Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[14] Epilogue

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v1

Outline

1. Introduction
2. Linux History
3. Kernels

14.1

14. Epilogue

Introduction

Epilogue

We finally completed our journey through the Linux Kernel. We started from the internals of Boot Process for the x86 and from there we saw all the most important Kernel Subsystems from the Memory to the Process Management, from the Interrupts to the System Calls Management.

We then concluded with Virtualization capabilities and some notions about the Security within Operating Systems.

Now for finally concluding this course let’s have a quick glance about the Linux history and the other kernels available around.

14.2

14. Epilogue

Linux History

UNIX

1969

In the late 1960s, Bell Labs was involved in a project with MIT and General Electric to develop a time-sharing system, called Multiplexed Information and Computing Service (**Multics**), allowing multiple users to access a mainframe simultaneously (the first time-sharing operating system). Dissatisfied with the project's progress, Bell Labs management ultimately withdrew.

Ken Thompson, a programmer in the Labs' computing research department, had worked on Multics. He decided to write his own operating system. While he still had access to the Multics environment, he wrote simulations for the new file and paging system but it needed a more efficient and less expensive machine to run on, and eventually he found a little-used Digital Equipment Corporation PDP-7 at Bell Labs.

https://en.wikipedia.org/wiki/History_of_Unix

UNIX

DEC PDP-7

https://en.wikipedia.org/wiki/History_of_Unix

UNIX

1969

On the PDP-7, in 1969, a team of Bell Labs researchers led by Thompson and Ritchie, including Rudd Canaday, implemented a hierarchical file system, the concepts of computer processes and device files, a command-line interpreter, and some small utility programs, modeled on the corresponding features in Multics, but simplified. Douglas McIlroy then ported TMG compiler-compiler to PDP-7 assembly, creating the first high-level language running on Unix. Thompson used this tool to develop the first version of his **B programming language**.

```
/* The following program will calculate the constant
e-2 to about
   4000 decimal digits, and print it 50 characters
to the line in
   groups of 5 characters.  The method is simple
output conversion
   of the expansion
     1/2! + 1/3! + ... = .111....
   where the bases of the digits are 2, 3, 4, . . .
*/

main() {
        extrn putchar, n, v;
        auto i, c, col, a;

        i = col = 0;
        while(i<n)
                v[i++] = 1;
        while(col<2*n) {
                a = n+1 ;
                c = i = 0;
                while (i<n) {
                        c =+ v[i] *10;
                        v[i++]  = c%a;
                        c =/ a--;
                }

                putchar(c+'0');
                if(!(++col%5))
                        putchar(col%50?' ': '*n');
        }
        putchar('*n*n');
}
v[2000];
  n 2000;
```

https://en.wikipedia.org/wiki/History_of_Unix

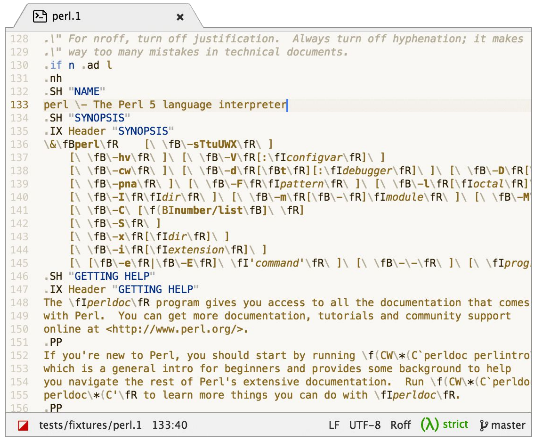
UNIX

1970s

The Unix operating system really needed a word processor, that was added by Thompson and Ritchie and they received funding for a PDP-11/20. For this machine the `roff` (from the Multics `runoff`) word processor program was added. At that time the OS were strictly not portable.

