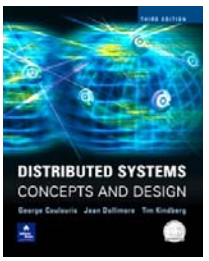


CHAPTER CDK3–18

Originally published as pp. 699-721 of Coulouris, Dolllimore and Kindberg, Distributed Systems, Edition 3, 2001.

MACH CASE STUDY

This chapter describes the design of the Mach microkernel. Mach runs on both multiprocessor and uniprocessor computers connected by networks. It was designed to allow new distributed systems to evolve while maintaining UNIX compatibility.

Mach's is a seminal design. It incorporates, in particular, sophisticated interprocess communication and virtual memory facilities. We describe the Mach architecture and discuss all of its major abstractions: threads and tasks (processes); ports (communication endpoints); and virtual memory, including its role in efficient inter-task communication and support for object sharing through paging between computers.

18.1 Introduction

The Mach project [Acetta *et al*. 1986, Loepere 1991, Boykin *et al*. 1993] was based at Carnegie-Mellon University in the USA until 1994. Its development into a real-time kernel continued there [Lee *et al*. 1996], and groups at the University of Utah and the Open Software Foundation continued its development. The Mach project was successor to two other projects, RIG [Rashid 1986] and Accent [Rashid and Robertson 1981, Rashid 1985, Fitzgerald and Rashid 1986]. RIG was developed at the University of Rochester in the 1970s, and Accent was developed at Carnegie-Mellon during the first half of the 1980s. In contrast to its RIG and Accent predecessors, the Mach project never set out to develop a complete distributed operating system. Instead, the Mach kernel was developed to provide direct compatibility with BSD UNIX. It was designed to provide advanced kernel facilities that would complement those of UNIX and allow a UNIX implementation to be spread across a network of multiprocessor and single-processor computers. From the beginning, the designers' intention was for much of UNIX to be implemented as user-level processes.

Despite these intentions, Mach version 2.5, the first of the two major releases, included all the UNIX compatibility code inside the kernel itself. It ran on SUN-3s, the IBM RT PC, multiprocessor and uniprocessor VAX systems, and the Encore Multimax and Sequent multiprocessors, among other computers. From 1989, Mach 2.5 was incorporated as the base technology for OSF/1, the Open Software Foundation's rival to System V Release 4 as the industry-standard version of UNIX. An older version of Mach was used as a basis for the operating system for the NeXT workstation.

The UNIX code was removed from the version 3.0 Mach kernel, however, and it is this version that we describe. Most recently, Mach 3.0 is the basis of the implementation of MkLinux, a variant of the Linux operating system running on Power Macintosh computers [Morin 1997]. The version 3.0 Mach kernel also runs on Intel x86-based PCs. It ran on the DECstation 3100 and 5000 series computers, some Motorola 88000-based computers and SUN SPARCStations; ports were undertaken for IBM's RS6000, Hewlett-Packard's Precision Architecture and Digital Equipment Corporation's Alpha.

Version 3.0 Mach is a basis for building user-level emulations of operating systems, database systems, language run-time systems and other items of system software that we call subsystems (Figure 18.1).

The emulation of conventional operating systems makes it possible to run existing binaries developed for them. In addition, new applications for these conventional operating systems can be developed. At the same time, middleware and applications that take advantage of the benefits of distribution can be developed; and the implementations of the conventional operating systems can also be distributed. Two important issues arise for operating system emulations. First, distributed emulations cannot be entirely accurate, because of the new failure modes that arise with

Figure 18.1 Mach supports operating systems, databases and other subsystems

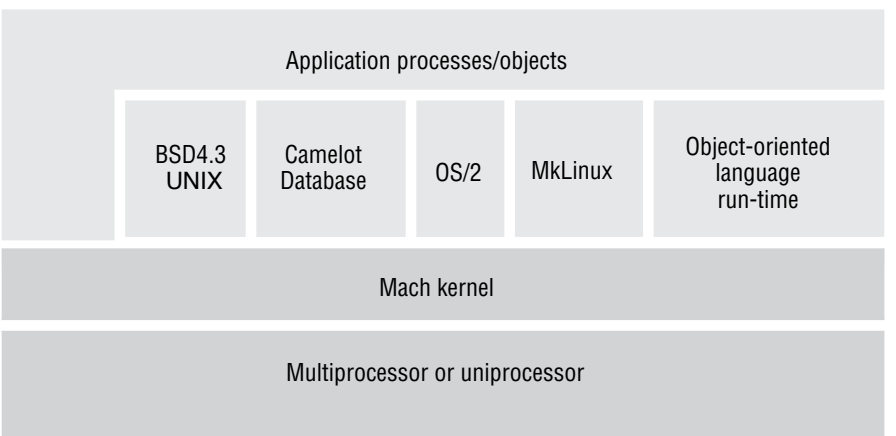
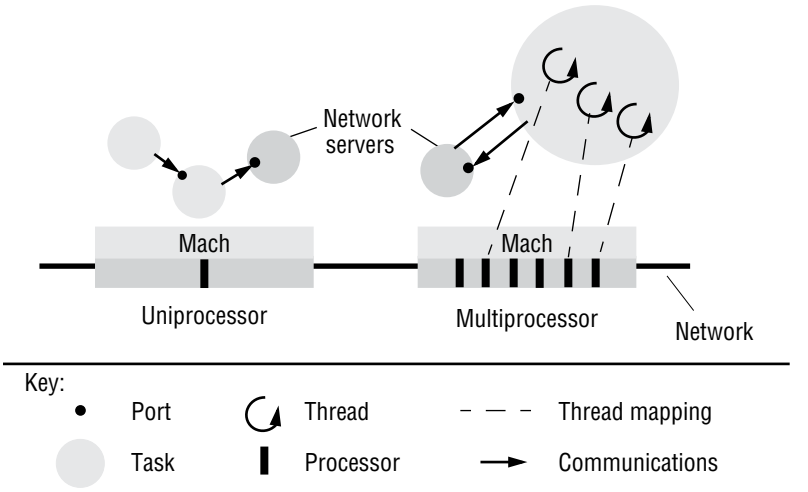


Figure 18.2 Mach tasks, threads and communication



distribution. Second, the question is still open of whether acceptable performance levels can be achieved for widespread use.

18.1.1 Design goals and chief design features

The main Mach design goals and features are as follows:

Multiprocessor operation: Mach was designed to execute on a shared memory multiprocessor so that both kernel threads and user-mode threads could be executed by any processor. Mach provides a multi-threaded model of user processes, with execution environments called *tasks*. Threads are pre-emptively scheduled, whether they belong to the same tasks or to different tasks, to allow for parallel execution on a shared-memory multiprocessor.

