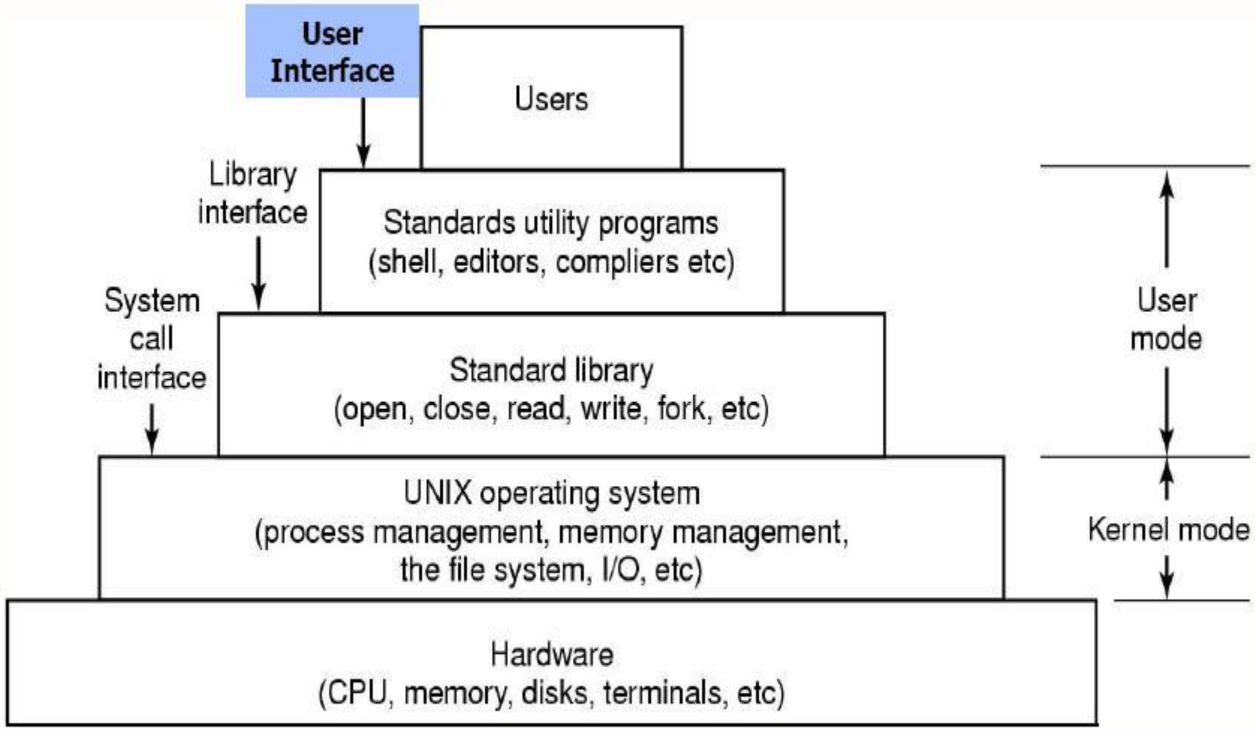
Introduction to Linux Operating System

Table of contents

- Operating system tasks
- UNIX history, Linux history
- Linux distributions
- Linux basic features
- Building OS kernels
- Linux kernel modules
- eBPF
- Kernel reports – what is going on in the kernel

Computer system layers (source: Stallings, Operating Systems)


User Interface
Users
Library interface
Standards utility programs (shell, editors, compliers etc)
System call interface
Standard library (open, close, read, write, fork, etc)
UNIX operating system (process management, memory management, the file system, I/O, etc)
Hardware (CPU, memory, disks, terminals, etc)
User mode
Kernel mode

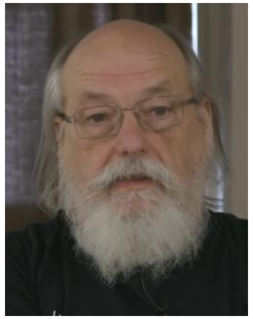
Operating System is a program that mediates between the user and the computer hardware.

- **Hides hardware details** of the computer system by creating abstractions (virtual machines).
- **Manages resources**: memory, processor (CPU), input/output, communication ports
- **Other activities**: security, job accounting, error detecting tools, etc.

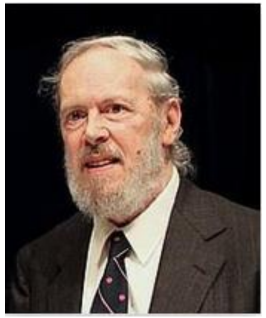
UNIX history

- Created in **1969**; authors: Ken Thompson, Denis Ritchie from Bell Laboratories, machine: old PDP-7; had many features of MULTICS.

  (Brian Kernighan participated in the creation of Unix, he is co-author of the first book about C).



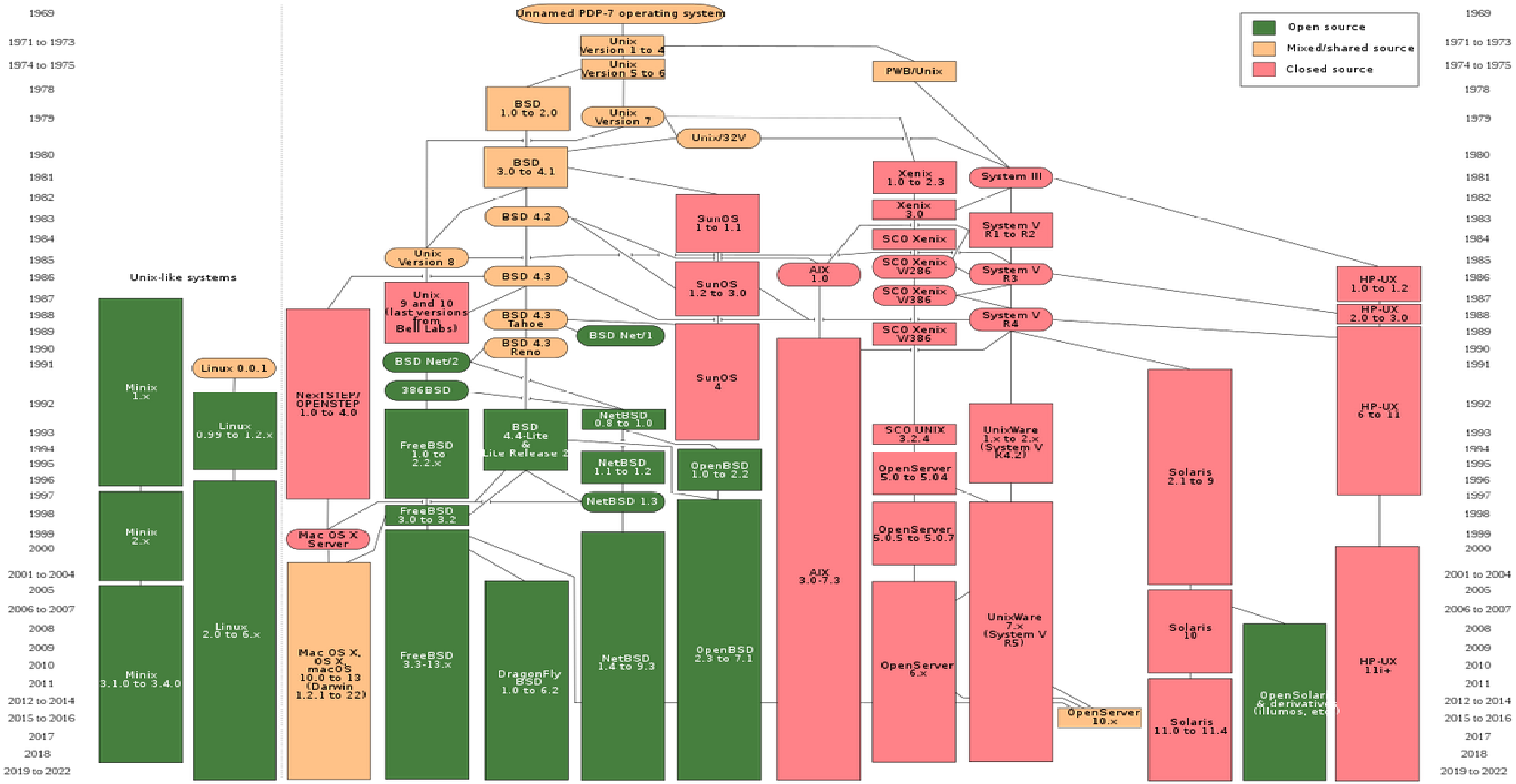
Ken Thompson



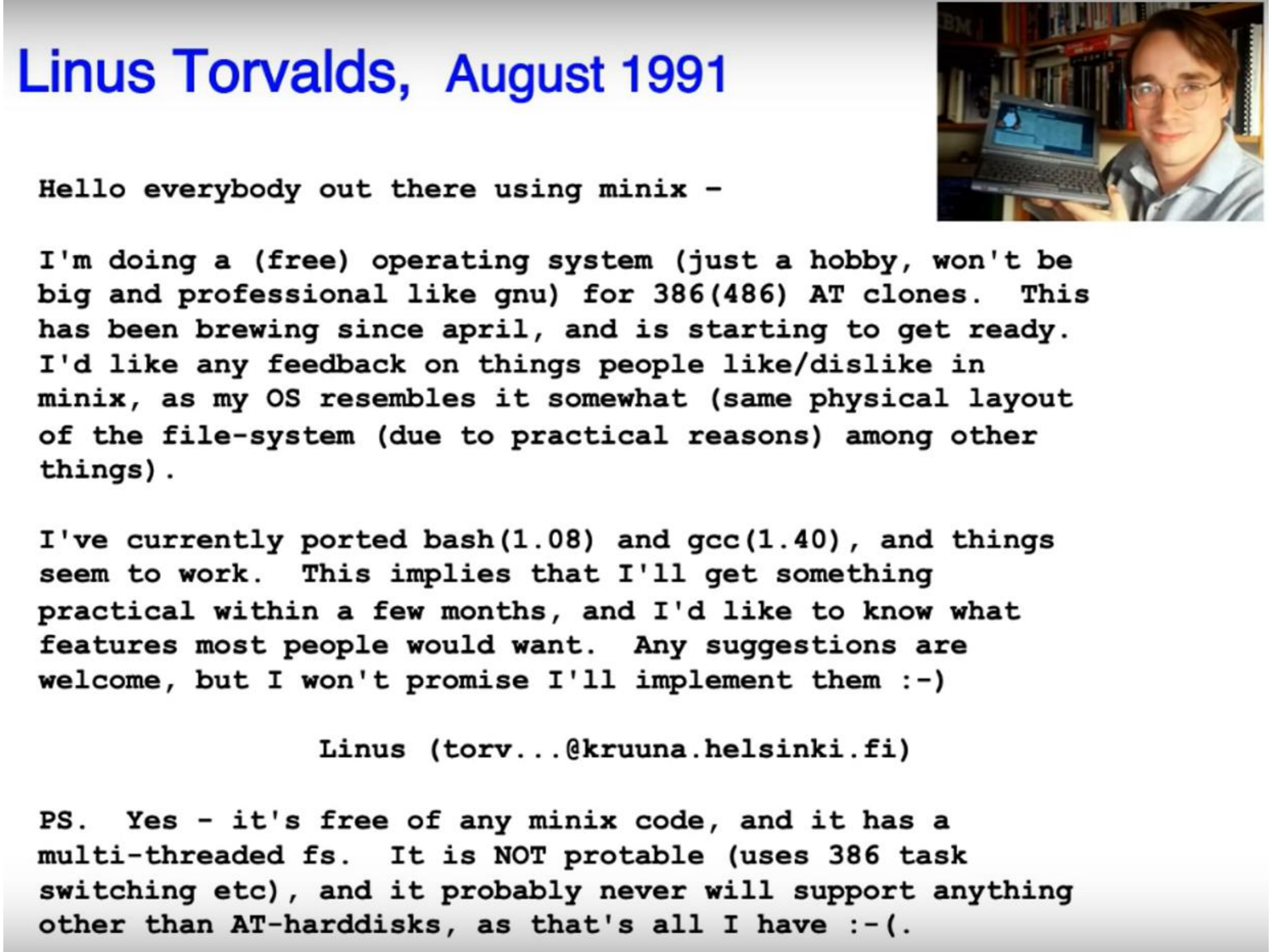
Denis Ritchie
died 12.10.2011

Brian Kernighan

- 1973: UNIX rewritten in C (language designed specifically for this purpose).
- 1974: presented on ACM Symposium on Operating Systems and in CACM, quickly gaining popularity.
- For hobbyists: Unix history, Unix, Linux, and variant history.
- The early days of Unix at Bell Labs, Brian Kernighan (LCA 2022 online).
- Ken Thompson interviewed by Brian Kernighan at VCF East 2019.



Unix History Diagram - short version (source: Wikipedia)



Linus Torvalds, August 1991

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torv...@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

Linux history

Linus Torvalds, Finland, born in the same year as UNIX, i.e. 1969, creator of the Linux kernel and the Git version control sysem.

Linus Torvalds announcing Linux 1.0, 30.03.1994

Linus Torvalds in 2023

in conversation with Dirk Hohndel at OSS Japan

Richard Stallman, founder of the GNU project and the Free Software Foundation, co-creator of the GNU GPL license, creator of the Emacs editor, GCC compiler, GDB debugger.

