ANSY - Getting in Linux Kernel details

version 2022-12-20

CENTRE DES RESSOURCES INFORMATIQUES

-- Cyril zarak Duval, root CRI/ACU 2020

Introduction

Generic information about the course

Why should you listen to the course ?

- → Linux is a state of the art in the industry
- → Understanding in details will help you as low-level engineers
- → Provide some tools useful to work with Linux
- → Help you acquire an analytic mind to tackle low-level issues
- → Subject somewhat difficult
- → Getting a decent grade

Notions

- ptrace
- strace & rr
- ebpf
- auditd
- kprobes/uprobes/tracepoint
- perf
- pseudo filesystems
- VFS
- iptables/nftables
- initramfs
- PXE
- dracut
- BCC
- CPU & scheduler metrics
- Memory management & metrics
- Systemd
- ...

Understand observability

Observability applied to Linux

Observability

- ➔ Observability is a high-level notion
- ➔ Observability means understanding what is going on in a system
- ➔ Observability in its modern approach has 3 pillars:
  - ◆ Metrics
  - ◆ Logs
  - ◆ Tracing
- ➔ Linux is the core of all our foundations
- ➔ We need to have observability in it
  - ◆ To better understand, to administrate it
  - ◆ To debug

Observability in Linux

- Linux thankfully offers interfaces for observability
- What are the first things that come to your mind when you think about observability and Linux ?
  - What are the things you want to observe ?
  - What are the interface(s) you will use ?

Get information about CPU usage

Let’s start with something “simple”

CPU and linux

- What does CPU usage means ?
  - 0, 50, 100% ?
  - 800% ?
- It a percentage of time spent working on stuff, otherwise idling
- How do we get this number ?
  - top, htop
  - mpstat

CPU and Linux

- ➔ How does *they* get the information ?
- ➔ Let's pause this question and investigation, and focus on the methodology here

“I have a tool behaving in a way that is unknown to me.

How do I figure out how it works?

How does it works ?

- **mpstat** returns CPU usage, along with some useful information
- Does it create this information ?
- Does it collect this information from somewhere ?
    - Is it on the network ?
    - Is it on the machine ?
        - Our filesystem ?
        - Any other mean ?

How do we get this info ?

- 2 hypothesis:
  - The CPU usage information is returned by the hardware directly
  - The CPU usage is computed by the kernel and reported
- How can we figure this out ?
  - Knowing that mpstat *knows* the answer
- The most straightforward solution would be to read mpstat source code
  - But before actually doing this, let’s play a small game

How do we contact the kernel ?

What are the interfaces offered ?

Kernel interfaces

- The “only” interface is a syscall
  - All other high-level interfaces are syscall-based
- A syscall can gives us information directly:
  - gethostname(2)
  - gettimeofday(2)
  - getcpu(2)
  - getcwd(2)
  - ...

Kernel interfaces

- ➔ Some syscalls are used to reach higher level interfaces
  - ◆ open(2), openat(2), read(2), write(2), close(2)
- ➔ What are higher level kernel interfaces ?
  - ◆ /proc/....
  - ◆ /sys/....
  - ◆ /dev/...
  - ◆ /sys/kernel/debug/...
  - ◆ /sys/kernel/security/...
  - ◆ /sys/firmware/efi/efivars/...
  - ◆ /sys/fs/cgroup/...

Kernel interfaces

- Are other syscalls used for higher level interfaces ?
- Yes:
  - socket(2)
  - ioctl(2)
  - bpf(2)
  - perf_event_open(2)
  - ptrace(2)

Kernel interfaces

- Let’s get back the special directories mentioned before (/sys, /proc, …)
- How are they special ?
- They aren’t “real files” on your SSD
  - In fact you can open your SSD on another machine and check that by yourself in a very easy and naive way
- The files there are kernel interfaces in the forms of a file
  - “In UNIX, everything is a file”
- Those are pseudo-filesystems
- But more about that later …

Back to our CPU usage analysis

So, what about mpstat ?

mpstat

- ➔ mpstats like almost everything on a classic linux distro is open-source
- ➔ Checking source code is therefore a good reflex for things like this
- ➔ Let's read [mpstats source code]()
- ➔ Code is well written
  - Follow many standards
  - Proper naming convention
  - Comments
  - ....

mpstat

- ➔ Code is quite short but finding the information still took time
- ➔ Can we make this more efficient ?
- ➔ What could be another approach than reading source code ?

- ➔ What do we know or deduced ?
  - ◆ The information is probably held by the kernel
  - ◆ mpstat gets this information
  - ◆ Communication between userland and kernel land is done via syscalls
  - ◆ Could we just look at the syscalls mpstats did ?

Let's ptrace mpstat

- ➔ Linux offers a syscall and its interface to debug softwares
- ➔ ptrace(2)
  - ◆ But more about it later ...
- ➔ Used to debug, like GDB, to see what is going on, inspect code, variable values, etc
- ➔ What if we have a special debugger ?
  - ◆ This debugger will just run the program
  - ◆ But whenever a function is called, it checks if it a syscall function ?
    - ● But wait, are syscall functions ?

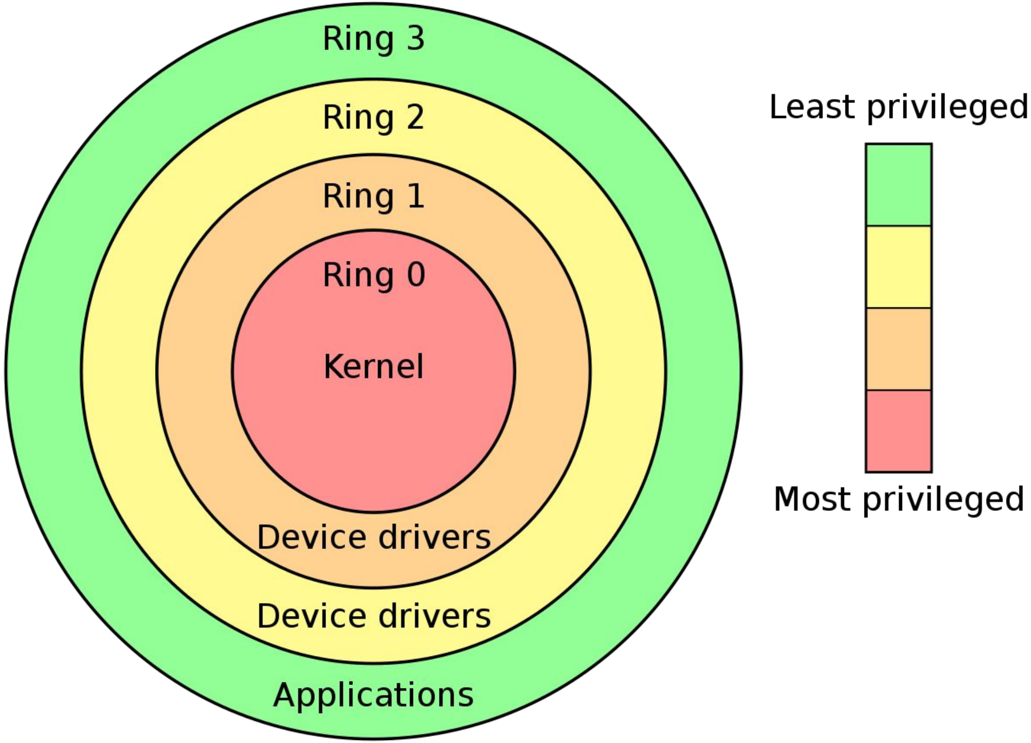
Some syscall digression

How is implemented a syscall in the end ?

What are syscalls ?

- We know that a syscall is a kernel function that is called from userland
  - Sort of
- But are we allowed to call directly a function like this ?
- In x86 (IA_32 and x86-64) we run code on the CPU in rings
  - ring 0 is the most privileged one
    - Allowed to access hardware and configure the CPU directly
  - ring 3 is the one userland runs in. Can do computation, but cannot run some privileged CPU instructions

CPU rings


Ring 3
Ring 2
Ring 1
Ring 0
Kernel
Device drivers
Device drivers
Applications
Least privileged
Most privileged

What are syscalls ?

- ➔ The kernel runs in ring 0
- ➔ The kernel can therefore do things regular process can't
- ➔ Regular process still need to access some protected devices or perform some privileged operations
  - ◆ In a controlled environment (permissions, ...)
- ➔ They contact the kernel for those operations via syscalls
- ➔ The kernel checks permissions, do sanity checks, etc and performs the operation
- ➔ The result, if any, is returned to the user

What are syscalls ?

- Syscalls are the interface between userland and the kernel
- Allows privileged operation, control kernel behavior or use kernel features
- Allow abstraction (disk drivers, network drivers, ...)
- How can we call a ring 0 function from ring 3 ?
  - Do we have symbols exported ?
- In x86 we have 3 ways:
  - INT 0x80 (legacy)
  - sysenter (IA_32)
  - syscall (AMD64)

Syscall implementation in x86

- ➔ INT 0x80 is hardly used anymore, it is a legacy way of making a syscall
  - Creates an interrupt to notify the kernel
    - But more about that later ...
- ➔ sysenter is also called fast system call, created by intel for IA_32
- ➔ syscall is the AMD64 version, mostly used now

Syscall implementation in x86

- ➔ When doing a syscall instruction, what happens exactly ?
- ➔ On syscall, the CPU looks in a specific MSR: IA32_LSTAR
- ➔ IA32_LSTAR MSR contains a ring 0 function address to execute
- ➔ In Linux, it's entry_SYSCALL_64
  - ◆ https://elixir.bootlin.com/linux/latest/source/arch/x86/entry/entry_64.S#L49
- ➔ Linux determines which syscall has been called in this function based on %rax
- ➔ The arguments to the syscall are in %rdi, %rsi, … as usual

Syscall implementation in x86

- In entry_SYSCALL_64 we prepare everything to call the actual kernel function
  - We save userland general purpose registers on the stack and do a few things
- We call the kernel function associated with the syscall requested, forwarding the arguments userland gave
- Once the kernel function returns, we put the return value on the stack
- We do a few things back, put the value from the stack back in registers and return

Syscall consequences

- ➔ Calling a syscall means calling a kernel function through some steps
- ➔ The steps are to ensure security when switching from ring 3 code to ring 0 (and vice-versa)
- ➔ These steps are called privilege switch
- ➔ Privilege switch is quite expensive:
  - More instructions to execute
  - No CPU pipelining/branch prediction/...
  - Data shouldn't be accessed directly and shall always go through copy_from_user/copy_to_user
  - Pointers must be handled carefully

About vDSO

Some more information about syscall implementation

Virtual Dynamic Shared Object

- ➔ Some syscalls are used a lot
- ➔ They don't have any security and will behave the same for every user (privileged or not)
- ➔ To allow better performances, the kernel exposes some syscall directly in userland
- ➔ Userland implementation is done in vDSO
- ➔ Shared ELF object to every userland process
  - Address fetcheable via auxiliary values
  - Glibc gets it for you

Virtual Dynamic Shared Object

- Contains some syscall, depending on the architecture
- gettimeofday(2), getcpu(2), ...
- No privilege switch = faster
  - No strace, no seccomp however

Virtual Dynamic Shared Object

```
$ file vdso64.so
vdso64.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, BuildID[sha1]=1e5d0c12baeaaebcc0ca26b4c4c4dd0be8f0e5d4, stripped
$ objdump -T vdso64.so

vdso64.so:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000940  w   DF .text  0000000000000264  LINUX_2.6   clock_gettime
0000000000000760 g    DF .text  00000000000001b0  LINUX_2.6   __vdso_gettimeofday
0000000000000bb0  w   DF .text  0000000000000063  LINUX_2.6   clock_getres
0000000000000bb0 g    DF .text  0000000000000063  LINUX_2.6   __vdso_clock_getres
0000000000000760  w   DF .text  00000000000001b0  LINUX_2.6   gettimeofday
0000000000000910 g    DF .text  000000000000002e  LINUX_2.6   __vdso_time
0000000000000c50 g    DF .text  00000000000000a1  LINUX_2.6   __vdso_sgx_enter_enclave
0000000000000910  w   DF .text  000000000000002e  LINUX_2.6   time
0000000000000940 g    DF .text  0000000000000264  LINUX_2.6   __vdso_clock_gettime
0000000000000000 g    DO *ABS*  0000000000000000  LINUX_2.6   LINUX_2.6
0000000000000c20 g    DF .text  000000000000002a  LINUX_2.6   __vdso_getcpu
0000000000000c20  w   DF .text  000000000000002a  LINUX_2.6   getcpu
```

Let’s get back to mpstats system calls

Now we do know what a system call is

Let’s ptrace mpstat

- ➔ Linux offers a syscall and its interface to debug softwares
- ➔ ptrace(2)
- ➔ Used to debug, like GDB, to see what is going on, inspect code, variable values, etc
- ➔ What if we have a special debugger ?
  - ◆ This debugger will just run the program
  - ◆ But whenever a function is called, it checks if it a syscall function
  - ◆ If so, prints arguments, resume execution and print return value

Discovering strace

- A famous debugging tool for such purposes exists
- strace(1)
- In the simplest usage:
  - Starts a process with given arguments
  - Gets notified of all the syscalls the tracee performs
  - Prints the syscall, its arguments and return value

Usefulness of strace

- When is strace useful ?
  - Find out why a software fails
  - Find out how it behaves if no documentation
    - Ex: location of config files read by the app
    - Ex: Interaction with other processes
    - Ex: Memory impact and behaviour
  - See where a software hangs (if on a syscall)
  - ....
- strace is a very popular and versatile debug tool
  - Simpler and quicker to use than GDB
    - Not suited for all workflows though

How to strace ?

- How to use strace(1) efficiently ?
- A few tips:
  - Use -f to follow and strace forks too
  - Use -z or -Z to see only successful or failed syscalls
  - -c will give you a summary/overview of the syscalls used. Can be useful at first to have a sneak peak
  - Discover the -e option

“Let’s discover strace -e and play with strace a bit

Strace can be difficult

- Since strace will show every syscalls, it might be difficult to find what you’re looking for
- Especially if the software is huge
  - Or we have limited knowledge on what to look for
- Example: pylint
  - Where’s the configuration file ?
  - Tracing open(2)/openat(2) ?

Strac

```
$ strace -c pylint ipxe_manager 2>&1

...

% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 45.49    0.006295           0     19086      9931 newfstatat
 12.04    0.001666           0      1677           read
 11.13    0.001540          12       128           munmap
  9.31    0.001288           1       991        10 openat
  4.94    0.000683           1       433           brk
  3.91    0.000541           0       985           close
  3.66    0.000506           0      1534         3 lseek
  2.51    0.000347           1       273           mmap
  2.29    0.000317           6        48           write
  2.19    0.000303           0       899       891 ioctl
  1.65    0.000228           2       106           getdents64
  0.30    0.000041           1        32           mprotect
  0.26    0.000036           0        66           rt_sigaction
  0.12    0.000017           1        16         9 readlink
  0.09    0.000012           0        53           getcwd
  0.05    0.000007           1         5           getrandom
  0.03    0.000004           4         1           sysinfo
  0.02    0.000003           3         1           futex
  0.02    0.000003           1         2           prlimit64
  0.01    0.000002           0         3           dup
  0.00    0.000000           0         2           pread64
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           fcntl
  0.00    0.000000           0         1         1 mkdir
  0.00    0.000000           0         2         1 arch_prctl
  0.00    0.000000           0         1           gettid
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           epoll_create1
  0.00    0.000000           0         1           rseq
------ ----------- ----------- --------- --------- ----------------
100.00    0.013839           0     26352     10847 total
```

Strace can be difficult

- ➔ Example: pylint
  - ◆ Where’s the configuration file ?
  - ◆ Tracing open(2)/openat(2) ?
    - 991 openat(2) in this example
  - ◆ Maybe grepping “cfg”, “yml” or “json” ?
  - ◆ Actually file name is pylintrc
  - ◆ And not even open/openat(2) if doesn’t exist !

Strace can be difficult

```
$ strace pylint ipxe_manager 2>&1 | grep pylintrc
                                                              30 ↵
newfstatat(AT_FDCWD, "pylintrc", 0x7ffcd8fc3b10, 0) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, ".pylintrc", 0x7ffcd8fc3b10, 0) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/home/zarak/.pylintrc", 0x7ffcd8fc3b10, 0) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/home/zarak/.config/pylintrc", 0x7ffcd8fc3b10, 0) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/etc/pylintrc", 0x7ffcd8fc3cd0, 0) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "pylintrc", 0x7ffcd8fc6c70, 0) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, ".pylintrc", 0x7ffcd8fc6c70, 0) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/home/zarak/.pylintrc", 0x7ffcd8fc6c70, 0) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/home/zarak/.config/pylintrc", 0x7ffcd8fc6c70, 0) = -1 ENOENT (No such file or directory)
newfstatat(AT_FDCWD, "/etc/pylintrc", 0x7ffcd8fc6e30, 0) = -1 ENOENT (No such file or directory)
```

Strace can be difficult

- Useful sometimes to simulate a failure or to simulate a success of a syscall or a set of syscalls
- Need to have some knowledge of the software
  - Or some intuition

CPU metrics

What is a CPU with linux ?

Understanding CPU metrics

- A CPU core or thread shall already be known to you
- We've seen already 2 kinds of things a CPU can execute in this course:
  - User code
  - Kernel code
- What are the other things a CPU can do ?
- Fortunately a CPU isn't always doing something: it can idle
- Let's check the metrics exported by the kernel in /proc/stat

Understanding CPU metrics

- User
- Nice
- System
- Idle
- Iowait
- Irq
- Softirq
- Steal
- Guest
- Guest_nice

Understanding CPU metrics

- User -> userland code
- Nice
- System -> kernel-land code
- Idle -> CPU literally doing nothing (~no power usage, C-state)
- Iowait
- Irq
- Softirq
- Steal
- Guest
- Guest_nice

Understanding CPU metrics

- User -> userland code
- Nice
- System -> kernel-land code
- Idle -> CPU literally doing nothing (~no power usage, C-state)
- Iowait
- Irq
- Softirq
- Steal
- Guest -> kernel KVM gave CPU time to VM
- Guest_nice -> kernel KVM gave nice CPU time to VM

Process niceness and scheduler

Why isn’t pulseaudio nice ?

CPU and multithreading

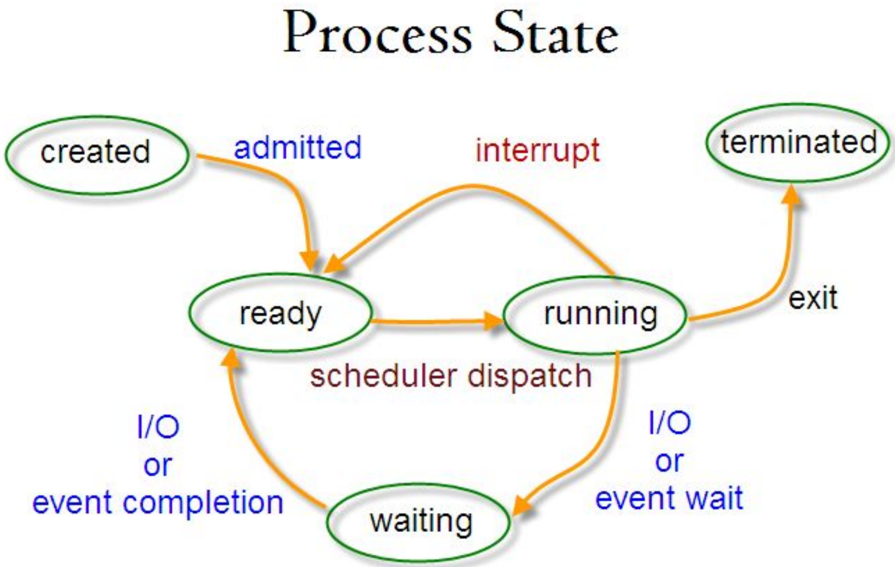
- ➔ A classic PC/server runs dozens if not hundreds of processes in “parallel”
- ➔ A modern CPU has multiple cores, and multiples threads or logical cores/hyper-thread
- ➔ Let’s say our CPU has 16 logical cores
- ➔ I can truly execute 16 processes in parallel
- ➔ How can I give the impression it’s running 150 ?