Transparent extension to network operation: In order to allow for distributed programs that extend transparently between uniprocessors and multiprocessors across a network, Mach has adopted a location-independent communication model involving ports as destinations. The Mach kernel, however, is designed to be 100% unaware of networks. The Mach design relies totally on user-level network server processes to ferry messages transparently across the network (Figure 18.2). This is a controversial design decision, given the costs of context switching that we examined in Chapter 6. However, it allows for absolute flexibility in the control of network communication policy.

User-level servers: Mach implements an object-based model in which resources are managed either by the kernel or by dynamically loaded servers. Originally, only user-level servers were allowed but later Mach was adapted to accommodate servers within the kernel's address space. As we have mentioned, a primary aim was for most UNIX facilities to be implemented at user level, while providing binary compatibility with existing UNIX. With the exception of some kernel-managed resources, resources are accessed uniformly by message passing, however they are managed. To every resource, there corresponds a port managed by a server. The *Mach Interface Generator* (MiG) was developed to generate RPC stubs used to hide message-based accesses at the language level [Draves *et al.* 1989].

Operating system emulation: To support the binary-level emulation of UNIX and other operating systems, Mach allows for the transparent redirection of operating system calls to emulation library calls and thence to user-level operating system servers – a technique known as *trampolining*. It also includes a facility that allows exceptions such as address space violations arising in application tasks to be handled by servers. Case studies of UNIX emulation under Mach and Chorus can be found at www.cdk3.net/oss.

Flexible virtual memory implementation: Much effort was put into providing virtual memory enhancements that would equip Mach for UNIX emulation and for supporting other subsystems. This included taking a flexible approach to the layout of a process's address space. Mach supports a large, sparse process address space, potentially containing many regions. Both messages and open files, for example, can appear as virtual memory regions. Regions can be private to a task, shared between tasks or copied from regions in other tasks. The design includes the use of memory mapping techniques, notably copy-on-write, to avoid copying data when, for example, messages are passed between tasks. Finally, Mach was designed to allow servers, rather than the kernel itself, to implement backing storage for virtual memory pages. Regions can be mapped to data managed by servers called *external pagers*. Mapped data can reside in any generalized abstraction of a memory resource such as distributed shared memory, as well as in files.

Portability: Mach was designed to be portable to a variety of hardware platforms. For this reason, machine-dependent code was isolated as far as possible. In particular, the virtual memory code was divided between machine-independent and machine-dependent parts [Rashid *et al*. 1988].

18.1.2 Overview of the main Mach abstractions

We can summarize the abstractions provided by the Mach kernel as follows (these will be described in detail later in this chapter):

Tasks: A Mach task is an execution environment. This consists primarily of a protected address space, and a collection of *kernel-managed* capabilities used for accessing ports.

Threads: Tasks can contain multiple threads. The threads belonging to a single task can execute in parallel at different processors in a shared-memory multiprocessor.

Ports: A port in Mach is a unicast, unidirectional communication channel with an associated message queue. Ports are not accessed directly by the Mach programmer and are not part of a task. Rather, the programmer is given handles to *port rights*. These are capabilities to send messages to a port or receive messages from a port.

Port sets: A port set is a collection of port receive rights local to a task. It is used to receive a message from any one of a collection of ports. Port sets should not be confused with port *groups*, which are multicast destinations but are not implemented in Mach.

Messages: A message in Mach can contain port rights in addition to pure data. The kernel employs memory management techniques to transfer message data efficiently between tasks.

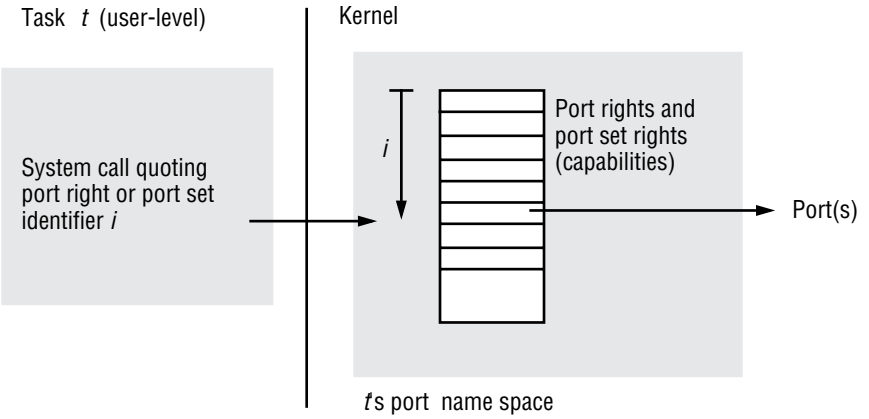
Devices: Servers such as file servers running at user level must access devices. The kernel exports a low-level interface to the underlying devices for this purpose.

Memory object: Each region of the virtual address space of a Mach task corresponds to a memory object. This is an object that in general is implemented outside the kernel itself but is accessed by the kernel when it performs virtual memory paging operations. A memory object is an instance of an abstract data type that includes operations to fetch and store data that are accessed when threads give rise to page-faults in attempting to reference addresses in the corresponding region.

Memory cache object: For every mapped memory object, there is a kernel-managed object that contains a cache of pages for the corresponding region that are resident in main memory. This is called a memory cache object. It supports operations needed by the external pager that implements the memory object.

We shall now consider the main abstractions. The abstraction of devices is omitted in the interests of brevity.

Figure 18.3 A task's port name space



18.2 Ports, naming and protection

Mach identifies individual resources with ports. To access a resource, a message is sent to the corresponding port. The Mach assumption is that servers will in general manage many ports: one for every resource. A single-server UNIX system uses about 2000 ports [Draves 1990]. Ports therefore have to be cheap to create and manage.

The problem of protecting a resource from illegal accesses amounts to that of protecting the corresponding port against illegal sends. This is achieved in Mach by kernel control over the acquisition of capabilities for the port, and also by network server control over messages that arrive over the network.

The capability to a port has a field specifying the port access rights belonging to the task that holds it. There are three different types of port rights. *Send rights* allow the threads in the task that possesses them to send messages to the corresponding port. A restricted form of these, *send-once rights*, allow at most one message to be sent, after which the rights are automatically destroyed by the kernel. This restricted form allows, for example, a client to obtain a reply from a server in the knowledge that the server can no longer send a message to it (thus protecting it from buggy servers); in addition, the server is spared the expense of garbage collecting send rights received from clients. Finally, *receive rights* allow a task's threads to receive messages from the port's message queue. At most one task may possess receive rights at any one time, whereas any number of tasks may possess send rights or send-once rights. Mach supports only *N*-to-one communication: multicast is not supported directly by the kernel.