In 1973, **Version 4 Unix** was rewritten in the **higher-level language C**, contrary to the general notion at the time that an operating system's complexity and sophistication required it to be written in assembly language. However, the C language appeared as part of Version 2

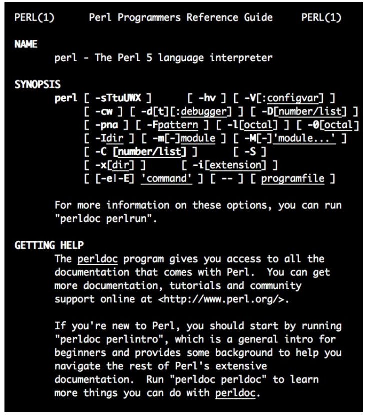
Version 4 Unix, however, still had considerable PDP-11-dependent code and was not suitable for porting. The first port to other platform was made five years later (1978) for Interdata 8/32.



perl.1

```
.\" For nroff, turn off justification.  Always turn off hyphenation; it makes
.\" way too many mistakes in technical documents.
.if n .ad l
.nh
.SH "NAME"
perl \- The Perl 5 language interpreter
.SH "SYNOPSIS"
.IX Header "SYNOPSIS"
\&\fBperl\fR    [\ \fB\-sTtuUWX\fR\ ]
    [\ \fB\-hv\fR\ ]\ [\ \fB\-V\fR[:\fIconfigvar\fR]\ ]
    [\ \fB\-cw\fR\ ]\ [\ \fB\-d\fR[\fBt\fR][:\fIdebugger\fR]\ ]\ [\ \fB\-D\fR[
    [\ \fB\-pna\fR\ ]\ [\ \fB\-F\fR\fIpattern\fR\ ]\ [\ \fB\-l\fR[\fIoctal\fR]
    [\ \fB\-I\fR\fIdir\fR\ ]\ [\ \fB\-m\fR[\fB\-\fR]\fImodule\fR\ ]\ [\ \fB\-M
    [\ \fB\-C\ [\f(BInumber/list\fB]\ \fR]
    [\ \fB\-S\fR\ ]
    [\ \fB\-x\fR[\fIdir\fR]\ ]
    [\ \fB\-i\fR[\fIextension\fR]\ ]
    [\ [\fB\-e\fR|\fB\-E\fR]\ \fI'command'\fR\ ]\ [\ \fB\-\-\fR\ ]\ [\ \fIprog
.SH "GETTING HELP"
.IX Header "GETTING HELP"
The \fIperldoc\fR program gives you access to all the documentation that comes
with Perl.  You can get more documentation, tutorials and community support
online at <http://www.perl.org/>.
.PP
If you're new to Perl, you should start by running \f(CW\*(C`perldoc perlintro
which is a general intro for beginners and provides some background to help
you navigate the rest of Perl's extensive documentation.  Run \f(CW\*(C`perldo
perldoc\*(C'\fR to learn more things you can do with \fIperldoc\fR.
.PP
```

tests/fixtures/perl.1 133:40 LF UTF-8 Roff (λ) strict master



```
PERL(1)      Perl Programmers Reference Guide      PERL(1)

NAME
       perl - The Perl 5 language interpreter

SYNOPSIS
       perl [ -sTtuUWX ]      [ -hv ] [ -V[:configvar] ]
            [ -cw ] [ -d[t][:debugger] ] [ -D[number/list] ]
            [ -pna ] [ -Fpattern ] [ -l[octal] ] [ -0[octal]
            [ -Idir ] [ -m[-]module ] [ -M[-]'module...' ]
            [ -C [number/list] ]      [ -S ]
            [ -x[dir] ]      [ -i[extension] ]
            [ [-e|-E] 'command' ] [ -- ] [ programfile ]

       For more information on these options, you can run
       "perldoc perlrun".

GETTING HELP
       The perldoc program gives you access to all the
       documentation that comes with Perl.  You can get
       more documentation, tutorials and community
       support online at <http://www.perl.org/>.

       If you're new to Perl, you should start by running
       "perldoc perlintro", which is a general intro for
       beginners and provides some background to help you
       navigate the rest of Perl's extensive
       documentation.  Run "perldoc perldoc" to learn
       more things you can do with perldoc.
```

