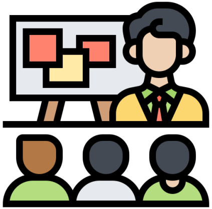
Linux debugging, profiling and tracing training

Latest update: August 26, 2024.

Document updates and training details:
https://bootlin.com/training/debugging

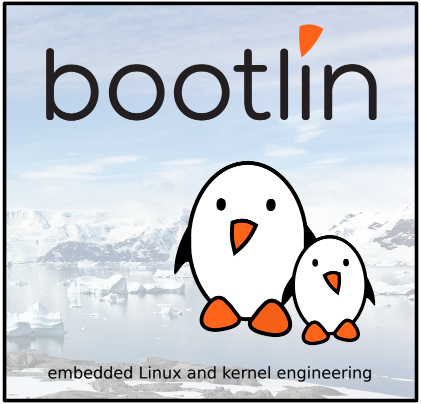
Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com

- These slides are the training materials for Bootlin's *Linux debugging, profiling and tracing* training course.
- If you are interested in following this course with an experienced Bootlin trainer, we offer:
  - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
  - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
  - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- Details and registrations: https://bootlin.com/training/debugging
- Contact: training@bootlin.com



Icon by Eucalyp, Flaticon

About Bootlin



Corrections, suggestions, contributions and translations are welcome!

Bootlin introduction

- Engineering company
  - In business since 2004
  - Before 2018: *Free Electrons*
- Team based in France and Italy
- Serving **customers worldwide**
- **Highly focused and recognized expertise**
  - Embedded Linux
  - Linux kernel
  - Embedded Linux build systems
- **Strong open-source** contributor
- Activities
  - **Engineering** services
  - **Training** courses
- https://bootlin.com

Bootlin engineering services

Bootloader / firmware development

U-Boot, Barebox, OP-TEE, TF-A, .../

Linux kernel porting and driver development

Linux BSP development, maintenance and upgrade

Embedded Linux build systems

Yocto, OpenEmbedded, Buildroot, ...

Embedded Linux integration

Boot time, real-time, security, multimedia, networking

Open-source upstreaming

Get code integrated in upstream Linux, U-Boot, Yocto, Buildroot, ...

Bootlin training courses

Embedded Linux system development On-site: 4 or 5 days Online: 7 * 4 hours	Linux kernel driver development On-site: 5 days Online: 7 * 4 hours	Yocto Project system development On-site: 3 days Online: 4 * 4 hours	Buildroot system development On-site: 3 days Online: 5 * 4 hours
Understanding the Linux graphics stack On-site: 2 days Online: 4 * 4 hours	Embedded Linux audio On-site: 2 days Online: 4 * 4 hours	Real-Time Linux with PREEMPT_RT On-site: 2 days Online: 3 * 4 hours	Linux debugging, tracing, profiling and performance analysis On-site: 3 days Online: 4 * 4 hours

All our training materials are freely available
under a free documentation license (CC-BY-SA 3.0)
See https://bootlin.com/training/

Bootlin, an open-source contributor

- Strong contributor to the **Linux** kernel
  - In the top 30 of companies contributing to Linux worldwide
  - Contributions in most areas related to hardware support
  - Several engineers maintainers of subsystems/platforms
  - 8000 patches contributed
  - https://bootlin.com/community/contributions/kernel-contributions/
- Contributor to **Yocto Project**
  - Maintainer of the official documentation
  - Core participant to the QA effort
- Contributor to **Buildroot**
  - Co-maintainer
  - 5000 patches contributed
- Significant contributions to U-Boot, OP-TEE, Barebox, etc.
- Fully **open-source training materials**

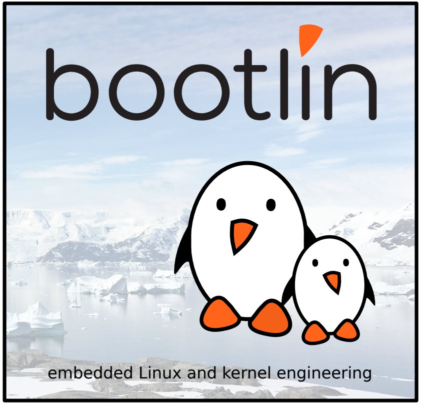
Bootlin on-line resources

- Website with a technical blog:
  https://bootlin.com
- Engineering services:
  https://bootlin.com/engineering
- Training services:
  https://bootlin.com/training
- Twitter:
  https://twitter.com/bootlincom
- LinkedIn:
  https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line:
  https://elixir.bootlin.com

Icon by Freepik, Flaticon

Generic course information

Corrections, suggestions, contributions and translations are welcome!



Supported hardware

Discovery Kits from STMicroelectronics: STM32MP157A-DK1, STM32MP157D-DK1, STM32MP157C-DK2 or STM32MP157F-DK2

- STM32MP157 (Dual Cortex-A7 + Cortex-M4) CPU from STMicroelectronics
- 512 MB DDR3L RAM
- Gigabit Ethernet port
- 4 USB 2.0 host ports, 1 USB-C OTG port
- 1 Micro SD slot
- On-board ST-LINK/V2-1 debugger
- Misc: buttons, LEDs, audio codec
- LCD touchscreen (DK2 only)

DK1 Discovery Kit

Board and CPU documentation, design files, software: A-DK1, D-DK1, C-DK2, F-DK2

Shopping list: hardware for this course

- STMicroelectronics STM32MP157D-DK1 Discovery kit
- USB-C cable for the power supply
- USB-A to micro B cable for the serial console
- RJ45 cable for networking
- A micro SD card with at least 128 MB of capacity

Training quiz and certificate

- You have been given a quiz to test your knowledge on the topics covered by the course. That's not too late to take it if you haven't done it yet!
- At the end of the course, we will submit this quiz to you again. That time, you will see the correct answers.
- It allows Bootlin to assess your progress thanks to the course. That's also a kind of challenge, to look for clues throughout the lectures and labs / demos, as all the answers are in the course!
- Another reason is that we only give training certificates to people who achieve at least a 50% score in the final quiz **and** who attended all the sessions.

Participate!

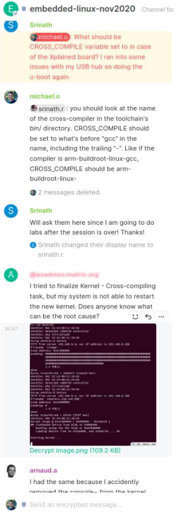
During the lectures...

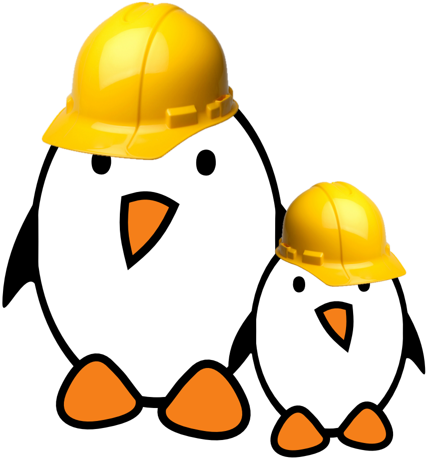
- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- Don't hesitate to share your experience too, for example to compare Linux with other operating systems you know.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- In on-line sessions
  - Please always keep your camera on!
  - Also make sure your name is properly filled.
  - You can also use the "Raise your hand" button when you wish to ask a question but don't want to interrupt.
- All this helps the trainer to engage with participants, see when something needs clarifying and make the session more interactive, enjoyable and useful for everyone.

Collaborate!

As in the Free Software and Open Source community, collaboration between participants is valuable in this training session:

- Use the dedicated Matrix channel for this session to add questions.
- If your session offers practical labs, you can also report issues, share screenshots and command output there.
- Don't hesitate to share your own answers and to help others especially when the trainer is unavailable.
- The Matrix channel is also a good place to ask questions outside of training hours, and after the course is over.





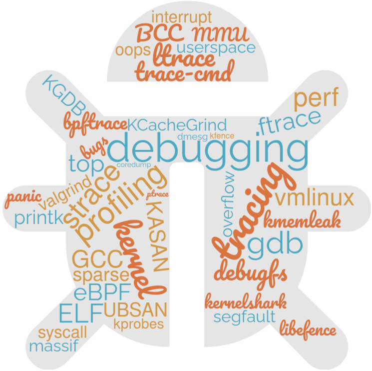
Prepare your lab environment

- Download and extract the lab archive

Debugging, Profiling, Tracing

Debugging, Profiling, Tracing

- Debugging, profiling and tracing are often used for development purposes
- All of these methods have different goals which aim at perfecting the software that is being developed
- Requires some knowledge about underlying mechanisms to correctly identify and fix bugs



Debugging

- Finding and fixing bugs that might exist in your software/system
- Use of various tools and methods to achieve that
  - Interactive debugging (With GDB for instance)
  - Postmortem analysis (Using coredump for instance)
  - Control flow analysis (With tracing tools)
  - Testing (Targeted tests)
- Most commonly done through debuggers in development environment
- Generally intrusive, allowing to pause and resume execution

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"
- Brian Kernighan

Profiling

- Analysis at program runtime to assist performance optimizations
- Often achieved by sampling counters during execution
- Uses specific tools, libraries and operating system features to measure performance.
  - Using *perf*, *OProfile* for instance.
- First step consists in gathering data from program execution
  - Function call count, memory usage, CPU load, cache miss, etc
- Then extracting meaningful information from these data and modify the program to optimize it

Tracing

- Following the execution flow of an application to understand the bottlenecks and problems.
- Achieved by instrumenting code either at compile time or runtime.
  - Can be done using specific tracers such as *LTTng*, *trace-cmd*, *SystemTap* etc
- Goes from the user space called functions up to the kernel ones
- Allows to identify functions and values that are used while application executes
- Often works by recording traces during runtime and then visualizing data.
  - Implies a large amount of recorded data since the complete execution trace is recorded
  - Often bigger overhead than profiling.
- Can also be used for debugging purpose since data can be extracted with tracepoints.

Linux Application Stack

User/Kernel mode

User/Kernel mode

- User mode vs Kernel mode are often used to refer to the privilege level of execution.
- This mode actually refers to the processor execution mode which is a hardware mode.
  - Might be named differently between architectures but the goal is the same
- Allows the kernel to control the full processor state (handle exceptions, MMU, etc) whereas the userspace can only do basic control and execute under the kernel supervision.

Introduction to Processes and Threads

Processes and Threads (1/2)

- A process is a group of resources that are allocated by the operating to allow the execution of a program.
  - Memory regions, threads, file descriptors, etc.
- A process is identified by a PID (**P**rocess **ID**) and all the information that are specific to this process are exposed in `/proc/<pid>`.
  - A special file named `/proc/self` accessible by the process points to the proc folder associated to it.
- When starting a process, it initially has one execution thread that is represented by a `struct task_struct` and that can be scheduled.
  - A process is represented in the kernel by a thread associated to multiple resources.

Processes and Threads (2/2)

- Threads are independent execution units that are sharing common resources inside a process.
  - Same address space, file descriptors, etc.
- A new process is created using the fork() system call (`man 2 fork`) and a new thread is created using `pthread_create()` (`man 3 pthread_create`).
  - Internally, both will call `clone()` with different flags
- At any moment, only one task is executing on a CPU core and is accessible using `get_current()` function (defined by architecture and often stored in a register).
- Each CPU core will execute a different task.
- A task can only be executing on one core at a time.

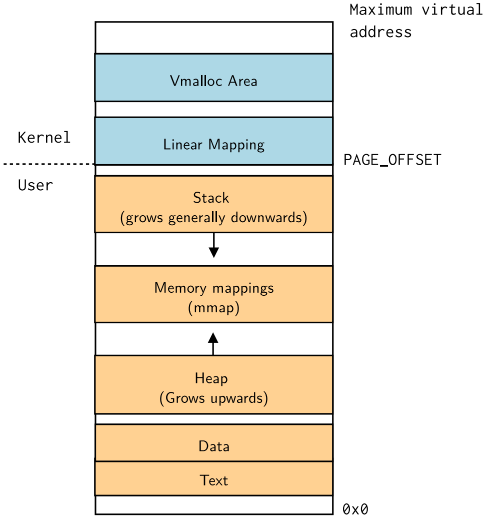
MMU and memory management

MMU and memory management

- Under Linux Kernel (when using `CONFIG_MMU=y`), all addresses that are accessed by the CPU are virtual
- The Memory Management Unit allows to map these virtual addresses to physical memory (either RAM or IO)
- All these mappings are inserted into the page table that is used by the MMU hardware to translate the CPU access from virtual to physical addresses
- The MMU allows to restrict access to the page mappings via some attributes
  - No Execute, Writable, Readable bits, Privileged/User bit, cacheability
- The MMU base unit for mappings is called a page
- Page size is fixed and depends on the architecture/kernel configuration.

