UNIX On Microkernels

An Overview Over Past And Current Developments

Technische Universität Dresden
Fakultät Informatik
Lehrstuhl für Betriebssysteme

Sebastian Humenda, 21/09/2016

Contents

0 License

1 Introduction

Unix was originally developed by Bell Laboratories and through the history, the trademark was often sold. This resulted in a variety of Unix implementations. Unix has been proprietary in the beginning, but inspired through the early days, where source code was freely available and also through the free software movement, the development of free and open source versions started [9].

Many unix concepts have been influential for today's operating system design. For instance, a hierarchical file system, a hierarchical process structure and pipes can be found on most systems today [6].

Unix was developed to be more modular than its predecessors. Programs should be simplistic and do exactly one thing. Through the introduction of pipes, it was possible to concatenate the output of one program as input of another, therefore solving complex tasks without requiring specialized programs. [9].

The kernel design of Unix systems, however, is often monolithic, for instance the kernels from the BSD family, Solaris or HPUX. It has been shown that microkernels offer better extensibility properties, ease the maintainability of the system and allow for better isolation of critical components [1, 4].

This work will give an overview over approaches and attempts to implement a Unix system on top of a microkernel. It starts with introducing Mach, the first microkernel with the intent to provide a next-generation modularized Unix system. Afterwards the QNX system is introduced, a commercial Unix system primarily deployed in embedded systems. The next section discusses Minix, a reliable and fault-tolerant operating system. Finally, the Hurd project with its Unix implementation on top of Mach is discussed.

2 Mach

Mach is a microkernel whose development started in the second half of the 1980s. With the increase in complexity of the systems back then, a more modular OS designed promised to help to adapt the system quicker to new developments. It intended to bring the modular Unix philosophy to the kernel, which consequently meant developing a microkernel [1].

2.1 Kernel Abstractions

The main abstractions that Mach provides are:

Task This is the smallest unit of resource allocation and provides the environment for execution. This includes virtual memory, protected access to system resources, etc.

Thread Threads provide the smallest unit of CPU utilisation and feature an independent program counter from other threads. They share access to resources within a task.[1]

Port This is a kernel-protected and queued communication channel endpoint for communication between tasks. It offers primitives like `send` and `receive`. This is also called inter-process communication (*IPC*).

Message A message is a typed collection of data used for communication between threads and may contain pointers or typed capabilities.

Capability Instead of maintaining a list of allowed actions on an object, the kernel can hand out a capability which enables the access to a certain kernel object.

[1, p. 3]

Mach is designed to provide a small set of primitive functions, which should enable building more complex systems. The primitives offered by Mach make it possible to represent services as objects, to which an entity can gain access by receiving a capability. The functionality of a system is determined by its servers,[2] not its kernel. Operations on objects[3] are done by sending messages to them.

Virtual Memory The handling of virtual memory is directly built into the Mach kernel. According to the authors, this was done mainly due to performance reasons. Virtual memory can be allocated/deallocated on a per-task basis, which in turn can set protection or inheritance attributes on the granted memory regions. A task may specify that a certain region is inherited to a child task and can do so either read-only, copied or not at all. Copying is done using copy-on-write to speed up the process. Inheritance is the only way of sharing memory regions between tasks in Mach [1, pp. 4,6].
Whenever a memory region is allocated, the requesting task gets a capability back which allows the usage of the memory. This capability can be passed to another task, which transfers the memory region. Even the whole address space of a task can be transferred this way.

2.2 Tasks And Threads

The original task abstraction in Unix was a single-threaded holder of resources and the smallest unit for scheduling. Blocking system calls suspended the execution, therefore

[1] Accetta et al. explain the abstraction of threads as a novel concept. It is common in today's monolithic and microkernel systems and it is not clear whether they introduced it. According to their description it was not found in Unix back then, which only supported tasks.

[2] Throughout this paper, server refers to a program, running in user space and serving a particular functionality and interacting with the environment through suitable communication mechanisms.