CPU and multithreading

- Most processes don’t need the CPU 100% of the time
- They need some time to work, and have to wait
  - Timer, user input, IO, being activated back, …
- If most of them don’t need to actually run in parallel, we can split execution in small timeshares, and simulate parallel execution
- This is the role of the scheduler to provide such timeshares and execute processes

CPU and multithreading

➔ Here is the classical representation of a Process state in Linux

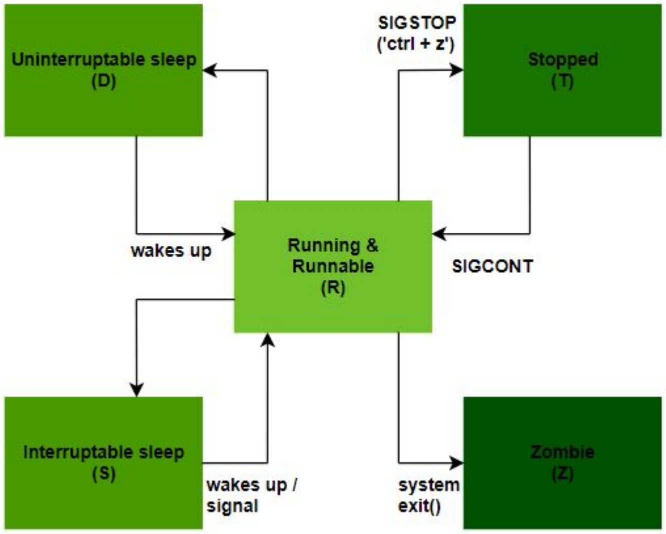


Process states

Giving meaning to R/S/D/Z/T

CPU and multithreading

➔ Actually it looks more like this



Process state

- R state means running or runnable
  - Either currently being executed on a CPU core (running)
  - Or waiting for a core to be free and for the scheduler to start it (runnable)
- S state is the state some process will spend the most time in
  - Waiting for an event, for I/O, for a timer, ...
- T state is fairly easy to grasp, one stopped the process by sending a SIGSTOP signal

Process state

- D state is a bit more shady
  - Some linux syscall are not interruptible. It means that a process waiting for the syscall to complete cannot be killed.
  - No signal can be transmitted, even SIGKILL
  - Examples include some I/O syscalls, KVM related calls, etc
    - https://elixir.bootlin.com/linux/latest/A/ident/TASK_UNINTERRUPTIBLE
  - Famous example often found is a NFS-related process stuck in D-state when NFS server is unreachable

Process state

- Z state is for a zombie
  - Zombie process is a process that has finished its execution but hasn’t been wait(2)-ed by its parent
  - Its information remains and must be collected for the process to be removed from the process list
  - Init process must wait for zombie process re-attached to it to maintain a clean system
  - When a zombie is create, SIGCHLD is sent to parent process

What does it have to do with niceness ?

Exploring CFS

CFS - Completely Fair Scheduler

- ➔ The role of the process scheduler is to run process when it makes sense
  - When they are ready to run
  - When they can (i.e. a CPU core is available)
- ➔ Linux default scheduler is called CFS
- ➔ It divides time in timeslices
- ➔ It gives a timeslice to the process that is ready to be run and has been starving CPU time the most first

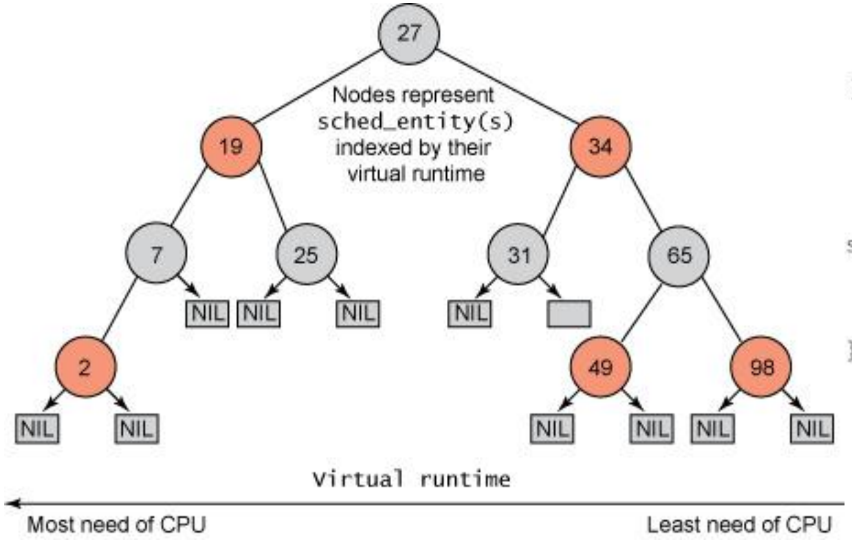
CFS - Completely Fair Scheduler

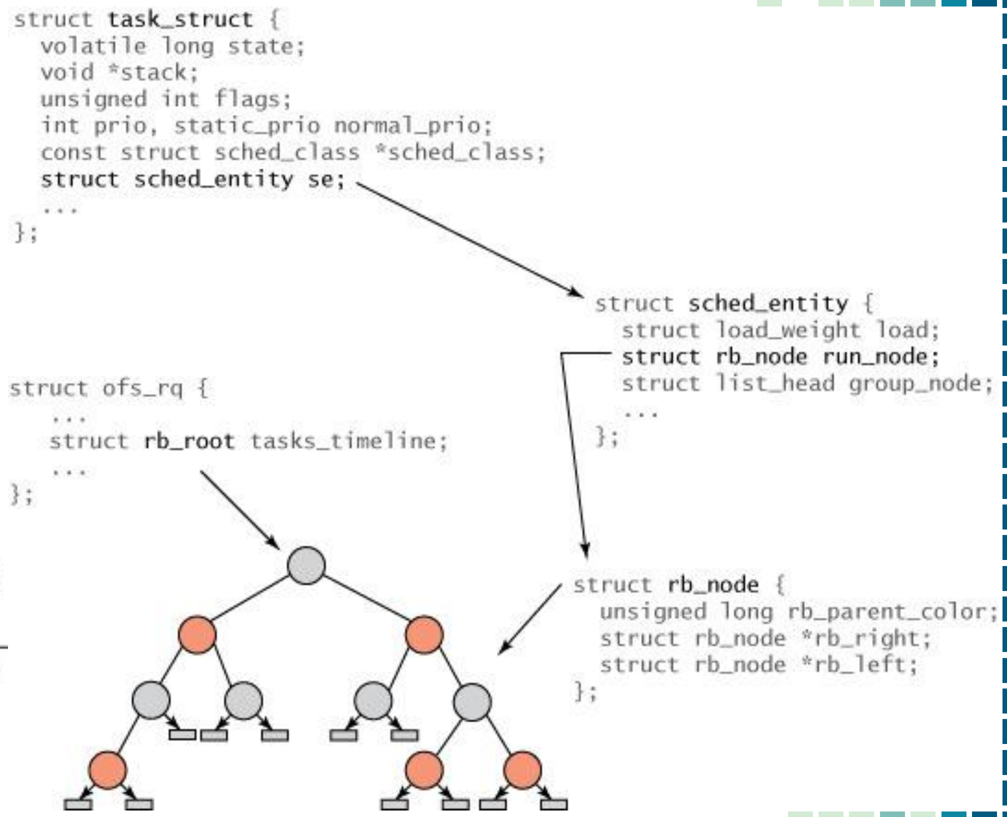
- ➔ If the system is not overloaded, CFS doesn't have to make important decisions
  - Most process are in D/S state, and therefore very few in R state. Decisions are easy
- ➔ But if the system starts to be overloaded, CFS comes to play
- ➔ CFS selects the process that is missing the most vruntime, i.e. the process that should have been running but hasn't
  - Takes decision based on total execution time and how long it has been waiting

CFS - Completely Fair Scheduler

- ➔ CFS tracks process via a red/black tree
  - ◆ On the left of the tree, process with the smallest vruntime
- ➔ It is also able to dynamically change the length of the CPU timeslice based on the load:
  - ◆ If a process is alone, it makes sense to give it a lengthy timeslice since it won't impact anyone
  - ◆ If 2 process requires each 50% of a single CPU core, to make them look like they run in parallel we need to alternate their execution
    - But mind context_switch ! Intervenes sched_min_granularity_ns

CFS


27
Nodes represent
sched_entity(s)
indexed by their
virtual runtime
19
34
7
25
31
65
2
49
98
NIL
Virtual runtime
Most need of CPU
Least need of CPU


struct task_struct {
volatile long state;
void *stack;
unsigned int flags;
int prio, static_prio normal_prio;
const struct sched_class *sched_class;
struct sched_entity se;
...
};
struct sched_entity {
struct load_weight load;
struct rb_node run_node;
struct list_head group_node;
...
};
struct ofs_rq {
...
struct rb_root tasks_timeline;
...
};
struct rb_node {
unsigned long rb_parent_color;
struct rb_node *rb_right;
struct rb_node *rb_left;
};

CFS - Completely Fair Scheduler

- But CFS is more complex than that
- Some process needs higher priority in their scheduling, because scheduling latency impacts
  - I.e. audio
    - audio doesn't need a lot of CPU time
    - But audio suffers heavily from latency
- This is the niceness of a process with linux
- The nicer the process, the less priority it gets
- Very nice process can still take 100% of a CPU core. They will just be descheduled if anyone else is asking for some CPU time

CFS - Completely Fair Scheduler

- CFS also takes into account various other configuration
- Internally, it also has a concept of priority
- Priority is changed by niceness, but to a range only
- To access the other priority values, a process must change its scheduling class
- More info on sched(7)

CPU load

A metric often misunderstood

CPU load

- ➔ CPU load can be understood as “how many operations my CPU is currently doing”
- ➔ This is a wrong understanding when it comes about the load metric reported by linux
- ➔ A better understanding would be “How much pressure is being applied to the CPU in average for a period of time”
- ➔ What does it measure exactly ?

CPU load

- Linux load represents the number of processes running, or waiting to be ran on the system, in average for a period of time
  - It also includes processes in uninterruptible sleep
    - I/O matters
  - It is not limited to a core -> all load values don't have the same meaning on each machine
  - Usually troubles begins when the load reaches the number of CPU cores

CPU load

- ➔ 3 values exported in /proc/loadavg
  - 1 min, 5min and 15min load
  - Number of processes in R state / schedulable entities
  - PID of the latest created process

```
$ cat /proc/loadavg
1.34 1.68 1.63 3/3253 2817976
```

CPU load

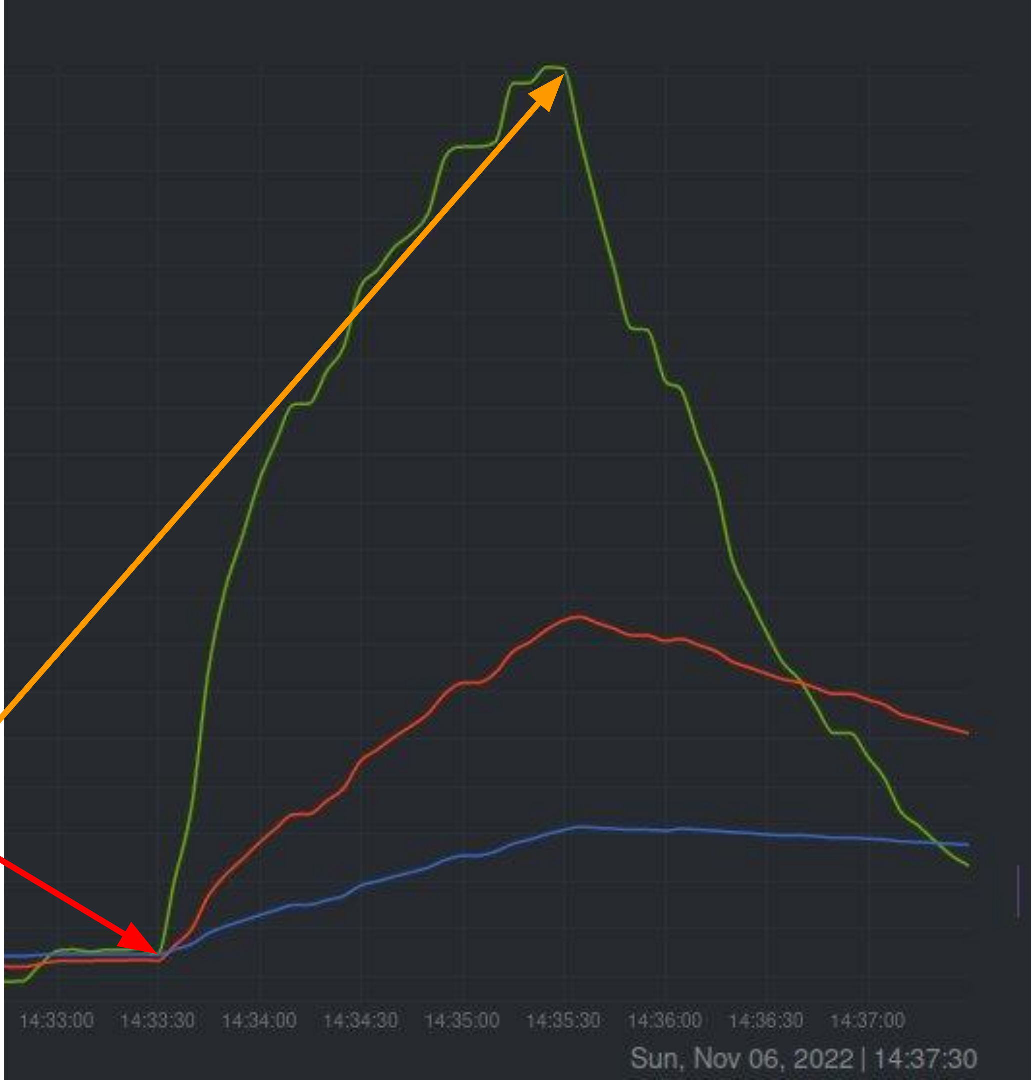
- ➔ Having 3 loads metrics, and them being averages has impact
- ➔ There is delay between event and possible visualization on the curves
- ➔ load1 is closer to “instant” load while load15 is really difficult to pull in any direction

CPU load

Load over time

Start of event

End of event



Let's get back to our CPU metrics

We do know now who's nice and who isn't

Understanding CPU metrics

- User -> userland code
- Nice -> process with high niceness
- System -> kernel-land code
- Idle -> CPU literally doing nothing (~no power usage, C-state)
- Iowait
- Irq
- Softirq
- Steal
- Guest -> kernel KVM gave CPU time to VM
- Guest_nice -> kernel KVM gave nice CPU time to VM

Understanding CPU metrics

- User -> userland code
- Nice -> process with high niceness
- System -> kernel-land code
- Idle -> CPU literally doing nothing (~no power usage, C-state)
- Iowait
- Irq
- Softirq
- Steal -> As a VM, hypervisor didn't schedule us
- Guest -> kernel KVM gave CPU time to VM
- Guest_nice -> kernel KVM gave nice CPU time to VM

Understanding CPU metrics

- ➔ User -> userland code
- ➔ Nice -> process with high niceness
- ➔ System -> kernel-land code
- ➔ Idle -> CPU literally doing nothing (~no power usage, C-state)
- ➔ Iowait -> time spent for a process waiting for I/O (unreliable)
- ➔ Irq
- ➔ Softirq
- ➔ Steal -> As a VM, hypervisor didn’t schedule us
- ➔ Guest -> kernel KVM gave CPU time to VM
- ➔ Guest_nice -> kernel KVM gave nice CPU time to VM

Quick tour of IRQ and softIRQ

Bringing some memory back

CPU interrupts

- The way for the hardware to notify the CPU something is happening is through IRQ
- For example, the user moved its mouse or typed on its keyboard
- Paquets reached the machine and are waiting on the network card
- Without getting into too many details, the CPU gets notified of these events through the PIC (Programmable Interrupt Controller)

CPU interrupts (x86)

- An interrupt stops the current CPU execution and executes an interrupt handler read on the IDT
  - The IDT (Interrupt Descriptor Table) maps interrupts to handlers
- An interrupt can be triggered by external device (like the network card) or by the CPU itself
  - In this case it's called a software interruption
    - Or an Exception (x86)
  - Examples include a division by 0, or an INT instruction

CPU interrupts (x86)

- ➔ Exceptions (or software interrupts) are of 3 categories: Traps, Fault and Abort
  - A trap is reported after the execution (ex: INT) and allow process continuity
  - A Fault is reported before the actual execution to allow to fix it (ex: div / 0)
  - An Abort is when everything is on fire. Run.
  - More about it in the x86 Intel manual

CPU interrupts (x86)

- In linux, it's translated as interrupts/IRQ (Interruption ReQuest) and softIRQ (software IRQ)
- Values are exposed in /proc/interrupts
- softIRQ in linux don't show all x86 exceptions
  - softIRQ displayed by Linux are limited, check /proc/interrupts
  - softIRQ is a "primitive" system that has been partially taken over by tasklets
- There is no direct mapping between linux exposed values and x86 events

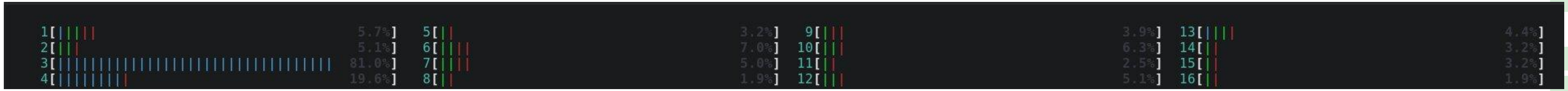
CPU metrics in the end

Putting everything together

Understanding CPU metrics

- ➔ User -> userland code
- ➔ Nice -> process with high niceness
- ➔ System -> kernel-land code
- ➔ Idle -> CPU literally doing nothing (~no power usage, C-state)
- ➔ Iowait -> time spent for a process waiting for I/O (unreliable)
- ➔ Irq -> hardware interrupts
- ➔ Softirq -> software interrupts
- ➔ Steal -> As a VM, hypervisor didn’t schedule us
- ➔ Guest -> kernel KVM gave CPU time to VM
- ➔ Guest_nice -> kernel KVM gave nice CPU time to VM

Understanding CPU metrics



- ➔ htop CPU bars have colors representing the different kind of CPU metrics
- ➔ By default:
  - ◆ blue = nice
  - ◆ green = user
  - ◆ red = kernel (+ iowait + irq + softirq)
  - ◆ orange = guest (+ steal)

PSI – how to represent pressure

Getting a higher level metric to abstract

Monitor system going wrong

- ➔ Let's say you want to monitor your system and try to detect problematic states
- ➔ What is a problematic state ?
    - ◆ Let's define this in this context by "a state when your workload doesn't run properly or in a degraded state, not exploiting your machine full capacity"
- ➔ In this case, is a 100% CPU usage defined as a problematic state ?