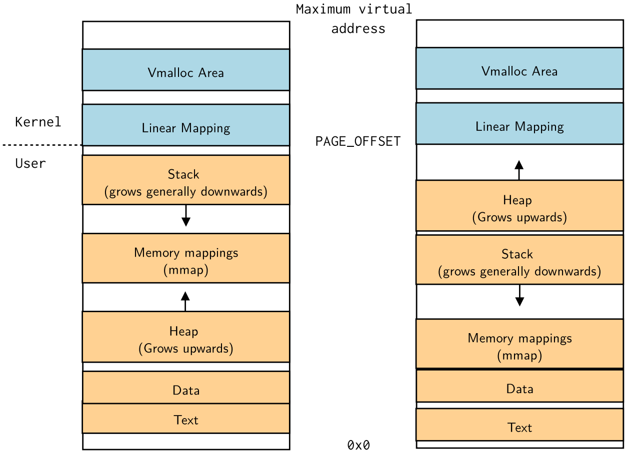
Userspace/Kernel memory layout

- Each process has its own set of virtual memory areas (`mm` field of `struct task_struct`).
- Also have their own page table
  - But share the same kernel mappings
- By default, all user mapping addresses are randomized to minimize attack surface (base of heap, stack, text, data, etc).
  - **A**ddress **S**pace **L**ayout **R**andomization
  - Can be disabled using `norandmaps` command line parameter



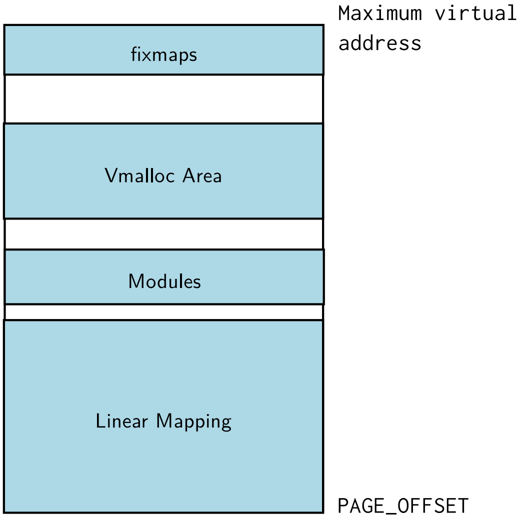
Userspace/Kernel memory layout

Multiple processes have different user memory spaces



Kernel memory map

- The kernel has it own memory mapping.
- Linear mapping is setup at kernel startup by inserting all the entries in the kernel init page table.
- Multiple areas are identified and their location differs between the architectures.
- **K**ernel **A**ddress **S**pace **L**ayout **R**andomization also allows to randomize kernel address space layout.
  - Can be disabled using `nokaslr` command line parameter



Userspace memory segments

- When starting a process, the kernel sets up several *Virtual Memory Areas* (VMA), backed by `struct vm_area_struct`, with different execution attributes.
- VMA are actually memory zones that are mapped with specific attributes (R/W/X).
- A segmentation fault happens when a program tries to access an unmapped area or a mapped area with an access mode that is not allowed.
  - Writing data in a read-only segment
  - Executing data from a non-executable segment
- New memory zones can be created using `mmap()` (`man 2 mmap`)
- Per application mappings are visible in */proc/<pid>/maps*

```
7f1855b2a000-7f1855b2c000 rw-p 00030000 103:01 3408650  ld-2.33.so
7ffc01625000-7ffc01646000 rw-p 00000000 00:00 0        [stack]
7ffc016e5000-7ffc016e9000 r--p 00000000 00:00 0        [vvar]
7ffc016e9000-7ffc016eb000 r-xp 00000000 00:00 0        [vdso]
```

Userspace memory types

	Private	Shared
Anonymous	- stack - malloc() - brk()/sbrk() - mmap(PRIVATE,ANON)	- POSIX shm* - mmap(SHARED, ANON)
File-backed	- mmap(PRIVATE,fd) - program (code) - shared libraries	- mmap(SHARED, fd)

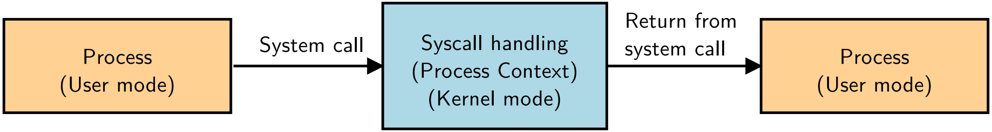
Terms for memory in Linux tools

- When using Linux tools, four terms are used to describe memory:
  - *VSS/VSZ*: Virtual Set Size (Virtual memory size, shared libraries included).
  - *RSS*: Resident Set Size (Total physical memory usage, shared libraries included).
  - *PSS*: Proportional Set Size (Actual physical memory used, divided by the number of times it has been mapped).
  - *USS*: Unique Set Size (Physical memory occupied by the process, shared mappings memory excluded).
- VSS >= RSS >= PSS >= USS.

The process context

Process context

- The *process context* can be seen as the content of the CPU registers associated to a process: execution register, stack register...
- This context also designates an execution state and allows to sleep inside kernel mode.
- A process that is executing in process context can be preempted.
- While executing in such context, the current process `struct task_struct` can be accessed using `get_current()`.



Scheduling

Scheduling

- The scheduler can be invoked for various reasons
  - On a periodic tick caused by interrupt (`HZ`)
  - On a programmed interrupt on tickless systems (`CONFIG_NO_HZ=y`)
  - Voluntarily by calling `schedule()` in code
  - Implicitly by calling functions that can sleep (blocking operations such as `kmalloc()`, `wait_event()`).
- When entering the schedule function, the scheduler will elect a new `struct task_struct` to run and will eventually call the `switch_to()` macro.
- `switch_to()` is defined by architecture code and it will save the current task process context and restore the one of the next task to be run while setting the new current task running.

The Linux Kernel Scheduler

- The Linux Kernel Scheduler is a key piece in having a real-time behaviour
- It is in charge of deciding which **runnable** task gets executed
- It also elects on which CPU the task runs, and is tightly coupled to CPUidle and CPUFreq
- It schedules both **userspace** tasks and **kernel** tasks
- Each task is assigned one **scheduling class** or **policy**
- The class determines the algorithm used to elect each task
- Tasks with different scheduling classes can coexist on the system

Non-Realtime Scheduling Classes

There are 3 **Non-RealTime** classes

- SCHED_OTHER: The default policy, using a time-sharing algorithm
- SCHED_BATCH: Similar to SCHED_OTHER, but designed for CPU-intensive loads that affect the wakeup time
- SCHED_IDLE: Very low priority class. Tasks with this policy will run only if nothing else needs to run.
- SCHED_OTHER and SCHED_BATCH use the **nice** value to increase or decrease their scheduling frequency
  - A higher nice value means that the tasks gets scheduled **less** often

Realtime Scheduling Classes

There are 3 **Realtime** classes

- Runnable tasks will preempt any other lower-priority task
- `SCHED_FIFO`: All tasks with the same priority are scheduled **First in, First out**
- `SCHED_RR`: Similar to `SCHED_FIFO` but with a time-sharing round-robin between tasks with the same priority
- Both `SCHED_FIFO` and `SCHED_RR` can be assigned a priority between 1 and 99
- `SCHED_DEADLINE`: For tasks doing recurrent jobs, extra attributes are attached to a task
  - A computation time, which represents the time the task needs to complete a job
  - A deadline, which is the maximum allowable time to compute the job
  - A period, during which only one job can occur
- Using one of these classes is necessary but not sufficient to get real-time behavior

Changing the Scheduling Class

- The Scheduling Class is set per-task, and defaults to `SCHED_OTHER`
- The `man 2 sched_setscheduler` syscall allows changing the class of a task
- The `chrt` tool uses it to allow changing the class of a running task:
  - `chrt -f/-b/-o/-r/-d -p PRIO PID`
- It can also be used to launch a new program with a dedicated class:
  - `chrt -f/-b/-o/-r/-d PRIO CMD`
- To show the current class and priority:
  - `chrt -p PID`
- New processes will inherit the class of their parent except if the `SCHED_RESET_ON_FORK` flag is set with `man 2 sched_setscheduler`
- See `man 7 sched` for more information

Context switching

Context switching

- Context switching is the action of changing the execution mode of the processor (Kernel $\leftrightarrow$ User).
  - Explicitly by executing system calls instructions (synchronous request to the kernel from user mode).
  - Implicitly when receiving exceptions (MMU fault, interrupts, breakpoints, etc).
- This state change will end up in a kernel entrypoint (often call vectors) that will execute necessary code to setup a correct state for kernel mode execution.
- The kernel takes care of saving registers, switching to the kernel stack and potentially other things depending on the architecture.
  - Does not use the user stack but a specific kernel fixed size stack for security purposes.

Exceptions

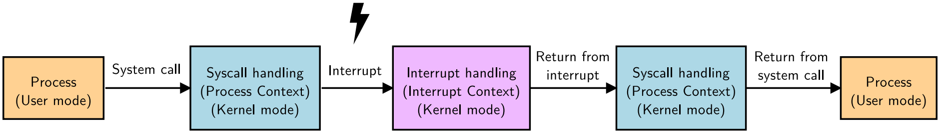
- Exceptions designate the kind of events that will trigger a CPU execution mode change to handle the exception.
- Two main types of exceptions exist: synchronous and asynchronous.
  - Asynchronous exceptions when a fault happens while executing (MMU, bus abort, etc) or when an interrupt is received (either software or hardware).
  - Synchronous when executing some specific instructions (breakpoint, syscall, etc)
- When such exception is triggered, the processor will jump to the exception vector and execute the code that was setup for this exception.

Interrupts

- Interrupts are asynchronous signals that are generated by the hardware peripherals.
  - Can also be synchronous when generated using a specific instruction (**I**nter **P**rocessor **I**nterrupts for instance).
- When receiving an interrupt, the CPU will change its execution mode by jumping to a specific vector and switching to kernel mode to handle the interrupt.
- When multiple CPUs (cores) are present, interrupts are often directed to a single core.
- This is called "IRQ affinity" and it allows to control the IRQ load for each CPU
  - See `core-api/irq/irq-affinity` and man irqbalance(1)

Interrupts

- While handling the interrupts, the kernel is executing in a specific context named *interrupt context*.
- This context does not have access to userspace and should not use `get_current()`.
- Depending on the architecture, might use an IRQ stack.
- Interrupts are disabled (no nested interrupt support)!



System Calls (1/2)

- A system call allows the user space to request services from the kernel by executing a special instruction that will switch to the kernel mode (`man 2 syscall`)
  - When executing functions provided by the libc (`read()`, `write()`, etc), they often end up executing a system call.
- System calls are identified by a numeric identifier that is passed via the registers.
  - The kernel exports some defines (in `unistd.h`) that are named __NR_<sycall> and defines the syscall identifiers.

```
#define __NR_read 63
#define __NR_write 64
```

System Calls (2/2)

- The kernel holds a table of function pointers which matches these identifiers and will invoke the correct handler after checking the validity of the syscall.
- System call parameters are passed via registers (up to 6).
- When executing this instruction the CPU will change its execution state and switch to the kernel mode.
- Each architecture uses a specific hardware mechanism (`man 2 syscall`)

```
mov w8, #__NR_getpid
svc #0
tstne x0, x1
```

Kernel execution contexts

Kernel execution contexts

- The kernel runs code in various contexts depending on the event it is handling.
- Might have interrupts disabled, specific stack, etc.

Kernel threads

- Kernel threads (kthreads) are a special kind of `struct task_struct` that do not have any user resources associated (`mm == NULL`).
- These processes are cloned from the `kthreadd` process and can be created using `kthread_create()`.
- Kernel threads are scheduled and are allowed to sleep much like a process executing in process context.
- Kernel threads are visible and their names are displayed between brackets under *ps*:

```
$ ps --ppid 2 -p 2 -o uname,pid,ppid,cmd,cls
USER         PID    PPID CMD                          CLS
root           2       0 [kthreadd]                    TS
root           3       2 [rcu_gp]                      TS
root           4       2 [rcu_par_gp]                  TS
root           5       2 [netns]                       TS
root           7       2 [kworker/0:0H-events_highpr   TS
root          10       2 [mm_percpu_wq]                TS
root          11       2 [rcu_tasks_kthread]           TS
```

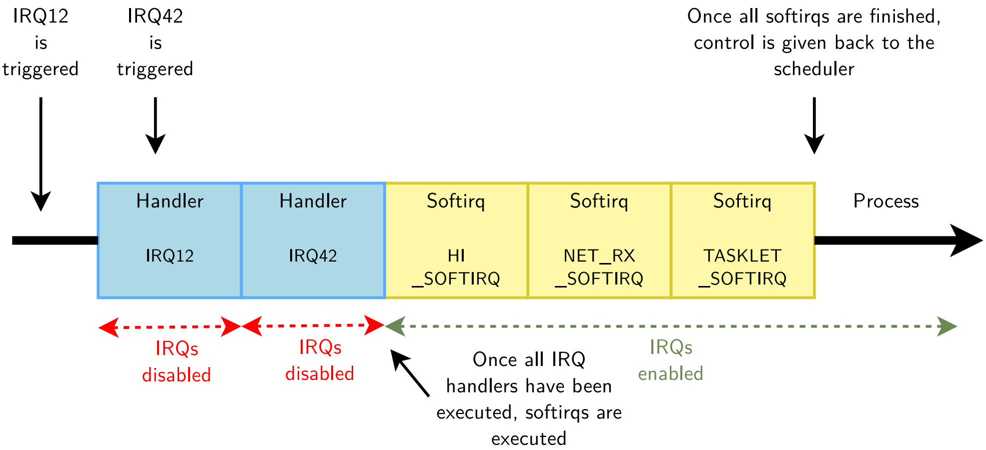
Workqueues

- Workqueues allows to schedule some work to be executed at some point in the future
- Workqueues are executing the work functions in kernel threads.
  - Allows to sleep while executing the deferred work.
  - Interrupts are enabled while executing
- Work can be executed either in dedicated work queues or in the default workqueue that is shared by multiple users.

softirq

- SoftIRQs is a specific kernel mecanism that is executed in software interrupt context.
- Allows to execute code that needs to be deferred after interrupt handling but needs low latency.
  - Executed right after hardware IRQ have been handled in interrupt context.
  - Same context as executing interrupt handler so sleeping is not allowed.
- Anyone wanting to run some code in softirq context should likely not create its own but prefer some entities implemented on top of it. There are for example tasklets, and the BH workqueues (Bottom Half workqueues) which aim to replace tasklets since 6.9.

Interrupts & Softirqs

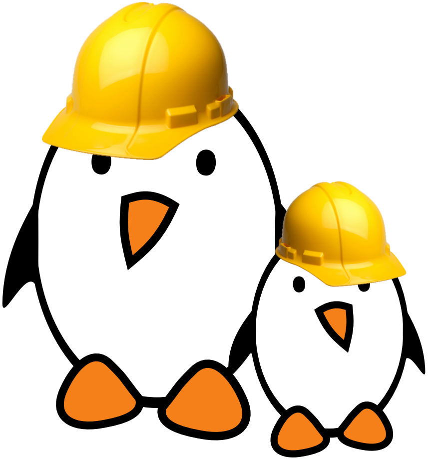


Threaded interrupts

- Threaded interrupts are a mecanism that allows to handle the interrupt using a hard IRQ handler and a threaded IRQ handler.
- A threaded IRQ handler will allow to execute work that can potentially sleep in a kthread.
- One kthread is created for each interrupt line that was requested as a threaded IRQ.
  - *kthread* is named `irq/<irq>-<name>` and can be seen using *ps*.

Allocations and context

- Allocating memory in the kernel can be done using multiple functions:
  - `void *kmalloc(size_t size, gfp_t gfp_mask);`
  - `void *kzalloc(size_t size, gfp_t gfp_mask);`
  - `unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)`
- All allocation functions take a `gfp_mask` parameter which allows to designate the kind of memory that is needed.
  - `GFP_KERNEL`: Normal allocation, can sleep while allocating memory (can not be used in interrupt context).
  - `GFP_ATOMIC`: Atomic allocation, won't sleep while allocating data.



