The Design and Implementation of the FreeBSD Operating System

FreeBSD version 5.2

Marshall Kirk McKusick
George V. Neville-Neil

Exhibit 1061
Microsoft Corporation v. Edge Networking Systems, LLC

UNIX is a registered trademark of X/Open in the United States and other countries. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales
(317) 581-3793
international@pearsontechgroup.com

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

McKusick, Marshall Kirk.
The design and implementation of the FreeBSD operating system / Marshall Kirk McKusick, George V. Neville-Neil.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-70245-2 (hc : alk. paper)
1. FreeBSD. 2. Free computer software. 3. Operating systems (Computers) I. Neville-Neil, George V. II. Title.

QA76.76.O63M398745 2004
005.3--dc22 2004010590

Copyright © 2005 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

Text printed on recycled and acid-free paper.
ISBN 0-201-70245-2
1 2 3 4 5 6 7 8 9 10–CRW–0807060504
First Printing, July 2004

Exhibit 1061
Microsoft Corporation v. Edge Networking Systems, LLC

If a controlling process exits, the system revokes further access to the controlling terminal and sends a SIGHUP signal to the foreground process group. If a process such as a job-control shell exits, each process group that it created will become an *orphaned process group*: a process group in which no member has a parent that is a member of the same session but of a different process group. Such a parent would normally be a job-control shell capable of resuming stopped child processes. The *pg_jobc* field in Figure 4.10 counts the number of processes within the process group that have the controlling process as a parent. When that count goes to zero, the process group is orphaned. If no action were taken by the system, any orphaned process groups that were stopped at the time that they became orphaned would be unlikely ever to resume. Historically, the system dealt harshly with such stopped processes: They were killed. In POSIX and FreeBSD, an orphaned process group is sent a hangup and a continue signal if any of its members are stopped when it becomes orphaned by the exit of a parent process. If processes choose to catch or ignore the hangup signal, they can continue running after becoming orphaned. The system keeps a count of processes in each process group that have a parent process in another process group of the same session. When a process exits, this count is adjusted for the process groups of all child processes. If the count reaches zero, the process group has become orphaned. Note that a process can be a member of an orphaned process group even if its original parent process is still alive. For example, if a shell starts a job as a single process A, that process then forks to create process B, and the parent shell exits; then process B is a member of an orphaned process group but is not an orphaned process.

To avoid stopping members of orphaned process groups if they try to read or write to their controlling terminal, the kernel does not send them SIGTTIN and SIGTTOU signals, and prevents them from stopping in response to those signals. Instead, attempts to read or write to the terminal produce an error.

Jails

The FreeBSD access control mechanism is designed for an environment with two types of users: those with and those without administrative privilege. It is often desirable to delegate some but not all administrative functions to untrusted or less trusted parties and simultaneously impose systemwide mandatory policies on process interaction and sharing. Historically, attempting to create such an environment has been both difficult and costly. The primary mechanism for partial delegation of administrative authority is to write a set-user-identifier program that carefully controls which of the administrative privileges may be used. These set-user-identifier programs are complex to write, difficult to maintain, limited in their flexibility, and are prone to bugs that allow undesired administrative privilege to be gained.

Many operating systems attempt to address these limitations by providing fine-grained access controls for system resources [P1003.1e, 1998]. These efforts

Exhibit 1061
Microsoft Corporation v. Edge Networking Systems, LLC

vary in degrees of success, but almost all suffer from at least three serious limitations:

1. Increasing the granularity of security controls increases the complexity of the administration process, in turn increasing both the opportunity for incorrect configuration, as well as the demand on administrator time and resources. Often the increased complexity results in significant frustration for the administrator, which may result in two disastrous types of policy: running with security features disabled and running with the default configuration on the assumption that it will be secure.
2. Usefully segregating capabilities and assigning them to running code and users is difficult. Many privileged operations in FreeBSD seem independent but are interrelated. The handing out of one privilege may be transitive to many others. For example, the ability to mount filesystems allows new set-user-identifier programs to be made available that in turn may yield other unintended security capabilities.
3. Introducing new security features often involves introducing new security management interfaces. When fine-grained capabilities are introduced to replace the set-user-identifier mechanism in FreeBSD, applications that previously did an appropriateness check to see if they were running with superuser privilege before executing must now be changed to know that they need not run with superuser privilege. For applications running with privilege and executing other programs, there is now a new set of privileges that must be voluntarily given up before executing another program. These changes can introduce significant incompatibility for existing applications and make life more difficult for application developers who may not be aware of differing security semantics on different systems.