At creation, a task is given a *bootstrap port right*, which is a send right that it uses to obtain the services of other tasks. After creation, the threads belonging to the task acquire further port rights either by creating ports, or by receiving port rights sent to them in messages.

Mach's port rights are stored inside the kernel and protected by it (Figure 18.3). Tasks refer to port rights by local identifiers, which are valid only in the task's local *port name space*. This allows the kernel's implementors to choose efficient representations for these capabilities (such as pointers to message queues), and to choose integer local names that are convenient for the kernel in looking up the capability from the name. In fact, like UNIX file descriptors, local identifiers are integers used to index a kernel table containing the task's capabilities.

The Mach naming and protection scheme thus allows rapid access to local message queues from a given user-level identifier. Against this advantage, we must set the expense of kernel processing whenever rights are transmitted in messages between tasks. At the very least, send rights have to be allocated a local name in the recipient task's name space and space in its kernel tables. And we note that, in a secure environment, the transmission of port rights by the network servers requires encryption of those rights to guard against forms of security attack such as eavesdropping [Sansom *et al.* 1986].

18.3 Tasks and threads

A task is an execution environment: tasks themselves cannot perform any actions; only the threads within them can. However, for convenience we shall sometimes refer to a task performing actions when we mean a thread within the task. The major resources associated directly with a task are its address space, its threads, its port rights, port sets and the local name space in which port rights and port sets are looked up. We shall now examine the mechanism for creating a new task, and the features related to the management of tasks and the execution of their constituent threads.

Creating a new task ◊ The UNIX *fork* command creates a new process by copying an existing one. Mach's model of process creation is a generalization of the UNIX model. Tasks are created with reference to what we shall call a *blueprint task* (which need not be the creator). The new task resides at the same computer as the blueprint task. Since Mach does not provide a task migration facility, the only way to establish a task at a remote computer is via a task that already resides there. The new task's bootstrap port right is inherited from its blueprint, and its address space is either empty or is inherited from its blueprint (address space inheritance is discussed in the subsection on Mach virtual memory below). A newly created task has no threads. Instead, the task's creator requests the creation of a thread within the child task. Thereafter, further threads can be created by existing threads within the task. See Figure 18.4 for some of the Mach calls related to task and thread creation.

Invoking kernel operations ◊ When a Mach task or thread is created, it is assigned a so-called *kernel port*. Mach 'system calls' are divided into those implemented directly as kernel traps and those implemented by message passing to kernel ports. The latter method has the advantage of allowing network-transparent operations on remote tasks and threads as well as local ones. A kernel service manages kernel resources in the same way that a user-level server manages other resources. Each task has send rights to its kernel port, which enables it to invoke operations upon itself (such as to create a new thread). Each of the kernel services accessed by message passing has an interface definition. Tasks access these services via stub procedures, which are generated from their interface definitions by the Mach Interface Generator.

Exception handling ◊ In addition to a kernel port, tasks and (optionally) threads can possess an *exception port*. When certain types of exception occur, the kernel responds by attempting to send a message describing the exception to an associated exception port. If there is no exception port for the thread, the kernel looks for one for the task. The thread that receives this message can attempt to fix the problem (it might, for example, grow the thread's stack in response to an address space violation), and it then returns a status value in a reply message. If the kernel finds an exception port and receives a reply indicating success, it then restarts the thread that raised the exception. Otherwise, the kernel terminates it.

For example, the kernel sends a message to an exception port when a task attempts an address space access violation or to divide by zero. The owner of the exception port could be a debugging task, which could execute anywhere in the network by virtue of Mach's location-independent communication. Page faults are handled by external pagers. Section 18.4 describes how Mach handles system calls directed to an emulated operating system.

Task and thread management ◊ About forty procedures in the kernel interface are concerned with the creation and management of tasks and threads. The first argument of each procedure is a send right to the corresponding kernel port, and message passing system calls are used to request the operation of the target kernel. Some of these task and thread calls are shown in Figure 18.4. In summary, thread scheduling priorities can be set individually; threads and tasks can be suspended, resumed and terminated; and the execution state of threads can be externally set, read and modified. The latter facility is important for debugging and also for setting up software interrupts. Yet more kernel interface calls are concerned with the allocation of a task's threads to particular *processor sets*. A processor set is a subset of processors in a multiprocessor. By assigning threads to processor sets, the available computational resources can be crudely divided between different types of activity. The reader is referred to Loepere [1991] for details of kernel support for task and

Figure 18.4 Task and thread creation

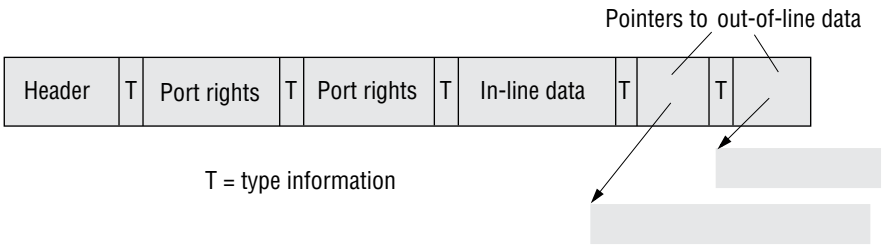
task_create(parent_task, inherit_memory, child_task)
parent_task is the task used as a blueprint in the creation of the new task, *inherit_memory* specifies whether the child should inherit the address space of its parent or be assigned an empty address space, *child_task* is the identifier of the new task.

thread_create(parent_task, child_thread)
parent_task is the task in which the new thread is to be created, *child_thread* is the identifier of the new thread. The new thread has no execution state and is suspended.

thread_set_state(thread, flavour, new_state, count)
thread is the thread to be supplied with execution state, f*lavour* specifies the machine architecture, *new_state* specifies the state (such as the program counter and stack pointer), *count* is the size of the state.

thread_resume(thread)
This is used to resume the suspended thread identified by *thread*.

Figure 18.5 A Mach message containing port rights and out-of-line data



thread management and processor allocation. Tokuda *et al*. [1990] and Lee *et al*. [1996] describe extensions to Mach for real-time thread scheduling and synchronization.

18.4 Communication model

Mach provides a single system call for message passing: *mach_msg*. Before describing this, we shall say more about messages and ports in Mach.

18.4.1 Messages

A message consists of a fixed-size header followed by a variable-length list of data items (Figure 18.5).

The fixed-size header contains:

The destination port: For simplicity, this is part of the message rather than being specified as a separate parameter to the *mach_msg* system call. It is specified by the local identifier of the appropriate send rights.

A reply port: If a reply is required, then send rights to a local port (that is, one for which the sending thread has receive rights) are enclosed in the message for this purpose.

