Originally published at pp. 579-84 of Coulouris, Dolllimore and Kindberg, Distributed Systems, Edition 2, 1994.

UNIX emulation in Mach and Chorus

Section 6.6 of edition 3 outlines the requirement for emulating operating systems at the binary level on top of distributed operating system kernels. Mach and Chorus are designed to emulate operating systems, notably UNIX; UNIX emulation has also been implemented on the V kernel.

Strict binary compatibility requires that all binary files compiled to run on a conventional implementation of a version of UNIX (for example, Linux, 4.3BSD or SVR4) should run correctly and without modification on the emulation, for the same machine architecture. This implies that the following list of requirements should be met:

Address space layout: The emulation must provide the regions expected by the program. If the code is non-relocatable, the machine instructions assume that regions such as the program text and heap occupy certain expected address ranges. Address space regions such as the stack must be grown as necessary.

System call processing: Whenever a program executes a system call with a valid set of arguments, the emulation must handle this correctly according to the defined call semantics; it must handle the associated *TRAP* instruction and obey the parameter-passing conventions expected by the program.

Error semantics: Whenever a program presents invalid arguments to a system call, the emulation must reproduce correctly the error semantics defined for the system call. In particular, if the user program provides an invalid memory address, the emulation should simply return an error status, and not raise a hardware exception at user-level.

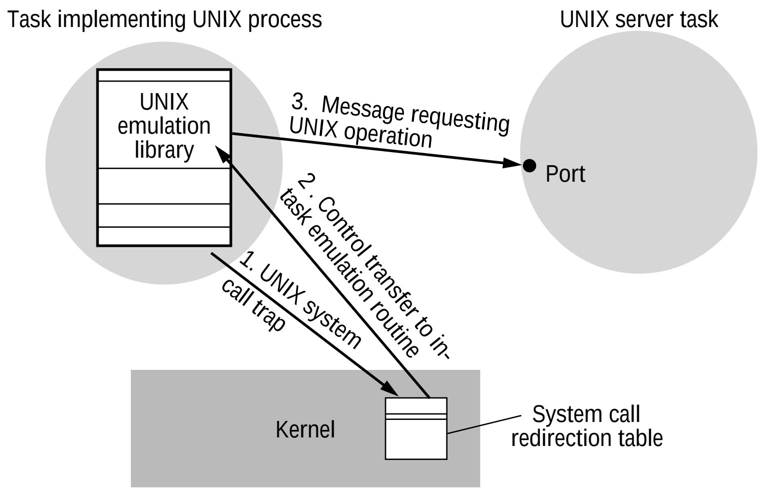
Failure semantics: The emulation should not introduce new system call failure modes. An example of a failure mode applicable to a conventional UNIX implementation is the inability to complete a call due to lack of system resources such as table space (for example, *fork* may fail in this way).

Protection: User data and the UNIX emulation system itself must not be compromised.

Signals: Signals must be generated and user-level handlers called as appropriate when a UNIX program causes an exception such as an address space violation.

Emulation software is required at every computer that can run UNIX processes. One of the aims when emulating UNIX in a distributed system is to implement a single UNIX image across several computers, so that, for example, UNIX processes have globally unique process identifiers, and signals can be transmitted transparently between computers. It becomes difficult or impossible to meet the requirement of reproducing UNIX failure semantics – in so far as they are defined – in these circumstances. Effectively, many UNIX system calls would have to be implemented as transactions, because of the independent failure modes of computers and networks. Moreover, suitable protection mechanisms are required, strictly speaking, when user data are transferred across a network.

Figure 1 UNIX emulation under Mach.


Task implementing UNIX process
UNIX server task
UNIX emulation library
3. Message requesting UNIX operation
Port
2. Control transfer to in-task emulation routine
1. UNIX system call trap
Kernel
System call redirection table

The Mach emulation

A UNIX process is implemented using a Mach task with a single thread (Figure 1). 4.3BSD UNIX services are provided by two software components: an *emulation library* and a *4.3BSD server*. The emulation library is linked as a distinct region into every task emulating a UNIX process. This region is inherited from */etc/init* by all UNIX processes. There is one 4.3BSD server (that is, one such task) for every computer running the emulation. This server both handles requests sent by clients and acts as an external pager when clients fault on mapped UNIX files, as we shall discuss.

Applications do not invoke the code in the emulation library directly. Mach provides a call *task_set_emulation*, which assigns the address of a handler in the emulation library to a given system call number; this is called for each UNIX system call when the emulation is initialized. When a UNIX process executes a system call, the *TRAP* instruction causes the Mach kernel to transfer control back to the thread in the UNIX task, so that this same thread executes the corresponding emulation library handler (Figure 1). The handler then either sends a message requesting the required service to the 4.3BSD server task and awaits a reply or, in some cases, performs the UNIX system service using data accessible to the emulation library itself.