Richard Stallman in 2019

May 1991, version 0.01: no support for the network, limited number of device drivers, one file system (Minix), processes with protected address spaces

The Linux Kernel Archives – https://www.kernel.org/

- **2024-02-23**, latest stable version **6.7.6**
- **2024-02-18**, latest mainline **6.8-rc05**

Numbering of the kernel versions – see lab notes or Wikipedia

Andrew Tanenbaum in 2012

Linux statistics and facts

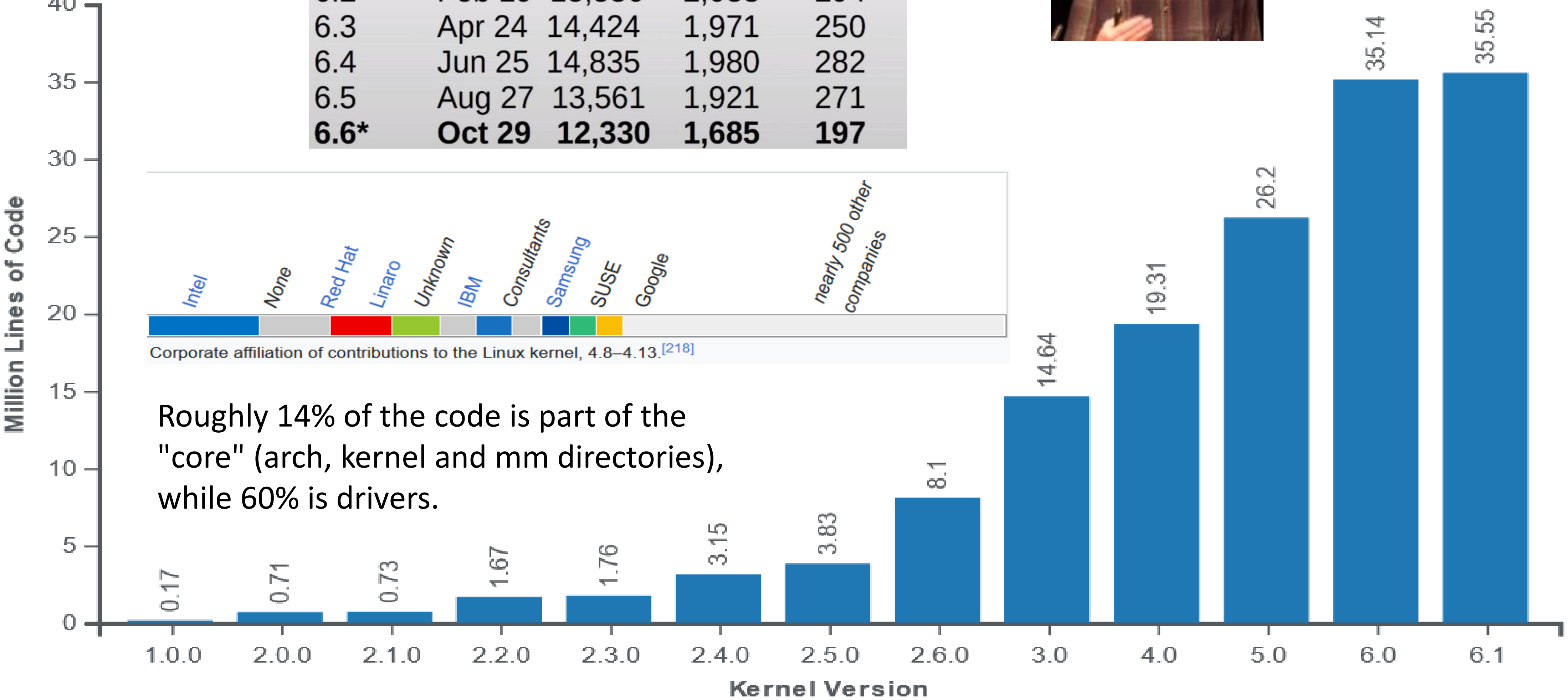
- In 2022, **100%** of the world’s top **500 supercomputers** run on Linux.
- All of the top **25 websites** in the world are using Linux.
- **96.3%** of the world’s top **one million servers** run on Linux.
- **90%** of all **cloud infrastructure** operates on Linux, and practically all the best cloud hosts use it.
- **90%** of Hollywood’s special effects are made on Linux

- In July 2022, **2.76% of all desktop operating systems worldwide ran on Linux**.
- In June 2022, Linux held a market share of **1.02%** of the global desktop/tablet/console market.
- In August 2022, the net market share of Linux was **2.35%**.
- In August 2022, **71.85%** of all mobile devices run on **Android**, which is Linux-based.

https://webtribunal.net/blog/linux-statistics/

Jonathan Corbet in 2023 Kernel Report :

Release	Date	Commits	Devs	1st time
6.0	Oct 2	15,402	2,034	236
6.1	Dec 11	13,942	2,043	303
6.2	Feb 19	15,536	2,088	294
6.3	Apr 24	14,424	1,971	250
6.4	Jun 25	14,835	1,980	282
6.5	Aug 27	13,561	1,921	271
6.6*	**Oct 29**	**12,330**	**1,685**	**197**



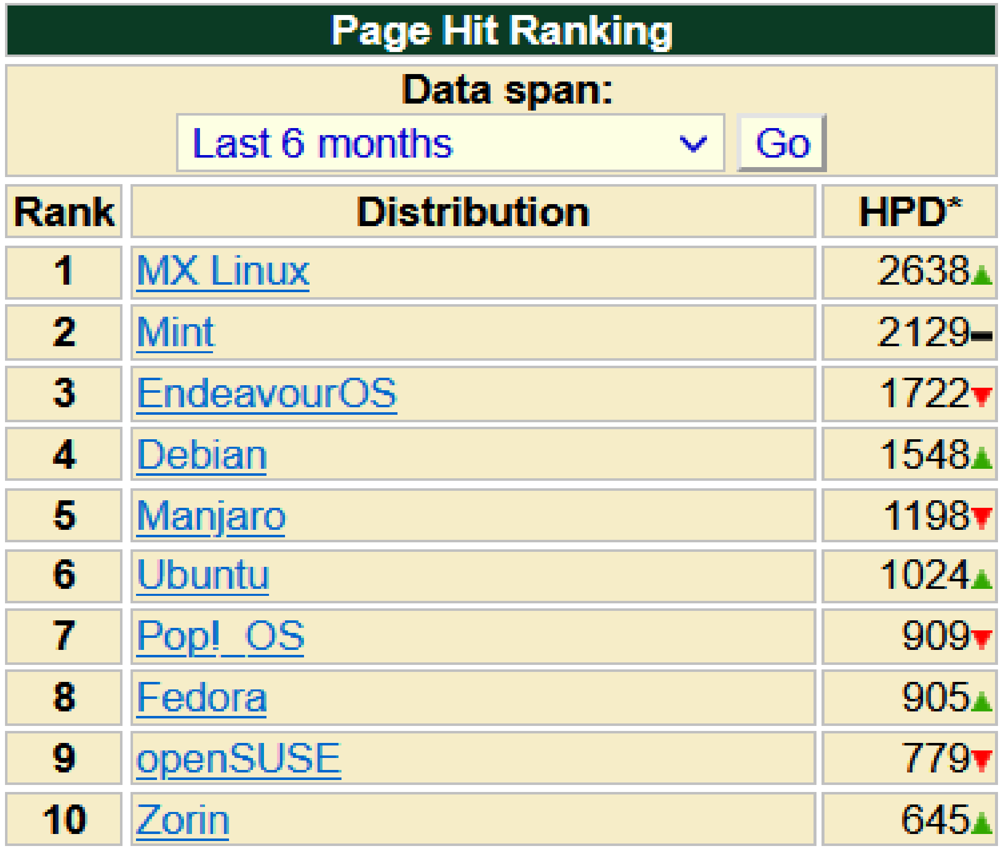
Corporate affiliation of contributions to the Linux kernel, 4.8–4.13.[218]

Roughly 14% of the code is part of the "core" (arch, kernel and mm directories), while 60% is drivers.

Linux kernel versions (source: Wikipedia)

Linux distributions

A set of ready-to-install, precompiled packages; tools for package installation and uninstallation (RPM: Red Hat Package Manager); kernel, but also many service programs; tools for file systems management, creation and maintenance of user accounts, network management etc.



Page Hit Ranking

Data span:

Last 6 months ⌄ Go

Rank	Distribution	HPD*
1	MX Linux	2638▲
2	Mint	2129▬
3	EndeavourOS	1722▼
4	Debian	1548▲
5	Manjaro	1198▼
6	Ubuntu	1024▲
7	Pop!_OS	909▼
8	Fedora	905▲
9	openSUSE	779▼
10	Zorin	645▲

DistroWatch is a website which provides news, popularity rankings, and other general information about Linux distributions as well as other free software/open source Unix-like operating systems.

← Debian used in labs

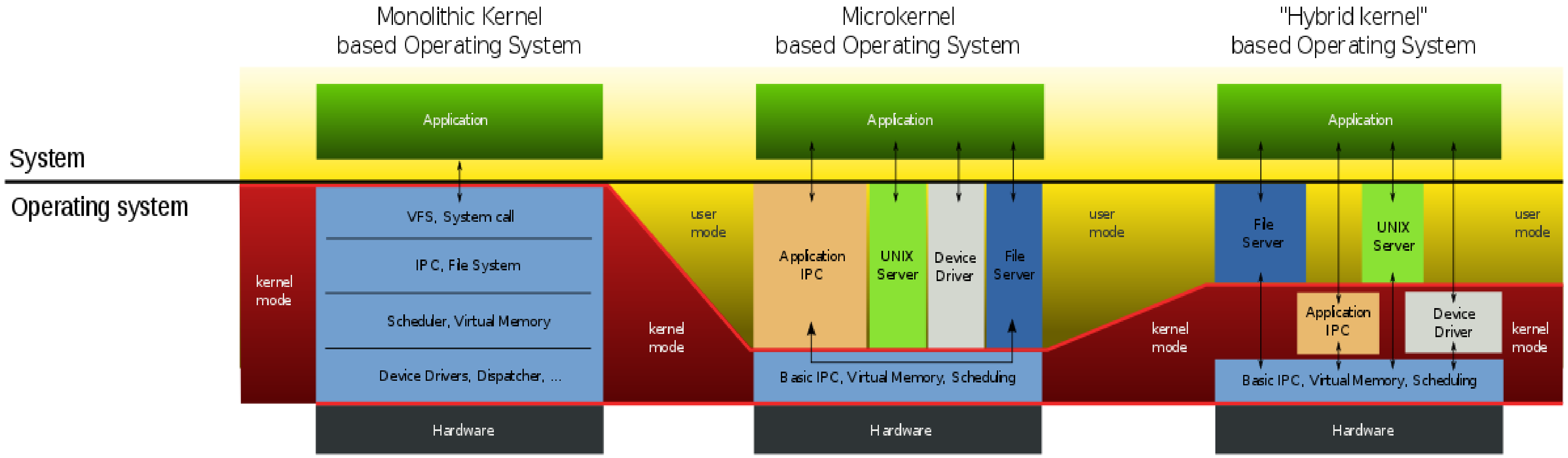
2023-11-12

Linux basic features

- Multi-access system (with time sharing) and multi-tasking.
- Multiprocess system, simple mechanisms to create hierarchy of processes, kernel preemption.
- Available for many architectures.
- Simple standard user interface that can be easily replaced (shell → command interpreter).
- Hierarchical file systems.
- Files are seen as strings of bytes (easy to write filters).
- Loading programs on demand (fork with copy on write).
- Virtual memory with paging.
- Dynamic hard disk cache.
- Shared libraries, loaded into memory dynamically (one code used simultaneously by many processes).
- Compliance with the POSIX 1003.1 standard.
- Different formats of executable files.

Building OS kernels

- **Monolithic kernel** (the only solution until the 1980s) – Linux belongs to this category.
  - the whole kernel runs in a single address space,
  - communication via direct function invocation.
- **Microkernel** (e.g. Mach, MINIX).
  - functionality of the kernel is broken down into separate processes (servers),
  - some servers run in kernel mode, but some in user mode – all servers have own address spaces,
  - communication is handled via message passing,
  - modularity – failure in one server does not bring down another, one server may be swapped out for another,
  - context switch and communication generate extra overhead so currently user mode servers are rarely used.
- **Macrokernel** or „**Hybrid kernel**" (e.g. Windows NT kernel on which are based Windows XP, Vista, Windows 7, Windows 10).



Structure of monolithic kernel, microkernel and hybrid kernel-based operating systems (source: Wikipedia)

Linus Torvalds :

“As to the whole ‘hybrid kernel’ thing - it’s just marketing. It’s ‘oh, those microkernels had good PR, how can we try to get good PR for our working kernel? Oh, I know, let’s use a cool name and try to imply that it has all the PR advantages that that other system has’.”

Readings

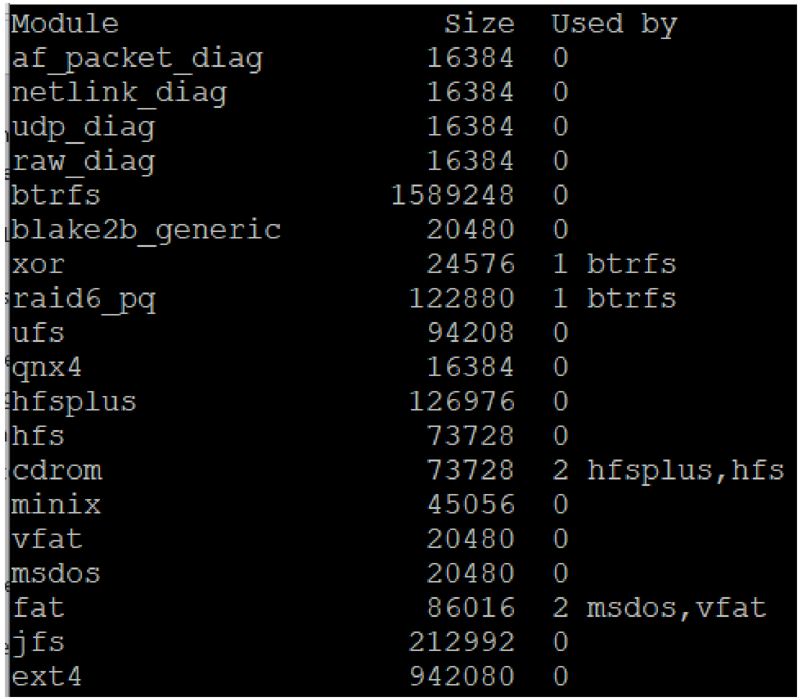
1. Tanenbaum – Torvalds debate on kernel architecture (MINIX vs Linux)
   - Wikipedia
   - Oreilly
2. Comparing Linux and Minix, February 5, 2007, Jonathan Corbet

Linux kernel modules

- Linux borrows much of the good from microkernels: modular design, capability to preempt itself, support for kernel threads, capability to dynamically load separate binaries (kernel modules).
- **Modules** – separately compiled, loaded into memory on demand and deleted when they are no longer needed.
- Examples: a device driver, a file system, an executable file format.