Understanding when a state is problematic

- ➔ CPU is a complex metric to grasp when trying to investigate problematic situations
  - ◆ Quite some metrics
  - ◆ Easy to get fooled
    - "OMG my CPU is spending all its resources on idle !!"
- ➔ CPU used at 100% doesn't mean your application is disturbed
- ➔ In some cases, it can be impacted without reaching 100%
- ➔ Try to put your metrics in correlation with your application

Understanding when a state is problematic

- 100% CPU usage when compiling the kernel
  - Usually not a problem
    - Indicates reaching your max capacity. You might want to upgrade your CPU maybe ?
  - Can be if done alongside other workload
  - Niceness to keep in mind
- It’s better to rely on what you observe
  - Latency, mouse lag, etc
  - How to program this ?

Monitor system going wrong

- ➔ Load is an indicator indeed but:
  - Relative value (number of cores)
  - No indication of actual waiting time the process had to wait
  - R+D state, so few faulty NFS process and the load goes up the roof
  - Average over time

Monitor system going wrong

- Linux proposes another metric: Pressure Stall Information
- From ~2018 by Facebook
- 3 metrics: io, memory and CPU
- Represent the % of time wasted because of processes conflicts for a resource
  - You can have 100% used CPU core and 0% CPU PSI
- Has a polling interface
  - Used to loadbalance workload in Facebook

What about memory ?

Yet another complex metric

Different kinds of memory

- Memory is a wide term with different kinds:
  - Volatile, fast memory (RAM)
  - Non-volatile, slower memory (swap)
- When trying to understand memory for your system, 2 kinds:
  - Virtual memory
  - Physical memory
- The kernel in combination with the MMU (Memory Management Unit) is responsible for abstracting memory to userland

Physical memory

- ➔ Physical memory is divided in multiple places
  - I.e. 4x16 GiB of RAM in 4 sticks
  - 4 GiB of swap on your NVME disk
- ➔ Physical memory has its own address space
  - Depends on the lanes you're plugging the memory in, the motherboard, …
- ➔ Different sources of memory may have different latencies
- ➔ ….

Virtual memory

- ➔ Userland doesn't want to deal with this
- ➔ Userland wants a unique address space for memory
- ➔ For security reasons, userland processes must not be able to access memory from each other
  - ◆ Per process address space
- ➔ Userland wants the kernel to do things for it
  - ◆ Maybe he wants to interfere a bit with the decisions
    - Advise, flush, ...
    - Control memory-related mechanisms (i.e. swap)

Virtual memory

- Memory will be used in multiple cases:
  - Variable storage (or generic computation needs)
  - Process executable binary
  - Disk cache
  - Kernel memory
  - Page tables
- Memory is used with pages
  - 4 kiB on AMD64
  - Possibility to increase with THP (Transparent Huge Pages)

Process virtual memory

How do a process have access to "memory" ?

Process virtual memory

- → Each process has its own address space
- → For obvious security reasons
- → Each process address space is virtual
  - ◆ 2 process can share the same address in their virtual memory that leads to completely different “real” memory
- → Each process address space is flat: no segmentation
- → Different sections in their address space however
  - ◆ Everything is not identical

Process virtual memory

- Example include a process' own executable code in a memory map, called text
- A process global variables in a data section
- ….
- Each process has a struct mm_struct to describe their virtual address space
  - Actually threads share the same struct since they have the same address space
  - https://elixir.bootlin.com/linux/v6.0.7/source/include/linux/mm_types.h#L486

Process virtual memory

- ➔ Various interesting implementation details about struct mm_struct
  - ◆ Like mmap & mm_rb fields
- ➔ Each process can have (and actually have) VMAs
- ➔ Virtual Memory Area
- ➔ Implements an area of virtual memory, with its property
- ➔ struct vm_area_struct
- ➔ https://elixir.bootlin.com/linux/v6.0.7/source/include/linux/mm_types.h#L397

Process virtual memory

- A VMA is associated to a mm_struct
- It has flags (including R / W / X)
- Makes the link to a file (if not anonymous memory)
- VMAs can be seen in /proc/<pid>/maps
- Each call to mmap creates a VMA
  - Sort of, sometimes if it possible, there are merged together

Pages and Huge pages

Lots of pages and yet not a book released

Virtual memory - pages

- A page of 4 kiB means 256 000 pages for 1 GiB of RAM
  - Memory overhead
  - TLB - Translation Lookaside Buffer
- Possibility to have bigger pages to reduce costs
  - 2 MiB instead of 4 kiB -> 512 times less TLB entries
- Can be “dangerous”
  - Pages allocated but not used don’t count
  - You can `malloc(1024 * 1024 * 1024);` no RAM will be taken
  - You write 1 bit, the page is actually taken
  - You will waste more with huge pages

Virtual memory - THP

- THP can be disabled system wide
  - Or only used when explicitly asked with madvise(2)
- madvise(2) indicates what usage a portion of memory will be subjected to
  - The kernel will apply optimization for such usage
- Usage includes:
  - Normal
  - Random
  - Sequential
  - willneed/dontneed
  - (un)mergeable (KSM)
  - (no)hugepage

Virtual memory - THP

- ➔ People usually don’t care nor pay attention to THP
- ➔ Many applications will malloc(4 * 1024) 512 times instead of allocating (4 * 1024 * 512) directly
  - ◆ Most never use madvise either
- ➔ Therefore, THP system would be unused and useless
- ➔ Linux introduces khugepaged and heuristics
  - ◆ When allocation more than 2 MiB of RAM at once, you usually allocate a THP without knowing it
  - ◆ khugepaged will look for pages to merge in a THP

Let's talk about swap

Why swap is bad but still good (??)

Swap memory

- ➔ For most people, memory = RAM
  - ◆ It's fast, usually big enough
  - ◆ However volatile, so we need to be careful
- ➔ However RAM is pricey
  - ◆ It's quite easy to run out of RAM even with normal (but relatively heavy) applications/processes
- ➔ What happens when we run out of RAM ?
  - ◆ If it's the only memory: oom-killer
  - ◆ If not: swap (then oom-killer if we abuse)

Swap memory

- ➔ Use a persistent disk as a backing storage for more memory
- ➔ Disk can be of various nature (various costs and speed)
  - HDD
  - SSD
  - NVME
  - the shitty 1 GiB USB 1.0 key Capgemini or Sopra Steria gave you in exchange for a CV no one will take time to read
- ➔ Performances will suffer: swap is used as a last resort option

Swap memory

- ➔ When do we swap ?
- ➔ Swap is used when memory pressure is high. You will not use swap before reaching a huge RAM usage first
  - ◆ Swapped memory stays in swap if unaccessed even if the system memory goes down again
- ➔ What are the consequences ?
  - ◆ Swapping out process that takes CPU time and I/O
  - ◆ “Volatile memory” written to non-volatile device (!)
  - ◆ (very) slow memory access on swapped out memory

Swap memory

- Where on disk is stored swap ?
- 2 options:
  - Dedicated partition on a disk
  - Dedicated file on your filesystem
    - Must be on a persistent storage (no tmpfs, duh)
- Swap device needs a specific partition type (mkswap(8))
- Can be enabled and disabled on runtime with swapon(2)(8)/swapoff(2)(8)

Swap memory

- Why is swap bad ?
  - Once you start swapping, performances goes down the drain (-ish)
  - .. ?
- Why swap is needed ?
  - Swap isn't used if the system isn't stressed
  - Most OS or applications don't have an efficient way to react on memory pressure to free-up memory
  - Reaching the limits often means bad things
  - There are traps when reaching high memory usage

Swap memory - vm.swappiness

- What pages of memory shall be sent to the disk ?
  - What are the best candidates ?
- Intuitively one will say:
  - Memory rarely accessed
  - Memory rarely written
  - LRU
- On top of those cases, one important case to not miss is the memory file-backed, or non-anonymous memory
  - i.e.: mmap() of a file, a process binary, ...
  - This memory is recoverable. We can evict it safely from memory altogether

Swap memory - vm.swappiness

- Behavior of what to do when running out of performing memory is controllable via vm.swappiness
- The value range goes from 0 to 200 (recent kernel)
- It is often misunderstood

Swap memory - vm.swappiness

- ➔ What people think (it's wrong):
  - vm.swappiness goes from 0 to 100
  - It indicates the memory threshold at which the kernel will start swapping
  - I.e: if vm.swappiness is at 60, if you take 59% of RAM, no swap, 61%, it will start swapping (maybe)
- ➔ This is stupid and wrong
  - Why 60 % ? Why would the kernel voluntarily drop performances to swap ?

Swap memory - vm.swappiness

- What it does:
  - It's a balance pressure indicator to put more pressure on swapping out anon pages and dropping recoverable file pages
  - It's from 0 to 200. 0 means aggressive on file pages, 200 on anon pages
  - The pressure finally applied is a bit complicated:
    - Swappiness is ignored in some cases
    - Pressure balance is ignored for some part of the list to ensure no leftovers
    - Swappiness = 0 -> no swap unless big troubles

Swap memory - vm.swappiness

- ➔ More information on https://elixir.bootlin.com/linux/v6.0.6/source/mm/vmscan.c#L2731

Fear the OOM-killer

How linux kills userland processes *by design*

OOM-killer

- ➔ When reaching the final limit of available memory
- ➔ Kernel mechanism triggered on allocation failure
- ➔ Find the most suited process to kill
  - ◆ Highest oom_score
- ➔ What is oom_score ?
  - ◆ Per process score always maintained
  - ◆ Amount of RAM being taken
  - ◆ oom_score_adj
  - ◆ Used to be more complex (user vs root process, HW direct access, ...)

OOM-killer

- Configurable in /proc (like most kernel mechanisms)
  - Can be disabled entirely
- Killing processes by design “omg wtf”
  - What do you expect from a system running out of memory anyway ?
- Invocation and its actions logged in /dev/kmsg (dmesg(1))

Getting rid of the OOM-killer

- Some people don't like the OOM-killer
  - But still reckon the job is useful
- Namely for a major reason: it intervenes when it's late
  - Often too late
- 3 projects exists to basically do the same thing, but in userland:
  - earlyoom
  - lmkd on Android
  - Systemd-oomd (which uses memory PSI)

Kernel threads

You’re not the only one having daemons

Kernel threads

- The kernel does some tasks synchronously:
  - Syscalls kernel code is executed when the user calls it
- But there are also asynchronous tasks to perform:
  - Kswapd for example will swap out memory
    - Even compress it with zcache enabled
  - khugepaged does periodic scans to reduce memory fragmentation by merging pages in THP
- Kernel threads are visible with ps or htop like other processes
- They don’t have an associated mm_struct

Understanding memory metrics

Why is my process taking 17 GiB of RAM on my 16 GiB laptop ?

Memory metrics

- The memory metrics we're the most interested in for basic usage is free memory
  - In fact it's incorrect. We want to know what memory is still usable for our process
- Free memory != Usable memory
  - Caches
  - Buffers
- When checking for available memory with `free -m` for example, be careful to read "available" and not "free"

Memory metrics

- ➔ The interface to check global memory usage for the machine is /proc/meminfo
- ➔ It lists memory and breaks it in different kind of usage
- ➔ It's has a lot of fields, some of them are overlapping or imprecise
- ➔ It can be misleading and quite difficult to understand it

Memory metrics /proc/meminfo

➔ Example on my 24 GiB laptop

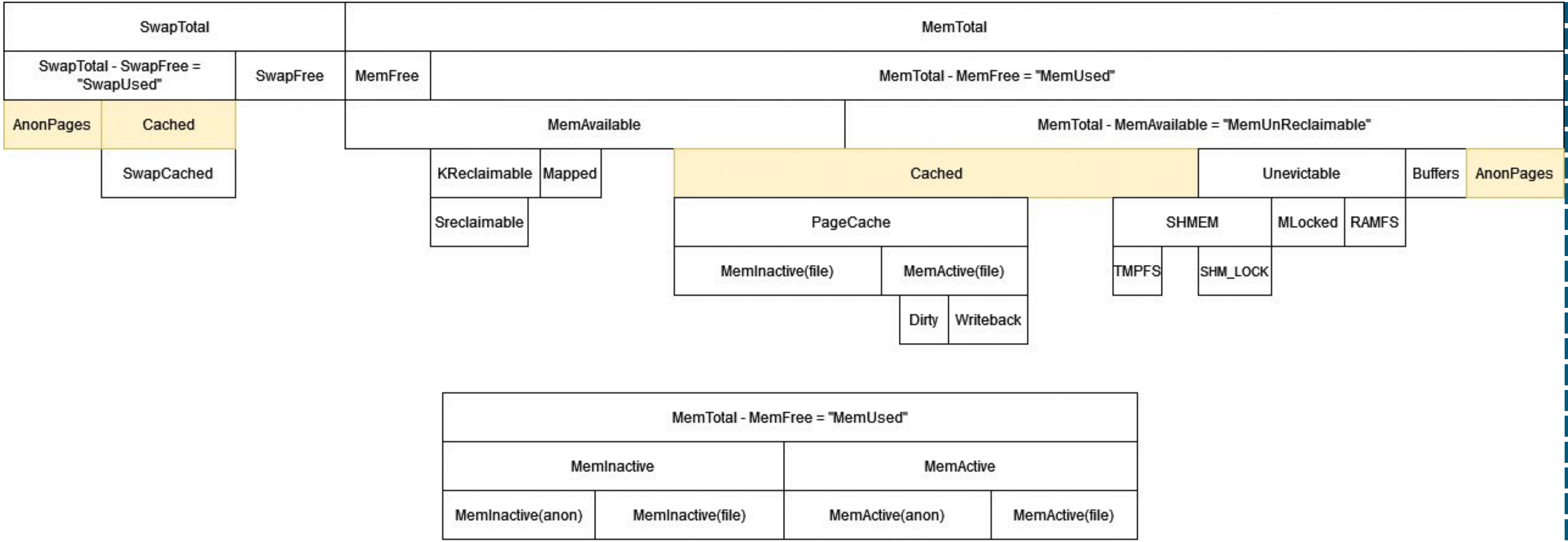
```
MemTotal:          23397872 kB
MemFree:            1394720 kB
MemAvailable:      15413048 kB
Buffers:             184188 kB
Cached:            13302188 kB
SwapCached:           41176 kB
Active:             8512216 kB
Inactive:          10476956 kB
Active(anon):       1169736 kB
Inactive(anon):     4611376 kB
Active(file):       7342480 kB
Inactive(file):     5865580 kB
```

Memory metrics /proc/meminfo

- Example on my 24 GiB laptop
- Buffers: kernel buffers, for block I/O & IPC
- Cached: file pages in memory
  - Include tmpfs & shmem
  - Exclude swapcached
- Swapcached: Memory that was in swap, was put back in RAM but kept in swap

```
MemTotal:          22.3139   GB
MemFree:           1.35636   GB
MemAvailable:      14.693    GB
Buffers:           181.113   MB
Cached:            12.6481   GB
SwapCached:        41.8359   MB
Active:            8.1002    GB
Inactive:          9.97824   GB
Active(anon):      1.10642   GB
Inactive(anon):    4.40702   GB
Active(file):      6.99377   GB
Inactive(file):    5.57122   GB
```

Memory metrics /proc/meminfo



Memory metrics

- Mem prefixed metrics don’t include swap
- Cached is a huge metric but also imprecise
  - Code in the kernel itself to check if cached < 0
  - Quite some subcomponents
  - Despite the name, everything is not “cache” memory
  - Everything can’t be reclaimed
  - Can send partially to swap, but != swapcached
- Unevictable means memory that can’t be sent to swap
- Mapped is mmap(2)-ed files
  - No anonymous mmap for example

Different kinds of memory

Complex graph shown above indicates how complex it actually is

Dirty memory

- ➔ When writing data to a file, by default the data isn't actually written
  - ◆ Well not directly, not always, and it's difficult to predict default behavior
- ➔ Because of performances reasons, when writing to a file, the data is actually put in a special cache in the kernel
- ➔ This cache has a special name: dirty memory
- ➔ Dirty memory is a trick played on the user:
  - ◆ We told them the data is written (write(2) succeeded)
  - ◆ It's actually not really on disk

Dirty memory

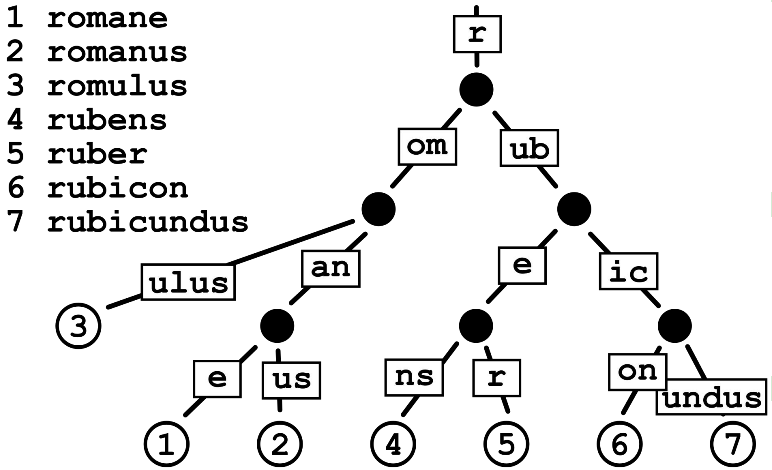
- Dirty memory is “dangerous”
- A hard failure of the system, bug in the kernel, or some nasty crash, and the data it lost
- Dirty memory must be flushed down to the disk
- Dirty memory helps for performances, but introduces a risk
- In fact, MacOS and windows do this as well
  - “Don’t unplug the USB key without ejecting it”
- How to control dirty memory ?
- sync(2)

Dirty memory & writeback

- open(2) flags like O_DIRECT
- Check dirty memory size and watch for high or constant high values
  - In might means that disks are a bottleneck
- The cache mechanism for write is called write-back
- It works with LRU lists
  - Active and inactive list
    - To handle one-access cache eviction case
  - Known as LRU/2

Dirty memory & writeback

- Dirty memory (and page cache in general) is implemented with a struct address_space
- These structs are kept in a radix tree
  - Meaning that the struct are ordered in a prefix tree by their address pointer



Dirty memory & writeback

- Flushing dirty pages to disk is done asynchronously
  - Unless cache is full during a cache manipulating operation
- A page is flushed when it has stayed in the cache long enough
  - Or when memory is running low
  - Or when manually requested with sync(2)
- Behavior is also tunable via knobs in /proc/sys/vm
  - There’s even a laptop_mode option !
    - Sadly mostly useless nowadays

Dirty memory & writeback

- Flushing dirty pages to disk is done asynchronously
  - Unless cache is full during a cache manipulating operation
- A page is flushed when it has stayed in the cache long enough
  - Or when memory is running low
  - Or when manually requested with sync(2)
- Behavior is also tunable via knobs in /proc/sys/vm
  - There's even a laptop_mode option !
    - Sadly mostly useless nowadays

Dirty memory handling

- ➔ How to have both performances and data integrity assurance ?
- ➔ Need to trick with concepts like WAL
  - ◆ Write-ahead Logging
- ➔ Imagine a database context
- ➔ You don’t want to lose data
- ➔ But transactions must be quick
  - ◆ As quick as possible
- ➔ Transactions can be complex. They can impact your whole data

