistency required that only one kernel hold the modified contents of a page at a time; this means that the failure of that kernel damages the memory object. If a kernel evicts modified page A which is subsequently modified on some other kernel without having evicted modified page B, the old contents of page B no longer correspond to the new value of page A. Loosening consistency allows multiple modified copies to be kept. Given a failure of a copy, the managers can reconstruct the latest values from recent values. It would be very difficult to do this in general but various “blackboard” servers with limited semantics have been built that provide this service.

CHAPTER 8 Network Shared Memory Server

The most typical use of an external memory manager is to provide shared memory between otherwise unrelated tasks. When this memory is to be shared between tasks on multiple hosts, it is referred to as distributed shared memory or netmemory. Mach provides a standard external memory manager server to provide such memory, suitable for sharing between tasks on a single host, or between multiple hosts. This server is the NetMemory server.

To share memory between tasks using the netmemory server, a task first creates a netmemory object with the **netmemory_create** call. This netmemory object can be distributed to all interested tasks either directly by IPC between tasks or indirectly by using a service such as the NetName server.

Each task then find its local netmemory server, and calls **netmemory_cache** on the local server with the netmemory object to obtain a local Mach memory object corresponding to the netmemory object. The **netmemory_cache** call allows the netmemory servers to use more efficient and more distributed protocols for maintaining object consistency than the standard memory manager interface.

Finally, the resulting memory object should be given to **vm_map** to map the object into the caller's address space. Again, the netmemory object is only for use by **netmemory_-cache**; it should not be handed directly to **vm_map**.

The netmemory object can be explicitly destroyed by calling **netmemory_destroy** on the *netmemory_control* port. This call is not necessary with the current implementation of the netmemoryserver which cleans up a netmemory object after the last kernel has unmapped the object.

The current netmemory server is actually a compatibility front-end to the External Memory Manager library.

The following is a routine which demonstrates how to use the netmemory server.

```
/*
    Create and map a shared object of given size with netname object_name.
*/
/*
    One task (the “master”) should call this routine with hostname = 0; the routine
    will then create a netmemory object and register it with the netname server
    under the supplied object name. All other tasks (the “slaves”) call this routine
    with the hostname where the master lives.
*/
kern_return_t map_object (char* object_name, char* hostname, vm_offset_t*
            address, vm_size_t size, boolean_t anywhere)
{
        kern_return_t           kr;
        mach_port_t             netmemory_server;
        mach_port_t             memory_object;
        mach_port_t             netmemory_object;
        mach_port_t             netmemory_control;
/*
        Find the local netmemory server. (If this routine is used a lot, this value
        can be cached.)
*/
        kr = netname_look_up (name_server_port, “”, “netmemoryserver”,
                                &netmemory_server);
        if (kr)
                return kr;
/*
        If a hostname is provided, then we are the slave and thus we simply
        look up the netmemory object on the given host by using the object
        name. If a hostname is not provided, then we are the master and thus
        have the responsibility of creating a netmemory object and registering
        it with the netname service under the given object name.
*/
        if (hostname)
        {
                kr = netname_look_up (name_server_port, hostname,
                            object_name, &netmemory_object);
                if (kr)
                        return kr;
        }
        else
        {
                kr = netmemory_create (netmemory_server, size,
                            &netmemory_object, &netmemory_control);
                if (kr)
                        return kr;
                kr = netname_check_in (name_server_port, object_name,
                            MACH_PORT_NULL, netmemory_object);
                if (kr)
                {
                        netmemory_destroy (netmemory_control);
                        return kr;
                }
        }
/*
        Cache the object locally. Note that even the master must do this.
*/
        kr = netmemory_cache (netmemory_server, netmemory_object,
                    &memory_object);
        if (kr)
                return kr;
/*
        Map the object, either anywhere or at the supplied address.
*/
        if (anywhere)
                *address = 0;
/*
                                        must be set
*/
        kr = vm_map (mach_task_self (), address, size, 0, anywhere,
                    memory_object, 0, FALSE, VM_PROT_DEFAULT,
                    VM_PROT_DEFAULT, VM_INHERIT_SHARE);
        if (kr)
                return kr;
        return KERN_SUCCESS;
}
```

APPENDIX A MIG Language Specification

The syntax of specification files is given semi-formally in what can be viewed as a slightly extended context free grammar.