A man page formatted in roff

https://en.wikipedia.org/wiki/History_of_Unix

UNIX

1970s

In 1973, AT&T released **Version 5 Unix** and licensed it to educational institutions, and licensed 1975's Version 6 to companies for the first time. While commercial users were rare because of the US$20,000 (equivalent to $95,028 in 2019) cost, the latter was the most widely used version into the early 1980s. Anyone could purchase a license, but the terms were very restrictive; licensees only received the source code, on an as-is basis.

Unix still only ran on DEC systems. As more of the operating system was rewritten in C (and the C language extended to accommodate this), portability also increased; in 1977, Bell Labs procured an Interdata 8/32 with the aim of porting Unix to a computer that was as different from the PDP-11 as possible, making the operating system more machine-independent in the process.

UNIX

1980s

Because the company widely and inexpensively licensed Unix, by the early 1980s thousands of people used Unix at AT&T and elsewhere, and as computer science students moved from universities into companies they wanted to continue to use it. Observers began to see Unix as a potential universal operating system, suitable for all computers. Less than 20,000 lines of code – almost all in C – composed the Unix kernel as of 1983, and more than 75% was not machine-dependent.

AT&T announced UNIX System III – based on Version 7, and PWB – in 1981. Licensees could sell binary sublicenses for as little as US$100 (equivalent to $281.22 in 2019), which observers believed indicated that AT&T now viewed Unix as a commercial product, even if they could not sell it as a commercial product, their policy was:

"No advertising, no support, no bug fixes, payment in advance."

UNIX

1980s

In 1983, the U.S. Department of Justice settled its second antitrust case against AT&T, causing the breakup of the Bell System. This **relieved** AT&T of the 1956 consent decree that had prevented the company from commercializing Unix. AT&T promptly introduced Unix System V into the market. The newly created competition nearly destroyed the long-term viability of Unix, because it **stifled the free exchanging** of source code and led to fragmentation and incompatibility. The GNU Project was founded in the same year by Richard Stallman.

Since the newer commercial UNIX licensing terms were not as favorable for academic use as the older versions of Unix, the Berkeley researchers **continued to develop BSD** as an alternative to UNIX System III and V. Many contributions to Unix first appeared in BSD releases, notably the C shell with job control (modelled on ITS). Perhaps the most important aspect of the BSD development effort was the addition of TCP/IP network code to the mainstream Unix kernel (Berkeley Sockets).

Linux

1980-1990

Due to an earlier antitrust case forbidding it from entering the computer business, AT&T was required to license the operating system's source code to anyone who asked. As a result, Unix grew quickly and became widely adopted by academic institutions and businesses.

In 1984, AT&T divested itself of Bell Labs; freed of the legal obligation requiring free licensing, Bell Labs began selling Unix as a proprietary product, where users were not legally allowed to modify Unix. The GNU Project, started in 1983 by Richard Stallman, had the goal of creating a "complete Unix-compatible software system" composed entirely of free software. Work began in 1984. Later, in 1985, Stallman started the Free Software Foundation and wrote the GNU General Public License (GNU GPL) in 1989. By the early 1990s, many of the programs required in an operating system (such as libraries, compilers, text editors, a command-line shell, and a windowing system) were completed, although low-level elements such as device drivers, daemons, and the kernel, called GNU Hurd, were stalled and incomplete

Linux

1990

MINIX was created by Andrew S. Tanenbaum, a computer science professor, and released in 1987 as a minimal Unix-like operating system targeted at students and others who wanted to learn operating system principles. Although the complete source code of MINIX was freely available, the licensing terms prevented it from being free software until the licensing changed in April 2000.