Dirty memory handling

- Likely, the database will be stored on disk on a file
- It can be huge, so a modification can introduce changes in quite some "random" places of the file
- Random access to different places of the file is expensive, in terms of I/O
- Writing the result after each transaction will take a lot of time
  - potentially

Dirty memory handling

- ➔ Instead, WAL technique allow to deal with this exact behavior
- ➔ The WAL is a log file that will record each transactions, in the right order
- ➔ When a client makes a query:
  - ◆ The transaction is written to the WAL
  - ◆ We make sure the WAL is written to disk
  - ◆ We perform the transaction, and return the value
  - ◆ Later, the modified db file might be flushed out to disk

Dirty memory handling

- ➔ If the database server crashes badly, the WAL is still there
- ➔ All successful transactions might not have been flushed to disk
- ➔ The database engine will check its WAL, and assure that data is correct
- ➔ If not, it can correct it since it has all the information in the WAL
- ➔ Regularly the WAL is reseted with a checkpoint
- ➔ Writing to disk the WAL is less expensive since it's append-only mode

NUMA nodes

Architecture comes to play

NUMA nodes

- On some architecture, not all memory is on the same access level
- Especially on “big” servers where it’s not uncommon to have 2 CPUS
  - And 2 memory zones
- Instead of having Unified Memory Access, we now have Non Unified Memory Access
- Reaching memory in node 1 from CPU on node 0 is possible
  - But more expensive

NUMA nodes

- ➔ Linux is NUMA aware
- ➔ numactl --hardware
- ➔ cat /proc/cpuinfo ; cpuinfo
- ➔ The scheduler runs in best-effort by default
- ➔ If a task has been running in a NUMA node, it will try to keep it there
- ➔ Has some functions and data structure to perform its NUMA assignation
- ➔ https://elixir.bootlin.com/linux/latest/source/kernel/sched/fair.c#L1439

NUMA nodes

- ➔ But this best-effort mode can actually be not good enough
- ➔ Especially in some cases where the machine is quite loaded
- ➔ It might actually sometimes be a good trade to force a task to run on a NUMA node
    - ◆ Reducing its CPU & RAM capacities
- ➔ Good example: VMs on an hypervisor

NUMA nodes

- Possibility to visualize NUMA memory allocation
- numastat
  - Has even a per-process information
  - /proc/<pid>/numa_maps
- Possibility to set a NUMA policy
  - set_mempolicy(2)
  - sched_setaffinity(2) also
  - Or via cgroups

Sidenote: Pushing debug tools even further

Carcinization of debugging tools

Advanced debug tools

- ➔ strace(1) is considered a debugging tool
  - ◆ syscall oriented
- ➔ gdb(1) remains the “true and only debugger”
  - ◆ Or is it ?
- ➔ gdb(1) while having tons of functionalities lacks a critical component
- ➔ record & replay
- ➔ Mozilla introduces rr

rr

- → The rr project is a debugger project built on top of GDB
- → Not a replacement
  - ◆ Allow you to keep using all the GDB features
  - ◆ Not asking you to learn everything again
- → rr works by recording your buggy software first
- → Like strace(1), it will inspect closely what your program does
- → Record it
- → And provide a way to replay it, in the exact same context

rr

- Replaying the exact same session is very useful for a few reasons:
  - No need to make the user interact the same way everytime
  - Ability to catch a misfortune once and work on it
    - Race conditions, thread problems, ...
  - Keep learned info in a debugging session across runs (pointer values, etc)

“ *Let’s check a quick example*

rr

- ➔ rr was designed by Mozilla to debug firefox
- ➔ It's able to debug complex software like firefox
- ➔ It has some limitations though
  - Single core machine emulated
  - x86 CPU
  - some syscalls not tracked
  - Can break on kernel update

rr - how does it work ?

- ➔ rr when started records everything to replay the exact same session
- ➔ ptrace(2)
- ➔ seccomp(2)
- ➔ Because one of the thing rr tries to catch is race conditions between threads, it must be able to catch them
- ➔ rr runs all thread on the same CPU core to be sure to capture such events
  - ◆ Impacts perf on heavily parallelized computation

rr - how does it work ?

- ➔ To be able to run all threads efficiently on the same core and catch their output, rr is preemptive
- ➔ When a thread enters a syscall, ptrace(2) catches the syscall and hand is given back to rr
- ➔ rr also periodically preempts threads with signals
- ➔ It chooses which thread to run while trying to respect linux scheduler and its priorities

rr - how does it work ?

- When a program do a syscall, rr catches it because of ptrace(2)
- It chooses to resume the syscall, but catches the return value
  - Like strace(1)
- It stores the syscall interaction in a replayable format
- It works for most syscalls, but ptrace(2) itself
- A process can only be ptrace-d once
  - And firefox and many other already use ptrace(2) on themselves

rr - how does it work ?

- rr emulates ptrace(2) syscall to bring compatibility
- rr has to deal with complex situations
  - Ioctl
  - Namespaces
  - ...
- To replay a recorded trace, ptrace(2) is also used
- rr replaces all syscalls with breakpoints
  - It moves past the breakpoint, and set the return value as recorded

rr - how does it work ?

- ➔ Some syscalls are harder to replay
  - ◆ mmap(2) - you need to have the same address
  - ◆ execve(2) - you have memory mappings that can change (ASLR)
- ➔ rr has to trick or implement complex logic to properly emulate them
- ➔ Asynchronous events must also be handled
  - ◆ Signals, interrupts
- ➔ They must be sent at the exact same time

rr - how does it work ?

- ➔ rr is able to time precisely when async events occurs to replay them the same way
- ➔ It relies on x86 specific performance counters
- ➔ rr must also catch race conditions happening on shared memory
- ➔ As they describe, famous cases includes X server, pulseaudio, GPU related function and vdso
- ➔ They disable shared memory for X and pulseaudio and remove direct access for GPU
  - ◆ Worse perfs, but ability to replay the bug

rr - how does it work ?

- ➔ For VDSO, rr live-patches vdso in the tracee address space to replace VDSO calls to actual syscalls
- ➔ rr must also be able to catch non-deterministic CPU instructions
- ➔ RDTSC is caught via prctl(2)
- ➔ RDRAND is rarely used, it’s replaced manually in the few places found, but this is not caught by rr
- ➔ CPUID returns the core number, so sched_setaffinity(2) is used to force a core

rr – how does it work ?

- ➔ For VDSO, rr live-patches vdso in the tracee address space to replace VDSO calls to actual syscalls
- ➔ rr must also be able to catch non-deterministic CPU instructions
- ➔ RDTSC is caught via prctl(2)
- ➔ RDRAND is rarely used, it’s replaced manually in the few places found, but this is not caught by rr
- ➔ CPUID returns the core number, so sched_setaffinity(2) is used to force a core
  - ◆ These instructions might be handled differently is recent versions thanks to CPUID faulting

rr - how does it work ?

- ➔ As a practical point of view, the trace created shall remain quite small
- ➔ It's compressed (and decompressed) on-the-fly by rr
- ➔ Shared libraries and binaries are stored via hard links or cow mechanisms
- ➔ Because ptrace(2) introduces a context switch (from tracee to tracer and vice-versa), and because it's used twice per syscall (before and after), it affects performances drastically
  - ◆ But rr is clever

rr - how does it work ?

- To avoid running to many ptrace(2), rr injects a library in each tracee
- The library overwrite syscalls wrappers
- The library performs the syscall, but write information to a shared buffer, shared with rr
- It tries to catch most frequently used syscalls this way
    - But fallbacks to the ptrace(2) + syscall in other cases
- ... there are many other challenges solved by rr
    - [Read there paper explaining most of them]

rr - how does it work ?

- ➔ The master of engineering put in rr leads to a very practical tool
- ➔ The overhead it adds is about 20% on firefox
- ➔ If firefox takes 10min to perform a task, it will take 12min max with rr as observed
- ➔ All these elements make rr also very powerful with fuzzers

Memory overcommiting

Let's go beyond limits

Memory overcommiting

- /proc/meminfo also has some metrics about virtual memory
- On linux, you can over-allocate
  - vm.overcommit_memory + vm.overcommit_ratio
- An allocation in virtual memory != necessarily bound to physical memory
  - It is if it’s used, meaning written to
- Useful because softwares tend to allocate more than they actually use
  - That’s also a reason why you’ll unlikely see a negative answer from malloc(3)

Memory overcommiting

- 3 overcommitting modes possible:
  - 0 -> heuristic, let the kernel decide (default)
  - 1 -> always allow, never check
  - 2 -> always check
- In /proc/meminfo:
  - Committed_AS is the sum of all committed (allocated virtual memory for all processes)
  - CommitLimit is the maximum amount of memory allocatable
    - Makes sense in mode 2 only

Memory overcommiting

- ➔ Overcommitment lead to memory limit being hit before a memory allocation syscall fails
- ➔ Checking return value of malloc(3) won’t guarantee the memory is yours
- ➔ You will trigger the OOM-killer in fact
- ➔ Still check return value …

memory metrics

- Other metrics are also available in:
  - /proc/vmstat
  - /proc/swap
  - /proc/buddyinfo
  - ….
- Also some per-process metrics
  - /proc/<pid>/maps
  - …

Can we pause a minute and finally explain /proc ?

A small dive in pseudofilesystems

What is a filesystem?

- A regular filesystem should be a well known notion
- A disk (HDD, SSD, ...) is exposed as a block device on linux
  - Special file, allows “raw” access to the disk
    - Not quite, but let’s keep this definition
- To be used as one would expect (put directories, files, etc), a filesystem must be created on the disk
- A filesystem is a data layout specs
  - A data structure
  - And its driver
  - Integrated in linux through abstraction interfaces

What is a filesystem?

- ➔ Different kinds of filesystems with different approaches, pros and cons
  - FAT, EXT4, XFS, ZFS, BTRFS, NFS, NTFS, ...
  - Can be thought for the network (NFS, CEPHFS, GLUSTERFS, ...)
  - Can have built-in snapshot mechanisms
  - Can have a journal
  - Can support extended attributes
  - Is more or less subject to fragmentation
  - ...

What to do with a filesystem?

- ➔ Once your disk is formatted with a filesystem, it can be used
- ➔ With windows, it’s directly accessible with a letter (C:, D:, ..)
  - ◆ It’s simpler for them, but also kind of stupid
  - ◆ No unified hierarchy
  - ◆ What about letter conflicts ?
- ➔ In linux, you have only one hierarchy: the Virtual FileSystem

Linux VFS

One hierarchy to rule them all

Linux VFS

- On linux there is no disk drive letter, only "/", the root
- Linux maintain internally the VFS, a unified file hierarchy
- You can put a disk filesystem somewhere in the VFS
  - This operation is called mounting
- Everything under the mount point will be bound to the filesystem
  - Read, writes, etc
- It's common to have the root of the VFS mounted on a disk partition
- The VFS is what you can see when "exploring files" on linux

Linux VFS

- The VFS is the concept that allows having multiple physical storage support under the same hierarchy
- It allows an abstraction of the actual operations performed to the user

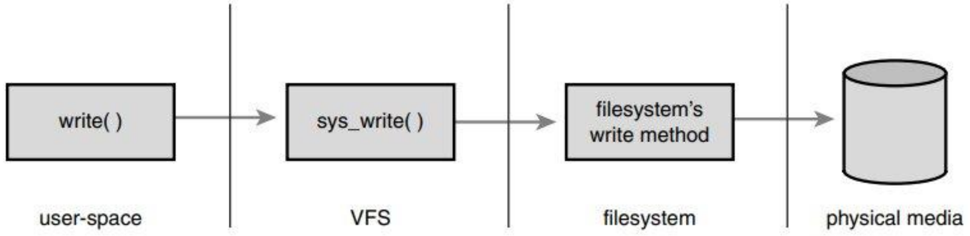


Figure 13.2 The flow of data from user-space issuing a `write()` call, through the VFS's generic system call, into the filesystem's specific write method, and finally arriving at the physical media.

Linux VFS

- ➔ The linux VFS is tightly tied to the concept of UNIX filesystem
- ➔ It was indeed built on top of the ext2 filesystem
- ➔ A UNIX filesystem in short is built with 4 concepts:
  - Files
  - Directory entries
  - Inodes
  - Mount points

Linux VFS

- If you want to access your USB key for example, you need:
  - To have a filesystem created on your disk, or on a partition
    - The filesystem needs to be compatible with your OS
  - To mount this filesystem somewhere in your VFS
    - If it's just to access its files, you should put it somewhere it doesn't impact your system, like /mnt

Linux VFS - mount

- To mount a filesystem in the VFS, one can use mount(1)
- This command (and its underlying syscall) will take a source device, and add it in the VFS at some path
  - This means that everything that used to be on this path and below isn't directly accessible anymore
    - It is still accessible by tricking
    - Opened files stay open, and modification are propagated

Linux VFS - mount

- The source device is usually a block device (a hard drive), but it can also be something else, like:
  - A network address, when mounting a NFS partition for example (or glusterfs, cephfs, etc)
  - A special kind of source known as a pseudo-filesystem
- You can check the supported filesystem in /proc/filesystem
  - Filesystem marked with “nodev” means that they don’t need a block device

Pseudo filesystem

Some filesystems are not like the others

Pseudo filesystem

- ➔ A filesystem is usually meant to store and access files
- ➔ But in Unix philosophy, everything is considered a file, even if it’s not truly is one
- ➔ For example, you might know the special file /dev/zero or /dev/null
- ➔ There is no such infinite file on your disk than you can read forever, or write to without it being actually written
- ➔ This is an interface the kernel exposes you

Pseudo filesystem

- ➔ When doing a open() syscall, the kernel will do a few things like checking the path, permissions, etc ...
- ➔ Then it will dispatch the syscall to the driver responsible for the file
  - ◆ If the file is on an ext4 partition for example, we need to run code specific to ext4 data structure (which is in the end what a filesystem is)

Pseudo filesystem

- ➔ We could come up with a special filesystem driver, that will execute functions for us depending on the file we read/write
- ➔ For example, a file that will execute this function when read:

```
def read_dev_zero(length, buff):
    if len(buff) > length:
        length = len(buff)
    memset(buff, 0, length)
    return length
```

This is obviously pseudo-code and not the actual linux implem of /dev/zero

Pseudo filesystem

- We can go a bit further, and imagine this as a whole interface
- For example, /proc
- It’s a pseudo filesystem mounted in /proc called procfs
- procfs exposes information about processes and various other runtime information
  - meminfo, filesystems supported, etc
- When reading a file there, you actually run kernel code that generates a response for you
- There is no disk space taken, only RAM for the responsible kernel code

Pseudo filesystem - procfs

- procfs goal is quite easy to understand, and is mostly read-only to return kernel runtime values
- But we can have other filesystems a bit more complex
- procfs for example which role is to expose current kernel parameters and settings for many things (memory, network, etc).
  - They can be read, but also written to, to dynamically change the kernel behaviour
  - You can for example disable IPv6, drop memory caches, etc ....

Pseudo filesystem - procfs

- procfs is more or less the config interface for the kernel, with the command line
- Regularly used with sysctl(1) binary

Pseudo filesystem - tmpfs

- tmpfs is a very useful pseudofilesystem
- Everything inside is stored in RAM
  - Very fast accesses
  - volatile , reboot = data gone
  - Usually mounted at least in /tmp
- When mounting this pseudofilesystem, size argument used to give the maximum size
  - Defaults to half the RAM

Pseudo filesystem - devfs

- ➔ devtmpfs is also a well known pseudo filesystem expected to be mounted on all platforms, on /dev
- ➔ It’s a bit special, being a tmpfs, another pseudofilesystem, but with special behaviour
- ➔ It differ from tmpfs by having automatically linux driver register block devices they create in the filesystem
- ➔ /dev – or devtmpfs – contains a block and chardevices:
  - Your disks – and their partition(s) if any
  - Special files like zero, null, urandom, kmsg …
  - Your tty(s)
  - ….

Pseudo filesystem – cgroups

- Another pseudofilesystem you might have encountered already is the cgroups (v1 or v2) fs
- Interface to manipulate the control groups
- Use extensively by systemd, docker, etc …
- Let's not get into too many details here

Understanding VFS structure

Will be useful to understand some metrics and do advanced monitoring

The VFS structure

- ➔ The linux VFS is tightly tied to the concept of UNIX filesystem
- ➔ It was indeed built on top of the ext2 filesystem
- ➔ A UNIX filesystem in short is built with 4 concepts:
    - Files
    - Directory entries
    - Inodes
    - Mount points

The VFS structure

- The UNIX filesystem build those 4 representations this way:
  - File
    - A file is a set of bytes, and doesn’t contain metadata
    - A directory is a special kind of file that lists its content
  - Inode
    - An inode represents the metadata of a file. It has a unique number in a given filesystem
  - Mount points - or superblock
    - Contains metadata information for the whole fs

The VFS structure

- The UNIX filesystem build those 4 representations this way:
  - Dentry
    - Directory Entry
    - Represents the components of a path

The VFS structure

- The VFS will be built on those 4 unix concepts
- Any filesystem not implementing a concept listed above will have to provide a compatibility layer
  - The driver will have to create one of those concept on-the-fly
  - With some (usually) negligible overhead
- Those are requirements for the VFS

The VFS structure

- ➔ The VFS is the abstract representation exposed to the user, more or less indirectly
- ➔ It needs to be abstract and compatible with any “backend”
- ➔ Which means its structure must be able to interact with any actual filesystem implementation
  - ◆ For pseudo-filesystems implemented in linux, it’s trivial, but for external ones, harder
- ➔ The VFS can be complicated with many mounts, filesystems mounted in multiple places, etc

The VFS structure - super_block

- The UNIX filesystem concept of a superblock is mapped to a struct super_block in the VFS
- https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h#L1451
- This struct contains information about a mount point
- It contains a struct super_operations that will provide functions to filesystem-specific pointers for the filesystem-specific operations
  - https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h#L2222

The VFS structure - super_block

- The super_block is usually a mapping to the filesystem's control block of superblock, stored on the disk for regular filesystems
  - The metadata information for a filesystem
  - How many files, its size, …
  - Generated on-the-fly for pseudo-filesystems
- Contains also run-time information for the mount-point
  - Is frozen ? Is dirty ? Mount flags, …
    - Frozen = block write operation on a fs

The VFS structure - super_block

- ➔ Its operations struct will allow operations of the super_block itself
  - ◆ Sync to the disk, remount, freeze, get statistics, ...
- ➔ But also on the inodes it handles
  - ◆ Create, delete, dirty, ...
- ➔ Having the super_operations allows genericity in the manipulation of super_block object, but filesystem-specific implementations of such operations

The VFS structure - inode

- Implemented in the VFS as the struct inode
  - https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h#L593
- Also contains a struct for an inode's operation
  - include/linux/fs.h - Bootlin
- This struct contains information about an inode (file metadata)
- The inode content is written on the disk but the struct is generated when the file is accessed

The VFS structure - inode

- ➔ Since an inode is generic for all files, and in UNIX everything is a file, struct inode contains a union for specific files
  - ◆ i_pipe, i_bdev, i_cdev
- ➔ An inode contains quite some fields that can be omitted in a driver implementation
  - ◆ For example i_atime
- ➔ Operations includes:
  - ◆ create, mkdir, mknod, symlink, permissions, ...
- ➔ No read, write !