[3] Objects are a general term to address anything which can be altered through a command (message) send via inter-process communication. This can be something within the kernel, but also within a user service.

eliminating the possibility of doing other work within the same task, while waiting for the system call to complete. The thread concept introduces independent execution units sharing all resources. This way, a multiprocessor can be used more efficiently, because multiple concurrent threads can run within a task and can be scheduled to run on different processors/cores in parallel.

Tasks in Mach are tree-structured and therefore have a parent-children relationship.

2.3 Inter-Process Communication

Ports Mach is designed for transparent extensibility of the kernel. Therefore it uses inter-process communication from and to ports. Sending to a port can only be done using a capability, serving as an authentication and also a way to locate a port. Capabilities for ports are created whenever a port is created. Ports are handed out to a task to provide a given service, communication takes place with messages and the kernel guarantees only that messages are sent and received. Since messages are buffered within the kernel, the sender will get blocked eventually, if the receiver is slower than the sender. It also means that data is copied from and to kernel space, so for each message twice. An alternate approach can be seen in section 3 on the following page.

Messages Messages have a fixed-length header, but a collection of data with variable size. They can contain typed pointers and capabilities.
It is possible to send messages synchronously and asynchronously. Synchronous receivers are blocked until a message arrives, asynchronous receivers register a signal handler which gets called whenever a message arrives. A message contains enough information to re-encode it for different machine types, when sent over a network.
An IPC interface is defined using a specialized definition language called Matchmaker. It compiles the definitions into RPC stubs. Interfaces generated with Matchmaker also do dynamic type checking on messages.

RPC over network Through the loosely coupled design of servers and user tasks, it is easy to add support for a network of processors or multiprocessors, because the coordination is already abstracted into communication between ports. Port endpoints may be a thread in the same task, in a different task or on a different processor. For the sender of a message, it is impossible to tell whether the message sent to a port is sent to a server on the same machine or over the network to another machine which will answer this request, because the location of a port is transparent to the sender.
Mach itself does not provide networking capabilities, but it is possible to redirect IPC using a networking server. The network server will take care of marshalling the message into the network protocol, map each port to a network address and send the message to that address. The other machine will then receive the package, map to the appropriate port and will send the unmarshalled message.

2.4 UNIX Interface

The Mach developers tried to provide a Unix system which was similar to BSD 4.3. This included e. g. a virtual file system and a debugger. Most of the services are implemented as user-level tasks. For instance, the virtual file system is runs in user space as a separate task and allows arbitrary servers to be registered as file system drivers. This is not limited to local file system implementations; network file systems can be used transparently as well.

Debugging of the kernel is possible with the built-in kernel debugger. However, since most of the system components are user space programs, debugging is far easier than in traditional Unix.

3 QNX

QNX was introduced around 1982 and had the main goal to provide the maximum performance of the underlying hardware to applications running on top of it. It is built on top of a real-time capable microkernel and has optional user processes offering Unix/POSIX services. All system services are optional to scale the system to tiny and big systems. The microkernel and the whole system architecture was written to not only match the performance of traditional monolithic systems, but outperform them.

The kernel implements exactly four services: inter-process communication, low-level network communication, process scheduling and interrupt dispatching. When the paper was published, only 14 system calls were implemented, which was enough to build a POSIX-compliant system on top. Because of its small size, the authors claimed that it would fit into the L1 cache of CPUs back then.

The process and thread scheduling contained in the kernel is POSIX-compatible. User-level resource managers can alter the policy of the scheduler to adapt it to the system needs. The scheduler is fully preemptive and designed with the goal that nothing will degrade the real-time property of the scheduling.

3.1 Inter-Process Communication

IPC in QNX is blocking, so that if one end is not ready, the other end (be it sending or receiving), is blocked (this happens on a per-thread basis). Messages are always copied from the sender to the receiver and don't take the indirection over the kernel; this saves a copy operation. The IPC implementation is very efficient, since `send` only takes place when both sides are ready and the number of context switches is minimized. If buffering of IPC is required, it can be implemented by a user-level service, interposing the communication.

Messages are tagged with a priority. If a server receives a message and reacts to it, it does not execute with its own priority, but with that one of the sending process. This prevents priority inversions, because lower-priority processes cannot keep the server busy and hence prevent high-priority senders from doing its job.

