History of FreeBSD

1979	3BSD Added virtual memory to UNIX/32V
1981	4.1BSD
1983	4.2BSD Final release from Berkeley DARPA UNIX project
1986	4.3BSD
1988	4.3BSD Tahoe
1989	4.3BSD Net1 - First free release of UNIX, but some files missing
1990	4.3BSD Reno
1991	4.3BSD Net2 - Free release with 6 files missing
1992	NetBSD 0.8 - Added the missing files to 4.3BSD Net2
1993	FreeBSD 1.0 - Based on NetBSD but oriented towards PC architecture
1993	4.4BSD Virtual memory code completely rewritten
1994	4.4BSD-Lite
1994	NetBSD and FreeBSD rewritten to use 4.4BSD-Lite code after lawsuit was settled
1995	4.4BSD-Lite Release 2 - Final free release from CSRG at Berkley
1995	OpenBSD Based on NetBSD - focus on security

FreeBSD Kernel Facilities

The FreeBSD kernel provides four basic facilities:

1. Processes are composed of an address space with one or more threads.
2. User interface to filesystem and devices.
3. Communication mechanisms as pipes, signals, sockets, fifos.
4. System startup routines.

Kernel

The FreeBSD kernel is a monolithic kernel.

This is both for historic reasons and for performance reasons.

A large part of the kernel is code that implements system services (system calls). This code is organized as:

- Basic kernel facilities: timer, process and descriptor management.
- Memory management
- Filesystems
- Terminal handling: pseudo terminals
- Interprocess communication: pipes sockets.
- Support for network communication.

Kernel

Most of the code in the kernel is machine independent.

Machine dependent software is isolated from the mainstream code. Machine dependent code includes:

- Low-level startup code.
- Trap and fault handling.
- Low-level manipulation of process context.
- Configuration and initialization of hardware devices.
- Runtime support for I/O devices.

Almost all code is written in the C language, only 0.6 percent is written in assembly language.

Kernel Services

- The kernel operates in a separate address space that is inaccessible to user processes.
- Only the kernel has direct access to system hardware.
- System calls are used to request services from the kernel.
- All system calls appear synchronous to the applications: The application do not run while the kernel performs the actions associated with a system call.
- User applications and the kernel operate independently of each other. The kernel do not store I/O control blocks or other operating-system related data structures in the application's address space.

Process Management

FreeBSD implements standard UNIX system calls for process management.

Signals

- Signals are delivered to a process as a result of certain events.
- Signals in FreeBSD are modeled after hardware interrupts.
- A process may specify a user level handler to which a signal should be delivered.
- When a signal is generated it is blocked from further occurrence while it is handled.
- Alternatively, a process may specify that a signal is to be ignored.
- Some signals like SIGKILL and SIGSTOP cannot be ignored.

Process Groups and Sessions

Related processes (such as all processes in a pipe) belongs to the same process group.

Process groups are used to get signals delivered to all related processes.

Memory Management

- Each process has its own address space.
- The address space is initially divided into three segments: *text, data* and *stack*.
- The entire address space need not be resident for a process to execute.
- The system implements a paged virtual memory system that allows some pages to be stored on a disk memory.
- An interface called *mmap()* was specified for 4.2BSD to allow processes to create new shared or private segments.
- Due to lack of time *mmap()* was not included in 4.2BSD.
- An improved version of *mmap()* was planned for 4.3BSD but once again not included due to lack of time.
- In 4.4BSD the virtual memory system was completely replaced and this was the first BSD system to include *mmap()*.
- The FreeBSD virtual memory is an extensively tuned version of the 4.4BSD implementation.

Memory Management - Read and Write

Data related with read and write system calls are usually copied into a kernel buffer.

Another possibility is to use remapping of the virtual memory to move the buffer into the kernel.

FreeBSD always copies data for the following reasons:

- Often, the user data is not page aligned.
- If the page is moved by the virtual memory, the process will no longer be able to reference it.
- If the process was allowed to keep a copy of the page, it must be made *copy-on-write*. Unfortunately, the typical process will immediately write into the buffer, forcing a copy anyway if it was *copy-on-write*.
- When pages are remapped by the virtual memory, most hardware requires that the hardware TLB is purged selectively. This purge is often slow resulting in remapping being slower than copying for block sizes less than 4 to 8 Kbyte.

Descriptors and I/O

The basic model for UNIX I/O system is a sequence of bytes that can be accessed sequentially or randomly.