- Advantages: saving memory (occupies memory only when it is needed), the error in the module does not suspend the system, but only removes the module from the memory, one can use conflicting drivers without the need to restart the system, etc.
- Disadvantages ???

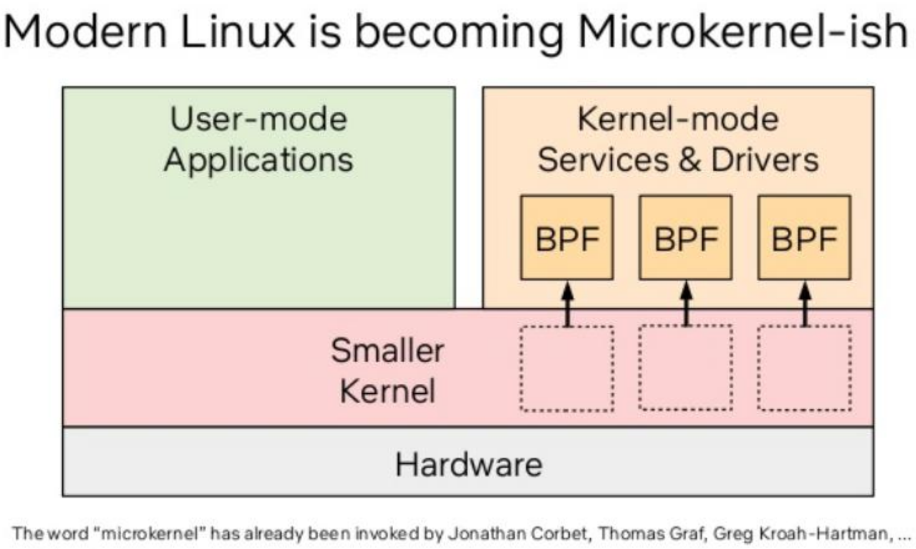
cat /proc/modules



```
Module                  Size  Used by
af_packet_diag         16384  0
netlink_diag           16384  0
udp_diag               16384  0
raw_diag               16384  0
btrfs                1589248  0
blake2b_generic        20480  0
xor                    24576  1 btrfs
raid6_pq              122880  1 btrfs
ufs                    94208  0
qnx4                   16384  0
hfsplus               126976  0
hfs                    73728  0
cdrom                  73728  2 hfsplus,hfs
minix                  45056  0
vfat                   20480  0
msdos                  20480  0
fat                    86016  2 msdos,vfat
jfs                   212992  0
ext4                  942080  0
```

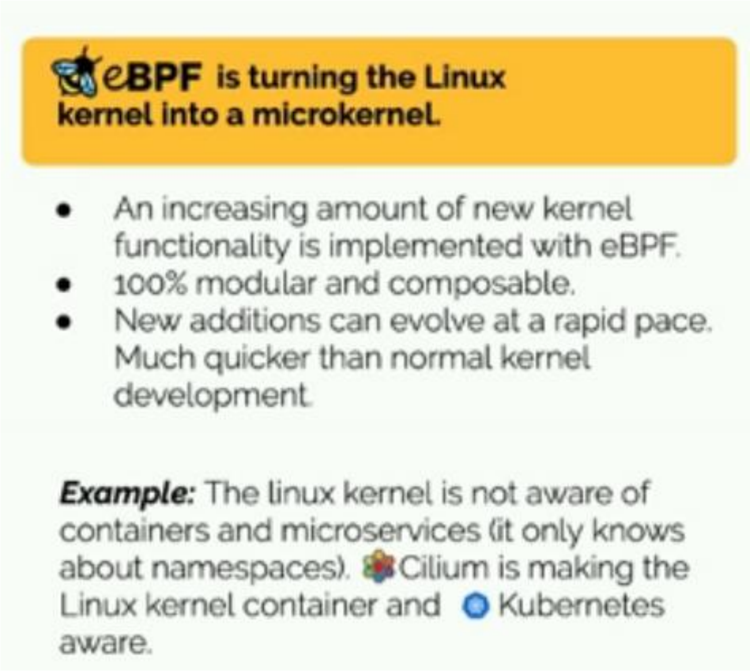
- *name of the module*
- *memory size of the module, in bytes*
- *how many instances of the module are currently loaded*
- *if the module depends upon another module(s)*

But – eBPF makes a change ...



Modern Linux is becoming Microkernel-ish

The word "microkernel" has already been invoked by Jonathan Corbet, Thomas Graf, Greg Kroah-Hartman, ...



eBPF is turning the Linux kernel into a microkernel.

- An increasing amount of new kernel functionality is implemented with eBPF.
- 100% modular and composable.
- New additions can evolve at a rapid pace. Much quicker than normal kernel development.

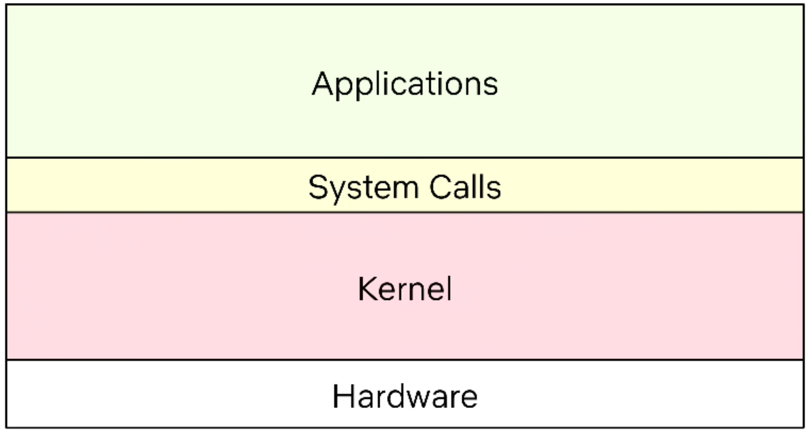
Example: The linux kernel is not aware of containers and microservices (it only knows about namespaces). Cilium is making the Linux kernel container and Kubernetes aware.

Extended BPF: A New Type of Software, Brendan Gregg at Ubuntu Masters Conf 2019
(presentation, slides)

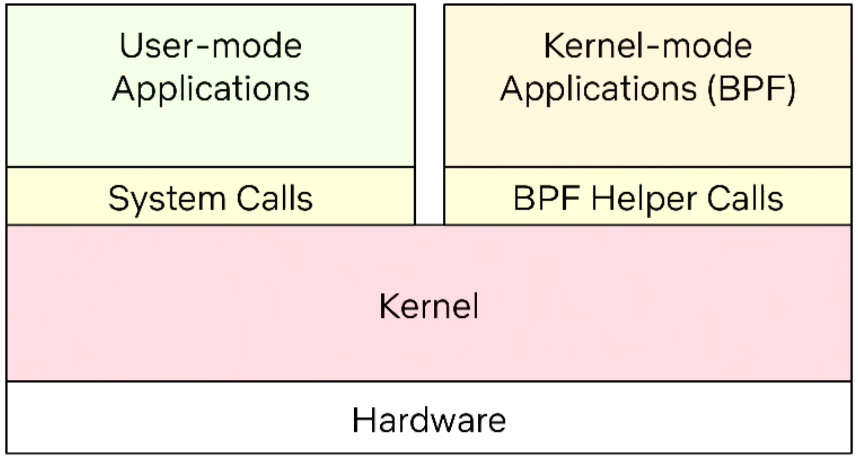
eBPF – Rethinking the Linux Kernel, Thomas Graf, QCon 2020
(presentation, transcript)

Thomas Graf: *With BPF, we're starting to implement a **microkernel model** where we can now dynamically load programs, we can dynamically replace logic in a safe way, we can make logic composable. We're going away from the requirement that every single Linux kernel change requires full consensus across the entire industry or across the entire development community and instead, you can define your own logic, you can define your own modules and load them safely and with the necessary efficiency.*

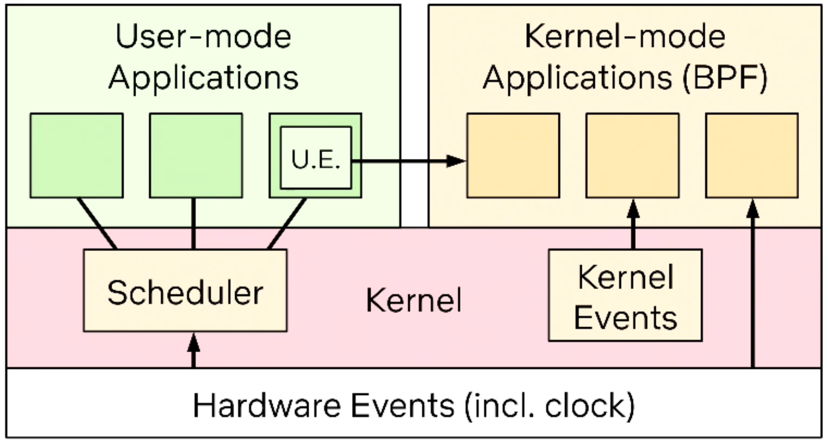
50 Years, one (dominant) OS model



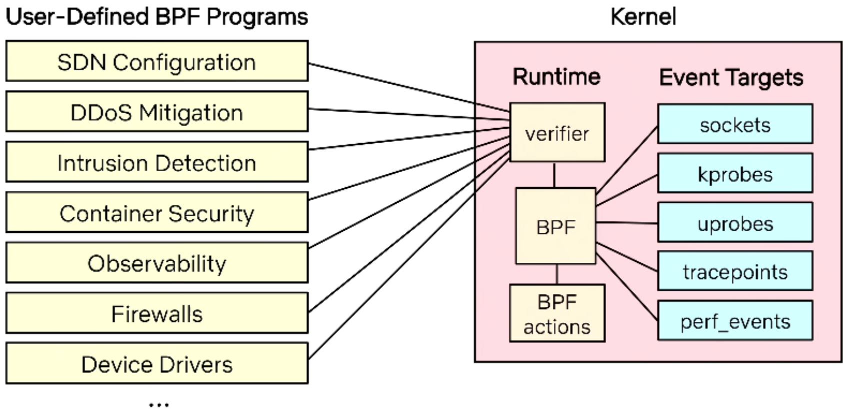
Modern Linux: A new OS model



Modern Linux: Event-based Applications

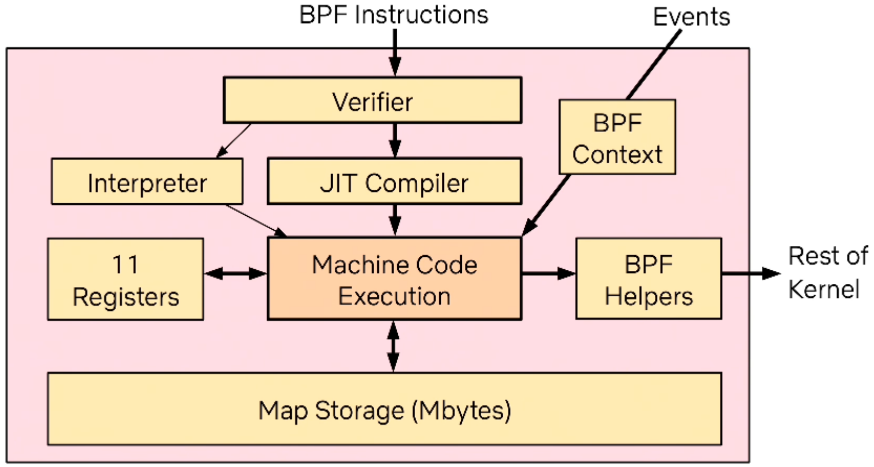


BPF 2019



Extended BPF: A New Type of Software, Brendan Gregg at Ubuntu Masters Conf 2019 (presentation, slides)

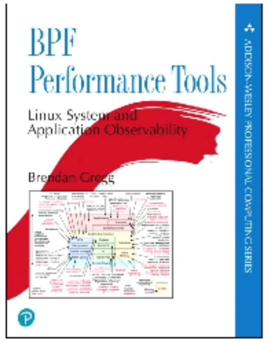
BPF Internals

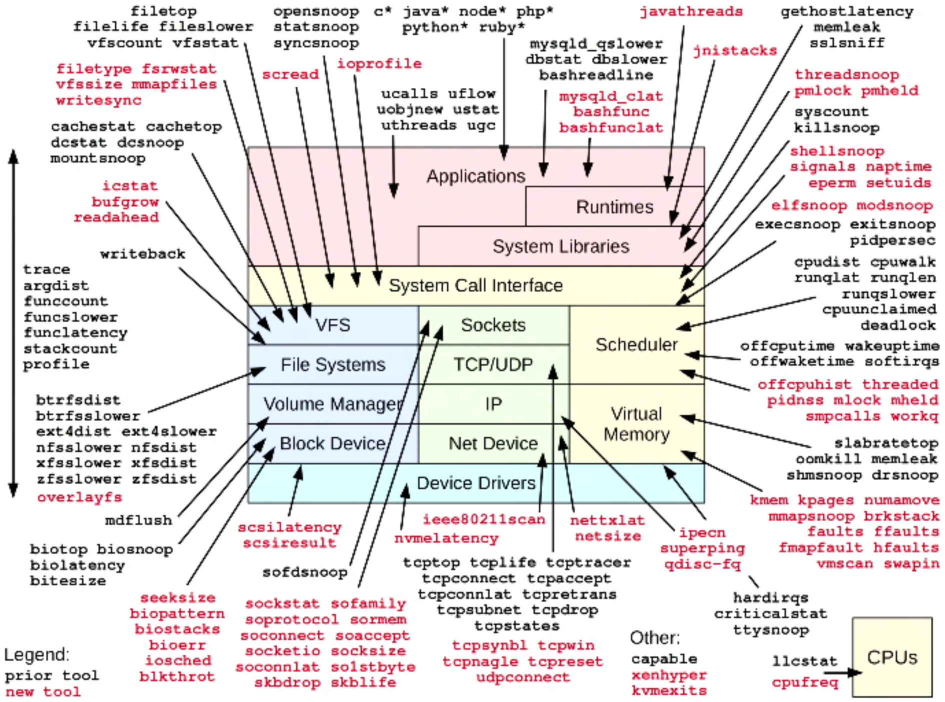


A New Type of Software

	Execution model	User defined	Compil-ation	Security	Failure mode	Resource access
User	task	yes	any	user based	abort	syscall, fault
Kernel	task	no	static	none	panic	direct
BPF	event	yes	JIT, CO-RE	verified, JIT	error message	restricted helpers