An operation identifier: This identifies an operation (procedure) in the service interface and is meaningful only to applications.

Extra data size: Following the header (that is, contiguous with it) there is, in general, a variable-sized list of typed items. There is no length limit to this, except the number of bits in this field and the total address space size.

Each item in the list after the message header is one of the following (which can occur in any order in the message):

Typed message data: individual, in-line type-tagged data items;

Port rights: referred to by their local identifiers;

Pointers to out-of-line data: data held in a separate non-contiguous block of memory.

Mach messages consist of a fixed-size header and multiple data blocks of variable sizes, some of which may be *out of line* (that is, non-contiguous). However, when out-of-line message data are sent, the kernel – not the receiving task – chooses the location in the receiving task's address space of the received data. This is a side effect of the copy-on-write technique used to transfer this data. Extra virtual memory regions received in a message must be de-allocated explicitly by the receiving task if they are no longer required. Since the costs of virtual memory operations outweigh those of data copying for small amounts of data, it is intended that only reasonably large amounts of data are sent out of line.

The advantage of allowing several data components in messages is that this allows the programmer to allocate memory separately for data and for metadata. For example, a file server might locate a requested disk block from its cache. Instead of copying the block into a message buffer, contiguously with header information, the data can be fetched directly from where they reside by providing an appropriate pointer in the reply message. This is a form of what is known as *scatter-gather I/O*, wherein data is written to or read from multiple areas of the caller's address space in one system call. The UNIX *readv* and *writev* system calls also provide for this [Leffler *et al*. 1989].

The type of each data item in a Mach message is specified by the sender (as, for example, in ASN.1). This enables user-level network servers to marshal the data into a standard format when they are transmitted across a network. However, this marshalling scheme has performance disadvantages compared with marshalling and unmarshalling performed by stub procedures generated from interface definitions. Stub procedures have common knowledge of the data types concerned, need not include these types in the messages and may marshal data directly into the message (See Section 4.2). A network server may have to copy the sender's typed data into another message as it marshals them.

18.4.2 Ports

A Mach port has a message queue whose size can be set dynamically by the task with receive rights. This facility enables receivers to implement a form of flow control. When a normal send right is used, a thread attempting to send a message to a port whose message queue is full will be blocked until room becomes available. When a thread uses a send-once right, the recipient always queues the message, even if the message queue is full. Since a send-once right is used, it is known that no further messages can be sent from that source. Server threads can avoid blocking by using send-once rights when replying to clients.

Sending port rights ◊ When port send rights are enclosed in a message, the receiver acquires send rights to the same port. When receive rights are transmitted, they are automatically de-allocated in the sending task. This is because receive rights cannot be possessed by more than one task at a time. All messages queued at the port and all subsequently transmitted messages can be received by the new owner of receive rights, in a manner that is transparent to tasks sending to the port. The transparent transfer of receive rights is relatively straightforward to achieve when the rights are transferred within a single computer. The acquired capability is simply a pointer to the local message queue. In the inter-computer case, however, a number of more complex design issues arise. These are discussed below.

Monitoring connectivity ◊ The kernel is designed to inform senders and receivers when conditions arise under which sending or receiving messages would be futile. For this purpose, it keeps information about the number of send and receive rights referring to a given port. If no task holds receive rights for a particular port (for example, because the task holding these rights failed), then all send rights in local tasks' port name spaces become *dead names*. When a sender attempts to use a name referring to a port for which receive rights no longer exist, the kernel turns the name into a dead name and returns an error indication. Similarly, tasks can request the kernel to notify them asynchronously of the condition that no send rights exist for a specified port. The kernel performs this notification by sending the requesting thread a message, using send rights given to it by the thread for this purpose. The condition of no send rights can be ascertained by keeping a reference count that is incremented whenever a send right is created and decremented when one is destroyed.

It should be stressed that the conditions of no senders/no receiver are tackled within the domain of a single kernel at relatively little cost. Checking for these conditions in a distributed system is, by contrast, a complex and expensive operation. Given that rights can be sent in messages, the send or receive rights for a given port could be held by any task, or even be in a message, queued at a port or in transit between computers.

Port sets ◊ Port sets are locally managed collections of ports that are created within a single task. When a thread issues a receive from a port set, the kernel returns a message that was delivered to some member of the set. It also returns the identifier of this port's receive rights so that the thread can process the message accordingly.

Ports sets are useful because typically a server is required to service client messages at all of its ports at all times. Receiving a message from a port whose message queue is empty blocks a thread, even if a message that it could process arrives on another port first. Assigning a thread to each port overcomes this problem but is not feasible for servers with large numbers of ports because a thread is a more expensive resource than a port. By collecting ports into a port set, a single thread can be used to service incoming messages without fear of missing any. Furthermore, this thread will block if no messages are available on any port in the set, so avoiding a busy-waiting solution in which the thread polls until a message arrives on one of the ports.

18.4.3 Mach_msg

The *Mach_msg* system call provides for both asynchronous message passing and request-reply-style interactions, which makes it extremely complicated. We shall give only an overview of its semantics. The complete call is as follows:

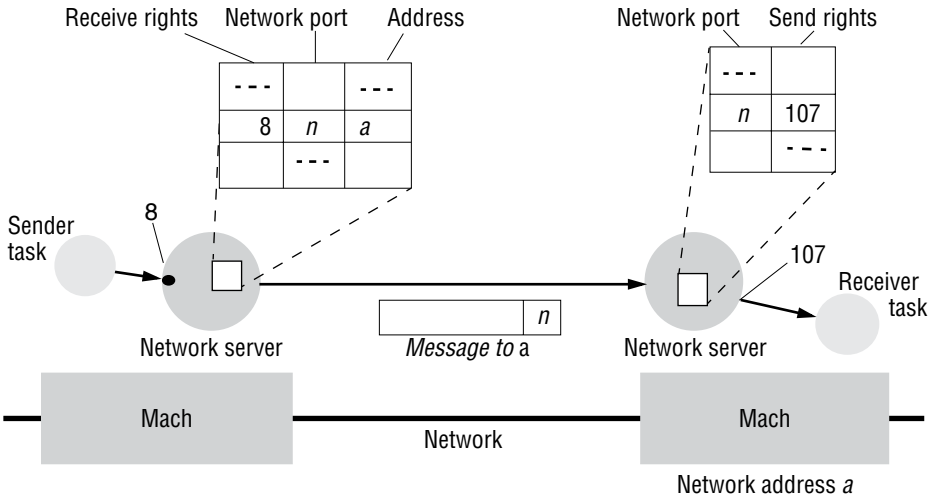
---

mach_msg(msg_header, option, snd_siz, rcv_siz, rcv_name, timeout, notify)
msg_header points to a common message header for the sent and received messages, *option* specifies send, receive or both, *snd_siz* and *rcv_siz* give the sizes of the sent and received message buffers, *rcv_name* specifies the port or port set receive rights (if a message is received), *timeout* sets a limit to the total time to send and/or receive a message, *notify* supplies port rights which the kernel is to use to send notification messages under exceptional conditions.