Prepare the STM32MP157D board

- Build an image using Buildroot
- Connect the board
- Load the kernel from SD card
- Mount the root filesystem over NFS

Linux Common Analysis & Observability Tools

Pseudo Filesystems

Pseudo Filesystems

- Some virtual filesystems are exposed by the kernel and provide a lot of information on the system.
- *procfs* contains information about processes and system information.
  - Mounted on `/proc`
  - Often parsed by tools to display raw data in a more user-friendly way.
- *sysfs* provides informations about hardware/logical devices, association between devices and drivers.
  - Mounted on `/sys`
- *debugfs* exposes information related to debug.
  - Typically mounted on `/sys/kernel/debug/`
  - `mount -t debugfs none /sys/kernel/debug`

procfs

- *procfs* exposes information about processes and system (man 5 proc).
  - `/proc/cpuinfo` CPU information.
  - `/proc/meminfo` memory information (used, free, total, etc).
  - `/proc/sys/` contains system parameters that can be tuned. The list of parameters that can be modified is available at admin-guide/sysctl/index
  - `/proc/interrupts`: interrupt count per CPU for each interrupt in use
    - We also have one entry per interrupt in `/proc/irq` for specific configuration/status for each interrupt line
  - `/proc/<pid>/` process related information
    - `/proc/<pid>/status` process basic information
    - `/proc/<pid>/maps` process memory mappings
    - `/proc/<pid>/fd` file descriptors of the process
    - `/proc/<pid>/task` descriptors of threads belonging to the process
  - `/proc/self/` will refer to the process used to access the file
- A list of all available *procfs* file and their content is described at filesystems/proc and man 5 proc

- *sysfs* filesystem exposes information about various kernel subsystems, hardware devices and association with drivers (`man 5 sysfs`).
- This allows to find the link between drivers and devices through a file hierarchy representing the kernel internal tree of devices.
- `/sys/kernel` contains interesting files for kernel debugging:
  - `irq` with information about interrupts (mapping, count, etc).
  - `tracing` for tracing control.
- `admin-guide/abi-stable`

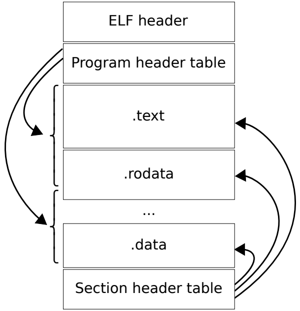
debugfs

- *debugfs* is a simple RAM-based filesystem which exposes debugging information.
- Used by some subsystems (*clk*, *block*, *dma*, *gpio*, etc) to expose debugging information related to the internals.
- Usually mounted on `/sys/kernel/debug`
  - Dynamic debug features exposed through `/sys/kernel/debug/dynamic_debug` (also exposed in `proc`)
  - Clock tree exposed through `/sys/kernel/debug/clk/clk_summary`.

ELF file analysis

Executable and **L**inkable **F**ormat

- File starting with a header which holds binary structures defining the file
- Collection of segments and sections that contain data
  - `.text` section: Code
  - `.data` section: Data
  - `.rodata` section: Read-only Data
  - `.debug_info` section: Contains debugging information
- Sections are part of a segment which can be loadable in memory
- Same format for all architectures supported by the kernel and also `vmlinux` format
  - Also used by a lot of other operating systems as the standard executable file format



binutils for ELF analysis

- The binutils are used to deal with binary files, either object files or executables.
  - Includes `ld`, `as` and other useful tools.
- *readelf* displays information about ELF files (header, section, segments, etc).
- *objdump* allows to display information and disassemble ELF files.
- *objcopy* can convert ELF files or extract/translate some parts of it.
- *nm* displays the list of symbols embedded in ELF files.
- *addr2line* finds the source code line/file pair from an address using an ELF file with debug information

binutils example (1/2)

- Finding the address of `ksys_read()` kernel function using *nm*:

```
$ nm vmlinux | grep ksys_read
c02c7040 T ksys_read
```

- Using *addr2line* to match a kernel OOPS address or a symbol name with source code:

```
$ addr2line -s -f -e vmlinux ffffffff8145a8b0
queue_wc_show
blk-sysfs.c:516
```

binutils example (2/2)

- Display an elf header with *readelf*:

```
$ readelf -h binary
ELF Header:
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF64
Data:                              2's complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              DYN (Position-Independent Executable file)
Machine:                           Advanced Micro Devices X86-64
...
```

- Convert an elf file to a flat binary file using *objcopy*:

```
$ objcopy -O binary file.elf file.bin
```

ldd

- In order to display the shared libraries used by an ELF binary, one can use *ldd* (Generally packaged with C library. See `man 1 ldd`).
- *ldd* will list all the libraries that were used at link time.
  - Libraries that are loaded at runtime using `dlopen()` are not displayed.

```
$ ldd /usr/bin/bash
linux-vdso.so.1 (0x00007ffdf3fc6000)
libreadline.so.8 => /usr/lib/libreadline.so.8 (0x00007fa2d2aef000)
libc.so.6 => /usr/lib/libc.so.6 (0x00007fa2d2905000)
libncursesw.so.6 => /usr/lib/libncursesw.so.6 (0x00007fa2d288e000)
/lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007fa2d2c88000)
```

Monitoring tools

Monitoring Tools

- Lots of monitoring tools on Linux to allow monitoring various part of the system.
- Most of the time, these are CLI interactive programs.
  - Processes with *ps*, *top*, *htop*, etc
  - Memory with *free*, *vmstat*
  - Networking
- Almost all these tools relies on the *sysfs* or *procfs* filesystem to obtain the processes, memory and system information but will display them in a more human readable way.
  - Networking tools uses a netlink interface with the networking subsystem of the kernel.

Process and CPU monitoring tools

Processes with *ps*

- The *ps* command allows to display a snapshot of active processes and their associated information (`man 1 ps`)
  - Lists both user processes and kernel threads.
  - Displays PID, CPU usage, memory usage, uptime, etc.
  - Uses */proc/<pid>/* directory to obtain process information.
  - Always present on almost all embedded platforms (provided by *Busybox*).
- By default, displays only the current user/current tty processes.
- Useful for scripting and parsing since its output is static.

Processes with *ps*

- Display all processes in a friendly way:

```
$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 168864 12800 ?        Ss   09:08   0:00 /sbin/init
root           2  0.0  0.0      0     0 ?        S    09:08   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   09:08   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   09:08   0:00 [rcu_par_gp]
root           5  0.0  0.0      0     0 ?        I<   09:08   0:00 [netns]
...
root         914  0.0  0.0 396216 16220 ?        Ssl  09:08   0:04 /usr/libexec/udisks2/udisksd
avahi        929  0.0  0.0   8728   412 ?        S    09:08   0:00 avahi-daemon: chroot helper
root         956  0.0  0.1 260304 19024 ?        Ssl  09:08   0:02 /usr/sbin/NetworkManager --n
root         960  0.0  0.0  17040  5704 ?        Ss   09:08   0:00 /sbin/wpa_supplicant -u -s -
root         962  0.0  0.0 317644 11896 ?        Ssl  09:08   0:00 /usr/sbin/ModemManager
vnstat       987  0.0  0.0   5516  3696 ?        Ss   09:08   0:00 /usr/sbin/vnstatd -n
```

Processes with *top*

- *top* command output information similar to *ps* but dynamic and interactive (`man 1 top`).
  - Also almost always present on embedded platforms (provided by *Busybox*)

```
$ top
top - 18:38:11 up  9:29,  1 user,  load average: 2.84, 2.74, 2.02
Tasks: 371 total,   1 running, 370 sleeping,   0 stopped,   0 zombie
%Cpu(s):  5.8 us,  2.1 sy,  0.0 ni, 77.4 id, 14.7 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15947.6 total,   1476.9 free,   7685.7 used,   6784.9 buff/cache
MiB Swap:  15259.0 total,  15238.7 free,     20.2 used.   7742.3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   2988 cleger    20   0 5184816   1.2g 430244 S  26.7   7.9  60:24.27 firefox-esr
   4326 cleger    20   0   16.4g 208104  81504 S  26.7   1.3   9:27.33 code
    909 root     -51   0       0      0      0 S  13.3   0.0  15:12.15 irq/104-nvidia
  41704 cleger    20   0   38.4g 373744 116984 S  13.3   2.3  13:25.76 code
  91926 cleger    20   0 2514784 145360  95144 S  13.3   0.9   1:29.85 Web Content
```

mpstat

- *mpstat* displays Multiprocessor statistics (`man 1 mpstat`).
- Useful to detect unbalance CPU workloads, bad IRQ affinity, etc.

```
$ mpstat -P ALL
Linux 6.0.0-1-amd64 (fixe)      19/10/2022      _x86_64_        (4 CPU)

17:02:50     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
17:02:50     all    6,77    0,00    2,09   11,67    0,00    0,06    0,00    0,00    0,00   79,40
17:02:50       0    6,88    0,00    1,93    8,22    0,00    0,13    0,00    0,00    0,00   82,84
17:02:50       1    4,91    0,00    1,50    8,91    0,00    0,03    0,00    0,00    0,00   84,64
17:02:50       2    6,96    0,00    1,74    7,23    0,00    0,01    0,00    0,00    0,00   84,06
17:02:50       3    9,32    0,00    2,80   54,67    0,00    0,00    0,00    0,00    0,00   33,20
17:02:50       4    5,40    0,00    1,29    4,92    0,00    0,00    0,00    0,00    0,00   88,40
```

Memory monitoring tools

free

- *free* is a simple program that displays the amount of free and used memory in the system (`man 1 free`).
  - Useful to check if the system suffers from memory exhaustion
  - Uses `/proc/meminfo` to obtain memory information.

```
$ free -h
              total        used        free      shared  buff/cache   available
Mem:           15Gi       7.5Gi       1.4Gi       192Mi       6.6Gi       7.5Gi
Swap:          14Gi        20Mi        14Gi
```

- *A small* `free` *value does not mean that your system suffers from memory depletion! Linux considers any unused memory as "wasted" so it uses it for buffers and caches to optimize performance. See also* `drop_caches` *from* `man 5 proc` *to observe buffers/cache impact on free/available memory*

vmstat

- *vmstat* displays information about system virtual memory usage
- Can also display stats from processes, memory, paging, block IO, traps, disks and cpu activity (`man 8 vmstat`).
- Can be used to gather data at periodic interval using `vmstat <interval> <number>`

```
$ vmstat 1 6
procs -----------memory----------   ---swap--  -----io---- -system-- ------cpu-----
r  b   swpd   free   buff  cache     si    so    bi    bo    in    cs  us sy id wa st
3  0 253440 1237236 194936 9286980    3     6   186   540   134   157   3  5 82 10  0
```

- *Note: vmstat consider a kernel block to be 1024 bytes*

pmap

- pmap displays process mappings more easily than accessing /proc/<pid>/maps (man 1 pmap).

```
# pmap 2002
2002:   /usr/bin/dbus-daemon --session --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
...
00007f3f958bb000     56K r---- libdbus-1.so.3.32.1
00007f3f958c9000    192K r-x-- libdbus-1.so.3.32.1
00007f3f958f9000     84K r---- libdbus-1.so.3.32.1
00007f3f9590e000      8K r---- libdbus-1.so.3.32.1
00007f3f95910000      4K rw--- libdbus-1.so.3.32.1
00007f3f95937000      8K rw---   [ anon ]
00007f3f95939000      8K r---- ld-linux-x86-64.so.2
00007f3f9593b000    152K r-x-- ld-linux-x86-64.so.2
00007f3f95961000     44K r---- ld-linux-x86-64.so.2
00007f3f9596c000      8K r---- ld-linux-x86-64.so.2
00007f3f9596e000      8K rw--- ld-linux-x86-64.so.2
00007ffe13857000    132K rw---   [ stack ]
00007ffe13934000     16K r----   [ anon ]
00007ffe13938000      8K r-x--   [ anon ]
 total            11088K
```

I/O monitoring tools

iostat

- *iostat* displays information about IOs per device on the system.
- Useful to see if a device is overloaded by IOs.

```
$ iostat
Linux 5.19.0-2-amd64 (fixe)     11/10/2022      _x86_64_        (12 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           8,43    0,00    1,52    8,77    0,00   81,28

Device        tps   kB_read/s   kB_wrtn/s   kB_dscd/s  kB_read  kB_wrtn  kB_dscd
nvme0n1     55,89     1096,88      149,33        0,00  5117334   696668        0
sda          0,03        0,92        0,00        0,00     4308        0        0
sdb        104,42      274,55     2126,64        0,00  1280853  9921488        0
```

iotop

- *iotop* displays information about IOs much like *top* for each process.
- Useful to find applications generating too much I/O traffic.
  - Needs `CONFIG_TASKSTATS=y`, `CONFIG_TASK_DELAY_ACCT=y` and `CONFIG_TASK_IO_ACCOUNTING=y` to be enabled in the kernel.
  - Also needs to be enabled at runtime: `sysctl -w kernel.task_delayacct=1`

```
# iotop
Total DISK READ:         20.61 K/s | Total DISK WRITE:        51.52 K/s
Current DISK READ:       20.61 K/s | Current DISK WRITE:      24.04 K/s
    TID  PRIO  USER      DISK READ DISK WRITE>    COMMAND
    2629 be/4 cleger      20.61 K/s   44.65 K/s firefox-esr [Cache2 I/O]
    322 be/3 root         0.00 B/s    3.43 K/s [jbd2/nvme0n1p1-8]
  39055 be/4 cleger       0.00 B/s    3.43 K/s firefox-esr [DOMCacheThread]
      1 be/4 root         0.00 B/s    0.00 B/s init
      2 be/4 root         0.00 B/s    0.00 B/s [kthreadd]
      3 be/0 root         0.00 B/s    0.00 B/s [rcu_gp]
      4 be/0 root         0.00 B/s    0.00 B/s [rcu_par_gp]
      ...
```

Networking observability tools

ss

- *ss* shows the status of network sockets
  - IPv4 and IPv6, UDP, TCP, ICMP and UNIX domain sockets
- Replaces *netstat*, now obsolete
- Gets info from `/proc/net`
- Usage:

  `ss` by default shows connected sockets

  `ss -l` shows listening sockets

  `ss -a` shows both listening and connected sockets

  `ss -4/-6/-x` shows only IPv4, IPv6, or UNIX sockets

  `ss -t/-u` shows only TCP or UDP sockets

  `ss -p` shows process using each socket

  `ss -n` shows numeric addresses

  `ss -s` shows a summary of existing sockets
- See [the ss manpage] for all the options

ss example output

```
# ss
Netid State  Recv-Q Send-Q                  Local Address:Port            Peer Address:Port  Process
u_dgr ESTAB  0      0                                   * 304840                   * 26673
u_str ESTAB  0      0      /run/dbus/system_bus_socket 42871                   * 26100
icmp6 UNCONN 0      0                                   *:ipv6-icmp                *:*
udp   ESTAB  0      0         192.168.10.115%wlp0s20f3:bootpc         192.168.10.88:bootps
tcp   ESTAB  0      136                    172.16.0.1:41376           172.16.11.42:ssh
tcp   ESTAB  0      273                  192.168.1.77:55494         87.98.181.233:https
tcp   ESTAB  0      0                 [2a02:...:dbdc]:38466     [2001:...:9]:imap2
...
#
```

iftop

- *iftop* displays bandwidth usage on an interface by remote host
- Visualizes bandwidth using histograms
- `iftop -i eth0`

```
                     19.1Mb                   38.1Mb                   57.2Mb                   76.3Mb          95.4Mb
|____________________|________________________|________________________|________________________|______________
localhost.localnet                         => ams.source.kernel.org                    424Kb    352Kb    403Kb
                                           <=                                         18.8Mb   15.6Mb   18.2Mb
localhost.localnet                         => bootlin.com                                 0b    113Kb   35.0Kb
                                           <=                                             0b   6.20Mb   1.84Mb

________________________________________________________________________________________________________________
TX:                cum:   3.06MB   peak:    836Kb                      rates:          424Kb    465Kb    439Kb
RX:                        146MB           44.3Mb                                     18.8Mb   21.8Mb   20.1Mb
TOTAL:                     149MB           45.1Mb                                     19.2Mb   22.3Mb   20.5Mb
```

- The output can be customized interactively
- See the iftop manpage for details

- *tcpdump* allows to capture network traffic and decode many protocols
- `tcpdump -i eth0`
- based on the *libpcap* library for packet capture
- It can also store captured packets to a file and read them back
  - In the *pcap* format or the newer *pcapng* format
  - `tcpdump -i eth0 -w capture.pcap`
  - `tcpdump -r capture.pcap`
- A capture filter can be used to avoid capturing irrelevant packets
  - `tcpdump -i eth0 tcp and not port 22`
- https://www.tcpdump.org/

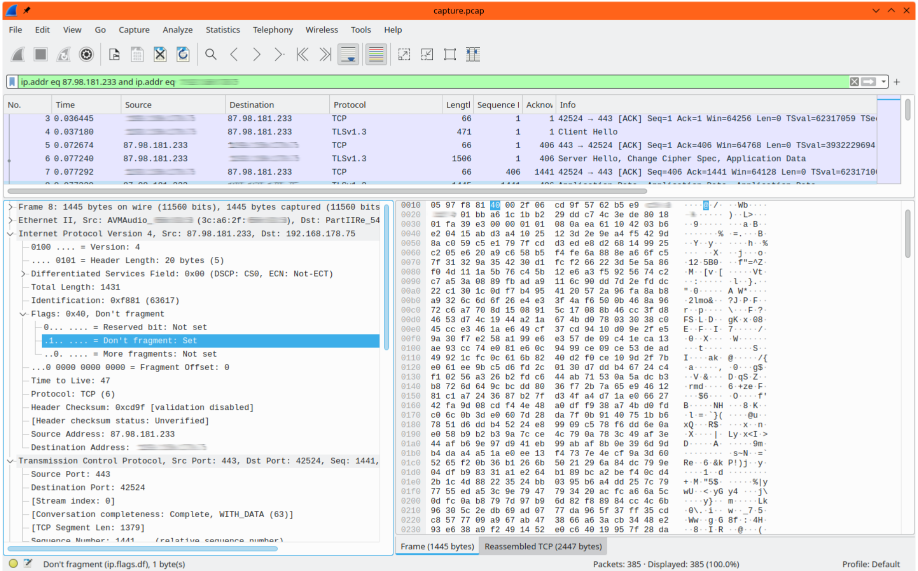
tcpdump example output

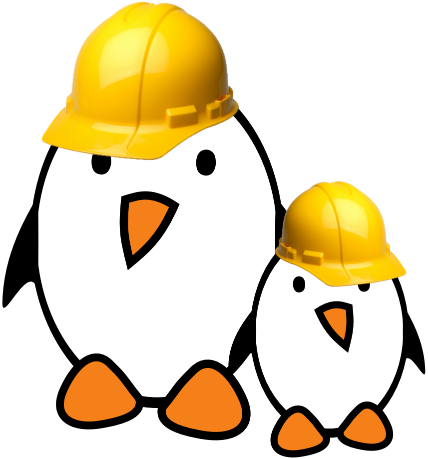
```
# tcpdump -i eth0
18:41:22.913058 IP localhost.localnet.40764 > srv.localnet: 14324+ AAAA? bootlin.com. (29)
18:41:22.913797 IP srv.localnet > localhost.localnet.40764: 14324 0/1/0 (89)
18:41:22.914268 IP localhost.localnet > bootlin.com: ICMP echo request, id 3, seq 1, length 64
18:41:23.933063 IP localhost.localnet > bootlin.com: ICMP echo request, id 3, seq 2, length 64
18:41:24.957027 IP localhost.localnet > bootlin.com: ICMP echo request, id 3, seq 3, length 64
18:41:24.996415 IP bootlin.com > localhost.localnet: ICMP echo reply, id 3, seq 3, length 64
^C
# tcpdump -i eth0 tcp and not port 22
... IP B.https > A.38910: Flags [.], ack 469, win 501, options [...], length 0
... IP B.https > A.38910: Flags [P.], seq 2602:2857, ack 469, win 501, options [...], length 255
... IP A.38910 > B.https: Flags [.], ack 2857, win 501, options [...], length 0
... IP A.38910 > B.https: Flags [P.], seq 469:621, ack 2857, win 501, options [...], length 152
... IP B.https > A.38910: Flags [.], ack 621, win 501, options [...], length 0
... IP B.https > A.38910: Flags [P.], seq 2857:3825, ack 621, win 501, options [...], length 968
... IP A.38910 > B.https: Flags [P.], seq 621:779, ack 3825, win 501, options [...], length 158
^C
#
```

Wireshark

- Similar to tcpdump, but with a GUI
- Also based on libpcap
  - Can capture and use the same BPF capture filters
  - Can load and save the same file formats
    - Useful for embedded: capture on the target with tcpdump, analyze on the host with Wireshark
- Has *dissectors* to decode hundreds of protocols
  - Each individual value from each packet is dissected into a separate field
  - Fields are very fine-grained, at least for the most common protocols
- Has *display filters* that allow filtering *already captured* packets
  - Each dissected field is also a filter key

Wireshark


capture.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
ip.addr eq 87.98.181.233 and ip.addr eq
No. Time Source Destination Protocol Length Sequence Acknow Info
3 0.036445 87.98.181.233 TCP 66 1 1 42524 → 443 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=62317059 TSe
4 0.037180 87.98.181.233 TLSv1.3 471 1 1 Client Hello
5 0.072674 87.98.181.233 TCP 66 1 406 443 → 42524 [ACK] Seq=1 Ack=406 Win=64768 Len=0 TSval=3932229694
6 0.077240 87.98.181.233 TLSv1.3 1506 1 406 Server Hello, Change Cipher Spec, Application Data
7 0.077292 87.98.181.233 TCP 66 406 1441 42524 → 443 [ACK] Seq=406 Ack=1441 Win=64128 Len=0 TSval=6231710
Frame 8: 1445 bytes on wire (11560 bits), 1445 bytes captured (11560 bits
Ethernet II, Src: AVMAudio_ (3c:a6:2f:), Dst: PartIIRe_54
Internet Protocol Version 4, Src: 87.98.181.233, Dst: 192.168.178.75
0100 .... = Version: 4
.... 0101 = Header Length: 20 bytes (5)
Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 1431
Identification: 0xf881 (63617)
Flags: 0x40, Don't fragment
0... .... = Reserved bit: Not set
.1.. .... = Don't fragment: Set
..0. .... = More fragments: Not set
...0 0000 0000 0000 = Fragment Offset: 0
Time to Live: 47
Protocol: TCP (6)
Header Checksum: 0xcd9f [validation disabled]
[Header checksum status: Unverified]
Source Address: 87.98.181.233
Destination Address:
Transmission Control Protocol, Src Port: 443, Dst Port: 42524, Seq: 1441,
Source Port: 443
Destination Port: 42524
[Stream index: 0]
[Conversation completeness: Complete, WITH_DATA (63)]
[TCP Segment Len: 1379]
Frame (1445 bytes) Reassembled TCP (2447 bytes)
Don't fragment (ip.flags.df), 1 byte(s)
Packets: 385 · Displayed: 385 (100.0%)
Profile: Default



Check what is running on a system and its load

- Observe processes and IOs
- Display memory mappings
- Monitor resources

Application Debugging

Good practices

Good practices

- Some good practices can allow you to save time before even needing to use a debugger
- Compiler are now smart enough to detect a wide range of errors at compile-time using warnings
  - Using `-Werror -Wall -Wextra` is recommended if possible to catch errors as early as possible
- Compilers now offer static analysis capabilities
  - GCC allows to do so using the -fanalyzer flag
  - LLVM provides dedicated tools that can be used in build process
- You can also enable component-specific helpers/hardening
  - If you are using the GNU C library, you can for example enable _FORTIFY_SOURCE macro to add runtime checks on inputs (e.g: buffers)

Building with debug information

Debugging with ELF files

- GDB uses ELF files since they are containing the debugging information
- Debugging information uses the DWARF format
- Allows the debugger to match addresses and symbol names, call sites, etc
- Debugging information is generated by the compiler and included in the ELF file when compiled with `-g`
  - `-g1`: minimal debug information (enough for backtraces)
  - `-g2`: default debug level when using `-g`
  - `-g3`: includes extra debugging information (macro definitions)
- See GCC documentation about debugging for more information

Debugging with compiler optimizations

- Compiler optimizations (`-O<level>`) can lead to optimizing out some variables and function calls.
- Trying to display them with GDB will display
  - `$1 = <value optimized out>`
- If one wants to inspect variables and functions, it is possible to compile the code using `-O0` (no optimization).
  - *Note: The kernel can only be compiled with* `-O2` *or* `-Os`
- It is also possible to annotate function with compiler attributes:
  - `__attribute__((optimize("O0")))`
- Remove function `static` qualifier to avoid inlining the function
  - *Note: LTO (Link Time Optimization) can defeat this.*
- Set a specific variable as `volatile` to prevent the compiler from optimizing it out.

Instrumenting code crashes

Instrumenting code crashes

- Displaying a backtrace from your application were the crash happened is useful to debug and can be done using backtrace() (man 3 backtrace) GNU extension function:

```
char **backtrace_symbols(void *const *buffer, int size);
```

- Thanks to signal() (man signal(3)) we can add hooks on specific signals to print our backtrace
  - This is for example very useful to catch SIGSEGV signal to dump our current backtrace

```
void (*signal(int sig, void (*func)(int)))(int);
```

Custom code crash report

```c
[...]
void callee(void *ptr) {
  int *myptr = (int *)ptr;
  printf("Executing suspicious operation\n");
  myptr[2] = 0;
}

void caller(void) {
  void *ptr = NULL;
  callee(ptr);
}

void segfault_handler(int sig) {
  void *array[20];
  size_t size;

  fprintf(stderr, "Segmentation fault!\n");
  size = backtrace(array, 20);
  backtrace_symbols_fd(array, size, STDERR_FILENO);
  exit(1);
}

int main() {
  signal(SIGSEGV, segfault_handler);
  printf("Calling a faulty function\n");
  caller();
  return 0;
}
```

```
[root@arch-bootlin-alexis custom_backtrace]# ./main
Calling a faulty function
Executing suspicious operation
Segmentation fault!
./main(segfault_handler+0x60)[0x55c6e4c1723c]
/usr/lib/libc.so.6(+0x38f50)[0x7fecb0a95f50]
./main(callee+0x2b)[0x55c6e4c171b4]
./main(caller+0x1c)[0x55c6e4c171d9]
./main(main+0x2c)[0x55c6e4c1729a]
/usr/lib/libc.so.6(+0x23790)[0x7fecb0a80790]
/usr/lib/libc.so.6(__libc_start_main+0x8a)[0x7fecb0a8084a]
./main(_start+0x25)[0x55c6e4c170b5]
```

The ptrace system call

ptrace

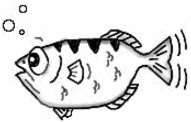
- The *ptrace* mechanism allows processes to trace other processes by accessing tracee memory and register contents
- A tracer can observe and control the execution state of another process
- Works by attaching to a tracee process using the `ptrace()` system call (see `man 2 ptrace`)
- Can be executed directly using the `ptrace()` call but often used indirectly using other tools.

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

- Used by *GDB*, *strace* and all debugging tools that need access to the tracee process state

GDB

GDB: GNU Project Debugger



- The debugger on GNU/Linux, available for most embedded architectures.
- Supported languages: C, C++, Pascal, Objective-C, Fortran, Ada...
- Command-line interface
- Integration in many graphical IDEs
- Can be used to
  - control the execution of a running program, set breakpoints or change internal variables
  - to see what a program was doing when it crashed: post mortem analysis
- https://www.gnu.org/software/gdb/
- https://en.wikipedia.org/wiki/Gdb
- New alternative: *lldb* (https://lldb.llvm.org/) from the LLVM project.

GDB crash course (1/3)

- GDB is used mainly to debug a process by starting it with *gdb*
  - `$ gdb <program>`
- GDB can also be attached to running processes using the program PID
  - `$ gdb -p <pid>`
- When using GDB to start a program, the program needs to be run with
  - `(gdb) run [prog_arg1 [prog_arg2] ...]`

GDB crash course (2/3)

A few useful GDB commands

- `break foobar` (b)
  Put a breakpoint at the entry of function `foobar()`
- `break foobar.c:42`
  Put a breakpoint in `foobar.c`, line 42
- `print var`, `print $reg` or `print task->files[0].fd` (p)
  Print the variable `var`, the register `$reg` or a more complicated reference. GDB can also nicely display structures with all their members
- `info registers`
  Display architecture registers

GDB crash course (3/3)

- `continue` (`c`)
  Continue the execution after a breakpoint
- `next` (`n`)
  Continue to the next line, stepping over function calls
- `step` (`s`)
  Continue to the next line, entering into subfunctions
- `stepi` (`si`)
  Continue to the next instruction
- `finish`
  Execute up to function return
- `backtrace` (`bt`)
  Display the program stack

GDB advanced commands (1/3)

- `info threads` (`i threads`)
  Display the list of threads that are available
- `info breakpoints` (`i b`)
  Display the list of breakpoints/watchpoints
- `delete <n>` (`d <n>`)
  Delete breakpoint <n>
- `thread <n>` (`t <n>`)
  Select thread number <n>
- `frame <n>` (`f <n>`)
  Select a specific frame from the backtrace, the number being the one displayed when using `backtrace` at the beginning of each line

GDB advanced commands (2/3)

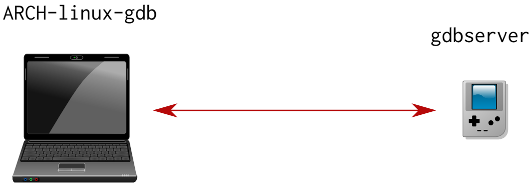
- `watch <variable>` or `watch \*<address>`
  Add a watchpoint on a specific variable/address.
- `print variable = value` (`p variable = value`)
  Modify the content of the specified variable with a new value
- `break foobar.c:42 if condition == value`
  Break only if the specified condition is true
- `watch <variable> if condition == value`
  Trigger the watchpoint only if the specified condition is true
- `display <expr>`
  Automatically prints expression each time program stops
- `x/<n><u> <address>`
  Display memory at the provided address. `n` is the amount of memory to display, `u` is the type of data to be displayed (`b/h/w/g`). Instructions can be displayed using the `i` type.