Meta language description: The right hand side of a production rule is indented from its left hand side (its heading). Terminal symbols are in plain font. Non-terminal symbols are in bold font. Symbols in italics are non-terminal symbols not defined within the grammar but that have obvious definitions:

- *number* - a string of digits from 0 to 9
- *identifier* - a name consisting of upper and lower case letters, digits 0 to 9 and underscores, with the first character being a letter or an underscore
- *string* - a string of letters, numbers, slashes, periods, underscores, dollar signs and dashes
- *quotedstring* - a string of arbitrary characters (other than ” and new-line) surrounded by quotes
- *anglestring* - a string of arbitrary characters (other than > and new-line) surrounded by angle brackets (<>)
- *filename* - a *quotedstring* or an *anglestring*

The symbolε indicates an empty string.

Symbols within a single production are separated by blanks. Alternative productions are listed on consecutive lines under a production rule heading.

Comments may be included in “*.defs*” file if surrounded by “/*” and “*/”. They are parsed and removed by **cpp**.

No attempt has been made to indicate in the grammar that types must be declared before they are used.

Statements

ε

Statements Statement

Statement

Subsystem;

WaitTime;

MsgOption;

ServerPrefix;

UserPrefix;

ServerDemux;

TypeDecl;

RoutineDecl;

skip;

Import;

RCSDecl;

;

Subsystem

subsystem **SubsystemMods SubsystemName SubsystemBase**

SubsystemMods

ε

SubsystemMods SubsystemMod

SubsystemMod

kerneluser

kernelserver

SubsystemName

identifier

SubsystemBase

number

WaitTime

waittime *string*

nowaittime

MsgOption

msgoption *string*

ServerPrefix

serverprefix *identifier*

UserPrefix

userprefix *identifier*

ServerDemux

serverdemux *identifier*

Import

ImportIndicant *filename*

ImportIndicant

import

uimport

simport

RCSDecl

rcsid *quotedstring*

TypeDecl

type **NamedTypeSpec**

NamedTypeSpec

identifier = **TransTypeSpec**

TransTypeSpec

TypeSpec

TransTypeSpec intran: *identifier identifier* (i*dentifier)*

TransTypeSpec outtran: *identifier identifier* (i*dentifier)*

TransTypeSpec destructor: *identifier (identifier)*

TransTypeSpec ctype: *identifier*

TransTypeSpec cusertype: *identifier*

TransTypeSpec cservertype: *identifier*

TypeSpec

BasicTypeSpec

PrevTypeSpec

VarArrayHead TypeSpec

ArrayHead TypeSpec

^ **TypeSpec**

StructHead TypeSpec

CStringSpec

BasicTypeSpec

IPCType

(I**PCType, IntExp IPCFlags**)