Mach_msg either sends a message, receives a message, or both. It is a single system call that clients use to send a request message and receive a reply, and servers use to reply to the last client and receive the next request message. Another benefit of using a combined send/receive call is that in the case of a client and server executing at the same computer the implementation can employ an optimization called *handoff scheduling*. This is where a task about to block after sending a message to another task 'donates' the rest of its timeslice to the other task's thread. This is cheaper than going through the queue of RUNNABLE threads to select the next thread to run.

Messages sent by the same thread are delivered in sending order, and message delivery is reliable. At least, this is guaranteed where messages are sent between tasks hosted by a common kernel – even in the face of lack of buffer space. When messages are transmitted across a network to a failure-independent computer, at-most-once delivery semantics are provided.

Figure 18.6 Network communication in Mach



The timeout is useful for situations in which it is undesirable for a thread to be tied up indefinitely, for example awaiting a message that might never arrive, or waiting for queue space at what turns out to be a buggy server's port.

18.5 Communication implementation

One of the most interesting aspects of the Mach communication implementation is the use of user-level network servers. The network servers (called *netmsgservers* in the Mach literature), one per computer, are collectively responsible for extending the semantics of local communication across the network. This includes preserving, as far as possible, delivery guarantees and making network communication transparent. It also includes effecting and monitoring the transfer of port rights. In particular, the network servers are responsible for protecting ports against illegal access, and for maintaining the privacy of message data across the network. Full details of Mach's treatment of protection issues are available in Sansom *et al*. [1986].

18.5.1 Transparent message delivery

Since ports are always local to a Mach kernel, it is necessary to add an externally imposed abstraction of *network port*, to which messages can be addressed across the network. A network port is a globally unique channel identifier that is handled only by the network servers and is associated by them with a single Mach port at any one time. Network servers possess send and receive rights to network ports in the same way that tasks possess send and receive rights to Mach ports.

The transmission of a message between tasks located at different computers is shown in Figure 18.6. The rights held by the sender task are to a local port, for which receive rights are held by the local network server. In the figure, the network server's local identifier for the receive rights is 8. The network server at the sender's computer looks up an entry for the rights identifier in a table of network ports for which it has send rights. This yields a network port and a network address hint. It sends the message, with the network port attached, to the network server at the address indicated in the table. There the local network server extracts the network port and looks this up in a table of network ports for which it has receive rights. If it finds a valid entry there (the network port might have been relocated to another kernel), then this entry contains the identifier of send rights to a local Mach port. This network server forwards the message using these rights, and the

message is thus delivered to the appropriate port. The whole process of handling by the network servers is transparent to both the sender and the receiver.

How are the tables shown in Figure 18.6 set up? The network server of a newly booted computer engages in an initialization protocol, whereby send rights are obtained to network-wide services. Consider what happens thereafter, when a message containing port rights is transferred between network servers. These rights are typed and therefore can be tracked by the network servers. If a task sends a local port's send rights, the local network server creates a network port identifier and a table entry for the local port – if none exists – and attaches the identifier to the message it forwards. The receiving network server also sets up a table entry if none exists.

When receive rights are transmitted, the situation is more complicated. This is an example in which migration transparency is required: clients must be able to send messages to the port while it migrates. First, the network server local to the sender acquires the receive rights. All messages destined for the port, local and remote, start arriving at this server. It then engages in a protocol whereby the receive rights are consistently transferred to the destination network server.

The main issue concerning this transfer of rights is how to arrange that messages sent to the port now arrive at the computer to which receive rights have been transferred. One possibility would be for Mach to keep track of all network servers possessing send rights to a given network port, and to notify these servers directly when receive rights were transferred. This scheme was rejected as being too expensive to manage. A cheaper alternative would have been to use a hardware broadcast facility to broadcast the change of location to all network servers. However, such a broadcast service is not reliable, and hardware broadcast is not available on all networks. Instead, responsibility for locating a network port was placed upon the network servers that hold send rights to the port.

Recall that a network server uses a location hint when it forwards a message to another network server. The possible responses are:

port here: the destination holds receive rights;

port dead: the port is known to have been destroyed;

port not here, transferred: receive rights were transferred to a specified address;

port not here, unknown: there is no record of the network port;

no response: the destination computer is dead.

If a forwarding address is returned, the sending network server forwards the message; but this in turn is only a hint and might be inaccurate. If at some point the sender runs out of forwarding addresses, then it resorts to broadcasting. How to manage chains of forwarding addresses and what to do when a computer holding a forwarding address crashes are both major design issues for this type of location algorithm, particularly over a WAN. Use of forwarding addresses over a WAN is described in Black and Artsy [1990].

A second issue in achieving migration transparency is how to synchronize message delivery. Mach guarantees that two messages sent by the same thread are delivered in the same order. How can this be guaranteed while receive rights are being transmitted? If care is not taken, a message could be delivered by the new network server before a prior message queued at the original computer was forwarded. The network server can achieve this by holding off delivery of all messages at the original computer until any queued messages have been transferred to the destination computer. Message delivery can thereafter be rerouted safely, and the forwarding address returned to senders.

18.5.2 Openness: protocols and drivers

Transport protocols ◊ Despite the intention that a range of transport protocols should be accommodated by running the network servers in user space, at the time of writing, Mach's network servers in widespread use employ only TCP/IP as the transport protocol. This was prompted in part by UNIX compatibility, and in part it was selected by Carnegie-Mellon University because of its complex network containing over 1700 computers, about 500 of which run Mach. TCP/IP is tuned to achieve robustness fairly efficiently in the face of such a network.

However, this is not necessarily suitable on LANs when request-reply interactions predominate, for performance reasons. We discuss the performance of Mach communication in Section 18.6 below.

User-level network drivers ◊ Some network servers provide their own, user-level network device drivers. The aim of this is to speed up network accesses. Apart from achieving flexibility in relation to using a range of hardware, placing device drivers at user level is largely a means of compensating for the performance degradation due to using a user-level network server. The kernel exports an abstraction of each device, which includes an operation to map the device controller's registers into user space. In the case of an Ethernet, both the registers and the packet buffers used by the controller can be mapped into the network server's address space. In addition, special code is run in the kernel to wake up a user-level thread (belonging, in this case, to the network server) when an interrupt occurs. This thread is thus able to handle the interrupt by transferring data to or from the buffers used by the controller and resetting the controller for the next operation.