UNIX processes uses descriptors to reference I/O streams.

Descriptors represent underlying objects supported by the kernel. In FreeBSD four kinds of objects can be represented by descriptors:

- A **file** is linear array of bytes with at least one name.
- A **pipe** is a linear array of bytes used as a unidirectional communication channel to another process.
- A **fifo** is often referred to as a named pipe.
- A **socket** is an endpoint for a general communication channel.

In systems before 4.2BSD, pipes were implemented in the filesystem.

In 4.2BSD pipes were reimplemented using sockets.

For performance reasons FreeBSD do not use sockets to implement pipes and fifos but use an implementation optimized for local communication.

Devices

- Hardware devices are available as *special files* located in the /dev directory.
- Most processes do not use these interfaces directly, terminals and mounted filesystems are accessed through the read and write system calls.
- Most network communication hardware do not have special files in the filesystem. Here the *raw-socket* interface is used instead.

Socket IPC

- The *socket* interface is a general interprocess and network communications interface, first introduced in 4.2BSD.
- Unlike pipes, sockets can be set up between arbitrary existing processes.
- *Fifos* are a kind of simplified sockets for local communication only.

Scatter/Gather I/O

- Scatter/gather I/O was introduced in 4.2BSD.
- It uses the system calls *readv* and *writev*.
- This facility allows buffers in different parts of a process address space to be written atomically without the need to copy them into a single buffer.

Multiple filesystem support

- Networked computers made it desirable to support both local and remote filesystems.
- If more than one filesystem is used a way is needed to direct system calls to the correct filesystem implementation.
- To make this possible a new data structure called *vnode* (virtual node) was added to the system.

Devices and Autoconfiguration

- Historically the connection between special files and the device driver was through a static array in the kernel source code.
- More complex and increasingly diverse hardware made a more flexible solution needed.
- Autoconfiguration is the process carried out to recognize and enable the hardware devices present in the system.
- Originally autoconfiguration was done only at system boot time.
- More dynamic device handling, especially in laptops, has made it necessary to be able to perform autoconfiguration operations at any time.
- FreeBSD uses a device-driver infrastructure called *newbus* to manage devices.

Filesystem operations

- The *mkdir* and *rmdir* system calls were added in 4.2BSD.
- In older systems these operations were implemented as a series of link and unlink operations.
- The *rename* system call was also introduced in 4.2BSD to make file renaming atomic.
- Also the *truncate* system call that is used to *decrease* or **increase** the size of a file was introduced in 4.2BSD.
- In addition to the usual UNIX file access controls FreeBSD5.2 also implement ACL:s (Access Control Lists)

Terminals

- Terminal access to mainframe and mini computers used hardwired terminals.
- Today the normal command line interface to UNIX systems uses terminal emulators such as **xterm** running in an X-window.
- These terminal emulators use pseudo terminals.
- A pseudo terminal is built from a device pair termed the *master* and *slave* devices.
- The slave device provides to a process an interface identical to that historically provided by a hardware device.
- Anything written on a master device appears as input on the slave device and anything written on a slave appears as input on the master device.

Terminals

- Terminals support the standard system read and write operations as well as terminal-specific operations to control input character editing and output formating.
- The character processing is handled by a line discipline.
- For command line oriented programs, the line discipline is run in *canonical mode.*
- In this mode input data are delivered to the process a line at the time.
- Screen editors usually run in *noncanonical mode (raw mode)*.
- In this mode every character is passed to the process without interpretation.
- On output the terminal handler provides simple formating services including:
  - → Converting line-feed to carriage-return + line-feed.
  - → Expanding tabs
  - → Displaying echoed nongraphic ASCII characters as two character sequences.

Interprocess Communication

- The socket interface is organized in *communication domains*.
- The most important domains are:
  - → The local domain.
  - → The IPv4 domain
  - → The IPv6 domain.
- Some of the communication domains provide access to network protocols.
- These protocols are implemented as a separate software layer logically below the socket layer in the kernel.
- The kernel provides many ancillary services such as buffer management, message routing, standardized interfaces to protocols and interfaces to network interface drivers.

Network Implementation

- The first protocol implemented in 4.2BSD was the TCP/IP suite (IPv4).
- The reason for this choice was that an 4.1BSD based implementation was publicly available from a DARPA-sponsored project at BBN (Bolt, Beranek and Newman).
- This implementation was probably the main reason for the very widespread use of the TCP/IP protocols today.