Messages in QNX can be either contiguous or fragmented. Fragmented messages are made up of a header (MX table) with the addresses of the different parts of the messages and their length. This way the message can be copied from different regions of the address space or they can be even remapped to the receivers address space. This avoids expensive copy operations for contiguous messages.

3.2 Resource Managers

All application-level OS interfaces are implemented by services. Resource managers provide a name space in which they control their resources and abstract from the management of these. A resource can be a file system or the set of all running processes.

A special kind of service is *Proc* , the first and only mandatory resource manager which acts as a process manager for all processes. It provides services as process creation, process accounting and memory allocation. It also manages the global name space, which is identical to an empty file system after boot. The name space is not limited to the local machine and many *Proc* instances can form a global name space, spanning multiple machines. *Proc* offers an API to set up resource managers below its own namespace and is able to transfer authority to a child resource manager.
Name spaces are strictly separated, only the name space of *Proc* is global, since it is the parent of all others. Name spaces can be used to implement independent subsystems and errors only propagate in the subsystem or to the resource manager of this subsystem.

Since all other services are optional, it is possible to build a very minimalistic OS with no file system which directly operates on RAM.

3.3 File System And Device Management

The virtual file system is provided by the *Fsys* service, which is also a resource manager. It gets the authority to manage the "/" root node from *Proc* and offers a Unix-alike file system tree.

According to Hildebrand, *Fsys* also contains a file system implementation, mixing virtual file system functionality and a file system driver. It is possible to add further nested resource managers to implement different file systems.

An `open` call for a file is sent to the *Proc* manager which then determines that a resource underneath / is requested and refers the request to *Fsys*. The requested path is then matched against its prefix and the match with the longest prefix wins. This can be either in the file system provided by *Fsys*, or in a different resource manager.

Fsys also supports FS aliasing, which is also known as mounting in Unix. Remote file systems can be aliased into the local file system hierarchy and are fully transparent for any application using them.

The *dev* resource manager manages devices and is a child process of *Fsys* . It binds itself to the name space under `/dev`. It offers the functionality of a device over the file system API. It can easily add and remove drivers from the dev name space during operation without a restart. Even *dev* itself can be removed if it is not required anymore.

Drivers in user space do not run slow, because interrupt handling is built directly into the kernel and switching to and from the microkernel is faster than in a monolithic system. The kernel provides a system call to let user space processes register for interrupts with a handler.

dev allows grouping most of the devices into block and character devices known from Unix. To further optimize the operation speed of character devices, it is possible to let the handler from a driver gather characters in a buffer and only invoke the driver, when a certain limit is reached. This way, the full driver program is only invoked when enough data for processing is available.

3.4 Networking

Low-level networking is carried out by the kernel. It enables the microkernel to use remote procedure calls for IPC. The actual network drivers and networking policies are not contained in the kernel though. The initialization and maintenance of a networking connection is done by a service called *Net*. It is a resource manager for network drivers and the network. All RPC calls to other machines are send by the kernel to *Net* and this takes care of calling the appropriate services to redirect the RPC to a different *Net* instance which transforms the RPC back into IPC.

4 Minix

Minix is a POSIX-compatible operating system, providing a fault-tolerant Unix environment running on a microkernel. The goal of Minix is to use the better fault isolation properties of a microkernel to deliver a more fault-tolerant operating system. It is designed to survive and to recover from failures in drivers and services. The detection and recovery of faulty components works automatically and transparent to the applications using them. For this to work, the operating system is split into many small components, enabling fine-grained control of each component. The design principles have been:

1. simple and effective IPC
2. detach interrupt handling from user-mode drivers
3. separation of policy and mechanisms

4. decoupling of servers and drivers through a publish/subscribe model
5. flexible runtime OS configuration

4.1 Inter-Process Communication

According to Herder et al., the kernel only implements the most basic mechanisms, which cannot be implemented in user space, i. e. interrupt handling, programming of CPU, MMU, device I/O, scheduling and IPC.