In 1991, while attending the University of Helsinki, Torvalds became curious about operating systems. Frustrated by the licensing of MINIX, which at the time limited it to educational use only, he began to work on his own operating system kernel, which eventually became the Linux kernel. Torvalds began the development of the Linux kernel on MINIX and applications written for MINIX were also used on Linux. Later, Linux matured and further Linux kernel development took place on Linux systems. GNU applications also replaced all MINIX components; code licensed under the GNU GPL can be reused in other computer programs as long as they also are released under the same or a compatible license. Torvalds initiated a switch from his original license, which prohibited commercial redistribution, to the GNU GPL. Developers worked to integrate GNU components with the Linux kernel, making a fully functional and free operating system.

Linux

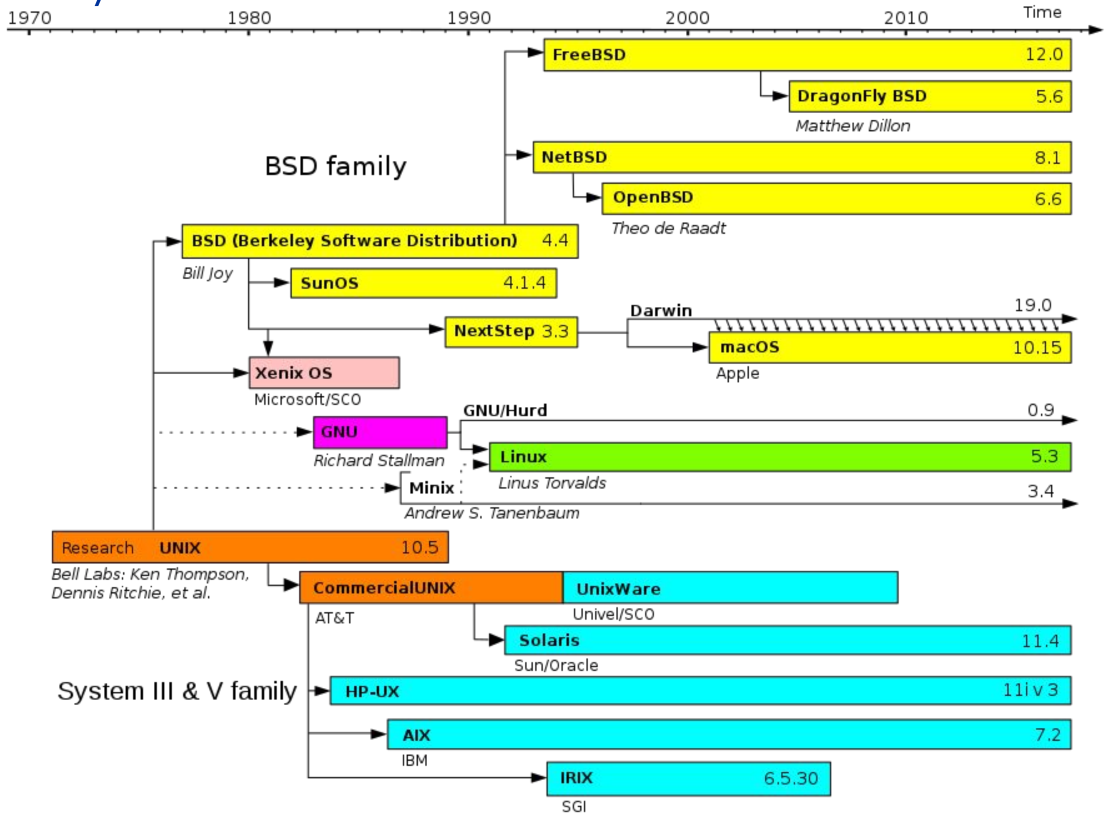
From 1991 to today

Linux kernel is licensed under the GNU General Public License (GPL), version 2. The GPL requires that anyone who distributes software based on source code under this license must make the originating source code (and any modifications) available to the recipient under the same terms.