18.6 Memory management

Mach is notable not only for its use of large, sparse address spaces but also for its virtual memory techniques allowing for memory sharing between tasks. Not only can memory be shared physically between tasks executing on the same Mach kernel but Mach's support for external pagers (called *memory managers* in the Mach literature) also allows for the contents of virtual memory to be shared between tasks, even when they reside at different computers. Lastly, the Mach virtual memory implementation is notable for being divided into machine-independent and machine-dependent layers to aid in porting the kernel [Rashid *et al.* 1988].

18.6.1 Address space structure

In Chapter 6, we introduced a generic model of an address space consisting of regions. Each region is a range of contiguous logical addresses with a common set of properties. These properties include access permissions (read/write/execute), and also extensibility. Stack regions, for example, are allowed to grow towards decreasing addresses; and heaps can grow upwards. The model used by Mach is similar.

However, the Mach view of address spaces is that of a collection of contiguous groups of pages named by their addresses, rather than of regions that are separately identifiable. Thus protection in Mach is applied to pages rather than regions. Mach system calls refer to addresses and extents rather than to region identifiers. For example, Mach would not 'grow the stack region'. Instead, it would allocate some more pages just below those that are currently used for the stack. However, for the most part this distinction is not too important.

We shall refer to a contiguous collection of pages with common properties as a region. As we have seen, Mach supports large numbers of regions, which can be used for varying purposes such as message data or mapped files.

Regions can be created in any of four ways:

- They can be allocated explicitly by a call to *vm_allocate*. A region newly created with *vm_allocate* is, by default, zero-filled.
- A region can be created in association with a *memory object*, using *vm_map*.
- Regions can be assigned in a brand new task by declaring them (or rather them and their contents) to be inherited from a blueprint task, using *vm_inherit* applied to the blueprint's region.
- Regions can be allocated automatically in a task's address space as a side effect of message passing.

All regions can have their read/write/execute permissions set, using *vm_protect*. Regions can be copied within tasks using *vm_copy*, and their contents can be read or written by other tasks using *vm_read* and *vm_write*. Any previously allocated region can be de-allocated, using *vm_deallocate*.

We shall now describe Mach's approach to the implementation of the UNIX *fork* operation and the virtual memory aspects of message passing, which are incorporated in Mach so as to enable memory sharing to take place wherever convenient.

18.6.2 Memory sharing: inheritance and message passing

Mach allows for a generalization of UNIX *fork* semantics through the mechanism of *memory inheritance*. We have seen that a task is created from another task, which acts as a blueprint. A region that is inherited from the blueprint task contains the same address range, and its memory is either:

shared: backed by the same memory; or

copied: a 'copy-inherited' region is one that is backed by memory that is a copy of the blueprint's memory at the time the child region was created.

It is also possible, in case a region is not required in the child, to set it to be non-inherited.

In the case of a UNIX *fork*, the program text of the blueprint task is set to be inherited for sharing by the child task. The same would be true of a region containing shared library code. The program heap and stack, however, would be inherited as copies of the blueprint's regions. In addition, if the blueprint is required to share a data region with its child (as is allowed by System V UNIX), then it could set up this region to be inherited for sharing.

Out-of-line message data is transferred between tasks in a way that is somewhat similar to copy-inheritance. A region is created by Mach in the receiver's address space, and its initial contents are a copy of the region passed as out-of-line data by the sender. Unlike inheritance, the received region does not in general occupy the same address range as the sent region. The address range of the sent region might already be used by an existing region in the receiver.

Mach uses copy-on-write for both copy inheritance and message passing. Mach thus makes an optimistic assumption: that some or all the memory that is copy-inherited or passed as out-of-line message data will not be written by either task, even when write permissions exist.

To justify this optimistic assumption, consider once again, for example, the UNIX *fork* system call. It is common for a *fork* operation to be followed soon by an *exec* call, overwriting the address space contents, including the writable heap and stack. If memory had been physically copied at the time of the fork, then most of this copying would have been wasted: few pages are modified between the two calls.

As a second important example, consider a message sent to the local network server for transmission. This message might be very large. If, however, the sending task does not modify the message, or modifies only parts of it, in the time taken to transmit the message, then much memory copying can be saved. The network server has no reason to modify the message. The copy-on-write optimization helps to offset the context switching costs incurred in transmitting via the network server.

By contrast to copy-on-write, Chorus and DASH [Tzou and Anderson 1991] support communication by *moving* (not copying) pages between address spaces. The Fbufs design exploits virtual memory manipulations for both copying and moving pages [Druschel and Peterson 1993].

18.6.3 Evaluation of copy-on-write

While copy-on-write assists in passing message data between tasks and the network server, copy-on-write cannot be used to facilitate transmission across a network. This is because the computers involved do not share physical memory.

Copy-on-write is normally efficient as long as sufficient data are involved. The advantage of avoiding physical copying has to outweigh the costs of page table manipulations (and cache manipulations, if the cache is virtually mapped – see Section 6.3). Figures for a Mach implementation on a Sun 3/60 are given by Abrossimov *et al*. [1989], and we reproduce some of

Figure 18.7 Copy-on-write overheads

Region size	*Simple copy*	*Create region*	*Amount of data copied (on writing)*		
			0 kilobytes (0 pages)	*8 kilobytes (1 page)*	*256 kilobytes (32 pages)*
8 kilobytes	1.4	1.57	2.7	4.82	–
256 kilobytes	44.8	1.81	2.9	5.12	66.4

Note: all times are in milliseconds.

them in Figure 18.7. The figures are given for illustration only and do not necessarily represent the most up-to-date performance of Mach.

Times for regions of two sizes, 8 kilobytes (1 page) and 256 kilobytes (32 pages), are given. The first two columns are for reference only. The column 'Simple copy' shows the time it would take simply to copy all the data involved between two pre-existing regions (that is, without using copy-on-write). The column 'Create region' gives the time required to create a zero-filled region (but one that is not accessed). The remaining columns give measured times taken from experiments in which a pre-existing region was copied into another region using copy-on-write. The figures include the time for creating the copy region, copying data on modification and destroying the copy region. For each region size, figures are given for cases of different amounts of data modified in the source region.

