



Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 18 Exercise Solutions

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

18.1 Ans.

Issues arise due to the need to coordinate and synchronize the kernels' accesses to shared data structures. The kernel of a uniprocessor computer achieves mutual exclusion between threads by manipulating the hardware interrupt mask. Kernels on a multiprocessor must instead use a shared-memory-based mechanism such as spinlocks. Moreover, care must be taken in enforcing caching protocols so as to produce a memory model with the required consistency guarantees.

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

18.2 Ans.

A binary-level OS emulation executes binary files that run on the native OS, without adaptation and with exactly the same results (functions and error behaviour). By contrast, a source-level emulation requires recompilation of the program and may behave differently because some 'system' functions are actually emulated in libraries.

Binary-level emulation is desirable so that users can retain their software investment while migrating to a system with superior functionality. Technically, achieving binary-level emulation is complex. The producers of the emulated OS may change that native OS's functionality, leaving the emulation behind.

18.3 Explain why the contents of Mach messages are typed.

18.3 Ans.

The contents of Mach messages are type-tagged because:

- a) The designers wanted to support marshalling of simple data types as a system service.
- b) The kernel monitors the movement of port rights, in order to monitor communication connectivity, to transmit port location hints for send rights and to implement port migration for receive rights. So the rights must be tagged in messages.
- c) Messages may contain pointers to out-of-line data, which the kernel must be aware of.

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.

18.4 Ans.

The Mach network server does in fact implement monitoring of the number of send rights to a port over the network. Clients do not always terminate gracefully, and so this feature is useful for servers that need to garbage-collect ports when no clients exist that can send to them. However, this convenience is gained at the expense of considerable management overhead. It would seem preferable to be able to switch off this feature if there is no strong need for it – perhaps on a port-by-port basis.

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

18.5 Ans.

Kernel-level Mach threads are an expensive and limited resource. Some tasks acquire thousands of ports. Instead of utilising a thousand threads, a small number of threads can read messages from a port set containing all the ports, and dispatch message processing according to which port each message arrives at.

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

18.6 Ans.

The single call replaces two system-calls per client-server interaction with one (saving two domain transitions). The client call sends a message and receives a reply; the server call sends the reply to the previous request and receives the next request.

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

18.7 Ans.

A 'network port' is an abstraction of a global port – one to which threads on multiple machines can send messages.

The network port is implemented by a local port and a globally-unique identifier, maintained by a network server at each machine that forwards network messages to the local port and participates in locating the port.

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.

18.8 Ans.

i) Associating a single port with all the thingumajigs minimises the number of ports. But it makes it difficult to relocate some but not all the thingumajigs with another server at run-time. Individual resource relocation can be desirable for performance reasons.

Associating a single port with each thingumajig requires potentially large numbers of ports, but it enables any resource to be relocated with another server at run time, independently of the others. It also makes it easy to vary the number and priority of threads associated with each resource.

Associating a port with each client would make it difficult to move resources to other servers. The only advantage of this scheme is that a server can take for granted the identity of the principal involved when it processes the requests arriving at a particular port (assuming that Mach ensures the integrity of the port).

ii) The server should send a send right for the server's port. The identifiers may differ, because each uses a local name, which is allocated from its own local name space and invalid beyond it.

iii) Assume that the server at computer *S* sends a send right back to a client at computer *C*. The message first arrives at the local network server at *S*, which examines the message and notices the send right *s* within it. If the network server has not received this right before, it generates a network port number *n* and sets up a table entry relating *n* to *s*. It forwards the message to *C*'s network server, enclosing *n* and the location of the network port, namely *S*. When *C*'s network server receives this message, it examines *n* and finds that it has no entry for it. It creates a port with receive rights *r* and send rights *t*, and creates a table entry relating *r* to *n* and *S*. It forwards the message to the local client, enclosing the send right *t*.

iv) When the network server at *C* later receives a message using *r*, it knows to forward the message quoting the network port *n* to the network server at *S*. The network server at *S*, in turn, knows that messages addressed to *n* should be forwarded using its stored send right *s*, which refers to the server's port.

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?

18.9 Ans.

Domain transitions are: task A → kernel A → Network server A, Network server B → kernel B → task B (4).

Using copy-on-write with no page faults (i.e. if the sending task does not write on the message before transmission is complete), the message is copied four times: once from the Network server's address space to kernel buffers, once from kernel buffers to the network interface, and *vice versa* at the receiving end.