Inter-process communication works synchronous and without memory allocation both within the kernel and the user space application. This is called *rendez-vous*: if the receiver is not awaiting the sender, the sender is blocked, until the receiver becomes ready. This also works the other way around. Data is always copied from one user process to another, which saves a copy operation in comparison to Mach (see 2.3 on page 6). This also eliminates the need for in-kernel allocation of buffers and of filled buffers. A certain set of notifications can be sent asynchronously to a process; to avoid resource depletions, all notification are saved in a per-process bitmap.

Whenever an exception happens within a process, the kernel converts it to an IPC message and sends it to the process manager. *PM* converts the IPC message into a Unix signal, which is either send to the process or leads to a termination of the process, if no handler was registered [4, p. 6].

To implement a capability-based permission system, the Minix kernel maintains several bitmaps and lists to track permissions of objects throughout the system. These include reachable IPC destinations, permitted kernel calls, IO ports, IRQ lines and memory regions [4, p. 9].

4.2 Architecture

The Unix interface and the fault-recovery mechanisms are implemented by a set of processes running in user space. Each service and driver is started with the least authority in order to operate. Normal user processes cannot execute system calls, but must use the POSIX interface (discussed below) [4, pp. 6, 11].

Two processes run in kernel mode, without being part of the kernel itself. They aide the kernel and shield it from user space services. They are:

SYS The SYS service provides the interface for user processes, which require low-level operations. These system calls are validated and checked for permissions and only then *SYS* delegates the work to the kernel. It never becomes active itself, but waits blocked for incoming messages.

CLOCK This service interacts with the hardware clock and controls all timers in the system. It also does the accounting on CPU usage, data which can be used by the scheduler. It is not reachable from user space directly, but a system call allows registering an alarm. [4, pp. 6 sq.]

The most basic Unix services are implemented by the process manager *PM* and the file system service *FS*.
PM is responsible for maintaining process relations across the system (process groups or parent-child relations). It implements the policy for process management, while the kernel provides the primitives for process creation. *PM* also contains the memory manager *MM*, although work is being done to split both. The memory manager uses hardware-independent segmented memory. This allows for easy sharing of regions across processes (e. g. text and data segment) and enables easier portability to other platforms. System processes can be granted special segments as for instance the video segment. The text segment is read-only and the stack non-executable by default [4, p. 7].

FS implements the Unix file system interface, with system calls such read, write and open. At the moment, only one file system is supported and it is directly built into the file system server. When the paper was written, work was in progress to transform the server into a virtual file system server enabling any file system to be supported.

The data store (*DS*) provides a global store for dynamic system configuration. It offers a publish and subscribe model, hence decoupling dependencies between servers. The subscriber can even subscribe to a pattern matching certain notification types (e. g. all new disk drivers). For instance, a file system server would subscribe to any disk driver and whenever such a new driver might be loaded or removed, the information is published to the registered file system server. A file system thus does not have a reference to the disk driver, but retrieves its information from the data store.
DS also provides private storage for each server running in the system and if a service fails, it can easily retrieve this data after its reincarnation. Furthermore, the data store provides a global naming service, but introduces a single point of failure as well. It is not apparent from the paper, whether the data store can be restarted without data loss [4, p. 7].

4.3 Failure Detection And Recovery

The reincarnation server (*RS*) runs as a parent of all processes. At system boot, all processes are started as a child of *RS*, which is also able to start and stop services. To determine fault-recovery policies, services can be equipped with a policy script, which regulate the actions on boot and on failure. The policy script may also specify actions to check whether a process is still alife, for instance by sending a periodic message and checking the reply [4, p. 6].

Whenever a server or driver crashes, the exception is converted to an IPC message to the parent by the kernel. Since all drivers and servers are a child of the reincarnation

server, it can detect whenever a service ceases to exist and execute the actions defined in the policy script. For different classes of drivers, different actions can be defined. For block devices, it is common to reissue the command after a restart, whereas character device failures are mostly propagated to the user.