This abstract risk becomes more clear when applied to a practical, real-world example: many Web service providers use FreeBSD to host customer Web sites. These providers must protect the integrity and confidentiality of their own files and services from their customers. They must also protect the files and services of one customer from (accidental or intentional) access by any other customer. A provider would like to supply substantial autonomy to customers, allowing them to install and maintain their own software and to manage their own services such as Web servers and other content-related daemon programs.

This problem space points strongly in the direction of a partitioning solution. By putting each customer in a separate partition, customers are isolated from accidental or intentional modification of data or disclosure of process information from customers in other partitions. Delegation of management functions within the system must be possible without violating the integrity and privacy protection between partitions.

FreeBSD-style access control makes it notoriously difficult to compartmentalize functionality. While mechanisms such as *chroot* provide a modest level of

Exhibit 1061
Microsoft Corporation v. Edge Networking Systems, LLC

compartmentalization, this mechanism has serious shortcomings, both in the scope of its functionality and effectiveness at what it provides. The *chroot* system call was first added to provide an alternate build environment for the system. It was later adapted to isolate anonymous **ftp** access to the system.

The original intent of *chroot* was not to ensure security. Even when used to provide security for anonymous **ftp**, the set of operations allowed by **ftp** was carefully controlled to prevent those that allowed escape from the *chroot*'ed environment.

Three classes of escape from the confines of a *chroot*-created filesystem were identified over the years:

1. Recursive *chroot* escapes
2. Escapes using **..**
3. Escapes using *fchdir*

All these escapes exploited the lack of enforcement of the new root directory.

Two changes to *chroot* were made to detect and thwart these escapes. To prevent the first two escapes, the directory of the first level of *chroot* experienced by a process is recorded. Any attempts to traverse backward across this directory are refused. The third escape using *fchdir* is prevented by having the *chroot* system call fail if the process has any file descriptors open referencing directories.

Even with stronger semantics, the *chroot* system call is insufficient to provide complete partitioning. Its compartmentalization does not extend to the process or networking spaces. Therefore, both observation of and interference with processes outside their compartment is possible. To provide a secure virtual machine environment, FreeBSD added a new "jail" facility built on top of *chroot*. Processes in a jail are provided full access to the files that they may manipulate, processes they may influence, and network services they may use. They are denied access to and visibility of files, processes, and network services outside their jail [Kamp & Watson, 2000].

Unlike other fine-grained security solutions, a jail does not substantially increase the policy management requirements for the system administrator. Each jail is a virtual FreeBSD environment that permits local policy to be independently managed. The environment within a jail has the same properties as the host system. Thus, a jail environment is familiar to the administrator and compatible with applications [Hope, 2002].

Jail Semantics

Two important goals of the jail implementation are to:

1. Retain the semantics of the existing discretionary access-control mechanisms
2. Allow each jail to have its own superuser administrator whose activities are limited to the processes, files, and network associated with its jail

Exhibit 1061
Microsoft Corporation v. Edge Networking Systems, LLC

The first goal retains compatibility with most applications. The second goal permits the administrator of a FreeBSD machine to partition the host into separate jails and provide access to the superuser account in each of these jails without losing control of the host environment.

A process in a partition is referred to as being "in jail." When FreeBSD first boots, no processes will be jailed. Jails are created when a privileged process calls the *jail* system call with arguments of the filesystem into which it should *chroot* and the IP address and hostname to be associated with the jail. The process that creates the jail will be the first and only process placed in the jail. Any future descendants of the jailed process will be in its jail. A process may never leave a jail that it created or in which it was created. A process may be in only one jail. The only way for a new process to enter the jail is by inheriting access to the jail from another process already in that jail.

Each jail is bound to a single IP address. Processes within the jail may not make use of any other IP address for outgoing or incoming connections. A jail has the ability to restrict the set of network services that it chooses to offer at its address. An application request to bind all IP addresses are redirected to the individual address associated of the jail in which the requesting process is running.

A jail takes advantage of the existing *chroot* behavior to limit access to the filesystem name space for jailed processes. When a jail is created, it is bound to a particular filesystem root. Processes are unable to manipulate files that they cannot address. Thus, the integrity and confidentiality of files outside the jail filesystem root are protected.

Processes within the jail will find that they are unable to interact or even verify the existence of processes outside the jail. Processes within the jail are prevented from delivering signals to processes outside the jail, connecting to processes outside the jail with debuggers, or even seeing processes outside the jail with the usual system monitoring mechanisms. Jails do not prevent, nor are they intended to prevent, the use of covert channels or communications mechanisms via accepted interfaces. For example, two processes in different jails may communicate via sockets over the network. Jails do not attempt to provide scheduling services based on the partition.