The VFS structure - file

- It is important to distinguish a file from the UNIX filesystem concept to the struct file, aka the VFS file concept
- A UNIX file is what people usually understand by a file
  - Without the metadata
- A struct file represents a per-process file interaction
- A struct file is what process usually interact with
  - Especially a file.f_op
- [include/linux/fs.h - Bootlin]

The VFS structure - file

- A struct file is created by open(2)
- A process can have multiple struct files pointing to the same UNIX file, the same inode
- The struct inode it however unique
- Its operations are all the operations you think about when thinking about file manipulation from a process
  - read, write, llseek, ioctl, mmap, ...

The VFS structure – dentry

- ➔ The difference between the kernel and users for files are how there are identified
- ➔ A user (usually) identifies a file by its path
  - ◆ open(2) syscall for example
- ➔ The kernel by an inode number
  - ◆ The path is used to translate to this inode concept
  - ◆ The same file (same inode) can have multiple paths for example
- ➔ A file is a generic term and can have multiple types, including being a directory

The VFS structure - dentry

- ➔ Each component of a path is decomposed in objects called dentry
- ➔ /bin/bash is .. 3 dentry objects
  - /, bin and bash
  - The first 2 are dentry representing a directory, the latest is a regular file
- ➔ A dentry object is a VFS specific object. There’s no direct information about underlying object pointed by it
- ➔ struct dentry

The VFS structure - dentry

- The role of the dentry object is to ease the user manipulation of file and directories
- Those operations are costly
  - String manipulation
  - Need to check if valid
  - Check its subcomponents
  - ...
- The dentry object is really meant to represent a path
- A mount point, a directory, a file will have a struct dentry
  - When needed

The VFS structure - dentry

- ➔ The role of the dentry object is to ease the user manipulation of file and directories
- ➔ Those operations are costly
  - ◆ String manipulation
  - ◆ Need to check if valid
  - ◆ Check its subcomponents
  - ◆ ...
- ➔ The dentry object is really meant to represent a path
- ➔ A mount point, a directory, a file will have a struct dentry
  - ◆ When needed

The VFS structure - dentry

- A dentry can be positive or negative
  - A positive ones means it has an inode associated to it
  - A negative one is the opposite
- A negative dentry (because the path is wrong for example) can be kept in cache to resolve queries quicker
- A dentry can also be considered as used or unused via d_count
  - d_count counts the number of active reference to the associated inode
    - Meaning if there are active users of the object

The VFS structure – dcache

- ➔ All those presented mechanisms lead to an obvious design: dcache
- ➔ dcache is a cache mechanism to store, access and remove dentry objects to have quicker accesses to files
- ➔ dcache keeps track of dentry objects, in both active (used) state, and inactive (unused but valid) and negative state (invalid)
- ➔ It provides a hash table to have quick access
  - ◆ d_lookup()

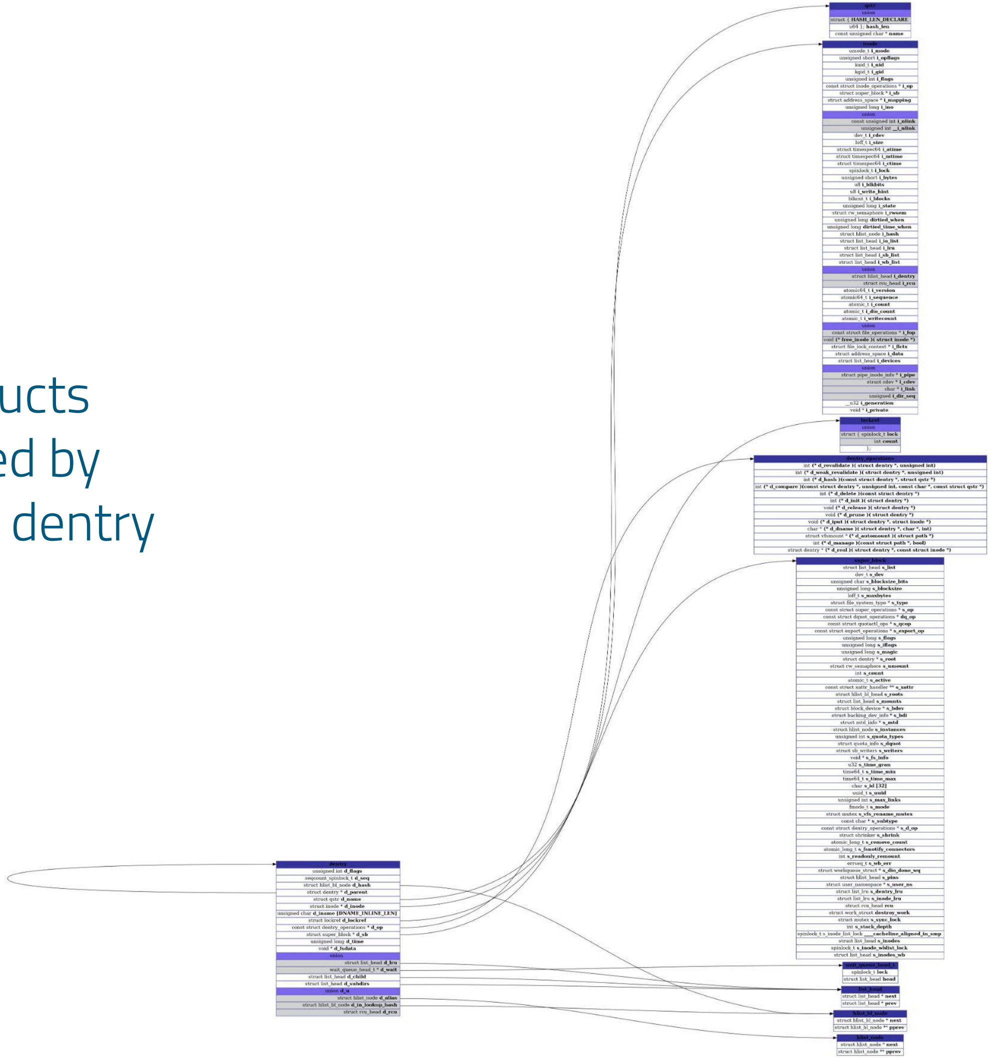
The VFS structure - dentry

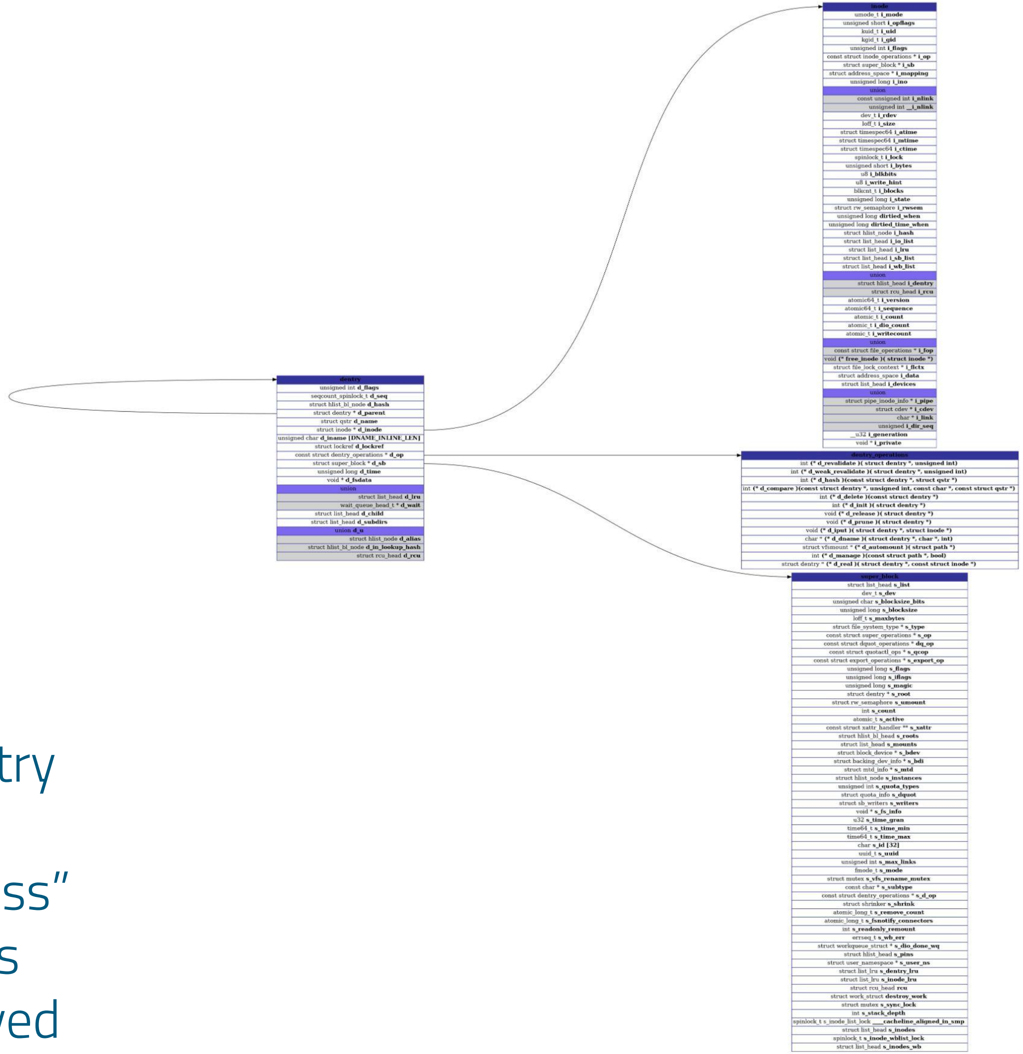
- ➔ The struct dentry also contains an operation struct
  - ◆ https://elixir.bootlin.com/linux/v6.0.7/source/include/linux/dcache.h#L127
- ➔ The operations of a dentry includes:
  - ◆ revalidate, hash, compare, ...

struct dentry

dentry
unsigned int **d_flags**
seqcount_spinlock_t **d_seq**
struct hlist_bl_node **d_hash**
struct dentry * **d_parent**
struct qstr **d_name**
struct inode * **d_inode**
unsigned char **d_iname [DNAME_INLINE_LEN]**
struct lockref **d_lockref**
const struct dentry_operations * **d_op**
struct super_block * **d_sb**
unsigned long **d_time**
void * **d_fsdata**
union
struct list_head **d_lru**
wait_queue_head_t * **d_wait**
struct list_head **d_child**
struct list_head **d_subdirs**
union **d_u**
struct hlist_node **d_alias**
struct hlist_bl_node **d_in_lookup_hash**
struct rcu_head **d_rcu**

- ➔ All structs pointed by struct dentry

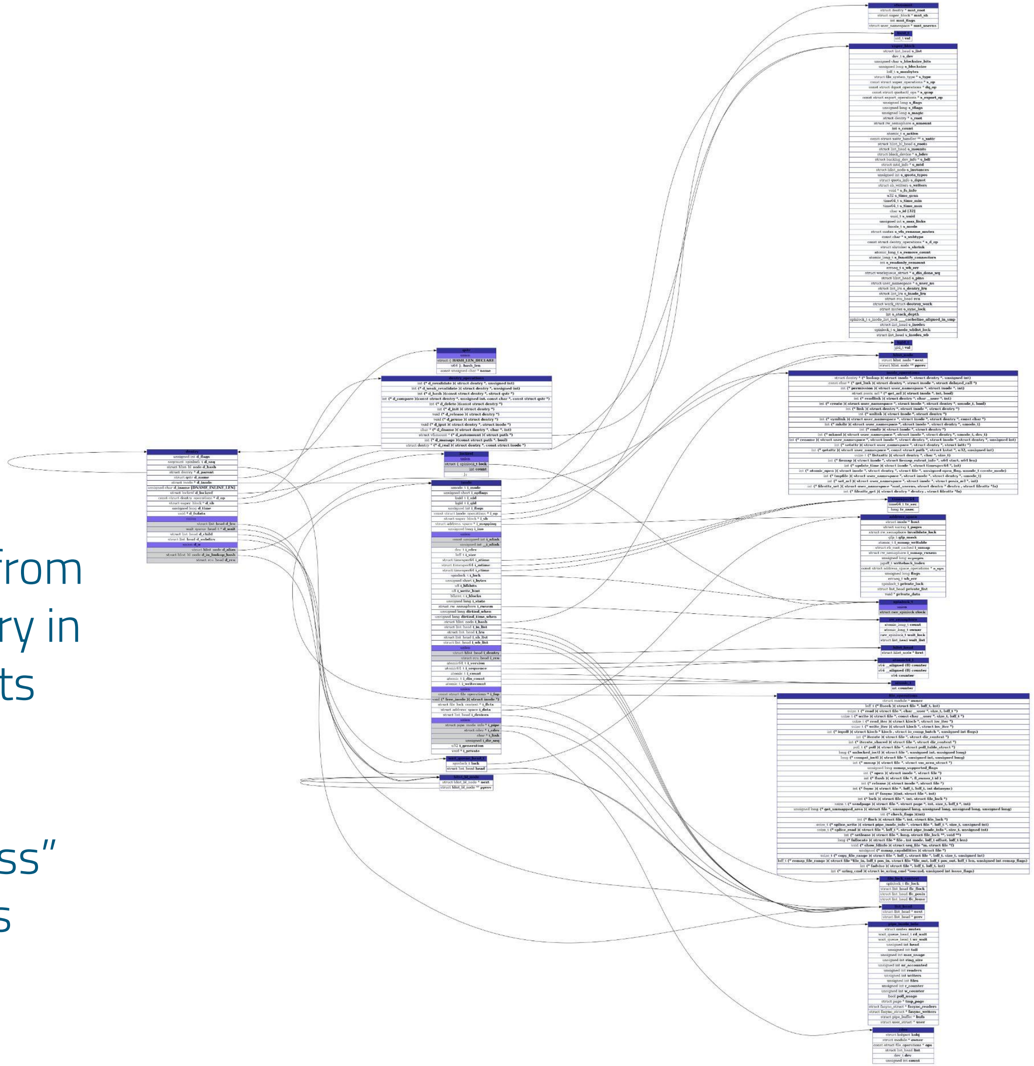




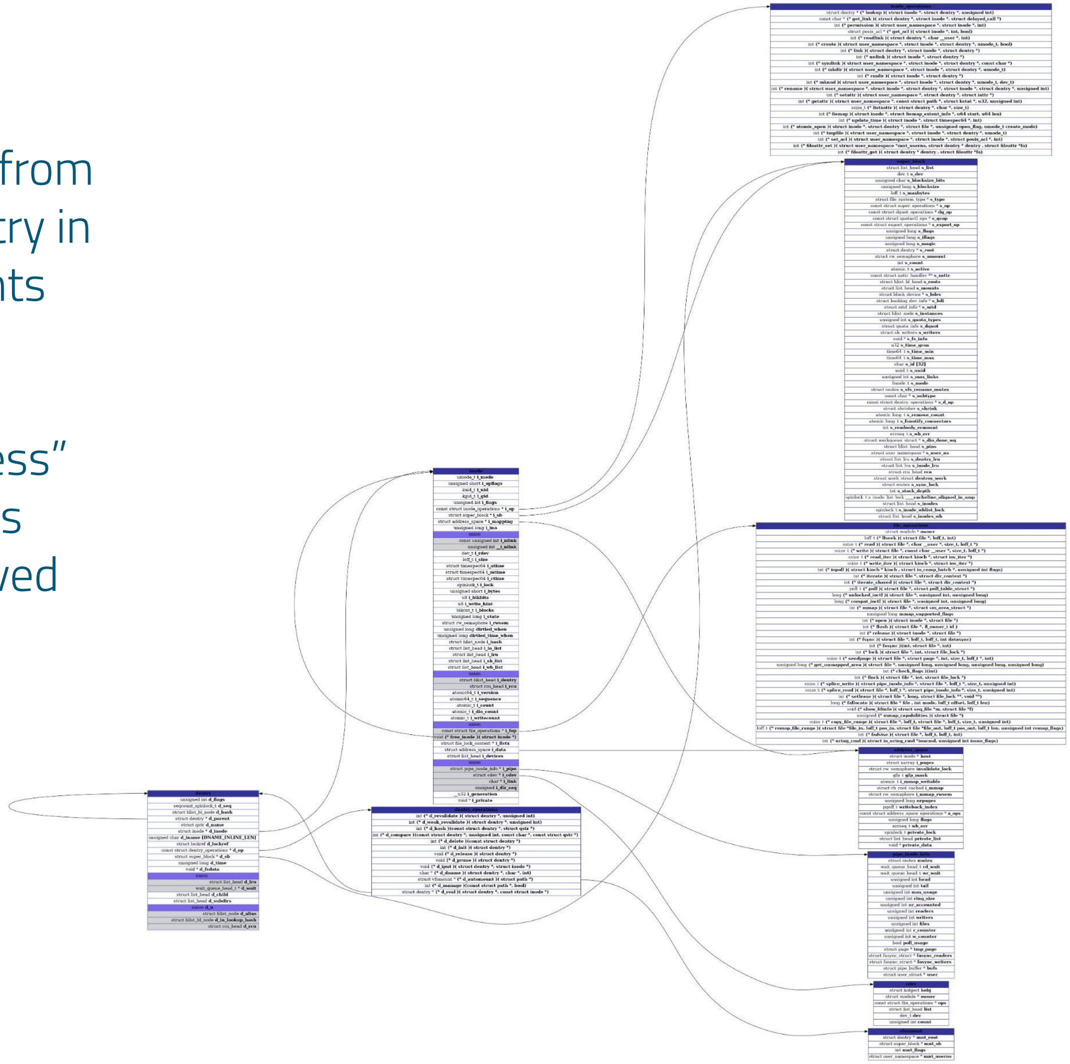
- ➔ All structs pointed by struct dentry
  - ◆ With “useless” structs removed

➔ All structs from struct dentry in a 2 elements range

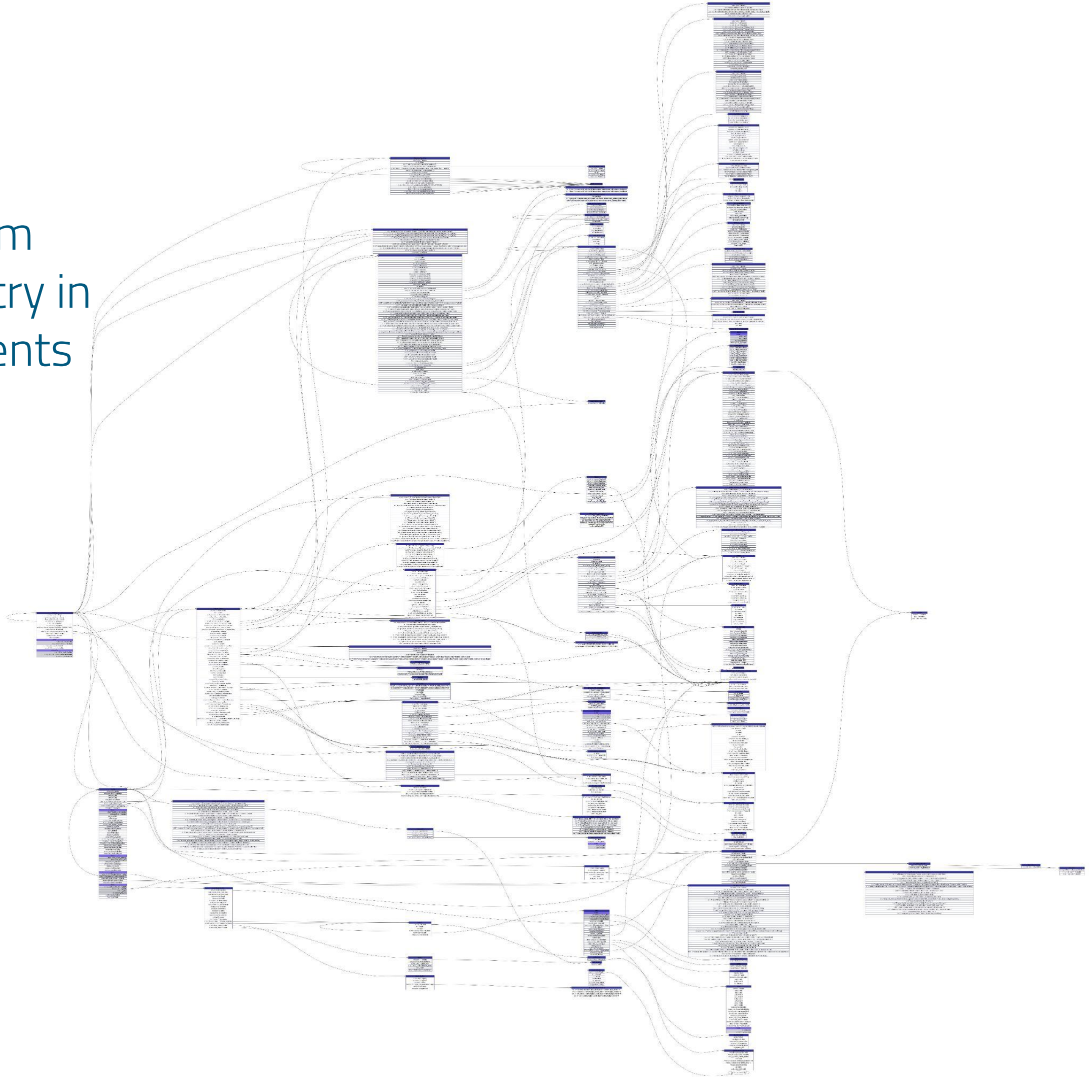
◆ With "useless" structs



- ➔ All structs from struct dentry in a 2 elements range
  - ◆ With “useless” structs removed



➔ For fun: All structs from struct dentry in a 10 elements range



➔ For fun: All structs from struct dentry in a 50 elements range

VFS observability

How can we observe what is going on with the VFS ?