GDB advanced commands (3/3)

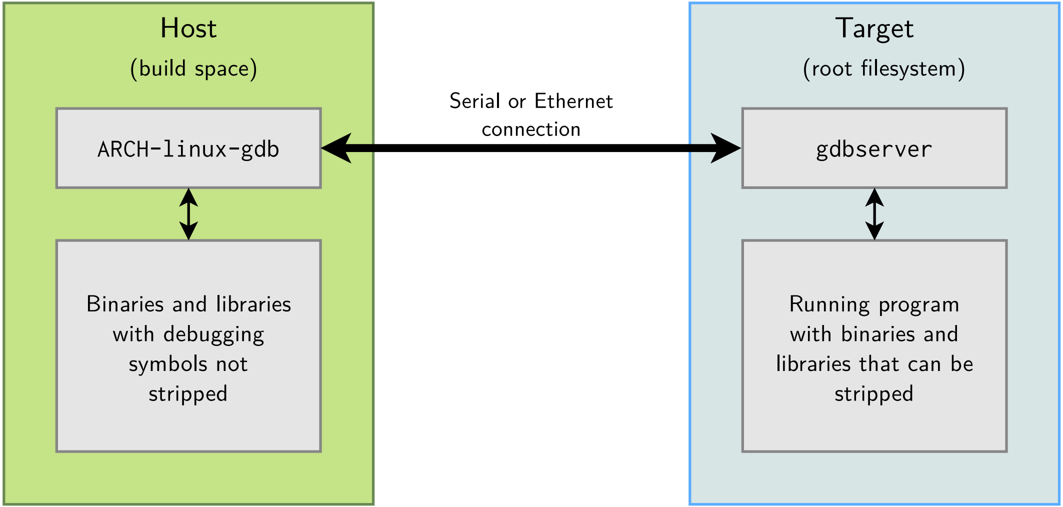
- `list <expr>`
  Display the source code associated to the current program counter location.
- `disassemble <location,start_offset,end_offset>` (`disas`)
  Display the assembly code that is currently executed.
- `p function(arguments)`
  Execute a function using GDB. NOTE: be careful of any side effects that may happen when executing the function
- `p $newvar = value`
  Declare a new gdb variable that can be used locally or in command sequence
- `define <command_name>`
  Define a new command sequence. GDB will prompt for the sequence of commands.

Remote debugging

- In a non-embedded environment, debugging takes place using gdb or one of its front-ends.
- gdb has direct access to the binary and libraries compiled with debugging symbols.
- However, in an embedded context, the target platform environment is often too limited to allow direct debugging with gdb (2.4 MB on x86).
- Remote debugging is preferred
  - `ARCH-linux-gdb` is used on the development workstation, offering all its features.
  - `gdbserver` is used on the target system (only 400 KB on arm).



Remote debugging: architecture



Remote debugging: target setup

- On the target, run a program through `gdbserver`.
  Program execution will not start immediately.
  `gdbserver :<port> <executable> <args>`
  `gdbserver /dev/ttyS0 <executable> <args>`
- Otherwise, attach `gdbserver` to an already running program:
  `gdbserver --attach :<port> <pid>`
- You can also start gdbserver without passing any program to start or attach (and set the target program later, on client side):
  `gdbserver --multi :<port>`

Remote debugging: host setup

- Then, on the host, start `ARCH-linux-gdb <executable>`, and use the following `gdb` commands:
  - To tell `gdb` where shared libraries are:
    `gdb> set sysroot <library-path>` (typically path to build space without `lib/`)
  - To connect to the target:
    `gdb> target remote <ip-addr>:<port>` (networking)
    `gdb> target remote /dev/ttyUSB0` (serial link)
    - Make sure to replace `target remote` with `target extended-remote` if you have started gdbserver with the `--multi` option
  - If you did not set the program to debug on gdbserver commandline:
    `gdb> set remote exec-file <path_to_program_on_target>`

Coredumps for post mortem analysis

- When an application crashes due to a *segmentation fault* and the application was not under control of a debugger, we get no information about the crash
- Fortunately, Linux can generate a `core` file that contains the image of the application memory at the moment of the crash in the ELF format. gdb can use this `core` file to let us analyze the state of the crashed application
- On the target
  - Use `ulimit -c unlimited` in the shell starting the application, to enable the generation of a `core` file when a crash occurs
  - The output name for the coredump file can be modified using `/proc/sys/kernel/core_pattern`. See `man 5 core`
  - On a system with *systemd*, core dumps are disabled by default as a security feature. To restore them temporarily, run: `echo core > /proc/sys/kernel/core_pattern`
- On the host
  - After the crash, transfer the `core` file from the target to the host, and run `ARCH-linux-gdb -c core-file application-binary`

minicoredumper

- Coredumps can be huge for complex applications
- minicoredumper is a userspace tool based on the standard core dump feature
  - Based on the possibility to redirect the core dump output to a user space program via a pipe
- Based on a JSON configuration file, it can:
  - save only the relevant sections (stack, heap, selected ELF sections)
  - compress the output file
  - save additional information from `/proc`
- https://github.com/diamon/minicoredumper
- "Efficient and Practical Capturing of Crash Data on Embedded Systems"
  - Presentation by minicoredumper author John Ogness
  - Video: https://www.youtube.com/watch?v=q2zmwrgLJGs
  - Slides: elinux.org/images/8/81/Eoss2023_ogness_minicoredumper.pdf

GDB: going further

- Tutorial: Debugging Embedded Devices using GDB - Chris Simmonds, 2020
  - Slides: https://elinux.org/images/0/01/Debugging-with-gdb-csimmonds-elce-2020.pdf
  - Video: https://www.youtube.com/watch?v=JGhAgd2a_Ck

GDB Python Extension

- GDB features a python integration, allowing to script some debugging operations
- When executing python under GDB, a module named *gdb* is available and all the GDB specific classes are accessible under this module
- Allows to add new types of commands, breakpoint, printers
  - Used by the kernel to create new commands with the python GDB scripts
- Allows full control and observability over the debugged program using GDB capabilities from Python scripts
  - Controlling execution, adding breakpoints, watchpoints, etc
  - Accessing the process memory, frames, symbols, etc

GDB Python Extension (1/2)

```
class PrintOpenFD(gdb.FinishBreakpoint):
  def __init__(self, file):
    self.file = file
    super(PrintOpenFD, self).__init__()

  def stop (self):
    print ("---> File " + self.file + " opened with fd " + str(self.return_value))
    return False

class PrintOpen(gdb.Breakpoint):
  def stop(self):
    PrintOpenFD(gdb.parse_and_eval("file").string())
    return False

class TraceFDs (gdb.Command):
  def __init__(self):
    super(TraceFDs, self).__init__("tracefds", gdb.COMMAND_USER)

  def invoke(self, arg, from_tty):
    print("Hooking open() with custom breakpoint")
    PrintOpen("open")

TraceFDs()
```

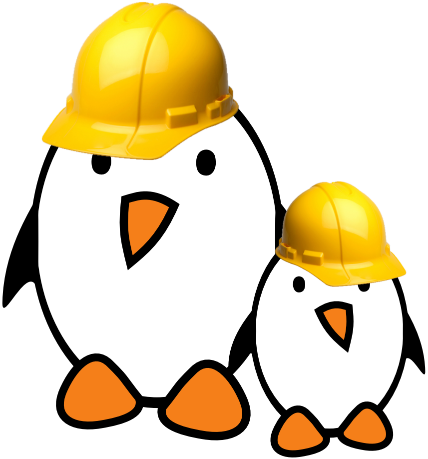
GDB Python Extension (2/2)

- Python scripts can be loaded using gdb `source` command
  - Or the script can be named <program>-gdb.py and will be loaded automatically by GDB

```
(gdb) source trace_fds.py
(gdb) tracefds
Hooking open() with custom breakpoint
Breakpoint 1 at 0x33e0
(gdb) run
Starting program: /usr/bin/touch foo bar
Temporary breakpoint 2 at 0x5555555587da
---> File foo opened with fd 3
Temporary breakpoint 3 at 0x5555555587da
---> File bar opened with fd 0
```

Common debugging issues

- You will likely encounter some issues while debugging, like poor address->symbols conversion, "optimized out" values or functions, empty backtraces...
- A quick checklist before starting debugging can spare you some troubles:
  - Make sure your host binary has debug symbols: with gcc, ensure `-g` is provided, and use non-stripped version with host gdb
  - Disable optimizations on final binary (`-O0`) if possible, or at least use a less intrusive level (`-Og`)
    - Static functions can for example be folded into caller depending on the optimization level, so they would be missing from backtraces
  - Prevent code optimization from reusing frame pointer register: with GCC, make sure `-fno-omit-frame-pointer` option is set
    - Not only true for debugging: any profiling/tracing tool relying on backtraces will benefit from it
- Your application is probably composed of multiple libraries: you will need to apply those configurations on all used components!



Debugging an application crash

- Code generation analysis with compiler-explorer
- Using GDB and its Python support
- Analyzing and using a coredump

Application Tracing

Corrections, suggestions, contributions and translations are welcome!

strace

strace

System call tracer - https://strace.io

- Available on all GNU/Linux systems
  Can be built by your cross-compiling toolchain generator or by your build system.
- Allows to see what any of your processes is doing: accessing files, allocating memory... Often sufficient to find simple bugs.
- Usage:
  `strace <command>` (starting a new process)
  `strace -f <command>` (**f**ollow child processes too)
  `strace -p <pid>` (tracing an existing process)
  `strace -c <command>` (time statistics per system call)
  `strace -e <expr> <command>` (use **e**xpression for advanced filtering)

See the strace manual for details.

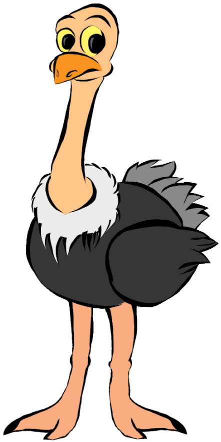


Image credits: https://strace.io/

strace example output

```
> strace cat Makefile
[...]
fstat64(3, {st_mode=S_IFREG|0644, st_size=111585, ...}) = 0
mmap2(NULL, 111585, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f69000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\320h\1\0004\0\0\0\344"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1442180, ...}) = 0
mmap2(NULL, 1451632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e06000
mprotect(0xb7f62000, 4096, PROT_NONE) = 0
mmap2(0xb7f66000, 9840, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7f66000
close(3) = 0
[...]
openat(AT_FDCWD, "Makefile", O_RDONLY)  = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=173, ...}, AT_EMPTY_PATH) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7290d28000
read(3, "ifneq ($(KERNELRELEASE),)\nobj-m "..., 131072) = 173
write(1, "ifneq ($(KERNELRELEASE),)\nobj-m "..., 173ifneq ($(KERNELRELEASE),)
```

Hint: follow the open file descriptors returned by `open()`. This tells you what files are handled by further system calls.

strace -c example output

```
> strace -c cheese
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 36.24    0.523807          19     27017           poll
 28.63    0.413833           5     75287       115 ioctl
 25.83    0.373267           6     63092     57321 recvmsg
  3.03    0.043807           8      5527           writev
  2.69    0.038865          10      3712           read
  2.14    0.030927           3     10807           getpid
  0.28    0.003977           1      3341        34 futex
  0.21    0.002991           3      1030       269 openat
  0.20    0.002889           2      1619       975 stat
  0.18    0.002534           4       568           mmap
  0.13    0.001851           5       356           mprotect
  0.10    0.001512           2       784           close
  0.08    0.001171           3       461       315 access
  0.07    0.001036           2       538           fstat
...
```

ltrace

ltrace

A tool to trace **shared** library calls used by a program and all the signals it receives

- Very useful complement to `strace`, which shows only system calls.
- Of course, works even if you don't have the sources
- Allows to filter library calls with regular expressions, or just by a list of function names.
- With the `-S` option it shows system calls too!
- Also offers a summary with its `-c` option.
- Manual page: https://linux.die.net/man/1/ltrace
- Works better with *glibc*. `ltrace` used to be broken with *uClibc* (now fixed), and is not supported with *Musl* (Buildroot 2022.11 status).

See https://en.wikipedia.org/wiki/Ltrace for details

ltrace example output

```
# ltrace  ffmpeg -f video4linux2 -video_size 544x288 -input_format mjpeg -i /dev
/video0 -pix_fmt rgb565le -f fbdev /dev/fb0
__libc_start_main([ "ffmpeg", "-f", "video4linux2", "-video_size"... ] <unfinished ...>
setvbuf(0xb6a0ec80, nil, 2, 0)                  = 0
av_log_set_flags(1, 0, 1, 0)                    = 1
strchr("f", ':')                                = nil
strlen("f")                                     = 1
strncmp("f", "L", 1)                            = 26
strncmp("f", "h", 1)                            = -2
strncmp("f", "?", 1)                            = 39
strncmp("f", "help", 1)                         = -2
strncmp("f", "-help", 1)                        = 57
strncmp("f", "version", 1)                      = -16
strncmp("f", "buildconf", 1)                    = 4
strncmp("f", "formats", 1)                      = 0
strlen("formats")                               = 7
strncmp("f", "muxers", 1)                       = -7
strncmp("f", "demuxers", 1)                     = 2
strncmp("f", "devices", 1)                      = 2
strncmp("f", "codecs", 1)                       = 3
...
```

ltrace summary

Example summary at the end of the ltrace output (`-c` option)

```
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 52.64    5.958660     5958660         1 __libc_start_main
 20.64    2.336331     2336331         1 avformat_find_stream_info
 14.87    1.682895         421      3995 strncmp
  7.17    0.811210      811210         1 avformat_open_input
  0.75    0.085290         584       146 av_freep
  0.49    0.055150         434       127 strlen
  0.29    0.033008         660        50 av_log
  0.22    0.025090         464        54 strcmp
  0.20    0.022836       22836         1 avformat_close_input
  0.16    0.017788         635        28 av_dict_free
  0.15    0.016819         646        26 av_dict_get
  0.15    0.016753         440        38 strchr
  0.13    0.014536         581        25 memset
...
------ ----------- ----------- --------- --------------------
100.00   11.318773                  4762 total
```

LD_PRELOAD

Shared libraries

- Shared libraries are provided as *.so* files that are actually ELF files
  - Loaded at startup by `ld.so` (the dynamic loader)
  - Or at runtime using `dlopen()` from your code
- When starting a program (an ELF file actually), the kernel will parse it and load the interpreter that needs to be invoked.
  - Most of the time `PT_INTERP` program header of the ELF file is set to `ld-linux.so`.
- At loading time, the dynamic loader `ld.so` will resolve all the symbols that are present in dynamic libraries.
- Shared libraries are loaded only once by the OS and then mappings are created for each application that uses the library.
  - This allows to reduce the memory used by libraries.