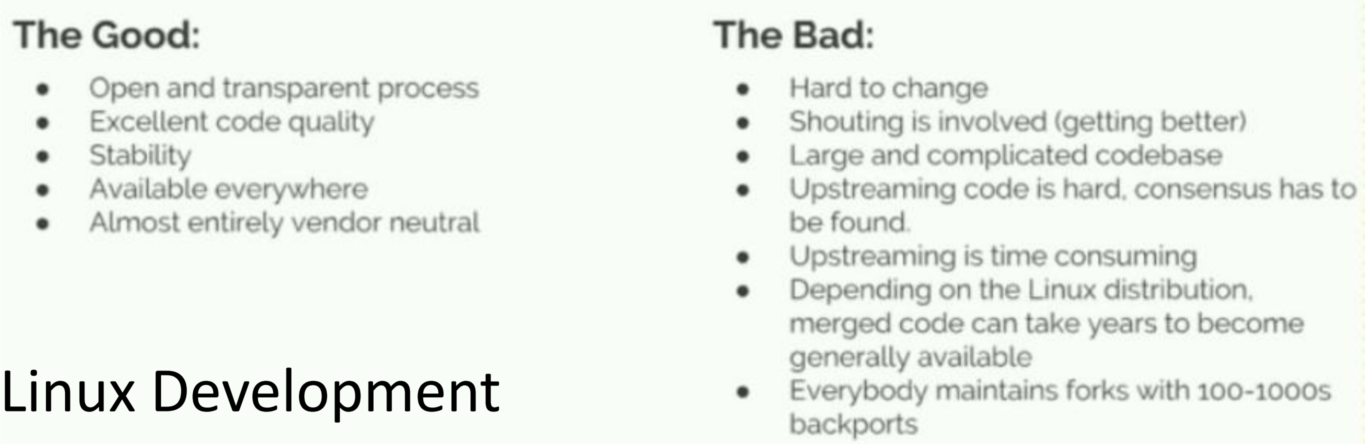
BPF Perf Tools





Extended BPF: A New Type of Software, Brendan Gregg at Ubuntu Masters Conf 2019 ([presentation](), [slides]())

http://brendangregg.com



The Good:

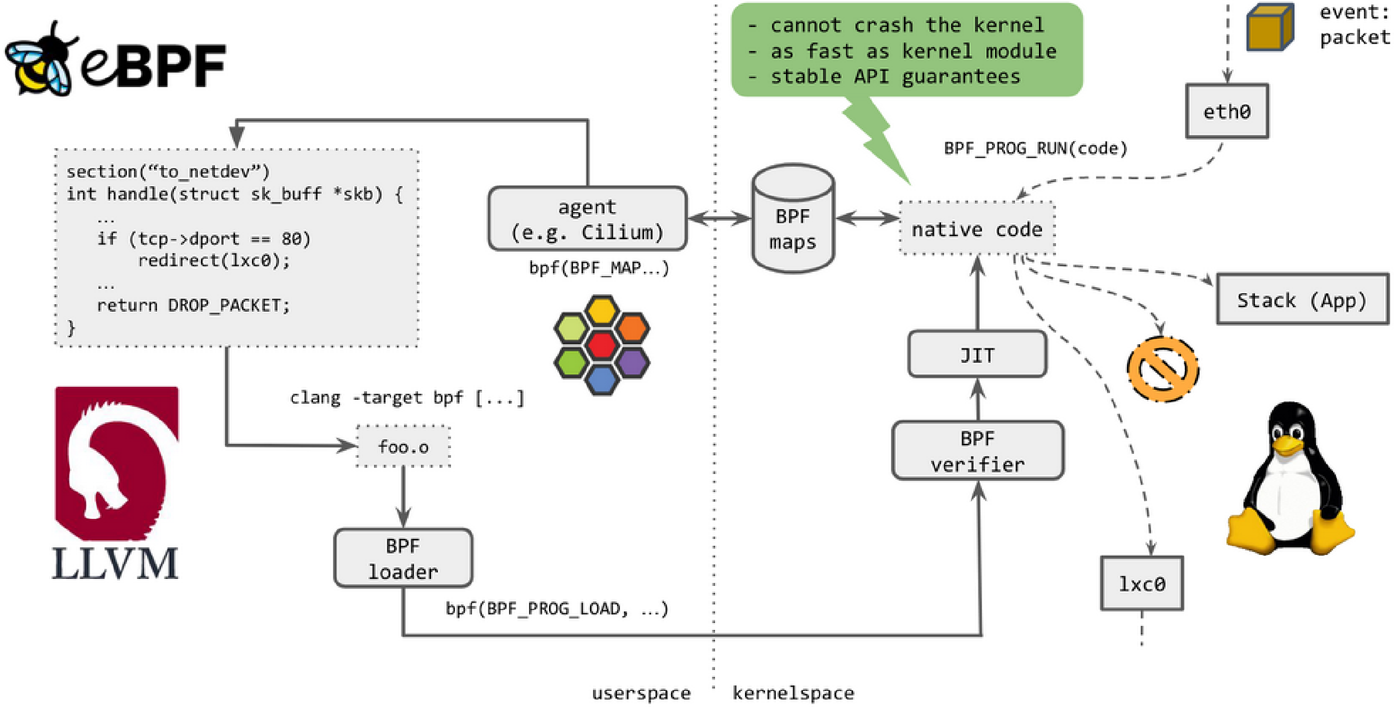
- Open and transparent process
- Excellent code quality
- Stability
- Available everywhere
- Almost entirely vendor neutral

The Bad:

- Hard to change
- Shouting is involved (getting better)
- Large and complicated codebase
- Upstreaming code is hard, consensus has to be found.
- Upstreaming is time consuming
- Depending on the Linux distribution, merged code can take years to become generally available
- Everybody maintains forks with 100-1000s backports

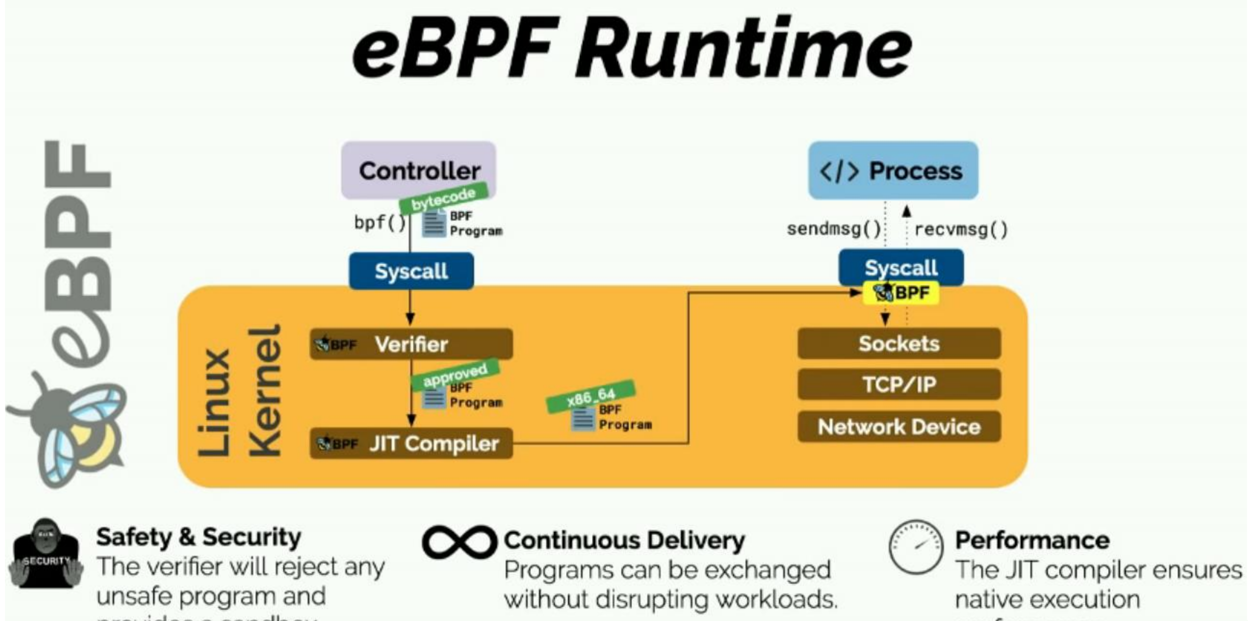
Linux Development

What is BPF?
Highly efficient sandboxed virtual machine in the Linux kernel making the Linux kernel programmable at native execution speed.



```
section(“to_netdev”)
int handle(struct sk_buff *skb) {
   ...
   if (tcp->dport == 80)
       redirect(lxc0);
   ...
   return DROP_PACKET;
}
```

How to Make Linux Microservice-Aware with Cilium and eBPF, Thomas Graf, QCon 2018, (presentation, transcript)



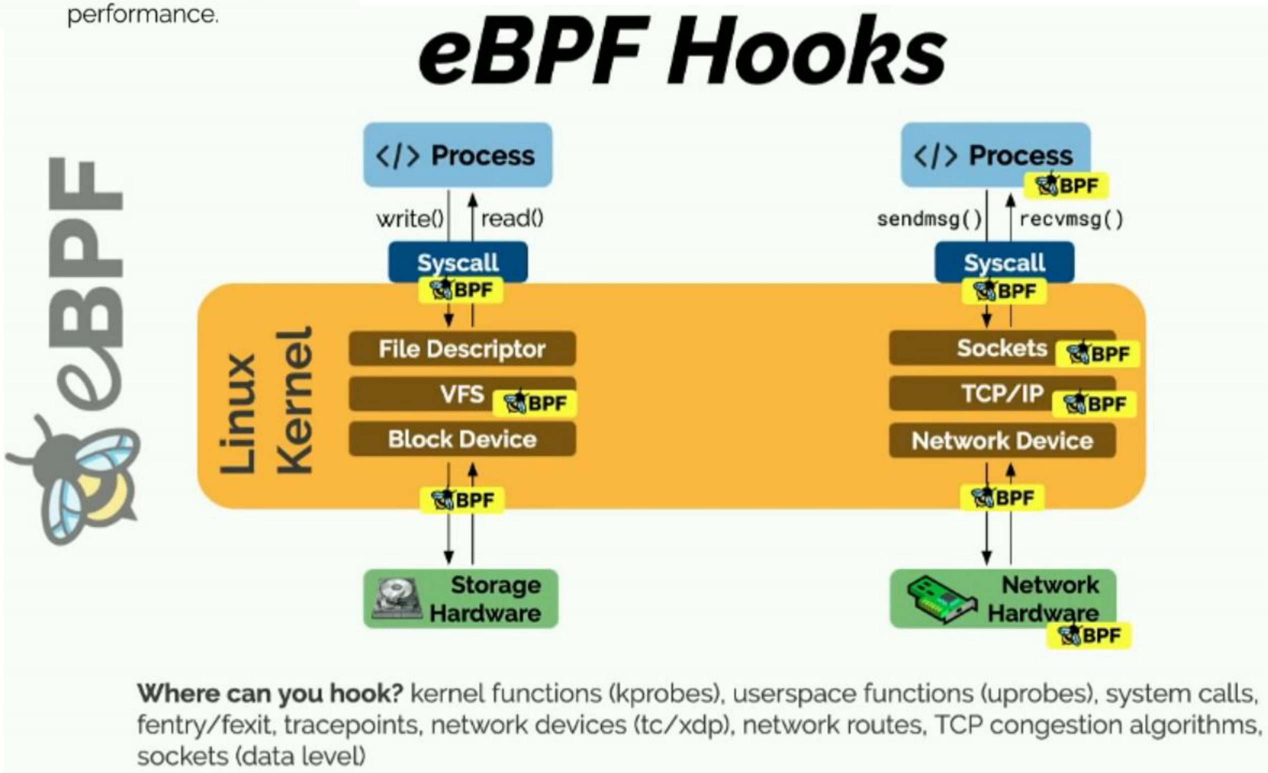
eBPF Runtime

Safety & Security
The verifier will reject any unsafe program and provides a sandbox.

Continuous Delivery
Programs can be exchanged without disrupting workloads.