A 2001 study of Red Hat Linux 7.1 found that this distribution contained 30 million source lines of code. Using the Constructive Cost Model, the study estimated that this distribution required about eight thousand person-years of development time. According to the study, if all this software had been developed by conventional proprietary means, it would have cost about $1.6 billion (2021 US dollars) to develop in the United States

Linux

From 1991 to today



14.3

14. Epilogue

Kernels

Kernels

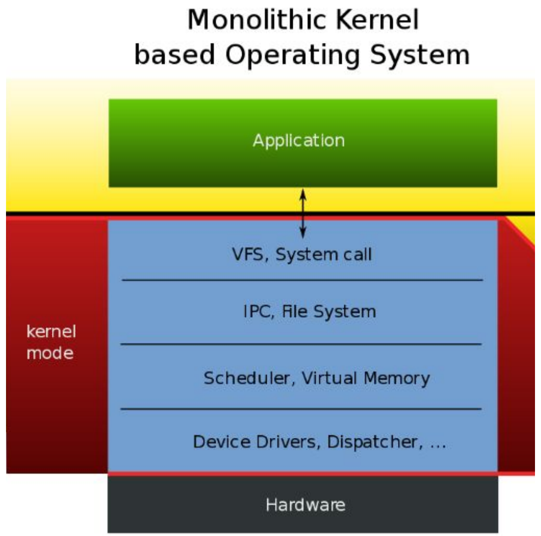
We can classify kernels in the following categories:

- **monolithic** kernel
- **microkernel**
- hybrid kernel
- nanokernel
- exokernel

Monolithic Kernel

In a monolithic kernel, all OS services run along with the main kernel thread, thus also residing in the same memory area. This approach provides rich and powerful hardware access. Some developers, such as UNIX developer Ken Thompson, maintain that it is "**easier** to implement a monolithic kernel" than microkernels. The main disadvantages of monolithic kernels are the dependencies between system components – a bug in a device driver **might crash** the entire system – and the fact that large kernels can become very difficult to maintain.

A monolithic kernel is one single program that contains all of the code necessary to perform every kernel-related task. Every part which is to be accessed by most programs which cannot be put in a library is in the kernel space: Device drivers, scheduler, memory handling, file systems, and network stacks.



Monolithic Kernels

Essential traits:

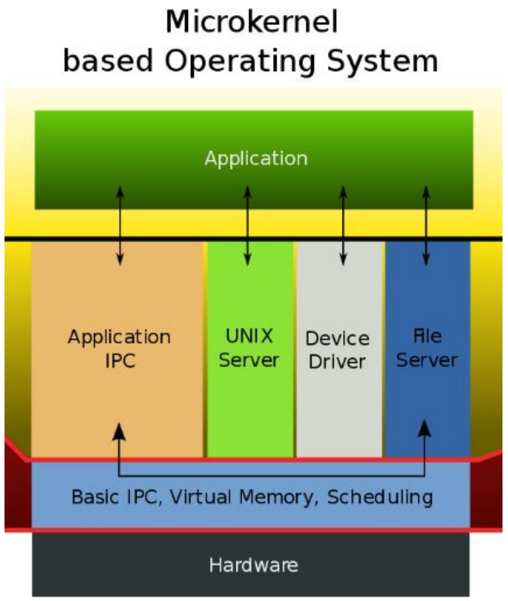
- most work in the monolithic kernel is done via **system calls**
- the monolithic Linux kernel can be made extremely small not only because of its ability to dynamically load modules but also because of its ease of customization
- coding in kernel can be challenging, **debugging** can be hard
- **bugs** in one part of the kernel have strong **side effects**, e.g. since the modules run in the same address space, a bug can bring down the entire system
- kernels often become very **large** and difficult to maintain
- even if the modules servicing these operations are separate from the whole, the code **integration** is tight and difficult to do correctly

Examples of monolithic kernels are Linux Kernel, AIX kernel, HP-UX kernel, Solaris kernel. But also the MS-DOS and BSD kernels (any unix-like kernel).