Hooking Library Calls

- In order to do some more complex library call hooks, one can use the *LD_PRELOAD* environment variable.
- *LD_PRELOAD* is used to specify a shared library that will be loaded before any other library by the dynamic loader.
- Allows to intercept all library calls by preloading another library.
  - Overrides libraries symbols that have the same name.
  - Allows to redefine only a few specific symbols.
  - "Real" symbol can still be loaded and used with `dlsym` (`man 3 dlsym`)
- Used by some debugging/tracing libraries (*libsegfault*, *libefence*)
- Works for C and C++.

LD_PRELOAD example

- Library snippet that we want to preload using *LD_PRELOAD*:

```c
#include <string.h>
#include <unistd.h>

ssize_t read(int fd, void *data, size_t size) {
  memset(data, 0x42, size);
  return size;
}
```

- Compilation of the library for *LD_PRELOAD* usage:

```
$ gcc -shared -fPIC -o my_lib.so my_lib.c
```

- Preloading the new library using *LD_PRELOAD*:

```
$ LD_PRELOAD=./my_lib.so ./exe
```

uprobes and perf

uprobes

- *uprobe* is a mechanism offered by the kernel allowing to trace userspace code.
- Tracepoints can be added dynamically on any userspace symbol
  - Internally patches the `.text` section with breakpoints that are handled by the kernel trace system
- Exposed by file `/sys/kernel/debug/tracing/uprobe_events`
- Often wrapped up by other tools (`perf`, `bcc` for instance).
- `trace/uprobetracer`

The *perf* tool

- *perf* tool was started as a tool to profile application under Linux using performance counters (`man 1 perf`).
- It became much more than that and now allows to manage tracepoints, kprobes and uprobes.
- *perf* can profile both user-space and kernel-space execution.
- *perf* is based on the `perf_event` interface that is exposed by the kernel.
- Provides a set of operations, each having specific arguments (see *perf* help).
  - `stat`, `record`, `report`, `top`, `annotate`, `ftrace`, `list`, `probe`, etc

Using *perf record*

- *perf record* allows to record performance events per-thread, per-process and per-cpu basis.
- Kernel needs to be configured with `CONFIG_PERF_EVENTS=y`.
- This is the first command that needs to be run to gather data from program execution and output them into `perf.data`.
- `perf.data` file can then be analyzed using `perf annotate` and `perf report`.
  - Useful on embedded systems to analyze data on another computer.

Probing userspace functions

- List functions that can be probed in a specific executable:

```
$ perf probe --source=<source_dir> -x my_app -F
```

- List lines number that can be probed in a specific executable/function:

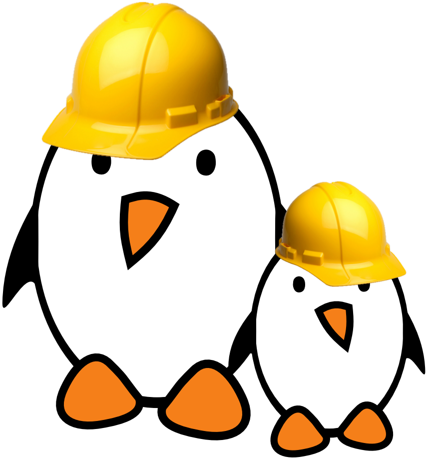
```
$ perf probe --source=<source_dir> -x my_app -L my_func
```

- Create uprobes on user-space library/executable functions:

```
$ perf probe -x /lib/libc.so.6 printf
$ perf probe -x app my_func:3 my_var
$ perf probe -x app my_func%return ret=%r0
```

- Record the execution of these tracepoints:

```
$ perf record -e probe_app:my_func -e probe_libc:printf
```



Analyzing of application interactions

- Analyze dynamic library calls from an application using *ltrace*.
- Overriding a library function with `LD_PRELOAD`.
- Using *strace* to analyze program syscalls.

Memory Issues

Usual Memory Issues

- Programming (almost) always involves accessing memory
- If done incorrectly, a large variety of errors can be triggered
  - Segmentation Faults can happen when accessing invalid memory addresses (NULL pointers or use-after-free for instance)
  - Buffer Overflows can happen if accessing a buffer outside its boundaries
  - Memory Leaks when allocating memory and forgetting to free it after usage
- Fortunately, there are tools to debug these errors

Segmentation Faults

- Segmentation Faults are generated by the kernel when a program tries to access a memory area that it is not allowed to or to access it in an incorrect way
  - Might be generated by a write on a read only memory zone
  - Can also be triggered when trying to execute memory that is not executable

```
int *ptr = NULL;
*ptr = 1;
```

- Execution will yield a `Segmentation fault` message in the terminal

```
$ ./program
Segmentation fault
```

Buffer Overflows

- Buffer Overflows are easily triggered when accessing an array outside of its boundaries (most often past the end)
- Such access might generate a crash or not depending on the access
  - Writing past the end of a `malloc()`'ed array will most often overwrite the malloc data structure leading to corruption
  - Writing past the end of an array allocated on the stack can corrupt data on the stack
  - Reading past the end of an array might generate a segfault but not always, this depends on the area of memory that is accessed

```
uint32_t *array = malloc(10 * sizeof(*array));
array[10] = 0xDEADBEEF;
```

Memory Leaks

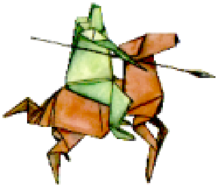
- Memory leaks are another class of memory errors that will not directly trigger a crash but will exhaust the system memory (sooner or later)
- This happens when allocating memory in your program and not releasing it after using it
- Can trigger in production when the program runs for a very long time
  - Better to debug that kind of problem early in the development process

```
void func1(void) {
  uint32_t *array = malloc(10 * sizeof(*array));
  do_something_with_array(array);
}
```

Valgrind memcheck

Valgrind (1/2)

- Valgrind is an instrumentation framework for building dynamic analysis tools
- `valgrind` is also a tool that is based on this framework and provides a memory error detector, heap profilers and others profilers.
- It is supported on all the popular platforms: Linux on x86, x86_64, arm (armv7 only), arm64, mips32, s390, ppc32 and ppc64.



Valgrind (2/2)

- Works by adding its own instrumentation to your code and then running in on its own virtual cpu core. Significantly slows down execution, and thus is suited for debugging and profiling
- *Memcheck* is the defaut *valgrind* tool and it detects memory-management errors
  - Access to invalid memory zones, use of uninitialized values, memory leaks, bad freeing of heap blocks, etc
  - Can be run on any application, no need to recompile them

```
$ valgrind --tool=memcheck --leak-check=full <program>
```

Valgrind Memcheck usage and report

```
$ valgrind ./mem_leak
==202104== Memcheck, a memory error detector
==202104== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==202104== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==202104== Command: ./mem_leak
==202104==
==202104== Conditional jump or move depends on uninitialised value(s)
==202104==    at 0x109161: do_actual_jump (in /home/user/mem_leak)
==202104==    by 0x109187: compute_address (in /home/user/mem_leak)
==202104==    by 0x1091A2: do_jump (in /home/user/mem_leak)
==202104==    by 0x1091D7: main (in /home/user/mem_leak)
==202104==
==202104== HEAP SUMMARY:
==202104==     in use at exit: 120 bytes in 1 blocks
==202104==   total heap usage: 1 allocs, 0 frees, 120 bytes allocated
==202104==
==202104== LEAK SUMMARY:
==202104==    definitely lost: 120 bytes in 1 blocks
==202104==    indirectly lost: 0 bytes in 0 blocks
==202104==      possibly lost: 0 bytes in 0 blocks
==202104==    still reachable: 0 bytes in 0 blocks
==202104==         suppressed: 0 bytes in 0 blocks
==202104== Rerun with --leak-check=full to see details of leaked memory
```

Valgrind and VGDB

- Valgrind can also act as a GDB server which can receive and process commands. One can interact with valgrind gdb server either with a gdb client, or directly with vgdb program (provided with valgrind). vgdb can be used in different ways:
  - As a standalone CLI program to send "monitor" commands to valgrind
  - As a relay between a gdb client and an existing valgrind session
  - As a server to drive multiple valgrind sessions from a remote gdb client
- See `man 1 vgdb` for available modes, commands and options

Using GDB with Memcheck

- *valgrind* allows to attach with GDB to the process that is currently analyzed.

```
$ valgrind --tool=memcheck  --leak-check=full --vgdb=yes --vgdb-error=0 ./mem_leak
```

- Then attach gdb to the valgrind gdbserver using *vgdb*

```
$ gdb ./mem_leak
(gdb) target remote | vgdb
```

- If valgrind detects an error, it will stop the execution and break into GDB.

```
(gdb) continue
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000109161 in do_actual_jump (p=0x4a52040) at mem_leak.c:5
5   if (p[1])
(gdb) bt
#0  0x0000000000109161 in do_actual_jump (p=0x4a52040) at mem_leak.c:5
#1  0x0000000000109188 in compute_address (p=0x4a52040) at mem_leak.c:11
#2  0x00000000001091a3 in do_jump (p=0x4a52040) at mem_leak.c:16
#3  0x00000000001091d8 in main () at mem_leak.c:27
```

Electric Fence

libefence (1/2)

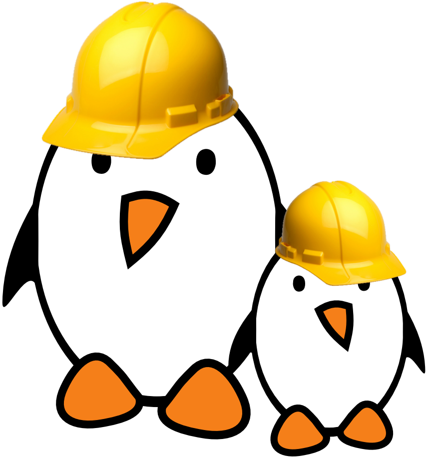
- *libefence* is more lightweight than *valgrind* but less precise
- Allows to catch two types of common memory errors
  - Buffer overflows and use after free
- *libefence* will actually trigger a segfault upon the first error encountered in order to generate a coredump.
- Uses a shared library that can either be linked with statically (`-lefence`) or preloaded using `LD_PRELOAD`.

```
$ gcc -g program.c -o program
$ LD_PRELOAD=libefence.so.0.0 ./program

  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
Segmentation fault (core dumped)
```

libefence (2/2)

- Upon segfault, a coredump will be generated in the current directory
- This coredump can be opened with GDB and will pinpoint the exact location where the error happened

```
$ gdb ./program core-program-3485
Reading symbols from ./libefence...
[New LWP 57462]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Core was generated by `./libefence'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  main () at libefence.c:8
8    data[99] = 1;
(gdb)
```



Debug various memory issues using specific tooling

- Memory leak and misbehavior detection with *valgrind* and *vgdb*.

Application Profiling

Profiling

- Profiling is the act of gathering data from a program execution in order to analyze them and then optimize or fix performance issues.
- Profiling is achieved by using programs that insert instrumentation in the code or leverage kernel/userspace mechanisms.
  - Profiling function calls and count of calls allow to optimize performance.
  - Profiling processor usage allows to optimize performance and reduce power usage.
  - Profiling memory usage allows to optimize memory consumption.
- After profiling, the data set must be analyzed to identify potential improvements (and not the reverse!).

Performance issues

"Premature optimization is the root of all evil", Donald Knuth

- Profiling is often useful to identify and fix performance issues.
- Performances can be affected by memory usage, IOs overload, or CPU usage.
- Gathering profiling data before trying to fix performance issues is needed to do the correct choices.
- Profiling is often guided by a first coarse-grained analysis using some classic tools.
- Once the class of problems has been identified, a fine grain profiling analysis can be done.

Profiling metrics

- Multiple tools allows to profile various metrics.
- Memory usage with *Massif*, `heaptrack` or memusage.
- Function calls using *perf* and callgrind.
- CPU hardware usage (Cache, MMU, etc) using *perf*.
- Profiling data can include both the user space application and kernel.

Visualizing data with flamegraphs

- Visualization based on hierarchical stacks
- Allows to quickly find bottlenecks and explore the call stack
- Popularized by Brendan Gregg tools which allows to generate flamegraphs from `perf` results.
  - Scripts to generate flamegraphs are available at https://github.com/brendangregg/FlameGraph

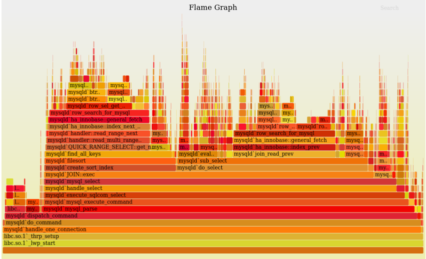
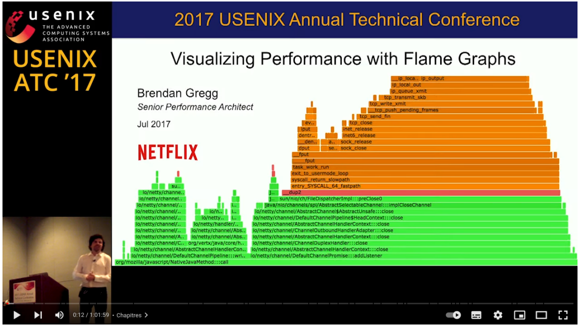


Image credits: https://www.brendangregg.com/flamegraphs.html

Going further with Flamegraphs

- Really nice technical presentation from Brendann Gregg explaining the use of flamegraphs for various metrics.
  - Video: https://www.youtube.com/watch?v=D53T1Ejig1Q
  - Slides: https://www.slideshare.net/brendangregg/usenix-atc-2017-visualizing-performance-with-flame-graphs



Memory profiling

Memory profiling

- Profiling memory usage (heap/stack) in a application is useful for optimization.
- Allocating too much memory can lead to system memory exhaustion.
- Allocating/freeing memory too much can lead to the kernel spending a considerable amount of time in `clear_page()`.
  - The kernel clears pages before giving them to processes to avoid data leakage.
- Reducing application memory footprint can allow optimizing cache usage as well as page miss.