Jailed processes are subject to the normal restrictions present for any processes including resource limits and limits placed by the network code, including firewall rules. By specifying firewall rules for the IP address bound to a jail, it is possible to place connectivity and bandwidth limitations on that jail, restricting the services that it may consume or offer.

The jail environment is a subset of the host environment. The jail filesystem appears as part of the host filesystem and may be directly modified by processes in the host environment. Processes within the jail appear in the process listing of the host and may be signalled or debugged.

Processes running without superuser privileges will notice few differences between a jailed environment or an unjailed environment. Standard system services such remote login and mail servers behave normally as do most third-party applications, including the popular Apache Web server.

Exhibit 1061
Microsoft Corporation v. Edge Networking Systems, LLC

Processes running with superuser privileges will find that many restrictions apply to the privileged calls they may make. Most of the limitations are designed to restrict activities that would affect resources outside the jail. These restrictions include prohibitions against the following:

- Modifying the running kernel by direct access or loading kernel modules
- Mounting and unmounting filesystems
- Creating device nodes
- Modifying kernel run-time parameters such as most sysctl settings
- Changing security-level flags
- Modifying any of the network configuration, interfaces, addresses, and routing-table entries
- Accessing raw, divert, or routing sockets. These restrictions prevent access to facilities that allow spoofing of IP numbers or the generation of disruptive traffic.
- Accessing network resources not associated with the jail. Specifically, an attempt to bind a reserved port number on all available addresses will result in binding only the address associated with the jail.
- Administrative actions that would affect the host system such as rebooting

Other privileged activities are permitted as long as they are limited to the scope of the jail:

- Signalling any process within the jail is permitted.
- Deleting or changing the ownership and mode of any file within the jail is permitted, as long as the file flags permit the requested change.
- The superuser may read a file owned by any UID, as long as it is accessible through the jail filesystem name space.
- Binding reserved TCP and UDP port numbers on the jail's IP address is permitted.

These restrictions on superuser access limit the scope of processes running with superuser privileges, enabling most applications to run unhindered but preventing calls that might allow an application to reach beyond the jail and influence other processes or systemwide configuration.

Jail Implementation

The implementation of the *jail* system call is straightforward. A *prison* data structure is allocated and populated with the arguments provided. The prison structure is linked to the process structure of the calling process. The prison structure's reference count is set to one, and the *chroot* system call is called to set the jail's root. The prison structure may not be modified once it is created.

Exhibit 1061
Microsoft Corporation v. Edge Networking Systems, LLC

Hooks in the code implementing process creation and destruction maintain the reference count on the prison structure and free it when the last reference is released. Any new processes created by a process in a jail will inherit a reference to the prison structure, which puts the new process in the same jail.

Some changes were needed to restrict process visibility and interaction. The kernel interfaces that report running processes were modified to report only the processes in the same jail as the process requesting the process information. Determining whether one process may send a signal to another is based on UID and GID values of the sending and receiving processes. With jails, the kernel adds the requirement that if the sending process is jailed, then the receiving process must be in the same jail.

Several changes were added to the networking implementation:

- Restricting TCP and UDP access to just one IP number was done almost entirely in the code that manages protocol control blocks (see Section 13.1). When a jailed process binds to a socket, the IP number provided by the process will not be used; instead, the preconfigured IP number of the jail is used.
- The loop-back interface, which has the magic IP number 127.0.0.1, is used by processes to contact servers on the local machine. When a process running in a jail connects to the 127.0.0.1 address, the kernel must intercept and redirect the connection request to the IP address associated with the jail.
- The interfaces through which the network configuration and connection state may be queried were modified to report only information relevant to the configured IP number of a jailed process.

Device drivers for shared devices such as the pseudo-terminal driver (see Section 10.1) needed to be changed to enforce that a particular virtual terminal cannot be accessed from more than one jail at the same time.

The simplest but most tedious change was to audit the entire kernel for places that allowed the superuser extra privilege. Only about 35 of the 300 checks in FreeBSD 5.2 were opened to jailed processes running with superuser privileges. Since the default is that jailed superusers do not receive privilege, new code or drivers are automatically jail-aware: They will refuse jailed superusers privilege.