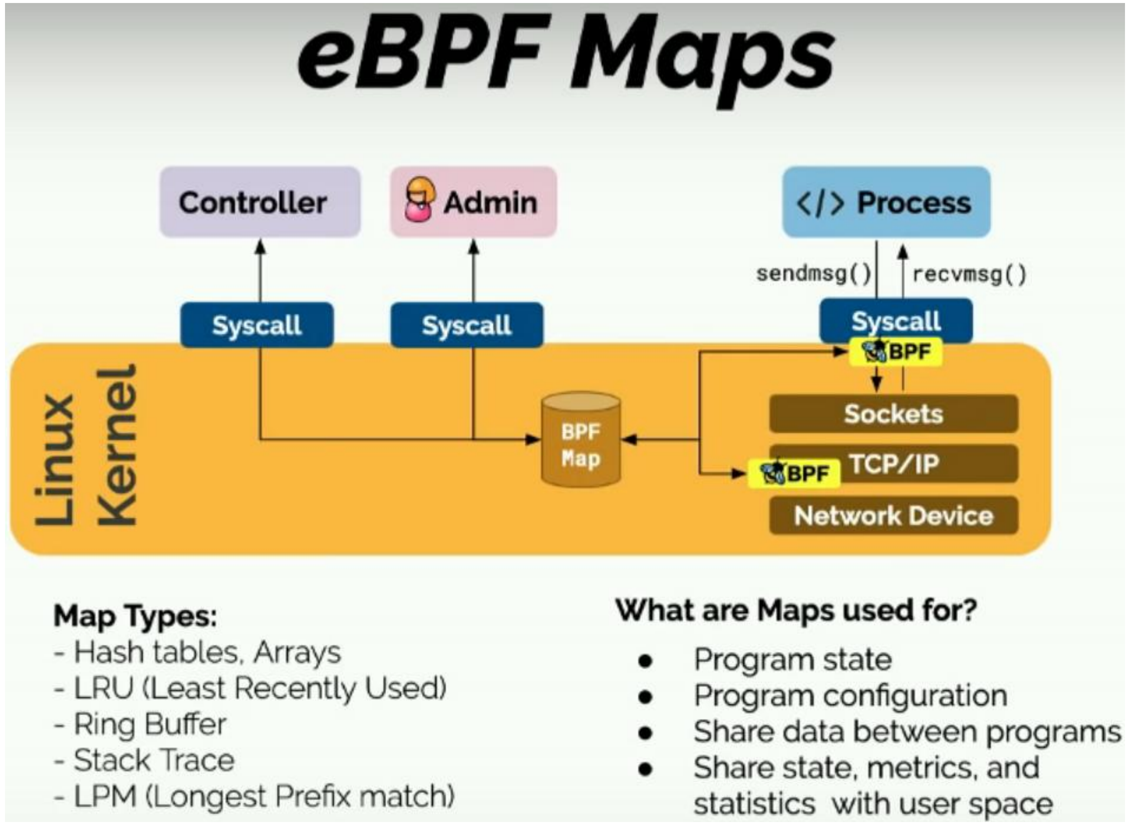
Performance
The JIT compiler ensures native execution performance.

eBPF – Rethinking the Linux Kernel, Thomas Graf, QCon 2020 (presentation, transcript)



eBPF Hooks

Where can you hook? kernel functions (kprobes), userspace functions (uprobes), system calls, fentry/fexit, tracepoints, network devices (tc/xdp), network routes, TCP congestion algorithms, sockets (data level)



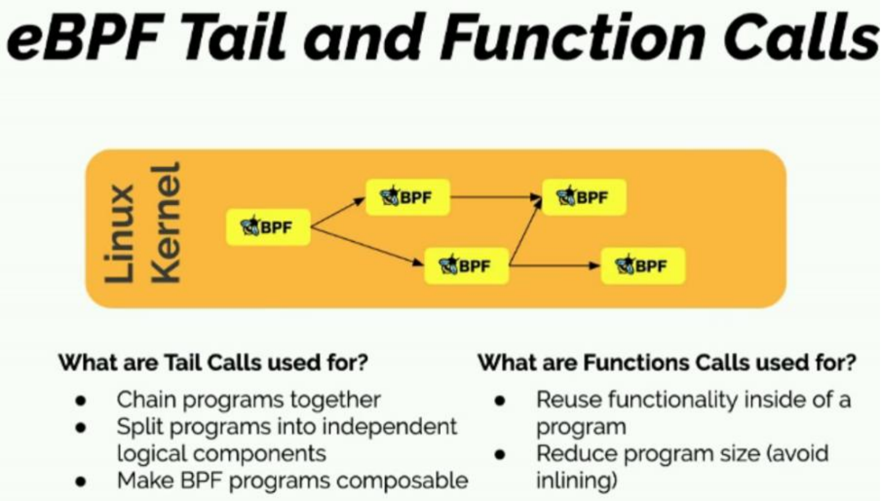
eBPF Maps

Map Types:
- Hash tables, Arrays
- LRU (Least Recently Used)
- Ring Buffer
- Stack Trace
- LPM (Longest Prefix match)

What are Maps used for?

- Program state
- Program configuration
- Share data between programs
- Share state, metrics, and statistics with user space

eBPF – Rethinking the Linux Kernel, Thomas Graf, QCon 2020 (presentation, transcript)



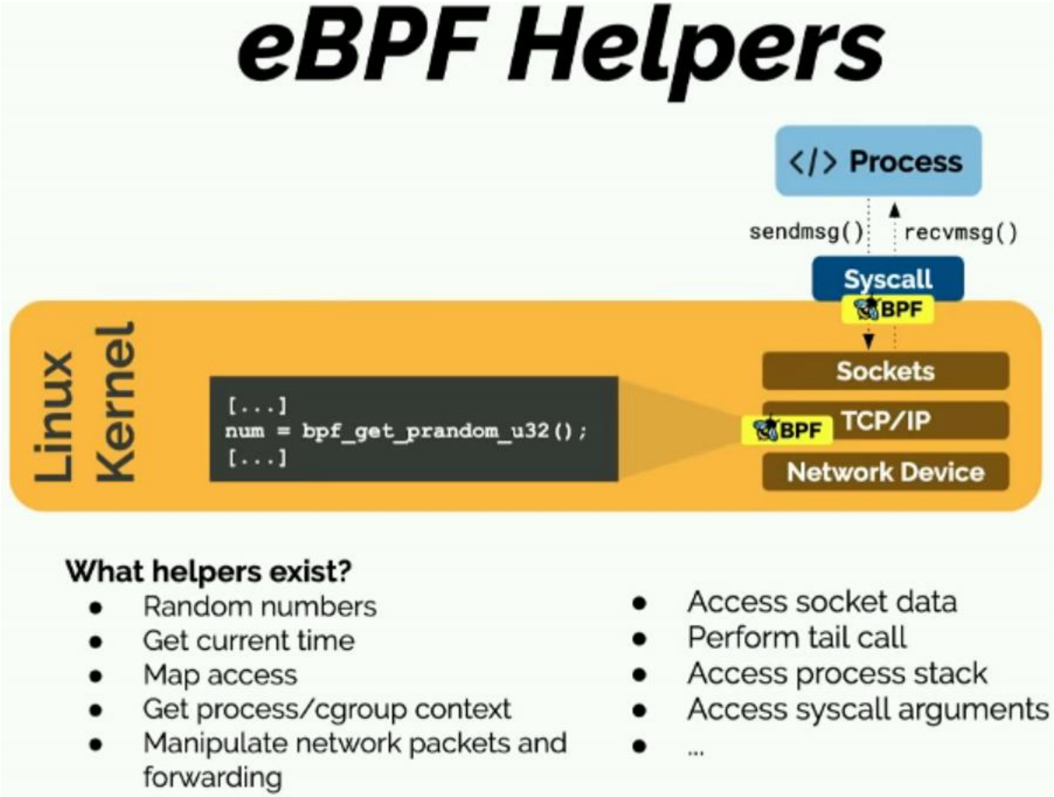
eBPF Tail and Function Calls

What are Tail Calls used for?

- Chain programs together
- Split programs into independent logical components
- Make BPF programs composable

What are Functions Calls used for?

- Reuse functionality inside of a program
- Reduce program size (avoid inlining)



eBPF Helpers

```
[...]
num = bpf_get_prandom_u32();
[...]
```

What helpers exist?

- Random numbers
- Get current time
- Map access
- Get process/cgroup context
- Manipulate network packets and forwarding
- Access socket data
- Perform tail call
- Access process stack
- Access syscall arguments
- ...

eBPF – summary

> “It seems that the Linux kernel continues its march towards becoming BPF runtime-powered microkernel.”
> — Toke Høiland-Jørgensen, December 2019

- In-kernel *just-in-time compiler*.
- Extensive verification for safety (**built-in verifier**).
- Many places to **attach programs**: packet filters, tracepoints, security policies, ...
- Enable the **addition of new functionality** – no kernel hacking required.
- Highly flexible kernel configuration.
- **Fast!**

Alexei Starovoitov

The Beginner’s Guide to eBPF, Liza Rice (live programming + source code)

What is eBPF? – eBPF portal

BPF at Facebook, Performance Summit 2019, Alexei Starovoitov

BPF at Facebook, (slides) Kernel Recipes 2019, Alexei Starovoitov

A thorough introduction to eBPF (four articles in lwn.net), Matt Fleming, December 2017.

BPF compiler collection (BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more)

eBPF – official documentary

https://www.youtube.com/watch?v=Wb_vD3XZYOA (30 min)

- In **2014**, a group of engineers at Plumgrid needed to find an innovative and cost-effective solution to handle network traffic in SDN environments. What they created was a landmark in the industry known as the extended **Berkeley Packet Filter** (or **eBPF**). This vital technology allows user-level code execution inside the Linux Kernel, transforming network traffic handling for SDN environments. Whether these engineers knew it or not, they had just revolutionized the Linux Kernel.
  - Growth of Linux and SDN
  - PLUMgrid
  - Initial Patch Submission
  - eBPF Merged into the Linux Kernel
  - Hyperscalers Adopt eBPF
  - Cilium Bring eBPF to End Users
  - DockerCon 2017 eBPF Takes Off
  - eBPF Expands to Security
  - eBPF on Windows
  - eBPF Everywhere

Thomas Graf
Daniel Borkmann
Chris Wright
Liz Rice
Purvi Desai (Google)
David Miller (network kernel maintainer)
Alexei Starovoitov
Brendan Gregg
Dave Thaler (Microsoft)

https://ebpfdocumentary.com/

What is going on in the kernel – kernel reports

- The Kernel Report, Jonathan Corbet, Open Source Summit EU 2023

  This talk will review recent events in the kernel development community, discuss the current state of the kernel and the challenges it faces, and look forward to how the kernel may address those challenges.
- The Kernel Report, Jonathan Corbet, Open Source Summit 2022

  The Kernel Report, Jonathan Corbet, Linux Plumbers Conference 2021 (starting from 6:45)
- The Kernel Report, Jonathan Corbet, LPC 2020, 2020 edition.
- The Kernel Report, Jonathan Corbet, linux.conf.au 2019 edition.
- The Kernel Report, Jonathan Corbet, Open Source Summit, 2018 edition.

- Linux Weekly News
  - Kernel index
  - Conference index

LWN.net

News from the source

The Kernel Report 2023

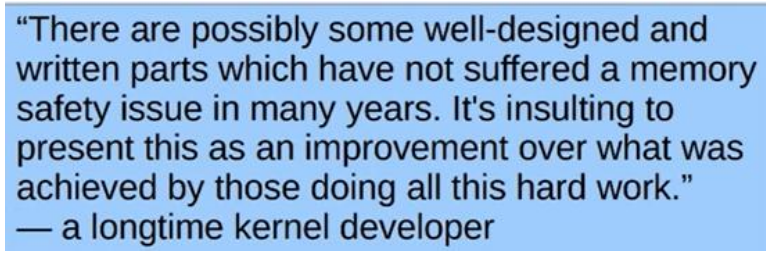
- **BPF – how far do we go?**
  - What BPF ***can*** do?

    Packet filtering, TCP congestion control, traffic control, rRouting++ w/XDP, infrared drivers, input drivers, system-call filtering (seccomp), tracing and analysis …
  - What BPF ***might*** do?
    - The extensible scheduler class (write complete CPU schedulers in BPF)

      - Developed by engineers from Meta and Google.
      - Why: easy experimentation, faster scheduler development, ad hoc schedulers for special workloads.
      - Why ***not***: added mainteance burden, benchmark gaming, vendors may require specific schedulers, ABI concerns, redirection of work on core scheduler.
      - Rejected by scheduler maintainer (**Peter Zijlstra**).
    - **Page aging** (why: adjust memory-management to workload).
    - **Io_uring integration** (why: better control over sequences of operations, create a complete programming environment).

The Kernel Report 2023

- **Rust**
  - Has a lot to offer (a stronger type system, no undefined behavior, attractive to newer developers).
  - Why ***not*** Rust in the kernel (a new language adds complexity, the language is still evolving – quickly, maintainers will need to learn Rust, lots of glue code, some things are hard to do in Rust, conservatism).
  - Initial Rust infrastructure has been merge into Linux 6.1 (October 2022).
  - More support code in subsequent kernels (access to existing types and functions … but safer).
  - Nothing in a production kernel yet, nothing that anybody is actually using.
  - Rust support was merged as an **experiment.**
  - The Rust **decision** point is coming soon.