Massif usage

- *Massif* is a tool provided by *valgrind* which allows to profile heap usage during the program execution (user-space only).
- Works by making snapshots of allocations.

```
$ valgrind --tool=massif --time-unit=B program
```

- Once executed, a *massif.out.<pid>* file will be generated in the current directory
- `ms_print` tool can then be used to display a graph of heap allocation

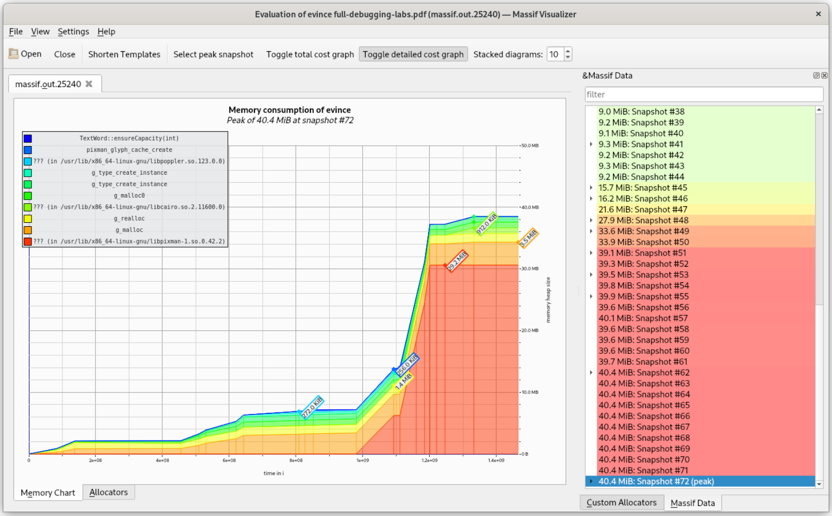
```
$ ms_print massif.out.275099
```

- `#`: Peak allocation
- `@`: Detailed snapshot (count can be adjusted thanks to `--detailed-freq`)

Massif report

```
    KB
547.0^                                               # :: : :@ : :: :
     |                                             @:#:::::::@::::::::@
     |                                           ::@:#:::::::@::::::::@::
     |                                        :::::@:#:::::::@::::::::@:::::
     |                                      :::::::@:#:::::::@::::::::@:::::::
     |                                      :::::::@:#:::::::@::::::::@:::::::
     |                                      :::::::@:#:::::::@::::::::@:::::::
     |                                      :::::::@:#:::::::@::::::::@:::::::
     |                              @@@@@@@@:::::::@:#:::::::@::::::::@:::::::
     |                              @       :::::::@:#:::::::@::::::::@:::::::
     |                              @       :::::::@:#:::::::@::::::::@:::::::
     |                       :::::::@       :::::::@:#:::::::@::::::::@:::::::
     |                       :      @       :::::::@:#:::::::@::::::::@:::::::
     |                 :::::::      @       :::::::@:#:::::::@::::::::@:::::::
     |                 :     :      @       :::::::@:#:::::::@::::::::@:::::::
     |            ::::::     :      @       :::::::@:#:::::::@::::::::@:::::::
     |            :    :     :      @       :::::::@:#:::::::@::::::::@:::::::
     |        :::::    :     :      @       :::::::@:#:::::::@::::::::@:::::::
     |     ::::   :    :     :      @       :::::::@:#:::::::@::::::::@:::::::
     |  ::::  :   :    :     :      @       :::::::@:#:::::::@::::::::@:::::::
   0 +----------------------------------------------------------------------->KB
     0                                                                   830.5

Number of snapshots: 52
 Detailed snapshots: [9, 19, 22 (peak), 32, 42]
```

massif-visualizer - Visualizing massif profiling data



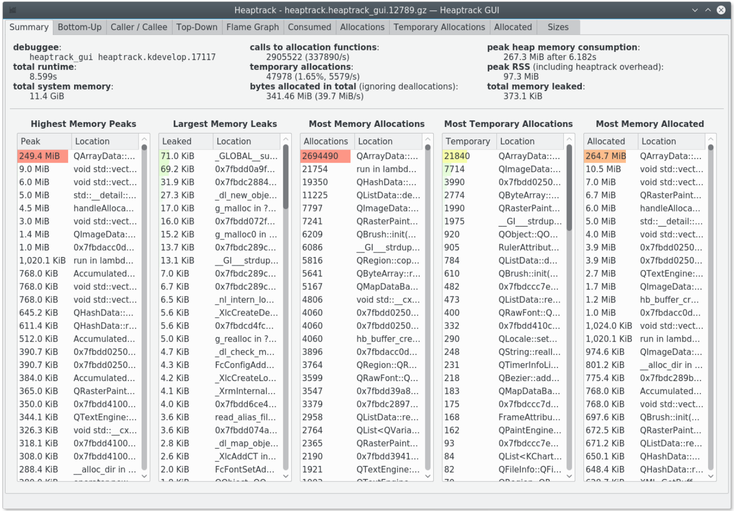
heaptrack usage

- *heaptrack* is a heap memory profiler for Linux.
  - Works with `LD_PRELOAD` library.
- Finer tracking than with Massif and visualizing tool is more advanced.
  - Each allocation is associated to a stacktrace.
  - Allows finding memory leaks, allocation hotspots and temporary allocations.
- Results can be seen using GUI (`heaptrack_gui`) or CLI tool (`heaptrack_print`).
- https://github.com/KDE/heaptrack

```
$ heaptrack program
```

- This will generate a `heaptrack.<process_name>.<pid>.zst` file that can be analyzed using `heaptrack_gui` on another computer.

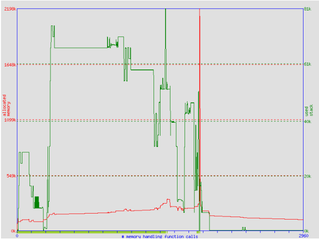
heaptrack_gui - Visualizing heaptrack profiling data



heaptrack_gui - Flamegraph view

memusage

- memusage is a program that leverages `libmemusage.so` to profile memory usage (`man 1 memusage`) (user-space only).
- Can profile heap, stack and also mmap memory usage.
- Profiling information can be shown on the console, logged to a file for post-treatment or catch on a PNG file.
- Lightweight solution compared to valgrind *Massif* tool since it uses the `LD_PRELOAD` mechanism.



memusage usage

```
$ memusage convert foo.png foo.jpg
Memory usage summary: heap total: 2635857, heap peak: 2250856, stack peak: 83696
         total calls   total memory   failed calls
 malloc|       1496        2623648              0
realloc|          6           3744              0  (nomove:0, dec:0, free:0)
 calloc|         16           8465              0
   free|       1480        2521334
Histogram for block sizes:
    0-15            329  21% ==================================================
   16-31            239  15% ====================================
   32-47            287  18% ===========================================
   48-63            321  21% ================================================
   64-79             43   2% ======
   80-95            141   9% =====================
  ...
21424-21439           1  <1%
32768-32783           1  <1%
32816-32831           1  <1%
   large              3  <1%
```

Execution profiling

Execution profiling

- In order to optimize a program, one may have to understand what hardware resources are used.
- Many hardware elements can have an impact on the program execution:
  - CPU cache performance can be degraded by an application without memory spatial locality.
  - Page miss due to using too much memory without spatial locality.
  - Alignment faults when doing misaligned accesses.

Using *perf stat*

- `perf stat` allows to profile an application by gathering performance counters.
  - Using performance counters might require *root* permissions. This can be modified using `# echo -1 > /proc/sys/kernel/perf_event_paranoid`
- The number of performance counters that are present on the hardware are often limited.
- Requesting more events than possible will result in multiplexing and perf will scale the results.
- Collected performance counters are then approximate.
  - To acquire more precise numbers, reduce the number of events observed and run `perf` multiple times changing the events set to observe all the expected events.
  - See perf wiki for more informations.

perf stat example (1/2)

```
$ perf stat convert foo.png foo.jpg

Performance counter stats for 'convert foo.png foo.jpg':

          45,52 msec task-clock                #    1,333 CPUs utilized
              4      context-switches          #   87,874 /sec
              0      cpu-migrations            #    0,000 /sec
          1 672      page-faults               #   36,731 K/sec
    146 154 800      cycles                    #    3,211 GHz                    (81,16%)
      6 984 741      stalled-cycles-frontend   #    4,78% frontend cycles idle   (91,21%)
     81 002 469      stalled-cycles-backend    #   55,42% backend cycles idle    (91,36%)
    222 687 505      instructions              #    1,52  insn per cycle
                                               #    0,36  stalled cycles per insn  (91,21%)
     37 776 174      branches                  #  829,884 M/sec                  (74,51%)
        567 408      branch-misses             #    1,50% of all branches        (70,62%)

    0,034156819 seconds time elapsed

    0,041509000 seconds user
    0,004612000 seconds sys
```

- *NOTE: the percentage displayed at the end denotes the time during which the kernel measured the event due to multiplexing*

perf stat example (2/2)

- List all events:

```
$ perf list
  List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  ...
```

- Count *L1-dcache-load-misses* and *branch-load-misses* events for a specific command

```
$ perf stat -e L1-dcache-load-misses,branch-load-misses cat /etc/fstab
...
Performance counter stats for 'cat /etc/fstab':

23 418      L1-dcache-load-misses
 7 192      branch-load-misses
...
```

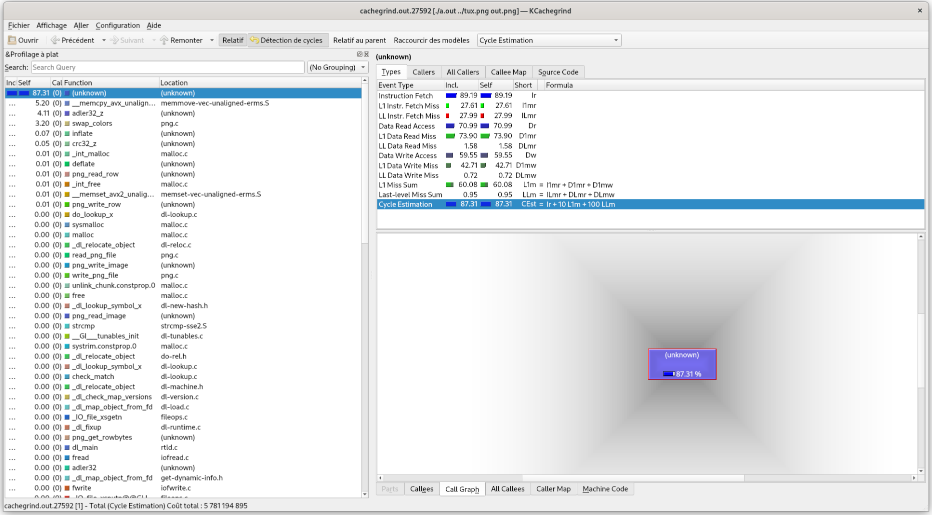
Cachegrind

- *Cachegrind* is a tool provided by *valgrind* for profiling program interactions with the instruction and data cache hierarchy.
  - *Cachegrind* also profiles branch prediction success.
- Simulate a machine with independent `I$` and `D$` backed with a unified L2 cache.
- Really helpful to detect cache usage problems (too many misses, etc).

```
$ valgrind --tool=cachegrind --cache-sim=yes ./my_program
```

- It generates a `cachegrind.out.<pid>` file containing the measures
- `cg_annotate` is a CLI tool used to visualize cachegrind simulation results.
- It also has a `--diff` option to allow comparing two measures files

Kcachegrind - Visualizing Cachegrind profiling data



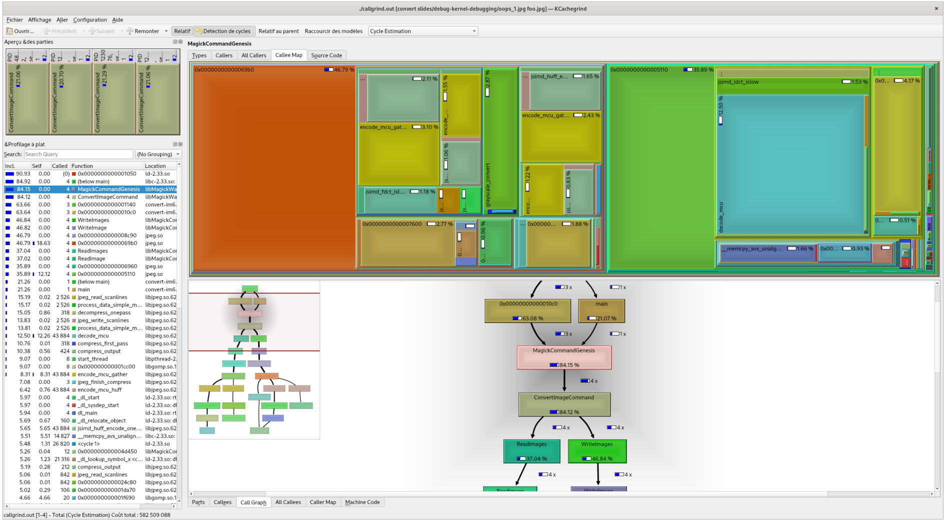
Callgrind

- Provided by *valgrind* and allowing to profile an application call graph (user-space only).
- Collects the number of instructions executed during your program execution and associate these data with the source lines
- Records the call relationship between functions and their call count.

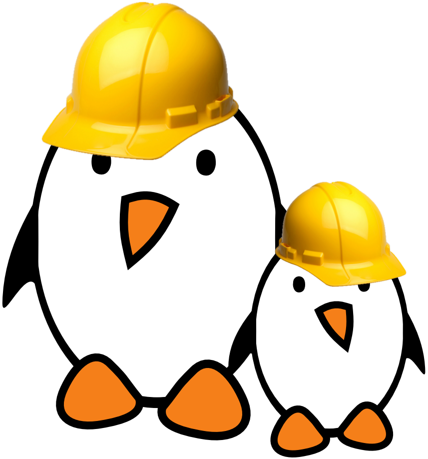
```
$ valgrind --tool=callgrind ./my_program
```

- `callgrind_annotate` is a CLI tool used to visualize callgrind simulation results.
- Kcachegrind can visualize *callgrind* results too.
- The cache simulation (done using cachegrind) has some accuracy shortcomings (See Cachegrind accuracy)

Kcachegrind - Visualizing Callgrind profiling data



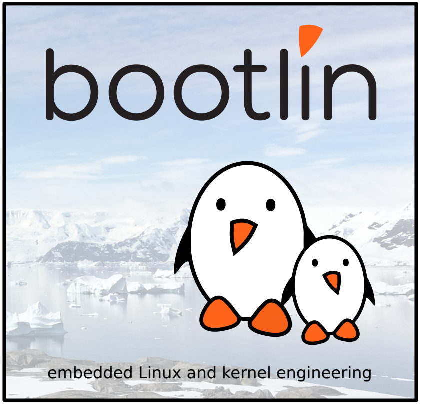
Practical lab - Profiling applications



Profiling an application using various tools

- Profiling application heap using *Massif*.
- Profiling an application with *Cachegrind*, *Callgrind* and *KCachegrind*.
- Analyzing application performance with *perf*.