IPCType

PrimIPCType

PrimIPCType | PrimIPCType

PrimIPCType

number

polymorphic

MACH_MSG_TYPE_UNSTRUCTURED

MACH_MSG_TYPE_BIT

MACH_MSG_TYPE_BOOLEAN

MACH_MSG_TYPE_INTEGER_8

MACH_MSG_TYPE_INTEGER_16

MACH_MSG_TYPE_INTEGER_32

MACH_MSG_TYPE_CHAR

MACH_MSG_TYPE_BYTE

MACH_MSG_TYPE_REAL

MACH_MSG_TYPE_STRING

MACH_MSG_TYPE_STRING_C

MACH_MSG_TYPE_PORT_NAME

MACH_MSG_TYPE_POLYMORPHIC

MACH_MSG_TYPE_MOVE_RECEIVE

MACH_MSG_TYPE_COPY_SEND

MACH_MSG_TYPE_MAKE_SEND

MACH_MSG_TYPE_MOVE_SEND

MACH_MSG_TYPE_MAKE_SEND_ONCE

MACH_MSG_TYPE_MOVE_SEND_ONCE

MACH_MSG_TYPE_PORT_RECEIVE

MACH_MSG_TYPE_PORT_SEND

MACH_MSG_TYPE_PORT_SEND_ONCE

IPCFlags

ε

IPCFlags, servercopy

IPCFlags, islong

IPCFlags, isnotlong

IPCFlags, notdealloc

IPCFlags, dealloc

IPCFlags, dealloc []

IPCFlags, countinout

PrevTypeSpec

identifier

VarArrayHead

array [] of

array [*] of

array [*: **IntExp**] of

ArrayHead

array [I**ntExp**] of

StructHead

struct [I**ntExp**] of

CStringSpec

c_string [I**ntExp**]

c_string [*: I**ntExp**]

IntExp

IntExp+ **IntExp**

IntExp– **IntExp**

IntExp* **IntExp**

IntExp/ **IntExp**

number

(I**ntExp**)

RoutineDecl

Routine

SimpleRoutine

Routine

routine *identifier* **Arguments**

SimpleRoutine

simpleroutine *identifier* **Arguments**

A**rguments**

()

(A**rgumentList**)

ArgumentList

Argument

Argument; ArgumentList

Argument

Direction *identifier* **ArgumentType IPCFlags**

Direction

ε

in

out

inout

requestport

replyport

sreplyport

ureplyport

waittime

msgoption

msgseqno

ArgumentType

: *identifier*

: **NamedTypeSpec**

APPENDIX B Standard MIG Types

This appendix provides the definition of the standard MIG data types.

IPC Typenames

MIG internally defines the following **ipc-typenames** in terms of their Mach IPC counterparts. These types are almost never directly used as the type in an argument specification because their type names do not have corresponding named C data types.

```
type MACH_MSG_TYPE_BOOLEAN =
        (MACH_MSG_TYPE_BOOLEAN,32);

type MACH_MSG_TYPE_INTEGER_32 =
        (MACH_MSG_TYPE_INTEGER_32,32);

type MACH_MSG_TYPE_INTEGER_16 =
        (MACH_MSG_TYPE_INTEGER_16,16);

type MACH_MSG_TYPE_INTEGER_8 =
        (MACH_MSG_TYPE_INTEGER_8,8);

type MACH_MSG_TYPE_CHAR = (MACH_MSG_TYPE_CHAR,8);

type MACH_MSG_TYPE_BYTE = (MACH_MSG_TYPE_BYTE,8);

type MACH_MSG_TYPE_BIT = (MACH_MSG_TYPE_BIT,1);
```

This next type does not define the passage of a port right, merely the name of a right. That is, this defines a sufficiently large integer type to hold a port name.

```
type MACH_MSG_TYPE_PORT_NAME =
        (MACH_MSG_TYPE_PORT_NAME,32);
```

The following types do not have their length specified, and so must be used in their long form ("(*ipc-typename, size*)") in type or argument declarations.

```
type MACH_MSG_TYPE_REAL = (MACH_MSG_TYPE_REAL,0);

type MACH_MSG_TYPE_STRING = (MACH_MSG_TYPE_STRING,0);

type MACH_MSG_TYPE_STRING_C =
        (MACH_MSG_TYPE_STRING_C,0);

type MACH_MSG_TYPE_UNSTRUCTURED =
        (MACH_MSG_TYPE_UNSTRUCTURED,0);

type MACH_MSG_TYPE_POLYMORPHIC =
        (MACH_MSG_TYPE_POLYMORPHIC,0);
```

Note that the Mach IPC type for MACH_MSG_TYPE_STRING and MACH_MSG_TYPE_STRING_C are the same.