Observe the VFS

- The main interface is /proc/sys/fs
- We have for example dentry-state
  - Exposes the content of dentry_stat_t
  - https://elixir.bootlin.com/linux/v6.0.7/source/fs/dcache.c#L118
- file-max
  - https://elixir.bootlin.com/linux/v6.0.7/source/include/uapi/linux/fs.h#L97
- A few other interfaces, but yet limited

Observe the VFS

- The VFS is hardly observable, and actually matters less than actual filesystem underneath
- Syscall to get some information about a filesystem: statfs(2)
  - Actually the recommended glibc wrapper in statvfs(3)
- Hardly any other interfaces for filesystems :/
- What is to be observed would mostly be I/O on actual physical devices
- It is frustrating because there are quite some structs, operations going on
  - But no interfaces …

eBPF

Let’s talk about the elephant in the room

Linux observability

- ➔ Most of this course has been focusing on two things:
  - ◆ Understanding linux kernel mechanisms
  - ◆ Understanding how to observe them
    - And observability in general
- ➔ For the past years, a fancy term appeared and is mentioned frequently
  - ◆ Everytime one talk about linux and observability in the same sentence
  - ◆ On lots of cool, modern and alpha projects

eBPF

- eBPF is relatively new feature added to linux “recently”
- It is actually an extension of a feature existing in the kernel for years: BPF
  - Berkeley Packet Filter
    - Berkeley being the same as in Berkeley sockets or BSD
  - Now it’s extended BPF

Getting to know BPF

- eBPF is a newer and more powerful BPF
  - But what is BPF already ?
- BPF is an old project that has its roots in BSD
- It's from the early 1990
- It has been integrated in Linux in the beginning of the 2000's
  - Around linux 2.5
- It has since been evolving gradually to become what is know known as eBPF

Getting to know BPF

- BPF until the 2010's had a simple purpose: filter network paquets
- It comes with a virtual machine running in kernel land
- It executes BPF scripts
  - BPF scripts are small programs with special instructions
  - It's not x86 (or else) instructions directly
- Scripts are limited
- Used by tcpdump for example

Getting to know BPF

```
$ sudo tcpdump -ni any dst host 127.0.0.1 and dst port 19999
tcpdump: data link type LINUX_SLL2
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapshot length 262144 bytes
17:42:23.798260 lo    In  IP 127.0.0.1.46842 > 127.0.0.1.19999: Flags [S], seq 3512779186, win 65495, options [mss 65495,sackOK,TS val 2009782134 ecr 0,nop,wscale 7], length 0
17:42:23.798328 lo    In  IP 127.0.0.1.46842 > 127.0.0.1.19999: Flags [.], ack 908325355, win 512, options [nop,nop,TS val 2009782134 ecr 2009782134], length 0
17:42:23.798508 lo    In  IP 127.0.0.1.46842 > 127.0.0.1.19999: Flags [P.], seq 0:79, ack 1, win 512, options [nop,nop,TS val 2009782134 ecr 2009782134], length 79
17:42:23.798651 lo    In  IP 127.0.0.1.46842 > 127.0.0.1.19999: Flags [.], ack 32769, win 379, options [nop,nop,TS val 2009782135 ecr 2009782135], length 0
17:42:23.799084 lo    In  IP 127.0.0.1.46842 > 127.0.0.1.19999: Flags [F.], seq 79, ack 40380, win 512, options [nop,nop,TS val 2009782135 ecr 2009782135], length 0
^C
5 packets captured
10 packets received by filter
0 packets dropped by kernel
```

Getting to know BPF

```
$ sudo tcpdump -ni any dst host 127.0.0.1 and dst port 19999 -d
tcpdump: data link type LINUX_SLL2
(000) ldh      [0]
(001) jeq      #0x800           jt 2    jf 14
(002) ld       [36]
(003) jeq      #0x7f000001      jt 4    jf 14
(004) ldb      [29]
(005) jeq      #0x84            jt 8    jf 6
(006) jeq      #0x6             jt 8    jf 7
(007) jeq      #0x11            jt 8    jf 14
(008) ldh      [26]
(009) jset     #0x1fff          jt 14   jf 10
(010) ldxb     4*([20]&0xf)
(011) ldh      [x + 22]
(012) jeq      #0x4e1f          jt 13   jf 14
(013) ret      #262144
(014) ret      #0
```

Getting to know BPF

- Executing code dynamically in the kernel in a virtual machine looks very interesting
- One could want to execute more than just packet filtering
- Maybe report information about the packets to userland ?
- Maybe include some packet manipulation ?
  - Even dropping them ?
- Etc ...
- As ideas came along, BPF subsystem grew

From BPF to eBPF: internal BPF

- The virtual machine idea looked seducing
- A JIT was introduced to compile BPF instruction to x86 instructions
  - So it was fast
- BPF had too many limitations, like the number of registers available (2), no 64 bits registers, ...
- 2 BPFs were created: classic BPF and internal BPF
- Internal BPF was more powerful, but was hidden from userspace

From BPF to eBPF: internal BPF

- ➔ The only interface was for classic BPF
- ➔ Classic BPF was then transformed in internal BPF in the kernel
  - ◆ Faster, using x86 instructions directly, …
- ➔ But the idea of exposing an interface for internal BPF to userland was already there
- ➔ The goal was also to work on GCC/LLVM to generate internal BPF directly
- ➔ Internal BPF was also able to call some limited set of kernel functions

From BPF to eBPF: internal BPF

- Internal BPF was also starting to draw attention for tracing
- Could be useful to use this dynamic language and virtual machine to run more things dynamically
  - No more real connection with the network
- BPF moved from the net/ to kernel/bpf
  - Removed also some ties with network as well
- This internal BPF was then renamed as eBPF for extended BPF

eBPF evolution

- ➔ What happened after ?
- ➔ bpf(2) syscall was introduced
- ➔ Load a BPF program in the kernel
- ➔ Of course, huge restrictions:
  - We can’t run arbitrary code in the kernel
  - We can’t call any function
  - We can’t have a loop
  - We can’t sleep

eBPF evolution

- ➔ The introduction of the bpf(2) syscall also introduce a BPF verifier:
  - ◆ Tries to prevent harmful programs from being loaded
  - ◆ Things mentioned on the previous slide
  - ◆ But also read from unallocated registers, bound checks, etc
- ➔ Obviously a privileged syscall requiring CAP_SYS_ADMIN (until 5.8 when CAP_BPF was added)

eBPF evolution

- Added feature to eBPF at that time was also maps
- eBPF maps are key/value store to exchange data from kernel to userland
  - They are created from userland though
    - Via a call to bpf(2) too
    - But not directly from a running eBPF program

eBPF evolution

- eBPF evolutions since 2014
  - Adding persistent eBPF programs
  - Adding a pseudo filesystem for eBPF (called bpf)
  - Adding different types of eBPF programs
    - Few related to network traffic
    - One to change socket(2) types
    - Few related to monitoring
  - Adding types of maps

eBPF evolution since 2014

- → Adding support to authz sysctl via eBPF
- → Ability to dump kernel structures
- → Allow sleepable eBPF programs
- → Ability to call some restricted kernel functions
- → Control scheduler decisions
- → Allow to loop (with still quite some restrictions)
- → ….

Running eBPF

How can one create and run such magical programs ?

How to run eBPF ?

- An eBPF program is setup in the kernel via bpf(2)
- It is checked, verified and then installed
- It can be referenced via an ID
- But a call to bpf(2) doesn’t run the program
- An ePBF program doesn’t run when userland asks for it to run
- An eBPF program is linked to an event, and is started from this event
  - Like classic BPF, when a socket receives data for example

The bpf(2) syscall

- ➔ All interactions with (e)bpf goes through the bpf(2) syscall
- ➔ The design intended to have only a single syscall for all operations
  - ◆ Like ioctl(2)
- ➔ First argument is the cmd:
  - ◆ Create, read elem, update elem, delete elem for maps
  - ◆ Load a eBPF program

The bpf(2) syscall

- A map to create must also have a type
  - BPF_MAP_TYPE_HASH
  - BPF_MAP_TYPE_ARRAY
  - BPF_MAP_TYPE_STACK_TRACE
  - …
- All map types don't behave the same way
- Read the documentation for specifics
- Same goes for eBPF programs
  - Different types for different usages
  - Changes in the verifier and capacities of your program

> Ok but why the fuss around eBPF ?

eBPF in action

Checkout why eBPF rhymes with observability

Linux kernel's own javascript

- eBPF is great because it adds dynamicity to the kernel
  - Could be seen as the equivalent of javascript, but for the kernel
- There's no intention of putting cats animation in the kernel though
- But instrumentality and observability is another subject
- Let's discover a few eBPF project

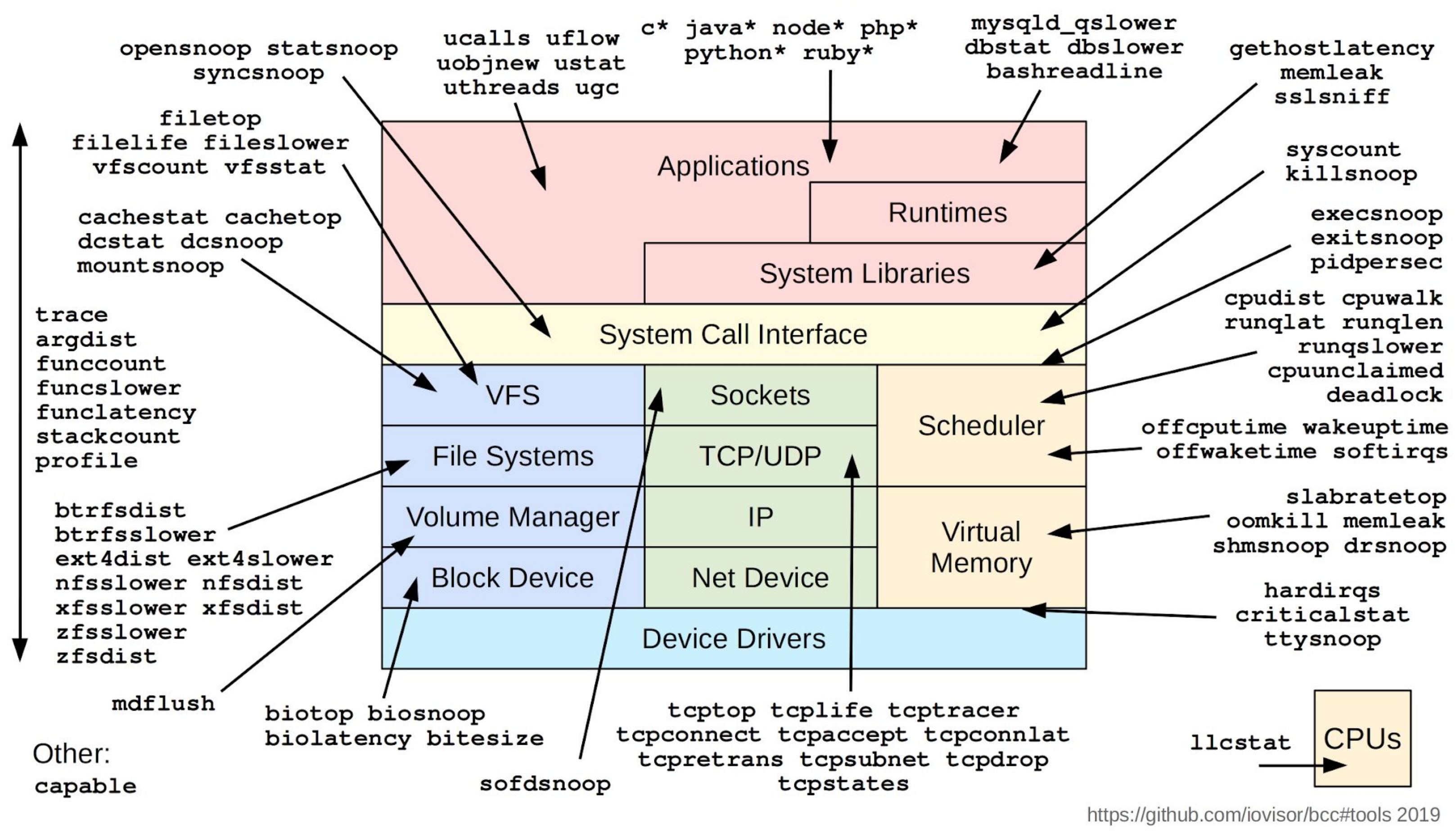
eBPF projects

- ➔ Netdata kernel collector
  - Collects metrics and allow monitoring on events that were inaccessible so far
  - Process-related, VFS, hardirqs, softirqs, shmem, sync-related syscalls, file access, mount, network-related, TCP-related internal functions calls, …

eBPF projects

- ➔ iovisor/bcc
  - Toolkit to manipulate eBPF easily: write eBPF programs in C-like language and compiled with LLVM, front-end for eBPF programs with python or lua
  - Comes with pre-defined tools to monitor, trace, snoop a machine

Linux bcc/BPF Tracing Tools



eBPF projects

- ➔ Cilium
  - Kubernetes-related projects for network
  - CNI to bring eBPF-aware networking to Kubernetes with:
    - Loadbalancing
    - Network policy L7 aware
    - ...
  - Hubble for observability in Kubernetes networking related stack
    - Metrics, tracing

eBPF projects

- ➔ Check a bigger, updated and more detailed list on https://ebpf.io/applications/
- ➔ Those projects look awesome and very promising
- ➔ The ability to expose metrics un-exposable otherwise is astonishing
- ➔ But how exactly is this possible ?
  - ◆ Let’s stop with the vagueness around observability and eBPF and let’s dig into implementation details

kprobes, uprobes, tracepoints, ...

They were there all along !

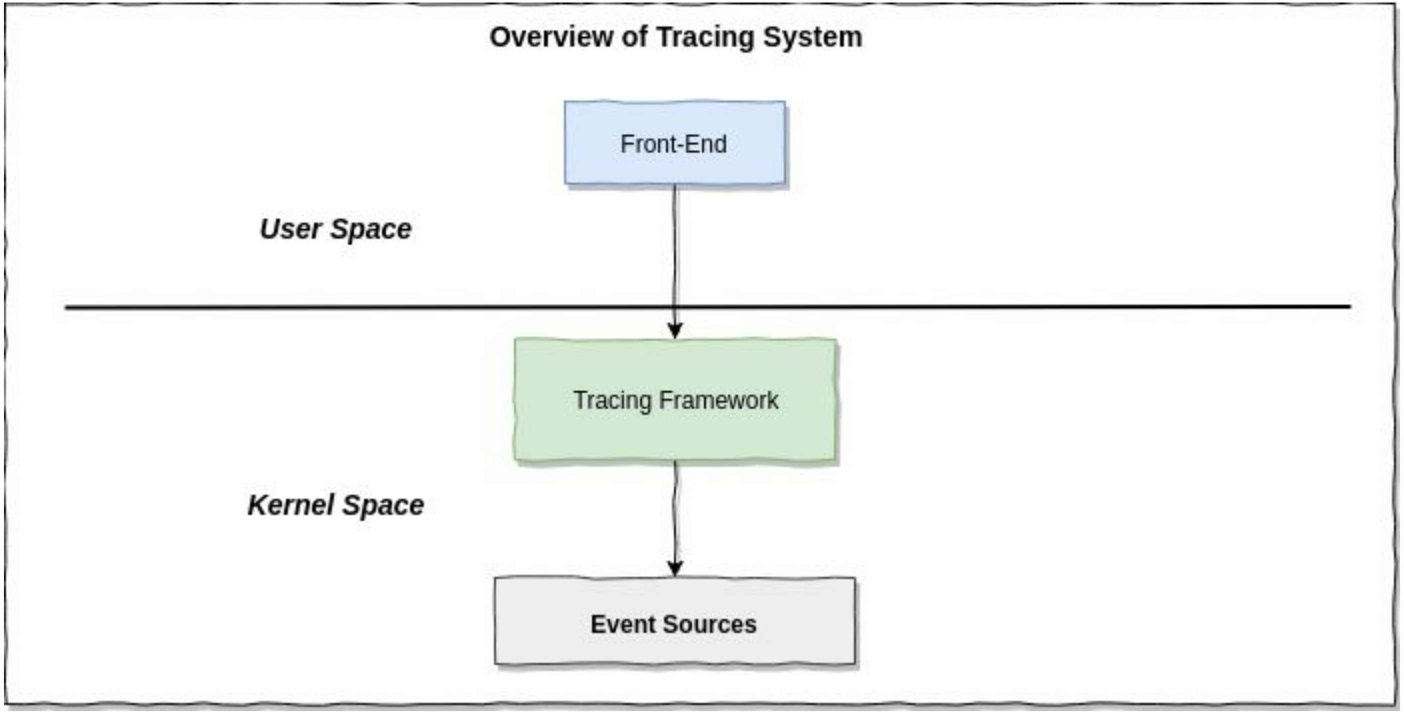
Probes and tracepoints

- ➔ Before “modern” monitoring like eBPF allow us to do, there were already concepts in the kernel to get events
  - ◆ From the early 2000’s
- ➔ It was mostly for instrumentation and debugging than observability
- ➔ Mostly aimed for kernel developers at first
- ➔ Then brought to more people