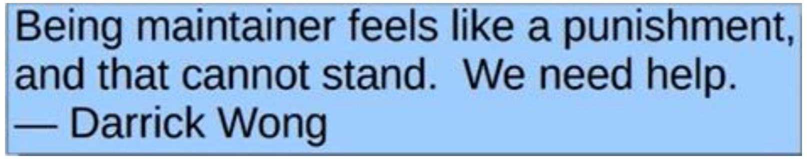


"There are possibly some well-designed and written parts which have not suffered a memory safety issue in many years. It's insulting to present this as an improvement over what was achieved by those doing all this hard work."
— a longtime kernel developer

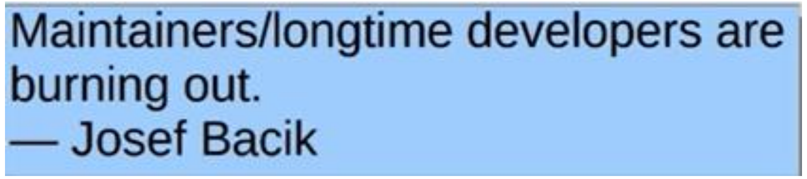
Rust-for-Linux developer Miguel Ojeda

The Kernel Report 2023

- **The maintainership crisis**
  - Increasing demands.
  - Understaffing.
  - Lack of employer support (many maintainers are not paid to maintain).
  - Kernel fuzzers (bad quality bug reports).
  - Dark areas (documentation, build system, many core-kernel areas, drivers for older hardware …).
  - Maintainers.
  - https://www.kernel.org/doc/html/latest/process/contribution-maturity-model.html



Being maintainer feels like a punishment, and that cannot stand. We need help.
— Darrick Wong



Maintainers/longtime developers are burning out.
— Josef Bacik

Slides: https://lwn.net/talks/2023/kr-osseu.pdf

2023 Kernel Maintainers Summit group photo

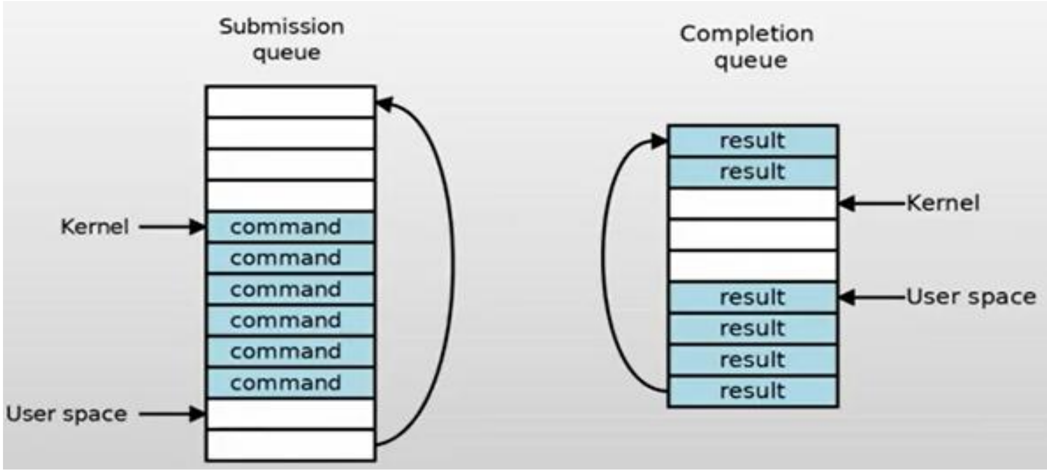
Back row: Alexei Starovoitov, Dave Chinner, Konstantin Ryabitsev, Thomas Gleixner, Jakub Kicinski, Josef Bacik, Dan Williams, Wolfram Sang, Rafael Wysocki, Kees Cook, Greg Kroah-Hartman, Jiri Kosina.

Front row: Tejun Heo, Jens Axboe, Sasha Levin, Ted Ts'o, Jan Kara, Linus Torvalds, Martin Petersen, Christian Brauner, Steve Rostedt, Arnd Bergmann.

Floor: Miguel Ojeda.

The Kernel Report 2022

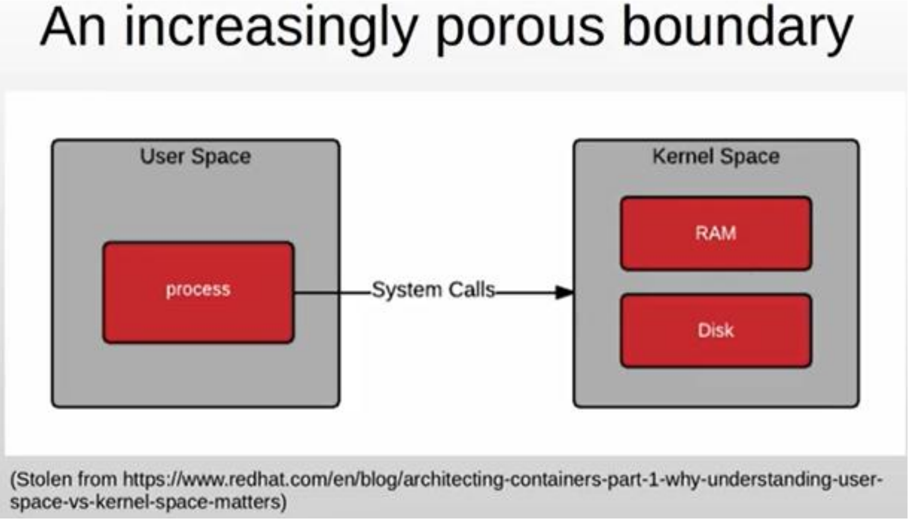
- **Bugs in the kernel**
  - Fixing bugs will take a long time.
  - Some bugs are very old.
- **Rust**
  - Can help (enforce rules, e.g. locking, eliminate undefinded behavior, bring in new developers).
  - What’s the holdup (a difficult learning curve, the language is still evolving, some things are hard to do in Rust, conservatism).
  - Initial Rust infrastructure has been merge into Linux 6.1 (October 2022).
  - A pair od Rust kernel modules (NVM Express driver, 9P filesystem server)
- **Io_uring**
  - System calls slow down your program.
  - Shared memory area (user, kernel).
  - What it brings
    - Asynchronous operations.
    - Submission/results without system calls.
    - Registered files and buffers
    - A wide range of commands.
    - Chained operations.



io_uring is an alternative, high-performance API that runs within the kernel

The Kernel Report 2022

- **Io_uring (continued)**
  - User-space block driver using io_uring (ublk)
  - Is io_uring the basis for future microkernel architecture?
- **Holes in the boundary**
  - BPF.
  - DAMON/DAMOS (memory management decisions to be pushed under user space control).
  - Userfaultfd().
  - Seccomp().
  - XDP (networking subsystem).



An increasingly porous boundary

(Stolen from https://www.redhat.com/en/blog/architecting-containers-part-1-why-understanding-user-space-vs-kernel-space-matters)

Linux systems will look a lot different in the future.

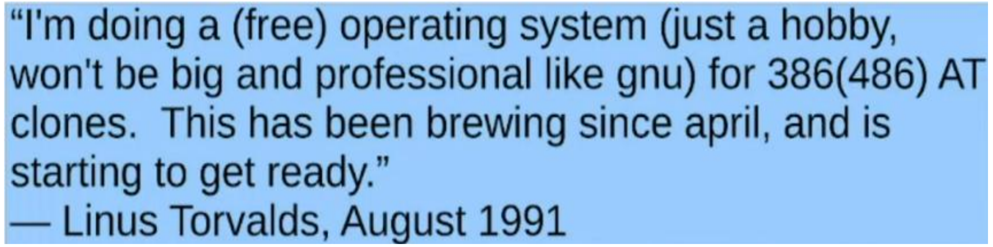
- **Generational change**
  - An unparalleled depth of skills and experience.
  - But also resistance to change (e.g. Rust), lack of diversity, increasingly tired single points of failure.
  - Preparing for change (shared maintenance duties, documenation, investment in tools).

The Kernel Report 2021

- **Security** (LLVM Control-flow integrity)
- **Core scheduling**
  - Allow processes to spy on each other or disable SMT (Simultaneous multi-threading).
  - Don’t let untrusting processes share an SMT core (v5.14 or later).
  - Processes can be assigned a „cookie” value, SMT siblings only shared by processes with the same cookie.
- **Landlock**
  - Load rules to restrict filesystem access.
  - An unprivileged sandboxing mechanism.
  - Merged for 5.13.
- **Patch attestation.**
- **The UMN affair** (five buggy patches sent under made-up names).
- **Rust in the kernel** (a memory-safe environment, avoid undefined behavior)
- **Runtime verification**.
- **Realtime** (work started in 2004, in 2022 will finally be merged).

The Kernel Report 2021

- **io_uring**
  - Asynchronous I/O that actually works.
  - More operations (not just I/O anymore).
  - File operations without file descriptors.
  - BPF support.
- **BPF**
  - BPF for Windows.
  - Atomic operations.
  - Sleepable BPF programs.
  - Direct calls to kernel functions.
  - Signed BPF programs (in progress).
- **30 years later – what have we learnt? (Linus Torvalds 1991)**
  - Tools matter.
  - Maintaining compatibility is important.
  - Vendor independence is crucial.
  - Code quality and maintainability over features.
  - Copyleft holds things together.
  - We can do it, we can do it better!

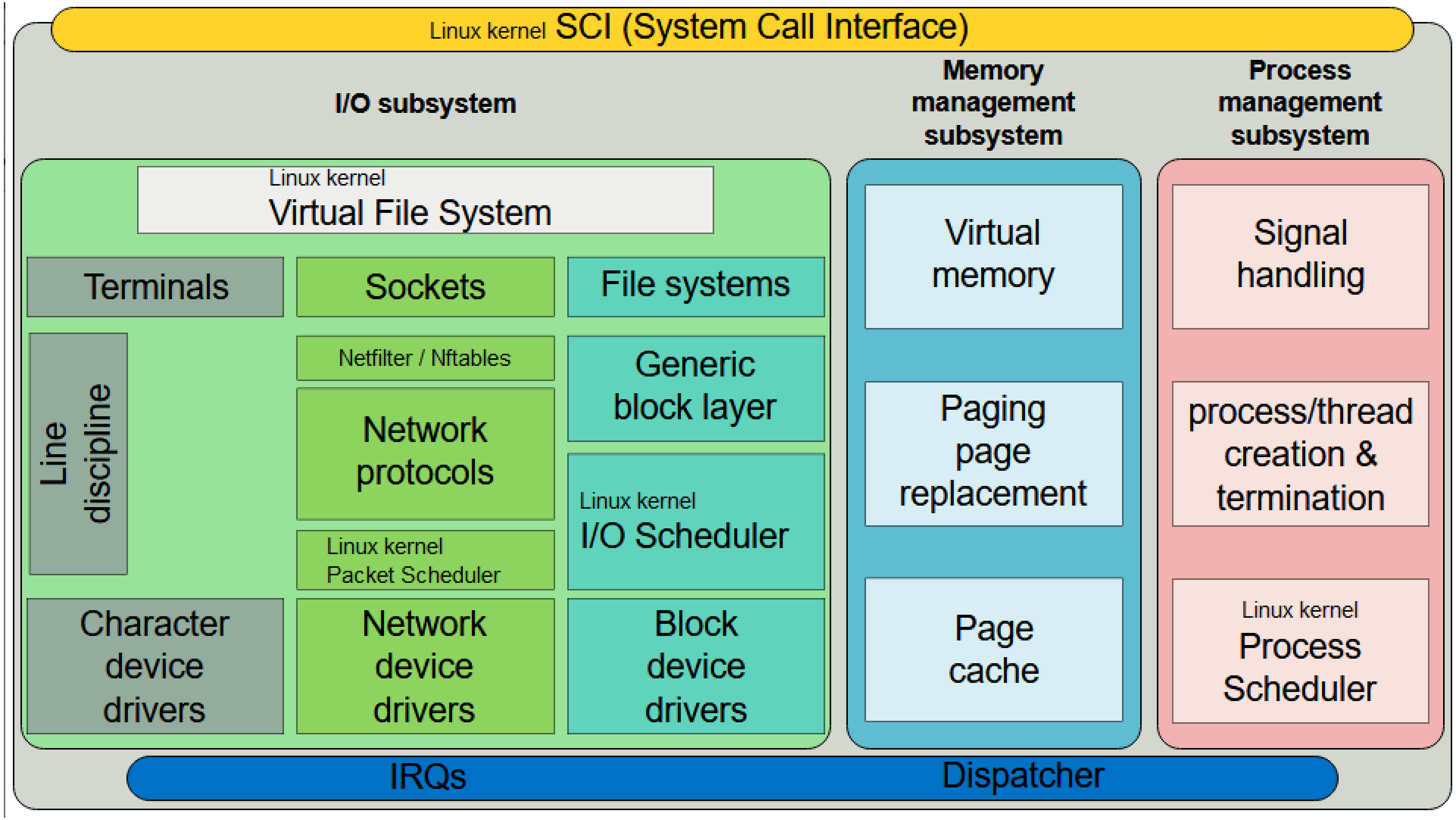


> “I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready.”
> — Linus Torvalds, August 1991

Linux structure and kernel functions
Basic concepts

- Linux structure and kernel functions
- Basic concepts – process, user mode and kernel mode, context switch, system calls, user stack and kernel stack, process state transitions

Linux – the structure and functions of the kernel



Source: Wikipedia

Process, address space, context

- **Process** is a program in execution; execution runs sequentially, according to the order of instructions in a **process address space**.
- **Process address space** is a collection of memory addresses, referenced by the process during execution.
- **Process context** is its operational environment. It includes contents of general and control registers of the processor, in particular:
  - program counter (PC),
  - stack pointer (SP),
  - processor status word (PSW),
  - memory management registers (allow access to code and data of a process).
- Linux is a **multiprogramming** system. The kernel dynamically allocates resources necessary for processes to operate and provides security.

  For this purpose, it needs **hardware support**:
  - processor executing in two modes: **user mode** and **system mode** (**kernel mode**),
  - privileged instructions and memory protection,
  - interrupts and exceptions.

Kernel address space

System address space or **kernel space** comprises code and kernel data structures. Access to them is only possible in system mode. The kernel has direct access to the address space of the current process. Occasionally, it can reach up to address space of the other process than the current one.

Kernel thread is executed in kernel mode.