The following type differs from MACH_MSG_TYPE_POLYMORPHIC in that it specifies a particular (hopefully useful) size.

```
type polymorphic = (MACH_MSG_TYPE_POLYMORPHIC, 32);
```

The following IPC types correspond to the standard treatment of port right passage.

```
type MACH_MSG_TYPE_MOVE_RECEIVE =
        (MACH_MSG_TYPE_MOVE_RECEIVE |
        MACH_MSG_TYPE_PORT_RECEIVE,32);

type MACH_MSG_TYPE_COPY_SEND =
        (MACH_MSG_TYPE_COPY_SEND |
        MACH_MSG_TYPE_PORT_SEND,32);

type MACH_MSG_TYPE_MAKE_SEND =
        (MACH_MSG_TYPE_MAKE_SEND |
        MACH_MSG_TYPE_PORT_SEND,32);

type MACH_MSG_TYPE_MOVE_SEND =
        (MACH_MSG_TYPE_MOVE_SEND |
        MACH_MSG_TYPE_PORT_SEND,32);
```

```
type MACH_MSG_TYPE_MAKE_SEND_ONCE =
            (MACH_MSG_TYPE_MAKE_SEND_ONCE |
            MACH_MSG_TYPE_PORT_SEND_ONCE,32);

type MACH_MSG_TYPE_MOVE_SEND_ONCE =
            (MACH_MSG_TYPE_MOVE_SEND_ONCE |
            MACH_MSG_TYPE_PORT_SEND_ONCE,32);
```

Note that the receiver always sees a Mach IPC type of ..._PORT_... Mach IPC does not indicate to the receiver whether the right was made or moved when sent. The sender, on the other hand, must specify the behavior.

The following IPC types define the *polymorphic* right passage; that is, where the sender must specify an additional parameter that provides the actual IPC type of the right being sent.

```
type MACH_MSG_TYPE_PORT_RECEIVE =
            (MACH_MSG_TYPE_POLYMORPHIC |
            MACH_MSG_TYPE_PORT_RECEIVE,32);

type MACH_MSG_TYPE_PORT_SEND =
            (MACH_MSG_TYPE_POLYMORPHIC |
            MACH_MSG_TYPE_PORT_SEND,32);

type MACH_MSG_TYPE_PORT_SEND_ONCE =
            (MACH_MSG_TYPE_POLYMORPHIC |
            MACH_MSG_TYPE_PORT_SEND_ONCE,32);
```

Standard Defined Types

The following types are defined in **<mach/std_types.defs>**. These types are used in preference to MIG internally defined types because these types have corresponding C data types, or have the corresponding C data type named explicitly.

Integer and Data Types

type *char* = MACH_MSG_TYPE_CHAR;

type *short* = MACH_MSG_TYPE_INTEGER_16;

type *int* = MACH_MSG_TYPE_INTEGER_32;

type *boolean_t* = MACH_MSG_TYPE_BOOLEAN;

type *unsigned* = MACH_MSG_TYPE_INTEGER_32;

type *pointer_t* = ^array[] of MACH_MSG_TYPE_BYTE;

Port Right Types

The following types define actual port rights to be passed.

The next types define passage of specific rights. Note that *mach_port_t* defaults to a send right copy.

type *mach_port_t* = MACH_MSG_TYPE_COPY_SEND;

type *mach_port_array_t* = array[] of *mach_port_t*;

type *mach_port_move_receive_t* = MACH_MSG_TYPE_MOVE_RECEIVE ctype: *mach_port_t*;

type *mach_port_copy_send_t* = MACH_MSG_TYPE_COPY_SEND ctype: *mach_port_t*;

type *mach_port_make_send_t* = MACH_MSG_TYPE_MAKE_SEND ctype: *mach_port_t*;

type *mach_port_move_send_t* = MACH_MSG_TYPE_MOVE_SEND ctype: *mach_port_t*;

type *mach_port_make_send_once_t* = MACH_MSG_TYPE_MAKE_SEND_ONCE ctype: *mach_port_t*;

type *mach_port_move_send_once_t* = MACH_MSG_TYPE_MOVE_SEND_ONCE ctype: *mach_port_t*;