Each UNIX process and its local 4.3BSD server share two regions, of size one page. One of these is read-only for the process, and it contains information such as the process's identifier, user identifier and group identifier. If a process calls *getpid*, for example, then the emulation library may read the process identifier directly from this page and return it without communicating with the 4.3BSD server. The other shared region can be written by the UNIX process; it contains signal-related information and an array of data structures relating to the process's open files.

When a UNIX process opens a file it is mapped into a region of its address space, with the 4.3BSD server acting as the external pager. The emulation assigns a region of 64 kilobytes for each file; if the file is larger, then the region is used as a movable window onto the file. When the process calls *read* or *write* on the file, the corresponding emulation library procedure copies the data between the mapped region and the user's buffer and updates the file pointer. The data copying requires no explicit communication with the 4.3BSD server. However, the library may generate page faults as it accesses the file region, which will result in the kernel communicating with the 4.3BSD server.

The emulation library has to synchronize with the 4.3BSD server before it accesses the file data if it needs the file window to be moved, or if the open file is shared (for example, with a child or parent process). In the latter case, the file's read-write pointer must be consistently updated. A token is used to obtain mutual exclusion over its use. The emulation library is responsible for requesting the token from the 4.3BSD server and releasing it. (See the subsection on distributed mutiual exclusion in Chapter 10 for a description of centralized token management.)

The Mach exception handling scheme facilitates the implementation of UNIX signals arising from exceptional conditions. A thread belonging to a 4.3BSD server can arrange to be sent messages pertaining to exceptions, and it can respond to these by adjusting the victim task's state so as to call a signal handler, before replying to the kernel. The exception handling scheme also facilitates automatic stack growth and task debugging.

The 4.3BSD server requires internal concurrency in order to handle the calls made upon it efficiently. Recall that UNIX processes undergo a context switch and execute within the kernel to handle their system calls in a conventional implementation; there is thus a process in the kernel for every system call. The 4.3BSD server uses many C threads to receive requests and process them. Most of the threads are kept in a pool and assigned dynamically to requests from emulation library calls. There are also a few dedicated threads: the *Device reply* thread requests device activity from the kernel; the *Softclock* thread implements timeouts; the *Netinput* thread handles network device interactions with the kernel; and the *Inode pager* thread implements an external pager corresponding to mapped UNIX files.

The Chorus emulation

In Chorus, any operating system emulation subsystem consists of two types of component: a *subsystem process manager* and zero or more other server actors. A subsystem process manager is a supervisor actor that operates at the 'front-line' of operating system emulation: it handles system call traps to its subsystem. A process manager runs at each site where the subsystem is implemented. In some cases, the process manager can service a system call itself. In other cases, it communicates by message passing with other, specialized subsystem actors to service the system call. Chorus/MiX, the UNIX SVR4 UNIX emulation subsystem, consists primarily of the following actors:

Process manager: This provides process creation and destruction and signal handling, and communicates with the other, specialized subsystem actors as necessary to handle system calls.

File (*Object*) *manager*: Performs file management and acts as the external pager for mapped files.

Device managers: Control particular devices such as the disk.

Streams manager: Manages pipes, networking, System V IPC and pseudo-terminals.

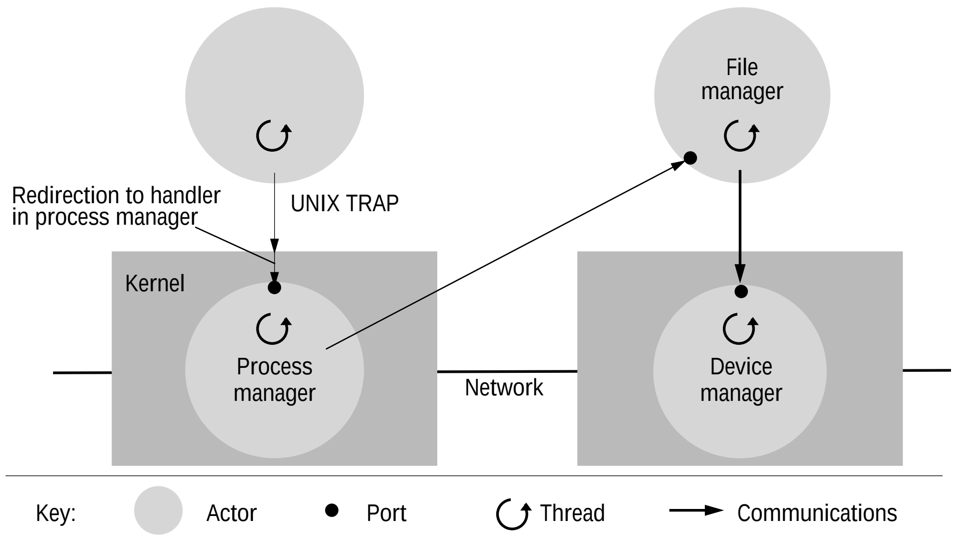
Apart from the Process Manager, the managers are run at each computer only as necessary: for example, a diskless node does not run the File Manager.