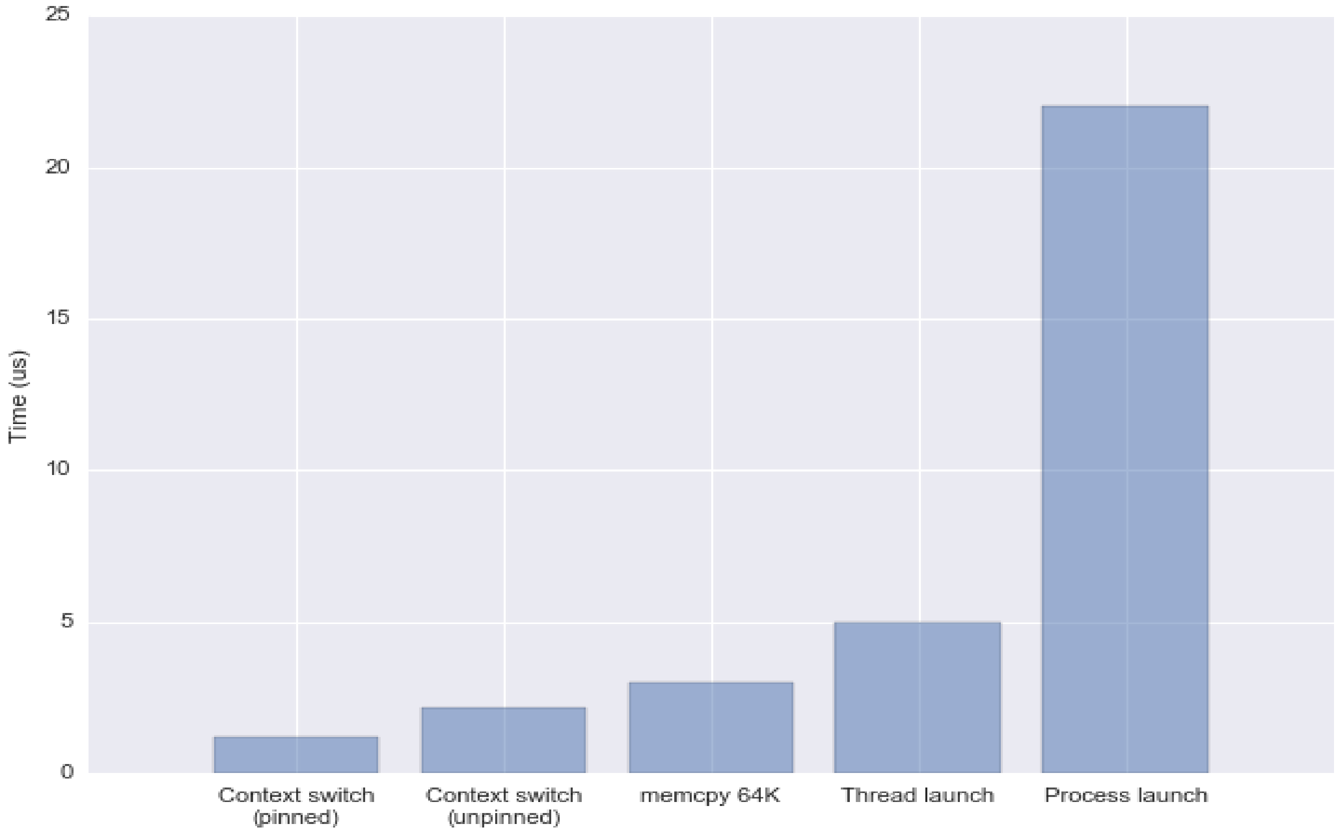
The transition to the execution of the kernel code can occur as a result of several events:

- The process calls the **system function** (*system call*). The user process instructs the kernel to perform certain actions (e.g. I/O operations) on its behalf.
- The processor reports **exception** while executing the process, e.g. a non-existent instruction. The kernel handles an exception on behalf of the process that caused it.
- An external device reports **an interrupt** to the CPU informing about the occurrence of an asynchronous event, e.g. completion of an input-output operation. Interrupt support is handled in the *interrupt handling routine*.

Context switching

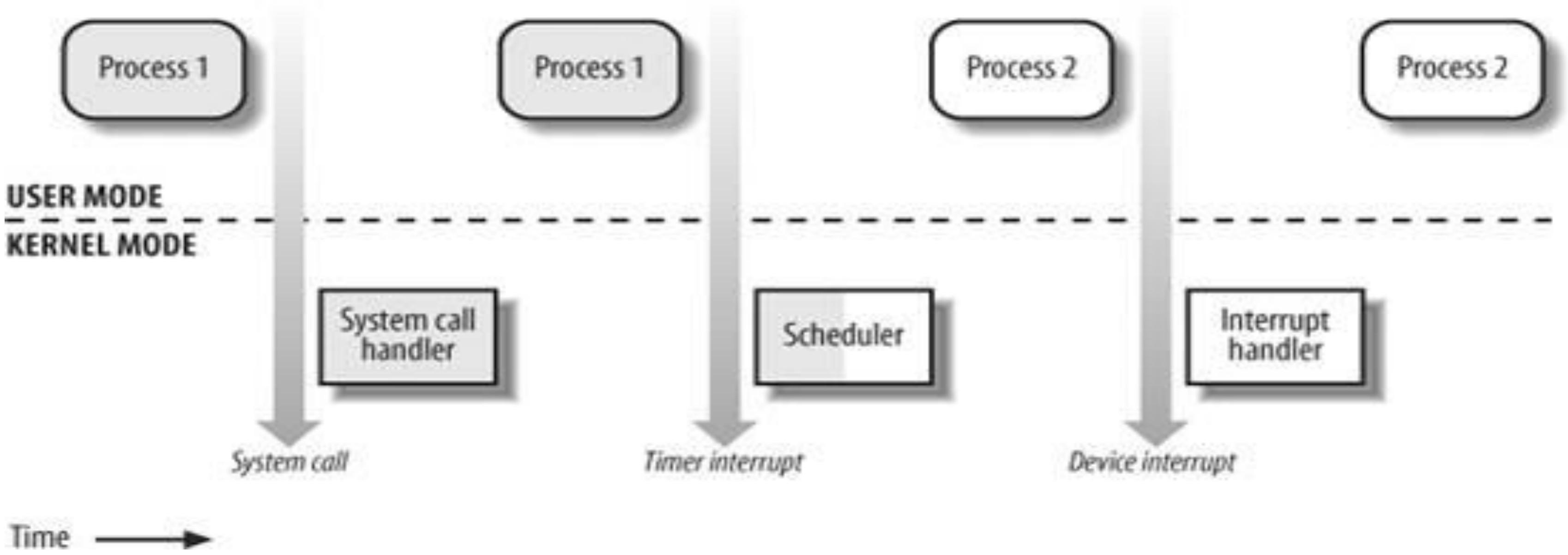
Context Switching – saving the context of the current process (in the structure that is part of the process address space) and loading the context of another process into the processor registers.

The context switch time is an overhead of the system and depends on hardware support (can take from a few **100 nanoseconds** to a **few microseconds**).

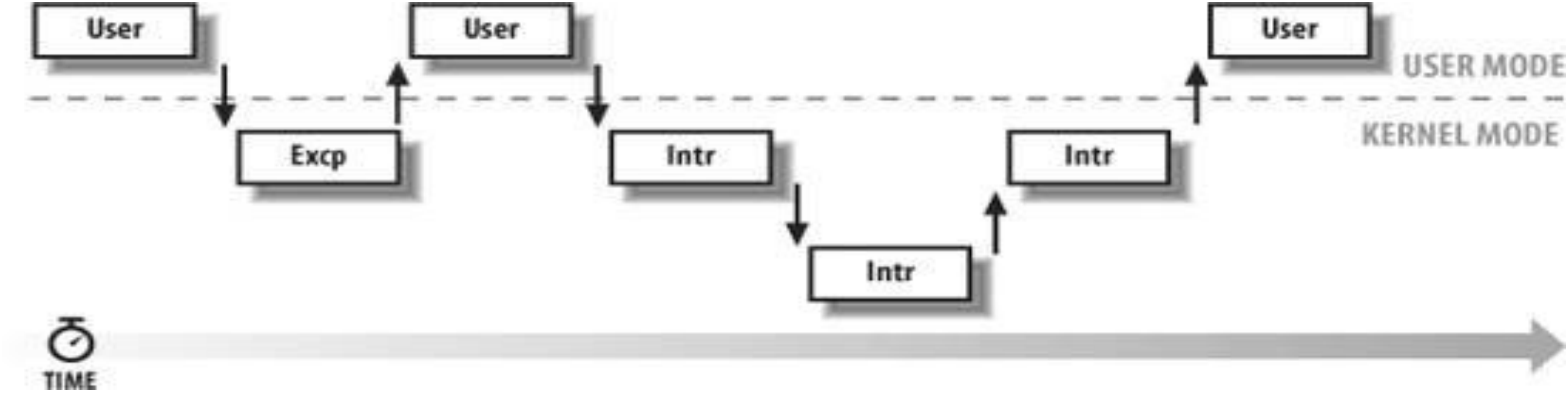


Measuring context switching and memory overheads

Context switching **itself has a cost in performance**, due to running the task scheduler, TLB flushes, and indirectly due to sharing the CPU cache between multiple tasks. L2 cache have substantial impact on the cost of context switch.

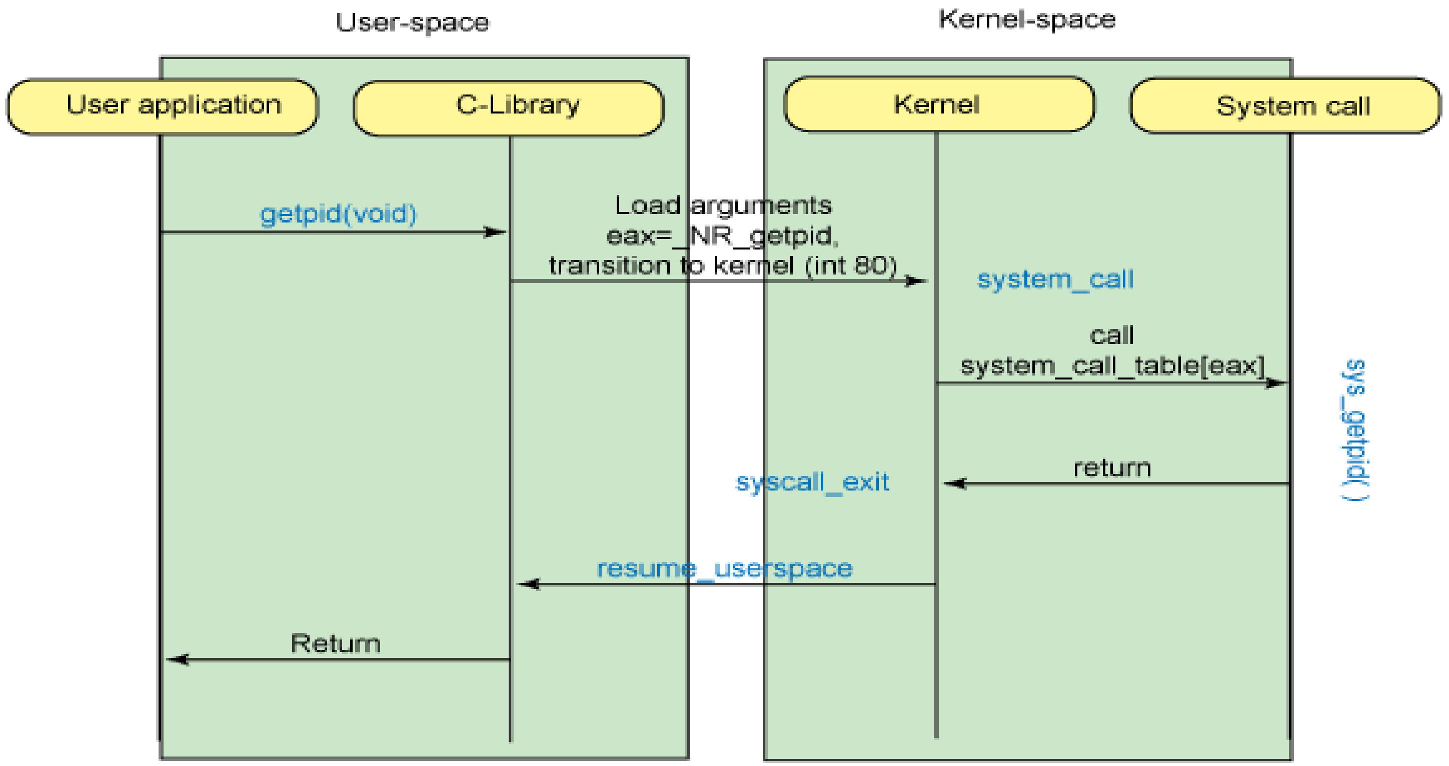


Transitions between user and kernel mode, source: Bovet, Cesati



Interleaving of kernel control paths, source: Bovet, Cesati

System function call with int 0x80



source:
Anatomy of the
Linux kernel,
M.Tim Jones

The details of the system function call depend on the architecture (the figure illustrates i386). The register **eax** is used to transmit the number of the function being called. The machine instruction **int 0x80** calls the program interrupt 0x80 (decimal 128) – context switching and calling the kernel function **system_call**. The function transfers control to the proper system function (uses **system_call_table** with eax treated as an index).