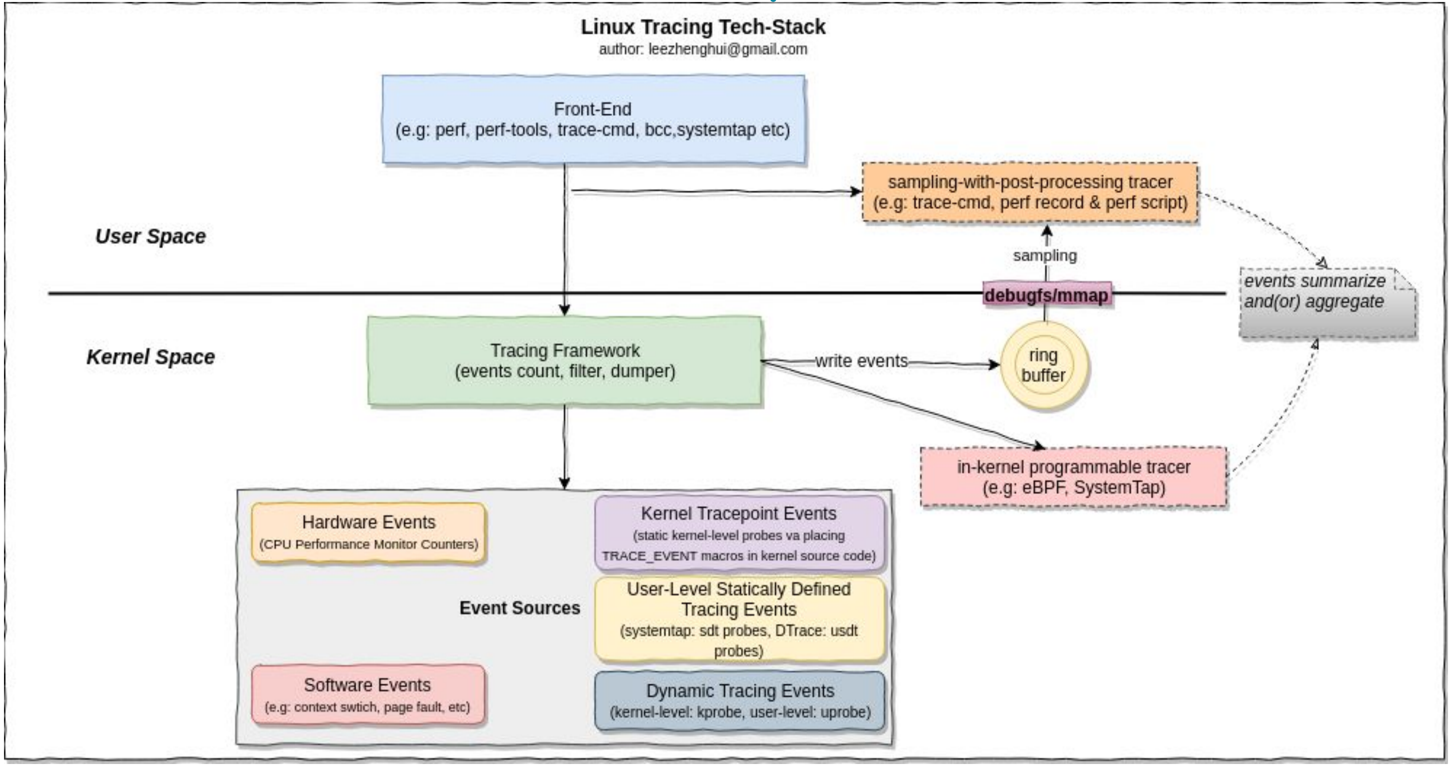
Probes and tracepoints

- ➔ There are 2 categories of event source in the kernel
  - ◆ Dynamically defined tracing points (Probes)
  - ◆ Statically defined tracing points (Tracepoints)
- ➔ Linux offers:
  - ◆ Tracepoints
  - ◆ Kprobes
- ➔ But also for userland:
  - ◆ Uprobes
  - ◆ (USDT)

Probes and tracepoints



Probes and tracepoints



Probes and tracepoints

- ➔ Probes and Tracepoints are to collect "data"
- ➔ The way there are used, called and how the data is then exposed depends on the tracing framework
- ➔ eBPF is, among other things, a tracing framework
- ➔ iovisor/bcc presented briefly is a front-end

Probes and tracepoints

- What is the difference between kprobes and tracepoints ?
- Tracepoints are defined statically
  - TRACE_EVENT macro in the kernel
- They had no overhead if disabled
  - Except for a small comparison
- Once enabled, notify with info observers

Probes and tracepoints

- What is the difference between kprobes and tracepoints ?
- kprobes are defined dynamically
- They don't require a "kprobe" event to be defined in the code
- You can compare it to a breakpoint with your debugger
- You can place it almost everywhere
  - Beginning and end of functions via k(ret)probe
- It replaces an instruction to be executed by an INT3
- The kprobe handler will check from where the trap comes from
- It will then report what is needed to the kprobe subscriber(s)

Tracing framework

A probe on its own is hardly usable

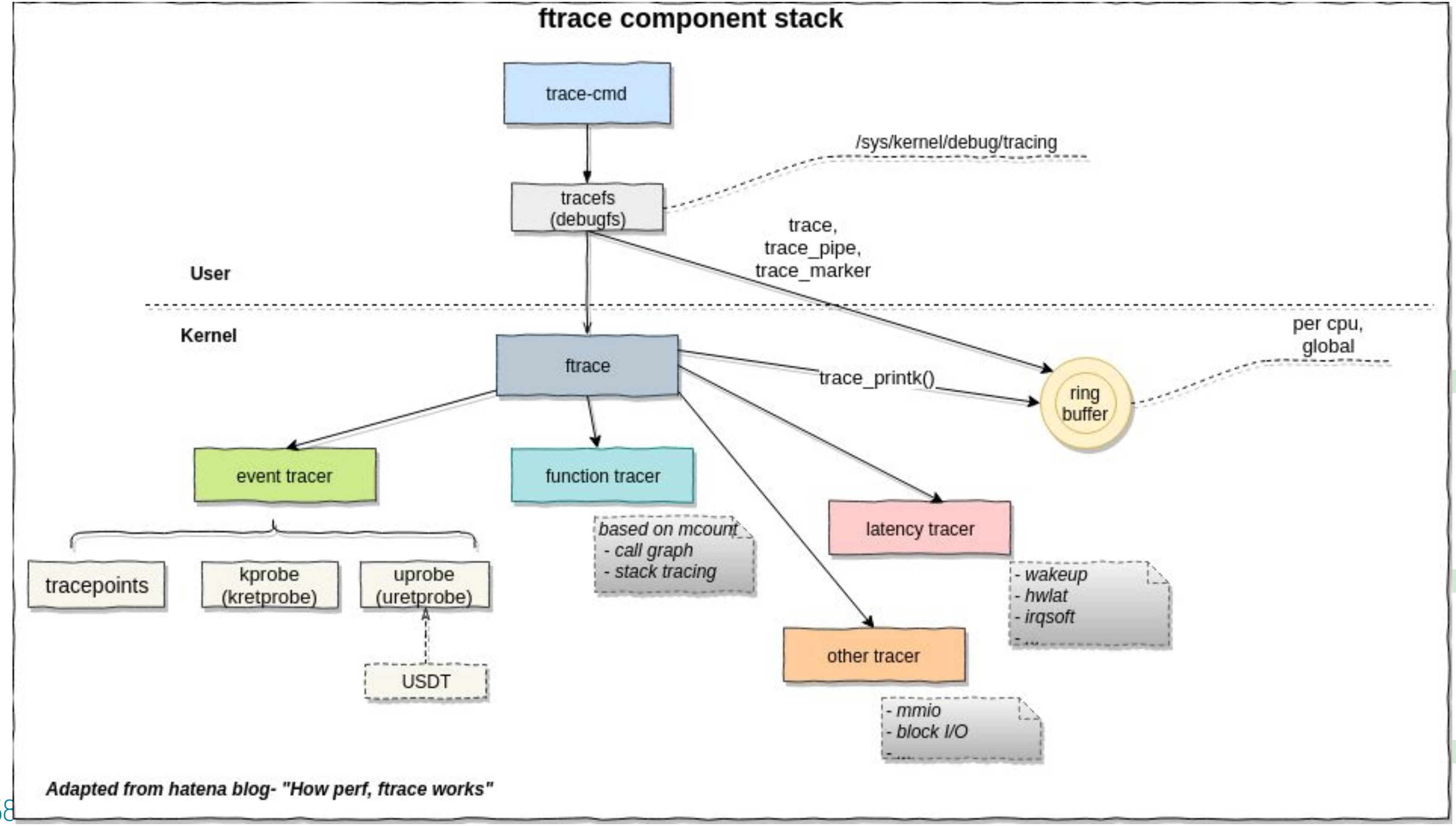
Tracing framework

- ➔ Probes and tracepoints reports data to a subscriber
- ➔ The subscriber is defined by the tracing framework used
- ➔ One example: eBPF
- ➔ eBPF allow you to define a small program and attach it to a probe/tracepoint
- ➔ Once the probe is fired, it calls your eBPF program
  - ◆ Argument to the probes are forwarded to your program
  - ◆ You can sometimes instrument them
  - ◆ Or do some logic and report things

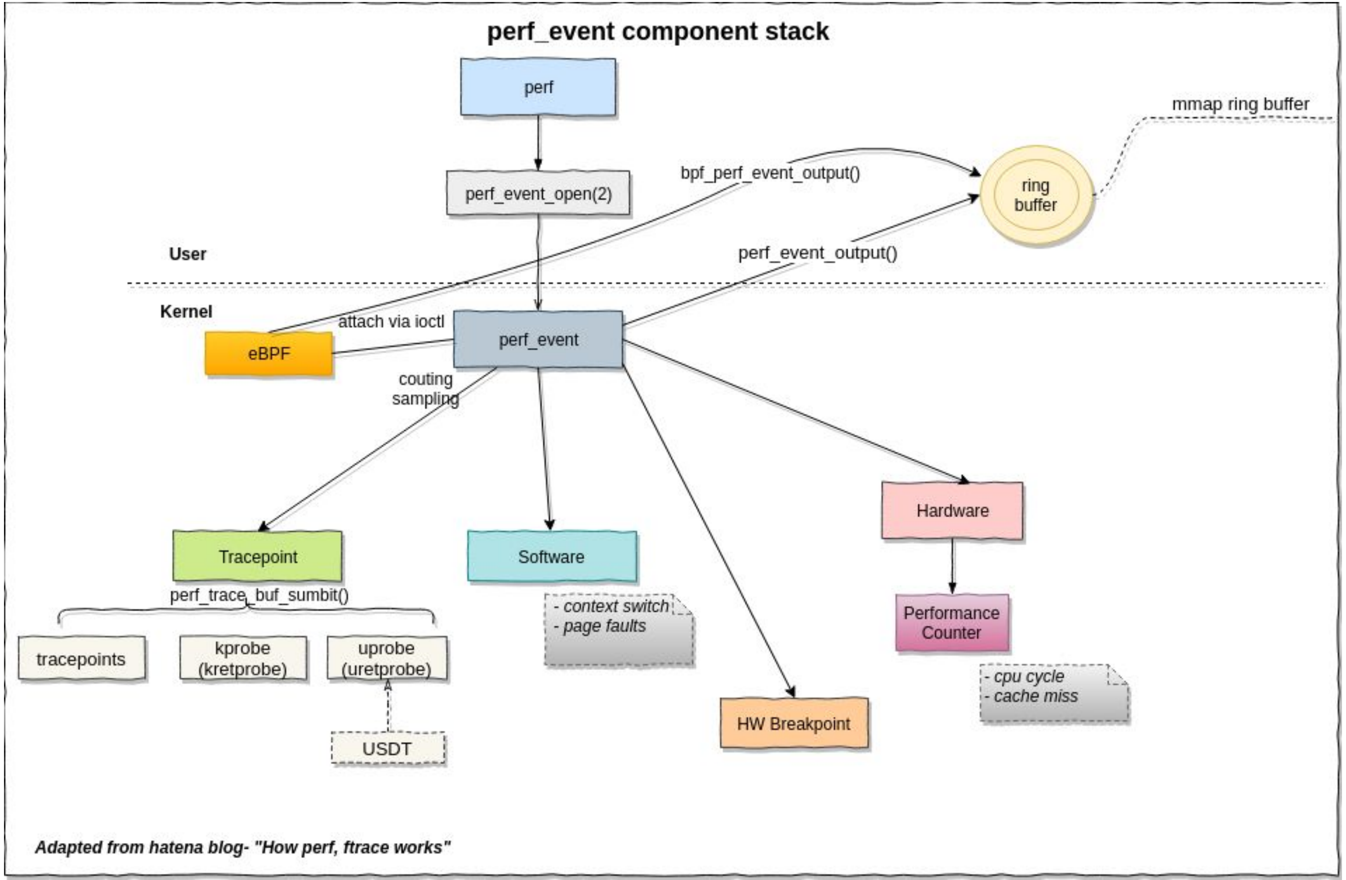
Tracing framework

- Only a limited amount of tracing frameworks are available in linux
- eBPF, ftrace and perf_event are the 3 main choices
  - There are also out-of-tree options (SystemTap, lttng, ...)

ftrace

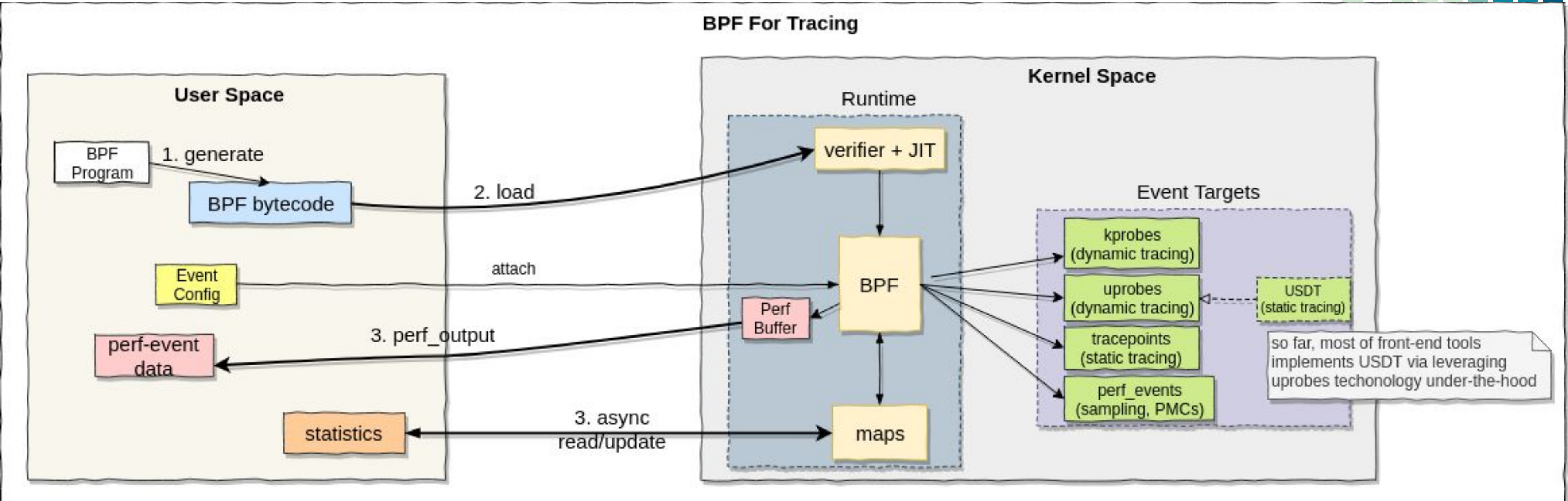

ftrace component stack
trace-cmd
/sys/kernel/debug/tracing
tracefs
(debugfs)
trace,
trace_pipe,
trace_marker
User
Kernel
ftrace
per cpu,
global
trace_printk()
ring
buffer
event tracer
function tracer
latency tracer
based on mcount
- call graph
- stack tracing
tracepoints
kprobe
(kretprobe)
uprobe
(uretprobe)
- wakeup
- hwlat
- irqsoft
USDT
other tracer
- mmio
- block I/O
Adapted from hatena blog- "How perf, ftrace works"

perf_event



perf_event component stack
perf
perf_event_open(2)
bpf_perf_event_output()
ring buffer
mmap ring buffer
User
perf_event_output()
Kernel
attach via ioctl
eBPF
perf_event
couting
sampling
Tracepoint
Software
Hardware
perf_trace_buf_sumbit()
- context switch
- page faults
Performance Counter
tracepoints
kprobe (kretprobe)
uprobe (uretprobe)
- cpu cycle
- cache miss
HW Breakpoint
USDT
Adapted from hatena blog- "How perf, ftrace works"

eBPF



Adapted from Brendan Gregg's blog post - "Linux Extended BPF (eBPF) Tracing Tools"

Frontends

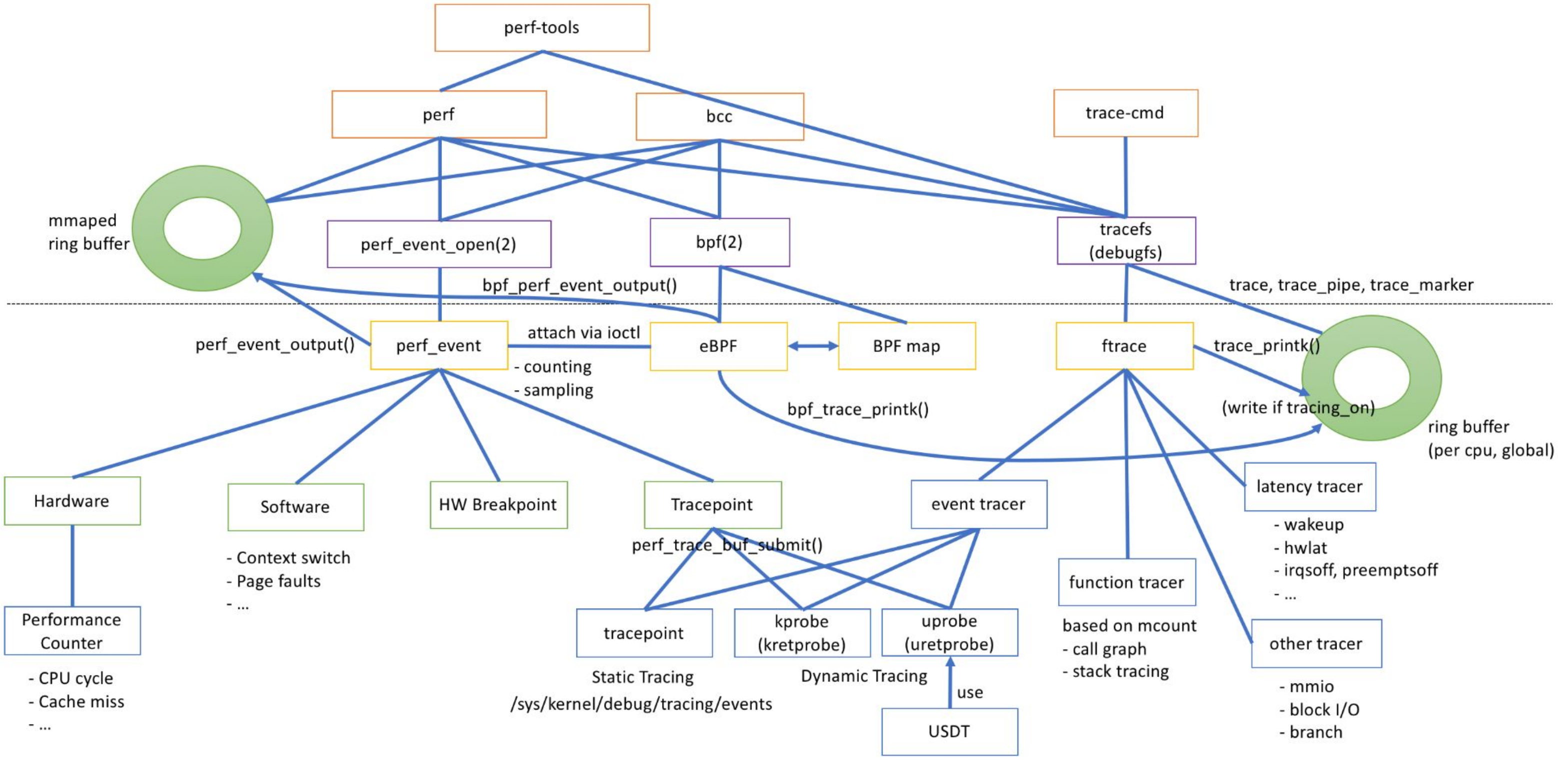
Let’s build some fancy tool on top of these