A supervisor or system actor can dynamically 'connect' a table of *TRAP* handler routines to be called by the kernel when an actor executes a corresponding *TRAP* instruction. The table contains the address of a routine for each emulated system call. When an actor implementing a UNIX process executes a *TRAP* instruction, the thread executing the call enters the kernel's protection domain. In other words, it becomes temporarily a supervisor thread. The kernel automatically switches the thread context so that it executes the corresponding *TRAP* handler previously installed by the Chorus/MiX process manager (Figure 2). This thread may be able to complete the system call itself using its own data – for example, it might achieve a file *read* or *write* by copying data between pages in the cache and the user address space. More generally, this thread will communicate by message passing with one of the other Chorus/MiX subsystem servers. For example, it will communicate with the File Manager if there is no cached data to satisfy a *read*. The File Manager will then communicate with the Disk Manager to fetch the data.

Since they are accessed by message passing, any of the other subsystem servers may provide system-wide facilities – unlike the process managers, which only handle local *TRAP*s.

The process manager establishes, in each actor representing a local UNIX process, a supervisor thread and a control port for it. This thread is used to handle operations upon the process resulting from system calls made by other UNIX processes. For example, if a process is sent a signal, a message is delivered to the process's control port and handled by the corresponding

Figure 2 The emulation of UNIX in Chorus.



Note: Only some subsystem servers shown.

supervisor thread. This thread can then, as appropriate, terminate the UNIX process or modify its state so that a signal handler is called within it.

Comparison

When a Mach task executes a UNIX system call *TRAP* the kernel passes control to a handler in the emulation library, whose code and data reside in the address space of the task. This has the disadvantage compared to the Chorus scheme, of allowing buggy code within the task to interfere with the data in the emulation library, and thus to give rise to non-standard failure modes. The Chorus scheme preserves complete isolation of system data structures from UNIX processes.

In Mach, all UNIX facilities are provided by the single UNIX (4.3BSD) server, as opposed to several servers in Chorus. The advantage of the Chorus design is its modularity. Since the separate servers communicate by message passing, any server can be re-implemented independently of the others, as long as the interfaces are adhered to. However, the consequence of this is that the system state relevant to a single emulated UNIX process is distributed across several servers – and perhaps several computers. The exact apportionment of state between subsystem servers is a difficult design issue. Extra communication can be required to maintain this state consistently if it is replicated, and functionality can be affected by a decision to split state without replication between servers. Mach, on the other hand, need concern itself only with the process state in the emulation library's data and that of the single UNIX server.

Subsystem servers require access to devices. Chorus differs from Mach in its approach to device access: in Mach, devices are normally managed by the kernel, and a message-based interface is provided to privileged tasks such as UNIX servers. In Chorus, by contrast, actors may dynamically introduce device drivers into the executing kernel, and take complete control over devices:

- Actors may dynamically connect interrupt handler routines to device interrupts. Whenever an interrupt occurs, the kernel calls the associated routine, which is part of the code of a supervisor actor. This routine can either handle the interrupt itself, or, for example, it can send a message to a pre-arranged port, so that a thread can handle it.
- System or supervisor actors can establish supervisor threads to control devices. These threads can handle requests for the device sent as messages to them, and they have direct access to device registers in handling the requests.

The Chorus approach to device management appears to offer a performance advantage over the Mach scheme, and it is flexible with respect to how much device processing is performed at kernel level and how much at user level. However, it is debatable whether a particular subsystem should gain complete access over a device, since this would normally be to the exclusion of other subsystems. If device accesses are multiplexed through the kernel, on the other hand, then in principle several subsystems can coexist at a single site. An example of this is Mach's use of a programmable *packet filter* installed in the kernel, which multiplexes incoming packets to different subsystem servers according to data in their headers. For example, all incoming NFS packets could be handled by a UNIX subsystem server, whereas Appletalk packets could be handled by a MacOS emulation server. These packets can be identified by simple programs provided by the respective servers.

Golub *et al*. [1990] and Armand *et al*. [1989] describe UNIX emulation on Mach and Chorus respectively; Dean and Armand [1992] describe UNIX emulation on Mach and Chorus in more detail, and give performance figures. Cheriton *et al*. [1990] describe UNIX emulation on the V kernel.

References

[Armand *et al.* 1989]	Armand, F., Gien, M., Herrman, F. and Rozier, M. (1989). Distributing UNIX brings it back to its original virtues. *Proc. Workshop on Experiences with Building Distributed and Multiprocessor Systems,* pp. 153-174, October.
[Cheriton *et al.* 1990]	Cheriton, D., Whitehead, G. and Sznyter, E. (1990). Binary emulation of Unix using the V kernel, *Proc. USENIX Summer Conference*, pp. 73-85.
[Dean and Armand 1992]	Dean, R. and Armand, F. (1992). Data movement in kernelized systems. *Proc. USENIX Workshop on Microkernels*.
[Golub *et al.* 1990]	Golub, D., Dean, R., Forin, A. and Rashid, R. (1990). UNIX as an application program. *Proc. USENIX Summer Conference*, pp. 87-96.