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

18.10 Ans.

Page 712 outlines some of the ingredients of such a protocol. The mechanisms available to us are (a) forwarding hints at nodes from which the port has migrated; (b) multicast, used to locate a port using its unique identifier. Sub-protocols are:

(1) a port-location and sender-rebinding protocol that operates when a sender attempts to send to a port that has migrated; this protocol must include provision to garbage-collect forwarding hints and it must cope with inaccurate hints and hints that point to crashed machines;

(2) a protocol that moves the message queue before other messages are appended at the new site.

We leave the details to the reader. Note that an implementation of such a protocol based on TCP/UDP sockets rather than Mach ports could be used in an implementation of object migration.

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

18.11 Ans.

If the device registers are memory-mapped, the process must be allowed to map the registers into its user-level address space. If the registers are accessible only by special instructions, then the process needs to be allowed to run with the processor in supervisor mode.

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.

18.12 Ans.

Two types of region sharing that Mach uses when emulating the UNIX *fork()* system call: (1) Physical sharing of pages in read-only shared regions, e.g. program text, libraries. (2) Copy-on-write sharing of copied regions, e.g. stack and heap.

If the child again calls *fork()*, then its copy-on-write page table entries need to refer to the correct page: that of its parent or its grandparent? If the grandparent modifies a shared page, then both parent and child's page table entries should be updated. Keeping track of dependencies despite arbitrary recursive calls to *fork()* is an implementation problem.

A doubly-linked tree structure of page-nodes representing the 'logical copy' relationship can be used to keep track of dependencies. If a process modifies a page, then the page is physically copied and the (bidirectional) link with the parent node is broken. Links from the descendant nodes are replaced by links to the parent node.

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.

18.13 Ans.

(i) In general, no. The copy-on-write mechanism is independent of the logical addresses in use. A process could therefore receive a message into a pre-specified region.

(ii) No: it is useful only for local memory copying. If a message is transferred remotely, then it must be copied to the remote machine anyway.

(iii) (1) Copying: 2 instructions/loop, 4 bytes at a time $\Rightarrow (10/2) * 4 = 20$ Mbyte/sec = 20K/ms. Sent asynchronously \Rightarrow receiver not waiting \Rightarrow 2 copies $\Rightarrow 2 * 16/20 = 1.6$ ms.

(2) In the best case, we create a new region but there are no page faults: $1000/10\text{MIPS} = 0.1$ ms (1000 instructions).

3) In the worst case, we create a new region and there are two page faults and two page copies: 0.1 ms + $0.02\text{ms} + 16/20$ (copy) = 0.92 ms.

Thus copy-on-write always wins in this example.

18.14 Summarize the arguments for providing external pagers.

18.14 Ans.

The central argument is to allow the kernel to support a variety of (distributed shared) memory abstractions, including files mapped from remote servers. See pp. 716-719.

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).

18.15 Ans.

Suppose that several tasks residing at different machines 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. If no special action is taken, a page can be modified at one computer while stale versions of the page exist in memory cache objects at other computers. Note that no consistency problem arises between tasks at a single kernel sharing a mapped memory object. The kernel keeps only one memory cache object in this case, and the associated frames are physically shared between the tasks.

In order to maintain consistency between a number of memory cache objects (managed by different kernels), the external pager controls, for each page of the shared file, whether or not the page is physically present at client machines, and if it is present, whether tasks there have read-only or read-write access to the page. It uses the single-writer/multiple-reader sharing scheme described in Chapter 16. To control access, the *memory_object_lock_request* message can be used to set permissions (as well as or instead of directing the kernel to write modified data). The *memory_object_data_provided* message contains a parameter to specify read-only or read-write permissions.

Enabling access to a page. A task takes a page-fault either when a) a non-resident page is required, or b) the page is resident but read-only, and an upgrade to write access is required. To handle the page-fault, the kernel respectively sends to the external pager (file server) either a *memory_object_data_request* message to request a non-resident page, or a *memory_object_data_unlock* message to request an upgrade to write access. To satisfy the request, the external pager may first downgrade other kernels' access to the page as described in the next paragraph. The external pager responds to the faulting machine with a *memory_object_data_provided* message to give the page data and set the access to it, or a *memory_object_lock_request* message to upgrade the access on the page to write access, as appropriate.

Downgrading access to a page. The external pager can issue a *memory_object_lock_request* message with appropriate parameters to fetch a page from a kernel if it has been modified, and at the same time to downgrade its access to read-only or none. Access becomes read-only if a page is required elsewhere for reading; it becomes null if the page is required elsewhere for writing.