Because the data store provides a mechanism to decouple dependencies, components can be transparently restarted or replaced. The underlying fault model can deal with transient failures, which are gone after a restart (e. g. aging bugs), but cannot cope with Byzantine faults. It is also not possible to recover from a crash of the reincarnation server [4, pp. 9 sq.].

5 GNU Hurd

GNU is the operating system developed by the Free Software Foundation, which aims at providing an entirely free open source operating system. GNU runs on the Hurd, which is a set of services on top of the Mach microkernel. When Hurd was started, no free implementation of Unix was available.[4] The focus of the project was to deliver a free Unix-alike operating system, providing as much freedom to the user as possible. The developers aimed at making each component of the system replaceable (or interceptable) by a user without disrupting other users. [7, 3].

The (Mach) kernel only handles tasks, threads, memory and IPC. Everything else is implemented in user space. Therefore the basic design is very similar to the original, unmodified Mach [1, 3]. While all processes are optional on a Mach system, a few processes are required for GNU/Hurd to provide a POSIX interface. These are the process server, the execution server, the root file system server and the authentication server. Any interface can be bypassed and Mach interfaces can be used instead. It is also possible to reimplement them to virtualize the functionality at a higher level.

The Hurd system brings some improvements to the user, which are not present in other Unix-alike operating systems. For instance, the console server, which provides a console (shell) session to the user, is able to load fonts dynamically and display e. g. Chinese in a VGA text mode. It is also possible for every user to swap or extend any service, since all services can be interposed. This is not present of any Unix today.

5.1 Process Handling

Process handling is done by the process server (*proc*), the execution server (*exec*) and the file system server. The process manager categorizes and manages process information and provides global host information such as the host name, which are not provided by Mach directly. It maintains a notion of POSIX sessions and process groups. In theory,

[4] Free as defined by the free software foundation is the right to copy, modify, redistribute and to use the software for any purpose.

the registration to *proc* is optional, but a (global) PID is always assigned. Due to limitations in Mach, it is necessary to run *proc* as privileged user, since only privileged users can see all processes [3].

exec performs loading of programs, libraries and also parses the interpreter from scripts to execute them. It sets up the process environment and initiates the task setup.

The Unix system calls are all carried out by libraries to which an application is linked. It contacts the appropriate servers using IPC. For example, a fork call in GNU Hurd is performed by the C library `libc`. The library would contact the file system server to check for the binary and to retrieve a handle to it, contact the execution server to set up the task, obtain a capability to the port representing the task and last but not least contact *proc* to register the new process [3].

5.2 File System

Hurd provides a virtual file system to which processes get access by receiving the root port capability. There is no designated virtual file system server, but the port from the mounted root file system is the main entry point for applications. All other file systems, which run in separate processes, are accessed through their port capability, which is retrieved using a path from the root file system. The program providing a file system services is called a translator and the process called translation. Every user can register any translator for any file system, as long as he/she has access to the resource. It will then run with the user ID, hence with the privileges of the user. Translators are invoked the first time when a file is accessed at the path at which they are bound to[5] and return a capability to a port with which the communication is performed.

A translator can be a normal file system implementation as e. g. ext3, but is not limited to it. In fact, it can hide everything behind the file system API.

Hurd ships with a FTP and ISO translator, which any user can use without special privileges or extensions to the OS infrastructure.[6] Drivers can be registered as a translator too and this is typically done under `/dev`. The null driver, which can output an arbitrary number of null bytes, registers itself under `/dev/null`.

5.3 Reusing Drivers

As already pointed out, drivers are ordinary user processes and hence benefit from easier recoverability and easier debugging, but they need to be written in the first place. With a limited amount of developers, reusing drivers is one of the main goals regarding drivers.

[5] Translators are only started once, so for two open files on an ext4 file system, only one translator is started.

[6] Therefore it is easily possible to register a translator under `ftp:` and then access `ftp://host/path`, which would get translated into a series of FTP commands.

Rump The Rump kernel is a FreeBSD kernel running in user mode. It provides many drivers as well as a TCP/IP stack and more. For instance, Hurd is able to play sound using certain sound cards when using the Rump kernel [7].