The next types define the polymorphic rights passage—those requiring an additional argument to the MIG stub to specify the actual type of right being passed.

type *mach_port_receive_t* = MACH_MSG_TYPE_PORT_RECEIVE ctype: *mach_port_t*;

type *mach_port_send_t* = MACH_MSG_TYPE_PORT_SEND ctype: *mach_port_t*;

type *mach_port_send_once_t* = MACH_MSG_TYPE_PORT_SEND_ONCE ctype: *mach_port_t*;

This last port right type defines a port right that is polymorphic to both the sender and receiver:

type *mach_port_poly_t* = *polymorphic* ctype: *mach_port_t*;

IPC Data Values

type *kern_return_t* = *int*;

type *mach_port_right_t* = *unsigned*;

type *mach_port_type_t* = *unsigned*;

type *mach_port_type_array_t* = array[] of *mach_port_type_t*;

type *mach_port_urefs_t* = *unsigned*;

type *mach_port_delta_t* = *int*;

type *mach_port_seqno_t* = *unsigned*;

type *mach_port_mscount_t* = *unsigned*;

type *mach_port_msgcount_t* = *unsigned*;

type *mach_port_rights_t* = *unsigned*;

type *mach_msg_id_t* = *int*;

type *mach_msg_type_name_t* = *unsigned*;

type *mach_port_name_t* = MACH_PORT_TYPE_PORT_NAME ctype: *mach_port_t*;

type *mach_port_name_array_t* = array[] of *mach_port_name_t* ctype: *mach_port_array_t*;

APPENDIX C Service Server

Each task inherits a set of *registered* ports upon creation (**mach_ports_register**). These are (send rights) for the name server, an environment server and the service server. All tasks in the system are intended to share the same name server (that is the point of the server) whereas different groups of tasks may have private environment servers. Although an environment server has been written, it is not typically used (and is not part of the standard distribution) so it is not discussed further. The name server was discussed in CHAPTER 2. This appendix discusses the purpose and need for the service server.

All tasks need access to the (same) name server. Given a task whose registered name server port names the name server, all tasks derived from that task will inherit the port and share access to the name server. Unfortunately, being a task itself, the name server initializes itself after certain other tasks (such as the init process, **/etc/init**) run. Thus, most tasks, which derive indirectly from the init task will be missing the name server (and environment server) port.

Because of this, the ports for the name and environment servers must be created (and register with the init process) before the servers the ports name exist. This mechanism is provided by the *service* server.

The privileged service server (**/mach_servers/mach_init**) is run as the first process. It creates the ports that will name the name and environment servers (as well as itself) and forcibly registers them with its parent, thus placing these registered ports at the root of the task derivation tree.

When the name or environment server is initialized, it registers itself with the service server, making the port created by the service server its own. Only the name and environment server have need for the service server. It is important that these servers initialize before user tasks are created as the service server will check-in the correct (the first) servers to announce their desire to register.

Given the service server port (one of all task's registered ports), *service_server*→**service_checkin** registers a given server. The name server, for example, locates the port that will name it (by finding the send right in its registered port set) and uses this port to name itself to the service server. The service server then returns receive rights for that port so that the name server can now respond to client name requests. (Up until this time, the service server merely ignored requests on the name server port.) The service server also requests a port destroyed notification so that the name server port never dies, and the same port always names the name server, no matter what task is registered as the name server at any given time.

The service server also provides a *service_server*→**service_waitfor** function that waits until the named server is registered.