Microkernels

Microkernel (also abbreviated μK or uK) is the term describing an approach to operating system design by which the **functionality** of the system is **moved out** of the traditional "kernel", into a set of "**servers**" that communicate through a "**minimal**" kernel, leaving as little as possible in "system space" and as much as possible in "user space". A microkernel that is designed for a specific platform or device is only ever going to have what it needs to operate. The microkernel approach consists of defining a simple abstraction over the hardware, with a set of primitives or system calls to implement minimal OS services such as memory management, multitasking, and inter-process communication.

Microkernels are easier to maintain than monolithic kernels, but the large number of system calls and context switches might slow down the system because they typically generate more overhead than plain function calls.



Microkernels

Only parts which really require being in a privileged mode are in kernel space: IPC (Inter-Process Communication), basic scheduler, or scheduling primitives, basic memory handling, basic I/O primitives. **Many critical parts** are now running in user space: the complete scheduler, memory handling, file systems, and network stacks.

Essential traits:

- **easier** to maintain, to debug and code
- **rapid** development time and new software can be tested without having to reboot the kernel
- most microkernels use a **message passing system** to handle requests from one server to another. The message passing system generally operates on a port basis with the microkernel. As an example, if a request for more memory is sent, a port is opened with the microkernel and the request sent through. Once within the microkernel, the steps are similar to system calls but more **expensive** and messaging bugs are difficult to debug
- if a kernel process crashes, it is still possible to prevent a crash of the system as a whole by merely **restarting** the service that caused the error

Microkernels

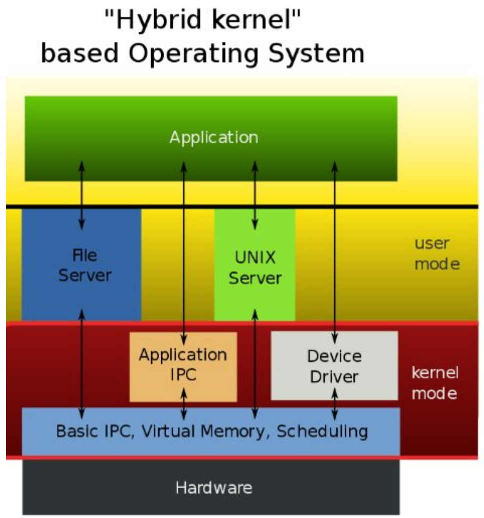
- a microkernel allows the implementation of the remaining part of the operating system as a normal application program written in a high-level language, and the use of different operating systems on top of the same unchanged kernel. It is also possible to dynamically switch among operating systems and to have more than one active simultaneously
- in terms of the source code size, microkernels are often smaller than monolithic kernels but can be larger when compiled

Examples of microkernels, GNU Hurd, MINIX 3 Kernel, first versions of the Mach kernel (originally from **M**ulti-**U**ser (or **M**ultiprocessor **U**niversal) **C**ommunication **K**ernel - MUCK) from Carnegie Mellon University.