DDEKit The Device Driver Environment Kit (DDEKit) is a library, which maps the interface for drivers found within the 2.6-series of Linux to an outside driver interface to reuse them on a different host system. To make drivers available on Hurd, the driver is linked against the DDE library and a bit of glue code is added to turn it into a server [7].

Missing Hardware Support According to Thibault, hardware support on Hurd is not extensive yet. USB support is missing, but being worked on. Sound support exists for a few sound cards and Rump kernel extends the list of supported cards. The AMD-64 architecture is also not yet officially supported, but being worked on. Although hardware support is limited, the software runs stable and it has been reported that some machines have not been reinstalled since a decade [7].

6 Conclusion

Since the 1980s, several attempts have been made to modularize the UNIX architecture by reimplementing all system services and user tools on top of a modular microkernel. Although there are many benefits, microkernels were not able to replace monolithic systems. The most popular Unix clone Linux is a monolithic kernel, this is also true for the BSD Unix systems.

Mac OS X, which builds on top of a hybrid kernel, has brought at least some of the advantages of a microkernel to modern desktop and mobile phone platforms. It incorporates code from Mach and from the FreeBSD kernel. This way, some drivers and interfaces are built-in and always available, but other mechanisms and features can be implemented on top of this kernel using IPC mechanisms [8].

QNX is the only fully microkernel-based operating system with a Unix API which is commercially successful. Several deployments are listed on the website `http://qnx.com`.

The Mach microkernel has been very influential. It was adopted for projects like Hurd or merged into another code base as for Mac Os X.[7] Mach's shortcomings motivated many researchers to improve the properties of microkernels and its design still is a good example.

[7] When the Mac OS X kernel was created, it was not part of Mac OS X. However, this is beyond the scope of this document.

More modern microkernels often do not fully implement the POSIX API and one reason could be that the POSIX API does not deliver good performance for nowadays systems with many cores or many processors [2].

References

[1] Mike Accetta et al.: "Mach: A New Kernel Foundation for UNIX Development". In: 1986, pp. 93–112.

[2] Nils Asmussen et al.: "M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores". In: *SIGOPS Oper. Syst. Rev.* 50.2 (Mar. 2016), pp. 189–203. ISSN: 0163-5980.
Web: `http://doi.acm.org/10.1145/2954680.2872371`.

[3] Thomas Bushnell: *Towards a New Strategy of OS Design*. retrieved 2016-08-16.
Web: `http://www.gnu.org/software/hurd/hurd-paper.html`.

[4] Jorrit N. Herder et al.: "Reorganizing UNIX for Reliability". In: *Proceedings of the 11th Asia-Pacific Conference on Advances in Computer Systems Architecture*. ACSAC'06. Shanghai, China: Springer-Verlag, 2006, pp. 81–94. ISBN: 3-540-40056-7, 978-3-540-40056-1.
Web: `http://dx.doi.org/10.1007/11859802_8`.

[5] Dan Hildebrand: "An Architectural Overview of QNX". In: *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*. Berkeley, CA, USA: USENIX Association, 1992, pp. 113–126. ISBN: 1-880446-42-1.
Web: `http://dl.acm.org/citation.cfm?id=646405.759105`.

[6] Dennis M. Ritchie and Ken Thompson: "The UNIX Time-sharing System". In: *Commun. ACM* 17.7 (July 1974), pp. 365–375. ISSN: 0001-0782. DOI: `10.1145/361011.361061`.
Web: `http://doi.acm.org/10.1145/361011.361061`.

[7] Samuel Thibault: *Hurd, Rump kernel, sound, and USB*. FOSDEM Conference. Jan. 2016.
Web: `https://fosdem.org/2016/schedule/event/microkernels_hurd_rump_sound_usb/attachments/slides/951/export/events/attachments`.

[8] Pawel Wall: *Die Architektur von MacOS X und iOS*. July 2007.
Web: `ps.informatik.uni-siegen.de/.../`.

[9] Wikipedia: *Unix*. [Online; accessed 15-August-2016]. 2016.
Web: `https://en.wikipedia.org/w/index.php?title=Unix&oldid=732116594`.