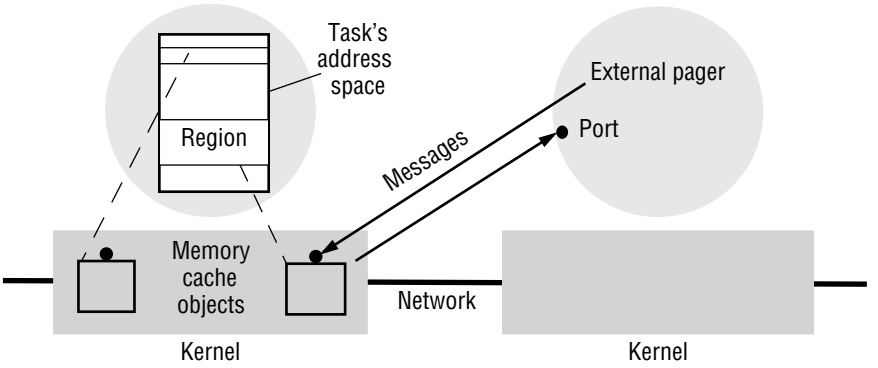
If we compare the times for a one-page region, there is an overhead of $4.82 - 2.7 = 2.12$ milliseconds due to the page being written, 1.4 milliseconds of which can be ascribed to copying the page; the remaining 0.72 milliseconds is taken up by write-fault handling and modification of internal virtual memory management data structures. For sending a message of size 8 kilobytes between two local tasks, sending the data out of line (that is, using copy-on-write) seems to be of dubious advantage over sending it in-line, when it will be simply copied. In-line transfer would involve two copies (user-kernel and kernel-user), and therefore would take somewhat more than 2.8 milliseconds in all. But the worst out-of-line case is significantly more expensive. On the other hand, transmitting a 256 kilobyte message out of line is far less expensive than in-line transmission if the optimistic assumption holds; and even in the worst case, 66.4 milliseconds is less than the 2×44.8 milliseconds required to copy 256 kilobytes of in-line data into and out of the kernel.

As a final point regarding virtual memory techniques for copying and sharing data, note that care has to be taken over specifying the data involved. Although we did not state this above, a user can specify regions as address ranges that do not begin and end on a page boundary. Mach, however, is forced to apply memory sharing at the granularity of a page. Any data that are within the pages concerned but were not specified by the user will nonetheless be copied between the tasks. This is an example of what is known as *false sharing* (see also the discussion of the granularity of shared data items in Chapter 16).

18.6.4 External pagers

In keeping with the microkernel philosophy, the Mach kernel does not support files or any other abstraction of external storage directly. Instead, it assumes that these resources are implemented by external pagers. Following Multics [Organick 1972], Mach has chosen the mapped access model for memory objects. Instead of accessing stored data using explicit *read* and *write* operations, the programmer is required only to access corresponding virtual memory locations directly. An advantage of mapped access is its uniformity: the programmer is presented with one model for access to data, not two. However, the question of whether mapped access is preferable to using explicit operations is complex in its ramifications, especially as regards performance, and we shall not attempt to deal with it here. We shall concentrate now on the distributed aspects of the

Figure 18.8 External pager



Mach virtual memory implementation. This consists primarily of the protocol between the kernel and an external pager that is necessary to manage the mapping of data stored by the latter.

Mach allows a region to be associated with contiguous data from a specified offset in a memory object, using a call to *vm_map*. This association means that read accesses to addresses in the region are satisfied by data backed by the memory object, and data in the region modified by write accesses are propagated back to the memory object. In general, the memory object is managed by an external pager, although a default pager may be supplied, implemented by the kernel itself. The memory object is represented by send rights to a port used by the external pager, which satisfies requests from the kernel concerning the memory object.

For each memory object mapped by it, the kernel keeps a local resource called a *memory cache object* (Figure 18.8). Essentially, this is a list of pages containing data backed by the corresponding memory object.

The roles of an external pager are (a) to store data that have been purged by a kernel from its cache of pages, (b) to supply page data as required by a kernel, and (c) to impose consistency constraints pertaining to the underlying memory resource abstraction in the case where the memory resource is shared and several kernels can hold memory cache objects for the same memory object simultaneously.

The main components of the message passing protocol between the kernel (K) and external pager (EP) are summarized in Figure 18.9. When *vm_map* is called, the local kernel contacts the external pager using the memory object port send right supplied to it in the *vm_map* call, sending it a message *memory_object_init*. The kernel supplies send rights in this message, which the external pager is to use to control the memory cache object. It also declares the size and offset of the required data in the memory object, and the type of access required (read/write). The external pager responds with a message *memory_object_set_attributes*, which tells the kernel whether the pager is yet ready to handle data requests, and supplies further information about the pager's requirements in relation to the memory object. When an external pager receives a *memory_object_init* message, it is able to determine whether it needs to implement a consistency protocol, since all kernels wishing to access the corresponding memory object have to send this message.

18.6.5 Supporting access to a memory object

We begin by considering the case in which a memory object is unshared – just one computer maps it. For the sake of concreteness, we can think of the memory object as a file. Assuming no pre-fetching of file data from the external pager, all pages in the mapped region corresponding to this file are initially hardware-protected against all accesses, since no file data is resident. When a thread attempts to read one of the region's pages, a page fault occurs. The kernel looks up the memory object port send right corresponding to the mapped region and sends a *memory_object_data_request* message to the external pager (which is, in our example, a file

Figure 18.9 External pager messages

Event	*Sender*	*Message*
vm_map called by task	K → EP	*memory_object_init*
	EP → K	*memory_object_set_attributes*, or
	EP → K	*memory_object_data_error*
Task page-faults when no data frame exists	K → EP	*memory_object_data_request*
	EP → K	*memory_object_data_provided*, or
	EP → K	*memory_object_data_unavailable*
Kernel writes modified page to persistent store	K → EP	*memory_object_data_write*
External pager directs kernel to write page/set access permissions	EP → K	*memory_object_lock_request*
	K → EP	*memory_object_lock_completed*
Task page-faults when insufficient page access	K → EP	*memory_object_data_unlock*
	EP → K	*memory_object_lock_request*
Memory object no longer mapped	K → EP	*memory_object_terminate*
External pager withdraws memory object	EP → K	*memory_object_destroy*
	K → EP	*memory_object_terminate*

server). If all is well, the external pager responds with the page data, in a *memory_object_data_provided* message.

If the file data is modified by the computer that has mapped it, then sometimes the kernel needs to write the page from its memory cache object. To do this, it sends a message *memory_object_data_write* to the external pager, containing the page data. Modified pages are transmitted to the external pager as a side effect of page replacement (when the kernel needs to find space for another page). In addition, the kernel can decide to write the page to backing store (but leave it in the memory cache object) in order to meet persistence guarantees. Implementations of UNIX, for example, write modified data to disk normally at least every 30 seconds, in case of a system crash. Some operating systems allow programs to control the safety of their data by issuing a *flush* command on an open file, which causes all modified file pages to be written to disk by the time the call returns.