Jail Limitations

As it stands, the jail code provides a strict subset of system resources to the jail environment, based on access to processes, files, network resources, and privileged services. Making the jail environment appear to be a fully functional FreeBSD system allows maximum application support and the ability to offer a wide range of services within a jail environment. However, there are limitations in the current implementation. Removing these limitations will enhance the ability to offer services in a jail environment. Three areas that deserve greater attention are the set of available network resources, management of scheduling resources, and support for orderly jail shutdown.

Exhibit 1061
Microsoft Corporation v. Edge Networking Systems, LLC

Currently, only a single IP version 4 address may be allocated to each jail, and all communication from the jail is limited to that IP address. It would be desirable to support multiple addresses or possibly different address families for each jail. Access to raw sockets is currently prohibited, as the current implementation of raw sockets allows access to raw IP packets associated with all interfaces. Limiting the scope of the raw socket would allow its safe use within a jail, thus allowing the use of **ping** and other network debugging and evaluation tools.

Another area of great interest to the current users of the jail code is the ability to limit the effect of one jail on the CPU resources available for other jails. Specifically, they require that the system have ways to allocate scheduling resources among the groups of processes in each of the jails. Work in the area of lottery scheduling might be leveraged to allow some degree of partitioning between jail environments [Petrou & Milford, 1997].

Management of jail environments is currently somewhat ad hoc. Creating and starting jails is a well-documented procedure, but jail shutdown requires the identification and killing of all the processes running within the jail. One approach to cleaning up this interface would be to assign a unique jail-identifier at jail creation time. A new *jailkill* system call would permit the direction of signals to specific jail-identifiers, allowing for the effective termination of all processes in the jail. FreeBSD makes use of an **init** process to bring the system up during the boot process and to assist in shutdown (see Section 14.6). A similarly operating process, **jailinit**, running in each jail would present a central location for delivering management requests to its jail from the host environment or from within the jail. The **jailinit** process would coordinate the clean shutdown of the jail before resorting to terminating processes, in the same style as the host environment shutting down before killing all processes and halting the kernel.

4.10 Process Debugging

FreeBSD provides a simplistic facility for controlling and debugging the execution of a process. This facility, accessed through the *ptrace* system call, permits a parent process to control a child process's execution by manipulating user- and kernel-mode execution state. In particular, with *ptrace*, a parent process can do the following operations on a child process:

- Attach to an existing process to begin debugging it
- Read and write address space and registers
- Intercept signals posted to the process
- Single step and continue the execution of the process
- Terminate the execution of the process

The *ptrace* call is used almost exclusively by program debuggers, such as **gdb**.

Exhibit 1061
Microsoft Corporation v. Edge Networking Systems, LLC

UNIX/Operating Systems

As in earlier Addison-Wesley books on the UNIX-based BSD operating system, Kirk McKusick and George Neville-Neil deliver here the most comprehensive, up-to-date, and authoritative technical information on the internal structure of open source FreeBSD. Readers involved in technical and sales support can learn the capabilities and limitations of the system; applications developers can learn effectively and efficiently how to interface to the system; system administrators can learn how to maintain, tune, and configure the system; and systems programmers can learn how to extend, enhance, and interface to the system.

The authors provide a concise overview of FreeBSD's design and implementation. Then, while explaining key design decisions, they detail the concepts, data structures, and algorithms used in implementing the systems facilities. As a result, readers can use this book as both a practical reference and an in-depth study of a contemporary, portable, open source operating system.

This book:

- Details the many performance improvements in the virtual memory system
- Describes the new symmetric multiprocessor support
- Includes new sections on threads and their scheduling
- Introduces the new jail facility to ease the hosting of multiple domains
- Updates information on networking and interprocess communication

Already widely used for Internet services and firewalls, high-availability servers, and general timesharing systems, the lean quality of FreeBSD also suits the growing area of embedded systems. Unlike Linux, FreeBSD does not require users to publicize any changes they make to the source code.

Marshall Kirk McKusick writes books and articles, consults, and teaches classes on UNIX- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast file system, and was the research computer scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD. He has twice served as the president of the board of the USENIX Association.

George V. Neville-Neil works on network and operating system code for fun and profit and teaches programming. He also serves on the editorial board of *Queue* magazine and is a member of the USENIX Association, ACM, and IEEE.

www.awprofessional.com/
www.FreeBSD.org

Cover design by Chuti Prasertsith
Cover art by John Lasseter

Text printed on recycled paper

Addison-Wesley
Pearson Education

9 780201 702453 55999

ISBN 0-201-70245-2

$59.99 US
$86.99 CANADA

Exhibit 1061
Microsoft Corporation v. Edge Networking Systems, LLC