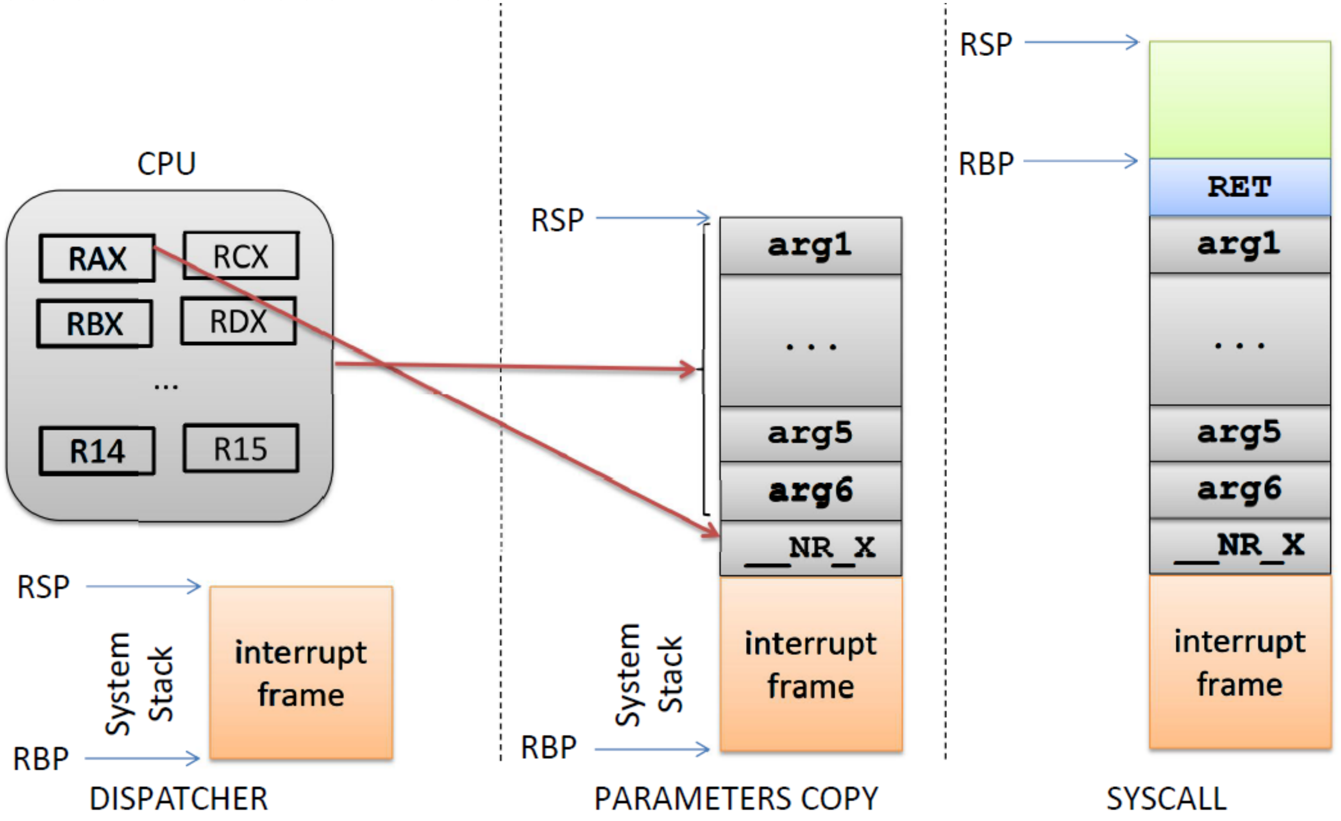
After returning from the system function, the **syscall_exit** function is executed, the **resume_userspace** function call returns the control back to the user space.

System call and process stacks

Each process uses **two stacks**:

- **user stack** – used in user mode (grows dynamically during program execution),
- **kernel stack** – in kernel mode (has a fixed, small size); is usually allocated in address space of the process, but it can not be accessed in the user mode.

system_call() starts by saving the registers in the kernel stack. After checking other things such as validating parameters, it will call the respective system call.



System call – sequence of steps

System calls: https://linux-kernel-labs.github.io/refs/heads/master/lectures/syscalls.html

This is what happens during a **system call**:

1. The application is setting up the **system call number** and **parameters** and it issues a trap instruction.
2. The execution mode switches from **user** to **kernel**; the CPU switches to a kernel stack; the user stack and the return address to user space is saved on the kernel stack.
3. The kernel entry point **saves registers** on the **kernel stack**.
4. The system call dispatcher **identifies the system call** function and **runs it**.
5. The user space **registers** are **restored** and **execution** is **switched back** to user (e.g. calling IRET).
6. The user space application **resumes**.

See also: Shadow stacks for user space**,** Jonathan Corbet February 21, 2022
Whenever a function is called, the return address is pushed onto both the **regular stack** and the **shadow stack**. When that function returns, the return addresses are popped off both stacks and compared; if they fail to match, the system goes into red alert and kills the process involved.

System call conventions

Definition of the system function from the C level (file include/linux/syscalls.h):

asmlinkage long sys_exit (int error_code);

asmlinkage tells compiler to look on the kernel stack for the function parameters, instead of registers.

In architecture x86 the registers **ebx**, **ecx**, **edx**, **esi** and **edi** are used to pass the first five parameters. If there are more parameters, it is through one register that a pointer to the user's address space is transferred, where all parameters are placed.

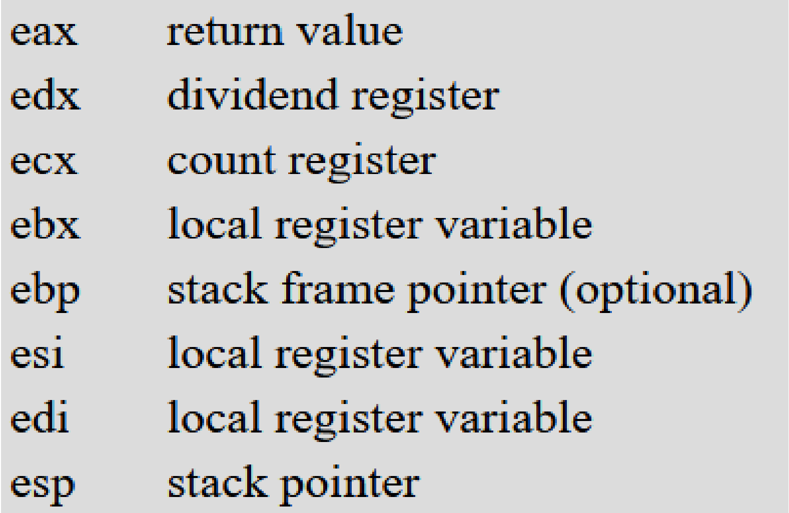
The value passed from the system function is placed in the **eax** register.

Other registers are used in 64-bit architecture:

- *x64 Architecture, registers, calling conventions, addressing modes*
- *syscall numbers*

Copying data between the kernel space and the user space is done using **copy_to_user()** and **copy_from_user()**.

When executing the system function, the kernel works **in the context of the process** (the variable **current** points to the current process).



eax	return value
edx	dividend register
ecx	count register
ebx	local register variable
ebp	stack frame pointer (optional)
esi	local register variable
edi	local register variable
esp	stack pointer

Sysenter and sysexit

Machine instructions **sysenter** and **sysexit** were added to x86 processors (newer than Pentium II). They allow a **faster transition** (return) to the kernel mode to perform a system function than using the **int** statement. Support for this mechanism has been added to the Linux kernel (Sysenter Based System Call Mechanism in Linux 2.6).

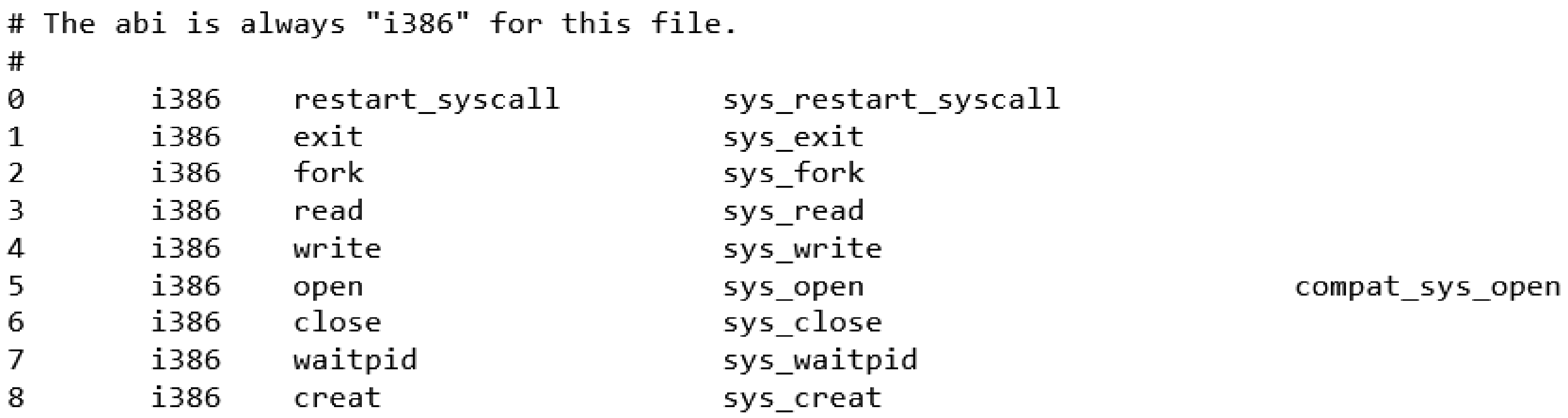
Calling the x86 function

- 64-bit version – defined in the file arch/x86/entry/entry_64.S
- 32-bit version – defined in the file arch/x86/entry/entry_32.S

Content of the system function table

- 64-bit version – defined in the file arch/x86/entry/syscalls/syscall_64.tbl
- 32-bit version – defined in the file arch/x86/entry/syscalls/syscall_32.tbl

This is the beginning



```
# The abi is always "i386" for this file.
#
0       i386    restart_syscall         sys_restart_syscall
1       i386    exit                    sys_exit
2       i386    fork                    sys_fork
3       i386    read                    sys_read
4       i386    write                   sys_write
5       i386    open                    sys_open                        compat_sys_open
6       i386    close                   sys_close
7       i386    waitpid                 sys_waitpid
8       i386    creat                   sys_creat
```

In other operating systems, there are many more functions than **435** in Linux 5.6 (32-bit).

Process and system context

Context of execution – summary:

- **user code** is executed in **user mode** and in **process context**, can only reach the address space of the process,

- **system functions and exceptions** (e.g. dividing by zero or violation of memory protection) are supported in **system mode**, but **in context of the process**, they have access to the **process and system address space**.

  The kernel acts on behalf of the current process (e.g. by executing a system function), it can reference the address space of the process and the process stack. It can also block the current process if it has to wait for resources.

- **interrupts** are handled in **system mode** in the **context of the system** with access only to the **system address space**.

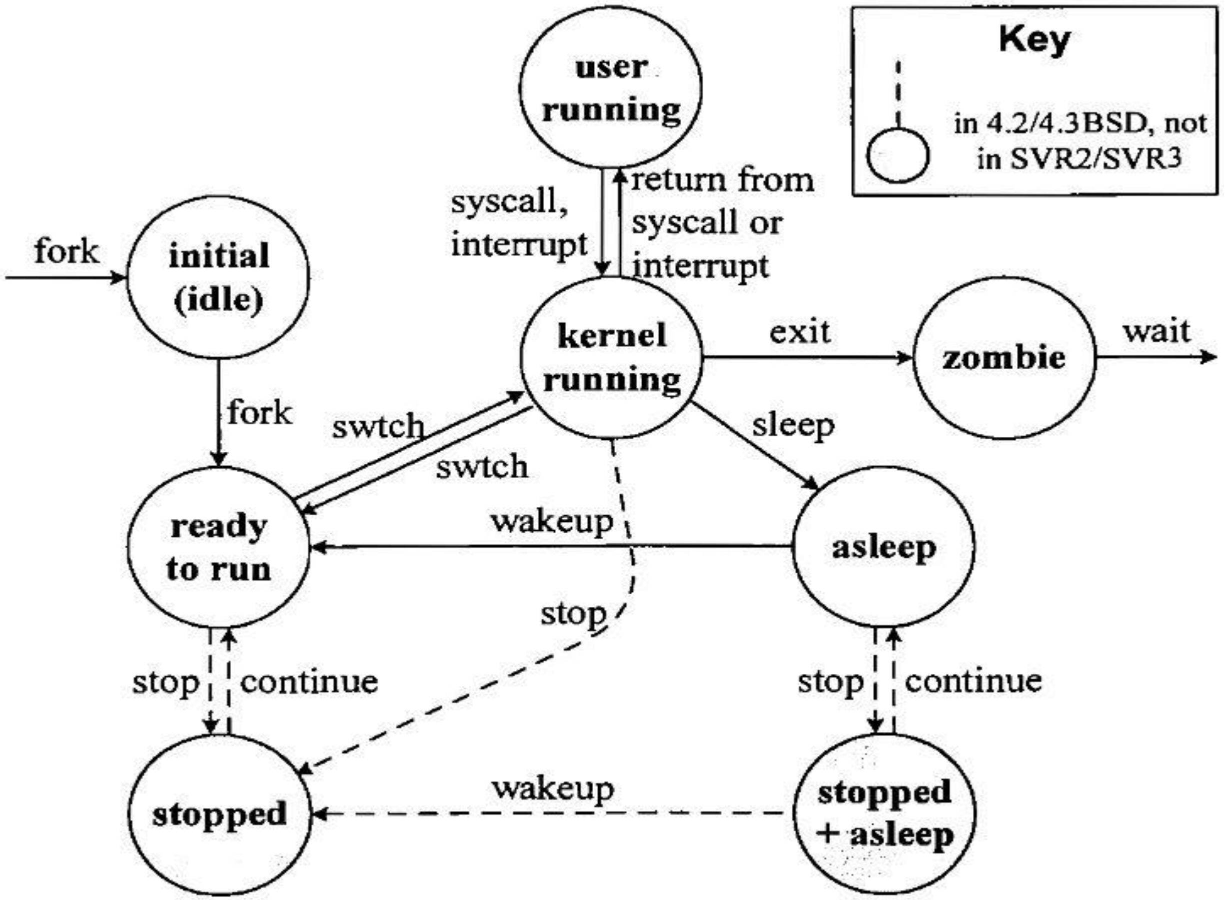
  System-wide operations, such as recalculating priorities or handling an external interrupt. Not performed on behalf of any particular process and therefore take place in the context of the system. The kernel does not reach to the address space or the stack of the current process, also it can not block.

Process state transitions

The Linux kernel is **preemptable** and **re-entrant**, it can support different processes concurrently.

The process during execution changes **state**. The basic states of the process are:

- **new**: the process has been created,
- **ready**: the process is waiting for the processor to be allocated,
- **executed** (more precisely: **executed in user mode** or **executed in system mode**): process instructions are executed,
- **waiting**: the process is waiting for an event to occur,
- **finished**: the process completed execution.



Process states and state transitions, source: U. Vahalia, UNIX Internals: The New Frontiers