Different types of memory resource can have differing persistence guarantees. The external pager can itself request, in a *memory_object_lock_request* message to a kernel, that modified data in a specified range be sent back to the pager for commitment to permanent storage in accordance with these guarantees. When the kernel has completed the requested actions, it sends a *memory_object_lock_completed* message to the external pager. (The external pager requires this, because it cannot know which pages have been modified and so need to be written back to it.)

Note that all the messages we are describing are sent asynchronously, even if they sometimes occur in request-reply combinations. This is, first, so that threads are not suspended but can get on with other work after issuing requests. Moreover, a thread is not tied up when it has issued a request to an external pager or kernel that turns out to have crashed (or when the kernel sends a request to an ill-behaved external pager that does not reply). Lastly, an external pager can use the asynchronous message-based protocol to implement a page pre-fetching policy. It can send page data in *memory_object_data_provided* messages to memory cache objects in anticipation of the data's use, instead of waiting for a page fault to occur and for the data then to be requested.

Supporting shared access to a memory object ◊ Let us now suppose, following our example, that several tasks residing at different computers map a common file. If the file is mapped read-only in every region used to access it, then there is no consistency problem and requests for file pages can be satisfied immediately. If, however, at least one task maps the file for writing, then the external

pager (that is, the file server) has to implement a protocol to ensure that tasks do not read inconsistent versions of the same page.

The reader is invited to translate the write-invalidate protocol for achieving sequential consistency, as discussed in Chapter 16, into messages sent between the Mach kernel and external pagers.

18.7 Summary

The Mach kernel runs on both multiprocessor and uniprocessor computers connected by networks. It was designed to allow new distributed systems to evolve while maintaining UNIX compatibility. Most recently, the Mach 3.0 microkernel is the basis for the MkLinux implementation of the Linux operating system.

Due in part to the sophistication of some of the facilities Mach is designed to emulate, the kernel itself is complex. The kernel's interface includes several hundred calls, although many of these are stubs which make only *mach_msg* system call traps. An operating system such as UNIX cannot be emulated using message passing alone. Sophisticated virtual memory facilities are required, and Mach provides these. Mach's model of tasks and threads and the integration of virtual memory management with communication all represent a considerable improvement over basic UNIX facilities, incorporating lessons learned in attempting to implement UNIX servers, in particular. Its model of inter-task communication is functionally rich and extremely complex in its semantics. However, it should be borne in mind that only a few system programmers should ever have to use it in its raw form: for example, simple UNIX pipes and remote procedure calls are both provided on top of it.

Although the Mach kernel is often referred to as a microkernel, it is of the order of 500 kilobytes of code and initialized data (including a substantial proportion of device driver code). The so-called second-generation microkernels that have succeeded it supply simpler memory management facilities and simpler inter-process communication.

Second-generation microkernels, with optimized inter-process communication, outstrip Mach's UNIX emulation performance. For example, the designers of the L4 microkernel [Härtig *et al*. 1997] report that their user-level implementation of Linux is within 5–10% of a native Linux implementation on the same machine. By contrast, they report a benchmark in which the MkLinux user-level emulation of Linux on Mach 3.0 provides an average of 50% worse throughput than native Linux on that machine.

The Mach philosophy of providing all extended functionality at user level was eventually dropped, and servers were allowed to be co-located with the kernel [Condict *et al*. 1994]. But Härtig *et al*. report that even the kernel-level MkLinux emulation of Linux is about 30% slower than native Linux.

Despite its performance limitations for the purposes of UNIX emulation, Mach, along with Chorus, is an innovative design that remains important. Both continue to be used, and they are an invaluable reference for developments in kernel architecture. The reader can find further material on kernel design at www.cdk3.net/oss: Amoeba, Chorus and the emulation of UNIX on top of Mach and Chorus.

EXERCISES

18.1 How does a kernel designed for multiprocessor operation differ from one intended to operate only on single-processor computers? *page 3*

18.2 Define (binary-level) operating system emulation. Why is it desirable and, given that it is desirable, why is it not routinely done? *page 3*

18.3 Explain why the contents of Mach messages are typed. *page 7*

18.4 Discuss whether the Mach kernel's ability to monitor the number of send rights for a particular port should be extended to the network. *page 9*

18.5 Why does Mach provide port sets, when it also provides threads? *page 9*

18.6 Why does Mach provide only a single communication system call, *mach_msg*? How is it used by clients and by servers? *page 9*

18.7 What is the difference between a network port and a (local) port? *page 10*

18.8 A server in Mach manages many *thingumajig* resources.

(i) Discuss the advantages and disadvantages of associating:

a) a single port with all the thingumajigs;

b) a single port per thingumajig;

c) a port per client.

(ii) A client supplies a thingumajig identifier to the server, which replies with a port right. What type of port right should the server send back to the client? Explain why the server's identifier for the port right and that of the client may differ.

(iii) A thingumajig client resides at a different computer from the server. Explain in detail how the client comes to possess a port right that enables it to communicate with the server, even though the Mach kernel can only transmit port rights between local tasks.

(iv) Explain the sequence of communication events that take place under Mach when the client sends a message requesting an operation upon a thingumajig, assuming again that client and server reside at different computers. *page 10*

18.9 A Mach task on machine *A* sends a message to a task on a different machine *B*. How many domain transitions occur, and how many times are the message contents copied if the message is page-aligned? *page 10*

18.10 Design a protocol to achieve migration transparency when ports are migrated. *page 11*

18.11 How can a device driver such as a network driver operate at user level? *page 12*

18.12 Explain two types of region sharing that Mach uses when emulating the UNIX *fork*() system call, assuming that the child executes at the same computer. A child process may again call *fork*(). Explain how this gives rise to an implementation issue, and suggest how to solve it.*page 13*

18.13 (i) Is it necessary that a received message's address range is chosen by the kernel when copy-on-write is used?

(ii) Is copy-on-write of use for sending messages to remote destinations in Mach?

(iii) A task sends a 16 kilobyte message asynchronously to a local task on a 10 MIPS, 32-bit machine with an 8 kilobyte page size. Compare the costs of (1) simply copying the message data (without using copy-on-write) (2) best-case copy-on-write and (3) worst-case copy-on-write. You can assume that:

- creating an empty region of size 16 kilobytes takes 1000 instructions;
- handling a page fault and allocating a new page in the region takes 100 instructions.*page 13*

18.14 Summarize the arguments for providing external pagers. *page 14*

18.15 A file is opened and mapped at the same time by two tasks residing at machines without shared physical memory. Discuss the problem of consistency this raises. Design a protocol using Mach external pager messages which ensures sequential consistency for the file contents (see Chapter 16). *page 16*