Frontends

- There are multiple frontends options for each framework
- The frontend usually is meant to leverage the framework easily
- Write human-readable code, and compile it in eBPF bytecode for example
- Provide awk-like scripts
- Essentially a simplification of the interfaces and syscalls
  - Sometimes shipped with a library in a given language

Frontends

- Quite some frontends can be named:
  - Perf (for perf framework and ftrace)
  - trace-cmd
  - Bcc
  - Bpftrace
  - LTTng
  - SystemTap
  -


perf-tools
perf
bcc
trace-cmd
mmaped
ring buffer
perf_event_open(2)
bpf(2)
tracefs
(debugfs)
bpf_perf_event_output()
trace, trace_pipe, trace_marker
perf_event_output()
perf_event
attach via ioctl
eBPF
BPF map
ftrace
trace_printk()
- counting
- sampling
bpf_trace_printk()
(write if tracing_on)
ring buffer
(per cpu, global)
Hardware
Software
HW Breakpoint
Tracepoint
event tracer
latency tracer
- wakeup
- hwlat
- irqsoff, preemptsoff
- ...
perf_trace_buf_submit()
- Context switch
- Page faults
- ...
Performance
Counter
tracepoint
kprobe
(kretprobe)
uprobe
(uretprobe)
function tracer
based on mcount
- call graph
- stack tracing
other tracer
- mmio
- block I/O
- branch
- CPU cycle
- Cache miss
- ...
Static Tracing
/sys/kernel/debug/tracing/events
Dynamic Tracing
use
USDT

nsswitch digression

Understanding glibc behavior as seen by strace(1)

nsswitch

- ➔ Unix world offers a few files to handle its system configuration
- ➔ Examples of those includes /etc/passwd, /etc/group, /etc/hosts, ...
- ➔ While those files works great and suit basic behavior, there are still a bit limited
- ➔ What if we wanted to handle servers' access for the employees of a company ?
  - There are hundreds of employees, thousands of systems
  - Handling each system individually is difficult and tedious

nsswitch

- How can we extend this behavior to use other kind of services in order to provide those information ?
- For example, connect to a database to get user information
  - LDAP is a famous protocol for this
- How to handle a DNS system a bit more clever than a simple /etc/hosts + /etc/resolv.conf ?
  - With cache
  - With per-interface domain resolution for example
  - ...

nsswitch

- GNU C Library allow us to extend and change the default behavior via a configuration file, /etc/nsswitch.conf
- nsswitch, for Name Service Switch in part of the glibc
  - And also introduced in other software due to its popularity
- The /etc/nsswitch.conf allow to change the configuration on how to find such Name Service information
- It has a pluggable approach, with shared libraries
  - Anyone can write a plugin to plug in the nsswitch system

nsswitch.conf(5)

- ➔ Default /etc/nsswitch.conf contains the basic configuration to use the default plugins for traditional UNIX config files

nsswitch.conf(5)

```
$ cat /etc/nsswitch.conf
# /etc/nsswitch.conf
#
# Example configuration of GNU Name Service Switch functionality.
# If you have the `glibc-doc-reference' and `info' packages installed, try:
# `info libc "Name Service Switch"' for information about this file.

passwd:         files
group:          files
shadow:         files
gshadow:        files

hosts:          files dns
networks:       files

protocols:      db files
services:       db files
ethers:         db files
rpc:            db files

netgroup:       nis
```

nsswitch.conf(5)

- Default /etc/nsswitch.conf contains the basic configuration to use the default plugins for traditional UNIX config files
- It has a simple format:
  - Name service: <plugin 1> <plugin 2> ...
    - There are some limited option to add on each plugin also
- Let’s check a few classic configurations

nsswitch.conf(5)

- passwd: files systemd ldap
- For the passwd name service, first is to check with the files plugin
- The files plugin is implemented via /usr/lib/libnss_files.so.2
- It implements the default UNIX behavior, by looking in /etc/passwd
- The next data source is implemented by libnss_systemd.so.2
  - It implements a connector to ask systemd(1) or some specific systemd service via a systemd API information
  - nss-systemd(8)

nsswitch.conf(5)

- passwd: files systemd ldap
- ldap is provided by nslcd and its libnss_ldap.so.2
- Used to query nslcd daemon which connect to remote configurable LDAP server and gets users, passwords and groups from
- Because of the multiple ways of finding passwd information (different name services), cat /etc/passwd is not enough
- Prefer using getent passwd

nscd digression

Understanding glibc behavior as seen by strace(1)

nscd

- ➔ On top of described behavior by nsswitch and glibc, another mechanism exists to provide cache for name service queries
- ➔ While /etc/passwd file for example is pretty much inexpensive to read, DNS queries or LDAP connection are expensive
- ➔ Having cache for them is great
- ➔ It's the role of nscd to provide such cache
  - ◆ Hence its name, Name Service Cache Daemon

nscd

- nscd as its name indicates is a daemon
  - It might not be installed on your machine, or not running
- It exposes a UNIX socket in /var/run/nscd/socket
- By default, the glibc connects to this socket automatically
  - Before contacting a name service source as provided by /etc/nsswitch.conf
- If the socket can't be opened, it … retries a second time
- If nscd is not running, or doesn't have the info in cache, it falls back to the default nsswitch mechanism

nscd

```
socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 3
connect(3, {sa_family=AF_UNIX, sun_path="/var/run/nscd/socket"}, 110) = 0
sendto(3, "\2\0\0\0\v\0\0\0\7\0\0\0passwd\0", 19, MSG_NOSIGNAL, NULL, 0) = 19
poll([{fd=3, events=POLLIN|POLLERR|POLLHUP}], 1, 5000) = 1 ([{fd=3,
revents=POLLIN|POLLHUP}])
recvmsg(3, {msg_name=NULL, msg_namelen=0, msg_iov=[{iov_base="passwd\0", iov_len=7},
{iov_base="\310O\3\0\0\0\0\0", iov_len=8}], msg_iovlen=2, msg_control=[{cmsg_len=20,
cmsg_level=SOL_SOCKET, cmsg_type=SCM_RIGHTS, cmsg_data=[4]}], msg_controllen=20,
msg_flags=MSG_CMSG_CLOEXEC}, MSG_CMSG_CLOEXEC) = 15
mmap(NULL, 217032, PROT_READ, MAP_SHARED, 4, 0) = 0x7fae19b4b000
getrandom("\x76\x85\x0c\xee\x32\xae\x07\x34", 8, GRND_NONBLOCK) = 8
brk(NULL)                                  = 0x55ae3363c000
brk(0x55ae3365d000)                        = 0x55ae3365d000
close(4)                                   = 0
close(3)                                   = 0
```

nscd

- ➔ In previous example
- ➔ nscd is answering
- ➔ Answers with a pointer to a shared memory to mount, that contains the asked database
- ➔ mmap(2) right under

```
socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 3
connect(3, {sa_family=AF_UNIX, sun_path="/var/run/nscd/socket"}, 110) = -1 ENOENT (No such file or directory)
close(3)                                = 0
socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 3
connect(3, {sa_family=AF_UNIX, sun_path="/var/run/nscd/socket"}, 110) = -1 ENOENT (No such file or directory)
close(3)                                = 0
getrandom("\x0e\x3e\xf0\xab\x82\xb2\xc4\x99", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55c619534000
brk(0x55c619555000)                     = 0x55c619555000
newfstatat(AT_FDCWD, "/etc/nsswitch.conf", {st_mode=S_IFREG|0644, st_size=359, ...}, 0) = 0
newfstatat(AT_FDCWD, "/", {st_mode=S_IFDIR|0755, st_size=4096, ...}, 0) = 0
openat(AT_FDCWD, "/etc/nsswitch.conf", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=359, ...}, AT_EMPTY_PATH) = 0
read(3, "# Name Service Switch configurat"..., 4096) = 359
read(3, "", 4096)                       = 0
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=359, ...}, AT_EMPTY_PATH) = 0
close(3)                                = 0
openat(AT_FDCWD, "/etc/passwd", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=1800, ...}, AT_EMPTY_PATH) = 0
lseek(3, 0, SEEK_SET)                   = 0
read(3, "root:x:0:0::/root:/usr/bin/zsh\nb"..., 4096) = 1800
close(3)                                = 0
```

Thanks !

Questions ?

Slides available on zarak.fr/

Contact: cyril@cri.epita.fr
zarak production#5492

Sources

lwn.net/Articles/904892/ - The ABI status of ELF hash tables [LWN.net]

lwn.net/Articles/330589/ - KSM tries again [LWN.net]

lwn.net/Articles/229096/ - SLUB: The unqueued slab allocator V6 [LWN.net]

lwn.net/Articles/288056/ - TASK_KILLABLE [LWN.net]

lwn.net/Articles/379748/ - Huge pages part 5: A deeper look at TLBs and costs [LWN.net]

lwn.net/Articles/317814/ - Taming the OOM killer [LWN.net]

lwn.net/Articles/83588/ - 2.6 swapping behavior [LWN.net]

lwn.net/Articles/82759/ - Kernel development [LWN.net]

lwn.net/Articles/793073/ - (hopefully) saner refcounting for mountpoint dentries [LWN.net]

lwn.net/Articles/330985/ - driver-core: devtmpfs - driver core maintained /dev tmpfs [LWN.net]

lwn.net/Articles/331818/ - The return of devfs [LWN.net]

lwn.net/Articles/612878/ - The BPF system call API, version 14 [LWN.net]

Sources

lwn.net/Articles/870269/ - Taming the BPF superpowers [LWN.net]

lwn.net/Articles/664688/ - Persistent BPF objects [LWN.net]

lwn.net/Articles/740157/ - A thorough introduction to eBPF [LWN.net]

lwn.net/Articles/779120/ - Concurrency management in BPF [LWN.net]

lwn.net/Articles/787856/ - BPF: what's good, what's coming, and what's needed [LWN.net]

lwn.net/Articles/785263/ - Managing sysctl knobs with BPF [LWN.net]

lwn.net/Articles/803890/ - Filesystem sandboxing with eBPF [LWN.net]

lwn.net/Articles/818714/ - Dumping kernel data structures with BPF [LWN.net]

lwn.net/Articles/825415/ - Sleepable BPF programs [LWN.net]

lwn.net/Articles/856005/ - Calling kernel functions from BPF [LWN.net]

lwn.net/Articles/873244/ - Controlling the CPU scheduler with BPF [LWN.net]

lwn.net/Articles/794934/ - Bounded loops in BPF for the 5.3 kernel [LWN.net]

lwn.net/Articles/132196/ - An introduction to KProbes [LWN.net]

Sources

lwn.net/Articles/346470/ - Fun with tracepoints [LWN.net]

lwn.net/Articles/379903/ - Using the TRACE_EVENT() macro (Part 1) [LWN.net]

github.com/brendangregg/perf-tools/blob/master/kernel/kprobe - perf-tools/kprobe at master · brendangregg/perf-tools

jvns.ca/blog/2017/07/05/linux-tracing-systems/ - Linux tracing systems & how they fit together

terenceli.github.io/%E6%8A%80%E6%9C%AF/2020/08/05/tracing-basic - Linux tracing - kprobe, uprobe and tracepoint

github.com/iovisor/bcc - iovisor/bcc: BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more

oreilly.com/library/view/understanding-the-linux/0596005652/ch04s07.html - 4.7. Softirqs and Tasklets - Understanding the Linux Kernel, 3rd Edition [Book]

docs.kernel.org/scheduler/index.html - Linux Scheduler — The Linux Kernel documentation

github.com/0xAX/linux-insides - GitHub - 0xAX/linux-insides: A little bit about a linux kernel

kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html - HugeTLB Pages — The Linux Kernel documentation

kernel.org/doc/html/latest/admin-guide/mm/idle_page_tracking.html - Idle Page Tracking — The Linux Kernel documentation

Sources

baeldung.com/linux/process-states - Linux Process States | Baeldung on Linux

haydenjames.io/what-is-iowait-and-linux-performance/ - What is iowait and how does it affect Linux performance?

unix.stackexchange.com/questions/16738/when-a-process-will-go-to-d-state - linux - When a process will go to 'D' state? - Unix & Linux Stack Exchange

stackoverflow.com/questions/71862781/how-to-make-a-process-to-enter-d-state - linux - How to make a process to enter D state? - Stack Overflow

unix.stackexchange.com/questions/539733/what-are-the-non-numeric-irqs-in-proc-interrupts - linux kernel - What are the non-numeric IRQs in /proc/interrupts? - Unix & Linux Stack Exchange

elixir.bootlin.com/linux/latest/source/arch/x86/include/asm/irq_vectors.h - irq_vectors.h - arch/x86/include/asm/irq_vectors.h - Linux source code (v5.19.2) - Bootlin

blog.dixo.net/irq.png

alexonlinux.com/smp-affinity-and-proper-interrupt-handling-in-linux - SMP affinity and proper interrupt handling in Linux - Alex on Linux

kernel.org/doc/html/latest/admin-guide/mm/ksm.html - Kernel Samepage Merging — The Linux Kernel documentation

Sources

kernel.org/doc/html/latest/RCU/whatisRCU.html - What is RCU? – "Read, Copy, Update" — The Linux Kernel documentation

zeph1912.github.io/notes_and_journal_repo/kernel_softirq.html - Softirq | Zephyr's study notes

unix.stackexchange.com/questions/591243/counting-the-number-of-issued-syscalls -

linux - Counting the number of issued syscalls - Unix & Linux Stack Exchange

rr-project.org/ - rr: lightweight recording & deterministic debugging

man7.org/linux/man-pages/man7/shm_overview.7.html - shm_overview(7) - Linux manual page

Linux-kvm.org - KVM

man7.org/linux/man-pages/man2/prctl.2.html - prctl(2) - Linux manual page

man7.org/linux/man-pages/man5/core.5.html - core(5) - Linux manual page

kernel.org/doc/html/latest/admin-guide/mm/index.html - Memory Management — The Linux Kernel documentation

kernel.org/doc/html/latest/admin-guide/mm/concepts.html - Concepts overview — The Linux Kernel documentation

llwn.net/Articles/306704/ - /dev/ksm: dynamic memory sharing [LWN.net]

kernel.org/doc/html/latest/admin-guide/mm/numa_memory_policy.html - NUMA Memory Policy — The Linux Kernel documentation

Sources

kernel.org/doc/html/latest/admin-guide/mm/numaperf.html - NUMA Locality — The Linux Kernel documentation

kernel.org/doc/html/latest/admin-guide/mm/pagemap.html - Examining Process Page Tables — The Linux Kernel documentation

kernel.org/doc/html/latest/admin-guide/mm/swap_numa.html - Automatically bind swap device to numa node — The Linux Kernel documentation

kernel.org/doc/html/latest/admin-guide/mm/zswap.html - zswap — The Linux Kernel documentation

stackoverflow.com/questions/7880784/what-is-rss-and-vsz-in-linux-memory-management - What is RSS and VSZ in Linux memory management - Stack Overflow

tothenew.com/blog/understanding-memory-utilization-in-linux/ - Understanding Memory Utilization in Linux | TO THE NEW Blog

man7.org/linux/man-pages/man7/user_namespaces.7.html - user_namespaces(7) - Linux manual page

redhat.com/sysadmin/dissecting-free-command - Dissecting the free command: What the Linux sysadmin needs to know | Enable Sysadmin

kernel.org/doc/gorman/html/understand/understand011.html - Slab Allocator

man7.org/linux/man-pages/man5/slabinfo.5.html - slabinfo(5) - Linux manual page

Sources

linuxize.com/post/free-command-in-linux/ - Free Command in Linux | Linuxize

man7.org/linux/man-pages/man5/procfs.5.html - proc(5) - Linux manual page

tecmint.com/linux-process-management/ - All You Need To Know About Processes in Linux [Comprehensive Guide]

devm.io/programming/linux-process-states-173858 - What are the process states in Unix/Linux?

stackoverflow.com/questions/67769737/check-if-the-process-in-in-running-state-or-runnable-state-in-linux - Check if the process in in RUNNING state or RUNNABLE state in Linux - Stack Overflow

elixir.bootlin.com/linux/latest/ident/task_struct - task_struct identifier - Linux source code (v5.19.5) - Bootlin

elixir.bootlin.com/linux/latest/source/include/linux/sched.h - sched.h - include/linux/sched.h - Linux source code (v5.19.5) - Bootlin

stackoverflow.com/questions/22101574/how-to-figure-out-if-process-is-really-running-or-waiting-to-run-on-linux - how to figure out if process is really running or waiting to run on Linux? - Stack Overflow

eklitzke.org/uninterruptible-sleep - Uninterruptible Sleep

elixir.bootlin.com/linux/latest/A/ident/TASK_UNINTERRUPTIBLE - TASK_UNINTERRUPTIBLE identifier - Linux source code (v5.19.5) - Bootlin

opensource.com/article/19/2/fair-scheduling-linux - CFS: Completely fair process scheduling in Linux

Sources

docs.kernel.org/scheduler/sched-design-CFS.html - CFS Scheduler — The Linux Kernel documentation

0xax.gitbooks.io/linux-insides/content/Interrupts/linux-interrupts-9.html - Softirq, Tasklets and Workqueues · Linux Inside

sites.google.com/site/masumzh/articles/x86-architecture-basics/interrupts-faults-and-traps - Masum Z Hasan, PhD - X86 Architecture basics: Interrupts, Faults and Traps and IO

wiki.osdev.org/Interrupts - Interrupts - OSDev Wiki

en.wikibooks.org/wiki/X86_Assembly/Programmable_Interrupt_Controller - x86 Assembly/Programmable Interrupt Controller - Wikibooks, open books for an open world

intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf - 64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

man7.org/linux/man-pages/man2/madvise.2.html - madvise(2) - Linux manual page

kernel.org/doc/html/latest/admin-guide/mm/transhuge.html - Transparent Hugepage Support — The Linux Kernel documentation

docs.kernel.org/admin-guide/mm/index.html - Memory Management — The Linux Kernel documentation

kernel.org/doc/html/latest/filesystems/vfs.html - Overview of the Linux Virtual File System — The Linux Kernel documentation

Sources

books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch12lev1sec7.html - The Dentry Object
developer.ibm.com/tutorials/l-completely-fair-scheduler/ - Inside the Linux 2.6 Completely Fair Scheduler