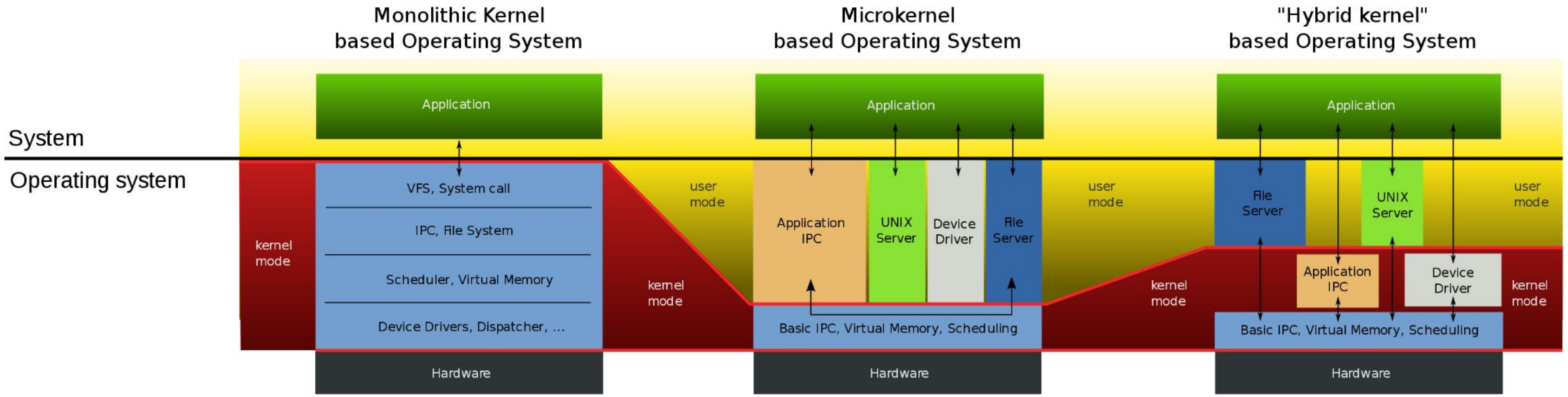
Hybrid Kernels

Hybrid kernels are used in most commercial operating systems such as Microsoft Windows NT 3.1, NT 3.5, NT 3.51, NT 4.0, 2000, XP, Vista, 7, 8, 8.1 and 10. Apple Inc's own macOS uses a hybrid kernel called XNU which is based upon code from OSF/1's Mach kernel (OSFMK 7.3) and FreeBSD's monolithic kernel. They are similar to micro kernels, except they include some additional code in kernel-space to increase performance.

Unlike monolithic kernels, these types of kernels are unable to load modules at runtime on their own. Hybrid kernels are micro kernels that have some "non-essential" code in kernel-space in order for the code to run more quickly than it would were it to be in user-space. Hybrid kernels are a compromise between the monolithic and microkernel designs. This implies running some services (such as the network stack or the filesystem) in kernel space to reduce the performance overhead of a traditional microkernel, but still running kernel code (such as device drivers) as servers in user space.



Kernel Types



Nano- and Exo- Kernels

Nanokernels

A nanokernel delegates virtually all services – including even the most basic ones like interrupt controllers or the timer – to device drivers to make the kernel memory requirement even smaller than a traditional microkernel.

Example: KeyKOS.

Exokernels

Exokernels are a still-experimental approach to operating system design. They differ from the other types of kernels in that their functionality is limited to the **protection and multiplexing of the raw hardware**, providing no hardware abstractions on top of which to develop applications. This separation of hardware protection from hardware management enables application developers to determine how to make the most efficient use of the available hardware for each specific program.

Nano- and Exo- Kernels

Exokernels

Exokernels in themselves are extremely small. However, they are accompanied by library operating systems (called **unikernel**), providing application developers with the functionalities of a conventional operating system. A major advantage of exokernel-based systems is that they can incorporate multiple library operating systems, each exporting a different API, for example one for high level UI development and one for real-time control.

Example of exokernel: Xen.

A **unikernel** is a specialised, single address space machine image constructed by using library operating systems (modular).

Example of tools for building unikernels: MirageOS

Bye

Final Course Survey

Please complete the following anonymous survey for helping me improve the course and my teaching approach. The survey will close on 31st May, 2021.

https://forms.gle/dpbgqLGDXYWKmEzG7

Thank you!

Advanced Operating Systems and Virtualization

[14] Epilogue

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

http://www.ce.uniroma2.it/~pellegrini/

gpm.name · proiettimattia@diag.uniroma1.it

DIAG