System-wide Profiling & Tracing



System-wide Profiling & Tracing

- Sometimes, the problems are not tied to an application but rather due to the usage of multiple layers (drivers, application, kernel).
- In that case, it might be useful to analyze the whole stack.
- The kernel already includes a large number of tracepoints that can be recorded using specific tools.
- New tracepoints can also be created statically or dynamically using various mechanisms (kprobes for instance).

kprobes

Kprobes

- Kprobes allows to insert breaks at almost any kernel address dynamically and allows to extract debugging and performance information
- Uses code patching to modify text code to insert calls to specific handlers
  - `kprobes` allows to execute specific handlers when the hooked instruction is executed
  - `kretprobes` will trigger when returning from a function allowing to extract the return value of functions but also display the parameters that were used for the function call
- Support should be enabled using `CONFIG_KPROBES=y`
- Moreover, since probes are inserted using modules, `CONFIG_MODULES=y` and `CONFIG_MODULE_UNLOAD=y` must be set to be able to register probes.
- Also requires `CONFIG_KALLSYMS_ALL=y` when hooking probes using `symbol_name` field
- See `trace/kprobes` for more information

Registering a Kprobe

- kprobes can be registered dynamically by loading a module that registers a `struct kprobe` with `register_kprobe()`
- Probes should be unregistered at module exit using `unregister_kprobe()`

```
struct kprobe probe = {
  .symbol_name = "do_exit",
  .pre_handler = probe_pre,
  .post_handler = probe_post,
};

register_kprobe(&probe);
```

Registering a kretprobe

- `kretprobes` can be registered the same way than regular probes but using a `struct kretprobe` with `register_kretprobe()`
  - Provided handlers will be called on function entry and exit
  - Probe should be unregistered at module exit using `unregister_kretprobe()`

```
int (*kretprobe_handler_t) (struct kretprobe_instance *, struct pt_regs *);
```

```
struct kretprobe probe = {
  .kp.symbol_name = "do_fork",
  .entry_handler = probe_entry,
  .handler = probe_exit,
};

register_kretprobe(&probe);
```

perf

- *perf* allows to do a wide range of tracing and recording operations.
- The kernel already contains events and tracepoints that can be used. The list is given using `perf list`.
- Syscall tracepoints should be enabled in kernel configuration using `CONFIG_FTRACE_SYSCALLS`.
- New tracepoint can be created dynamically on all symbols and registers when debug info are not present.
- Tracing functions, recording variables and parameters content using their names will require a kernel compiled with `CONFIG_DEBUG_INFO`.
- If perf does not find `vmlinux` you have to provide it using `-k <vmlinux>`.

perf example

- List all events that matches `syscalls:*`

```
$ perf list syscalls:*
List of pre-defined events (to be used in -e):

  syscalls:sys_enter_accept                          [Tracepoint event]
  syscalls:sys_enter_accept4                         [Tracepoint event]
  syscalls:sys_enter_access                          [Tracepoint event]
  syscalls:sys_enter_adjtimex_time32                 [Tracepoint event]
  syscalls:sys_enter_bind                            [Tracepoint event]
...
```

- Record all `syscalls:sys_enter_read` events for `sha256sum` command into `perf.data` file.

```
$ perf record -e syscalls:sys_enter_read sha256sum /bin/busybox
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.018 MB perf.data (215 samples) ]
```

perf report example

- Display the collected samples ordered by time spent.

```
$ perf report
Samples: 591  of event 'cycles', Event count (approx.): 393877062
Overhead  Command      Shared Object                   Symbol
  22,88%  firefox-esr  [nvidia]                        [k] _nv031568rm
   3,21%  firefox-esr  ld-linux-x86-64.so.2            [.] __minimal_realloc
   2,00%  firefox-esr  libc.so.6                       [.] __stpncpy_ssse3
   1,86%  firefox-esr  libglib-2.0.so.0.7400.0         [.] g_hash_table_lookup
   1,62%  firefox-esr  ld-linux-x86-64.so.2            [.] _dl_strtoul
   1,56%  firefox-esr  [kernel.kallsyms]               [k] clear_page_rep
   1,52%  firefox-esr  libc.so.6                       [.] __strncpy_sse2_unaligned
   1,37%  firefox-esr  ld-linux-x86-64.so.2            [.] strncmp
   1,30%  firefox-esr  firefox-esr                     [.] malloc
   1,27%  firefox-esr  libc.so.6                       [.] __GI___strcasecmp_l_ssse3
   1,23%  firefox-esr  [nvidia]                        [k] _nv013165rm
   1,09%  firefox-esr  [nvidia]                        [k] _nv007298rm
   1,03%  firefox-esr  [kernel.kallsyms]               [k] unmap_page_range
   0,91%  firefox-esr  ld-linux-x86-64.so.2            [.] __minimal_free
```

perf probe

- *perf* allows to create dynamic tracepoints on both kernel functions and user-space functions.
- In order to be able to insert probes, `CONFIG_KPROBES` must be enabled in the kernel.
  - Note: *libelf* is required to compile *perf* with *probe* command support.
- New dynamic probes can be created and then used using *perf record*.
- Often on embedded platforms, `vmlinux` is not present on the target and thus only symbols and registers can be used.

perf probe examples (1/3)

- List all the kernel symbols that can be probed (no debug info needed):

```
$ perf probe --funcs
```

- Create a new probe on `do_sys_openat2` with *filename* named parameter (debug info required).

```
$ perf probe --vmlinux=vmlinux_file do_sys_openat2 filename:string
Added new event:
  probe:do_sys_openat2 (on do_sys_openat2 with filename:string)
```

- Execute `tail` and capture previously created probe event:

```
$ perf record -e probe:do_sys_openat2 tail /var/log/messages
...
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.003 MB perf.data (19 samples) ]
```

perf probe examples (2/3)

- Display the recorded tracepoints with *perf script*:

```
$ perf script
tail   164 [000]  3552.956573: probe:do_sys_openat2: (c02c3750) filename_string="/etc/ld.so.cache"
tail   164 [000]  3552.956642: probe:do_sys_openat2: (c02c3750) filename_string="/lib/tls/v7l/neon/vfp/libresolv.so.2"
...
```

- Create a new probe on `ksys_read` return value using register `r0` (ARM) alias with "ret" name:

```
$ perf probe ksys_read%return ret=%r0
```

- Execute `sha256sum` and capture previously created probe events:

```
$ perf record -e probe:ksys_read__return sha256sum /etc/fstab
```

perf probe examples (3/3)

- List all probes that have been created:

```
$ perf probe -l
  probe:ksys_read__return (on ksys_read%return with ret)
```

- Remove an existing tracepoint:

```
$ perf probe -d probe:ksys_read__return
```

perf record example

- Record all events for all cpus (system-wide mode):

```
$ perf record -a
^C
```

- Display recorded events from perf.data using `perf script`

```
$ perf script
...
klogd    85 [000]   208.609712:     116584   cycles:          b6dd551c memset+0x2c (/lib/libc.so.6)
klogd    85 [000]   208.609898:     121267   cycles:          c0a44c84 _raw_spin_unlock_irq+0x34 (vmlinux)
klogd    85 [000]   208.610094:     127434   cycles:          c02f3ef4 kmem_cache_alloc+0xd0 (vmlinux)
 perf   130 [000]   208.610311:     132915   cycles:          c0a44c84 _raw_spin_unlock_irq+0x34 (vmlinux)
 perf   130 [000]   208.619831:     143834   cycles:          c0a44cf4 _raw_spin_unlock_irqrestore+0x3c (vmlinux)
klogd    85 [000]   208.620048:     143834   cycles:          c01a07f8 syslog_print+0x170 (vmlinux)
klogd    85 [000]   208.620241:     126328   cycles:          c0100184 vector_swi+0x44 (vmlinux)
klogd    85 [000]   208.620434:     128451   cycles:          c096f228 unix_dgram_sendmsg+0x46c (vmlinux)
kworker/0:2-mm_    44 [000]   208.620653:     133104   cycles:          c0a44c84 _raw_spin_unlock_irq+0x34 (vmlinux)
 perf   130 [000]   208.620859:     138065   cycles:          c0198460 lock_acquire+0x184 (vmlinux)
...
```

Using *perf trace*

- `perf trace` captures and displays all tracepoints/events that have been triggered when executing a command

```
$ perf trace -e "net:*" ping -c 1 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
      0.000 ping/37820 net:net_dev_queue(skbaddr: 0xffff97bbc6a17900, len: 98,
        name: "enp34s0")
      0.005 ping/37820 net:net_dev_start_xmit(name: "enp34s0",
        skbaddr: 0xffff97bbc6a17900, protocol: 2048, len: 98,
        network_offset: 14, transport_offset_valid: 1, transport_offset: 34)
      0.009 ping/37820 net:net_dev_xmit(skbaddr: 0xffff97bbc6a17900, len: 98,
        name: "enp34s0")
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.867 ms
```

Using *perf top*

- `perf top` allows to do a live analysis of the running kernel
- It will sample all function calls and display them ordered by most time consuming one.
- This allows to profile the whole system usage

```
$ perf top
Samples: 19K of event 'cycles', 4000 Hz, Event count (approx.): 4571734204 lost: 0/0 drop: 0/0
Overhead  Shared Object                       Symbol
   2,01%  [nvidia]                            [k] _nv023368rm
   0,94%  [kernel]                            [k] __static_call_text_end
   0,89%  [vdso]                              [.] 0x0000000000000655
   0,81%  [nvidia]                            [k] _nv027733rm
   0,79%  [kernel]                            [k] clear_page_rep
   0,76%  [kernel]                            [k] psi_group_change
   0,70%  [kernel]                            [k] check_preemption_disabled
   0,69%  code                                [.] 0x000000000623108f
   0,60%  code                                [.] 0x0000000006231083
   0,59%  [kernel]                            [k] preempt_count_add
   0,54%  [kernel]                            [k] module_get_kallsym
   0,53%  [kernel]                            [k] copy_user_generic_string
```

ftrace and trace-cmd

ftrace

- *ftrace* is a tracing framework within the kernel which stands for "Function Tracer".
- It offers a wide range of tracing capabilities allowing to observe the system behavior.
  - Trace static tracepoints already inserted at various locations in the kernel (scheduler, interrupts, etc).
  - Relies on GCC mcount() capability and kernel code patching mechanism to call *ftrace* tracing handlers.
- All traces are recorded in a ring buffer that is optimized for tracing.
- Uses *tracefs* filesystem to control and display tracing events.
  - `# mount -t tracefs nodev /sys/kernel/tracing`.
- *ftrace* support must be enabled in the kernel using `CONFIG_FTRACE=y`.
- `CONFIG_DYNAMIC_FTRACE` allows to have a zero overhead tracing support.

ftrace files

- *ftrace* controls are exposed through some specific files located under `/sys/kernel/tracing`.
  - `current_tracer`: Current tracer that is used.
  - `available_tracers`: List of available tracers that are compiled in the kernel.
  - `tracing_on`: Enable/disable tracing.
  - `trace`: Acquired trace in human readable format. Format will differ depending on the tracer used.
  - `trace_pipe`: same as `trace`, but each read consumes the trace as it is read.
  - `trace_marker{_raw}`: Emit comments from userspace in the trace buffer.
  - `set_ftrace_filter`: Filter some specific functions.
  - `set_graph_function`: Graph only the specified functions child.
- Many other files are exposed, see `trace/ftrace`.
- *trace-cmd* CLI and *Kernelshark* GUI tools allow to record and visualize tracing data more easily.

ftrace tracers

- ftrace provides several "tracers" which allow to trace different things.
- The tracer to be used should be written to the `current_tracer` file
  - `nop`: Trace nothing, used to disable all tracing.
  - `function`: Trace all kernel functions that are called.
  - `function_graph`: Similar to `function` but traces both entry and exit.
  - `hwlat`: Trace hardware latency.
  - `irqsoff`: Trace sections where interrupts are disabled.
  - `branch`: Trace likely()/unlikely() prediction errors.
  - `mmiotrace`: Trace all accesses to the hardware (`read[bwlq]/write[bwlq]`).
- **Warning: Some tracers can be expensive!**

```
# echo "function" > /sys/kernel/tracing/current_tracer
```

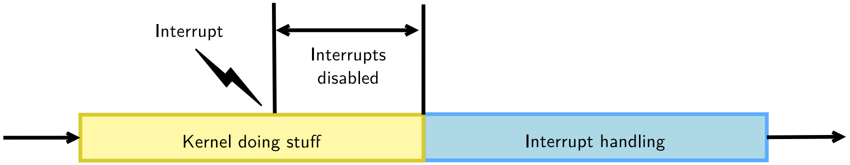
function_graph tracer report example

- The *function_graph* traces all the function that executed and their associated callgraphs
- Will display the process, CPU, timestamp and function graph:

```
$ trace-cmd report
...
dd-113   [000]   304.526590: funcgraph_entry:                   |  sys_write() {
dd-113   [000]   304.526597: funcgraph_entry:                   |    ksys_write() {
dd-113   [000]   304.526603: funcgraph_entry:                   |      __fdget_pos() {
dd-113   [000]   304.526609: funcgraph_entry:        6.541 us   |        __fget_light();
dd-113   [000]   304.526621: funcgraph_exit:       + 18.500 us  |      }
dd-113   [000]   304.526627: funcgraph_entry:                   |      vfs_write() {
dd-113   [000]   304.526634: funcgraph_entry:        6.334 us   |        rw_verify_area();
dd-113   [000]   304.526646: funcgraph_entry:        6.208 us   |        write_null();
dd-113   [000]   304.526658: funcgraph_entry:        6.292 us   |        __fsnotify_parent();
dd-113   [000]   304.526669: funcgraph_exit:       + 43.042 us  |      }
dd-113   [000]   304.526675: funcgraph_exit:       + 78.833 us  |    }
dd-113   [000]   304.526680: funcgraph_exit:       + 91.291 us  |  }
dd-113   [000]   304.526689: funcgraph_entry:                   |  sys_read() {
dd-113   [000]   304.526695: funcgraph_entry:                   |    ksys_read() {
dd-113   [000]   304.526702: funcgraph_entry:                   |      __fdget_pos() {
dd-113   [000]   304.526708: funcgraph_entry:        6.167 us   |        __fget_light();
dd-113   [000]   304.526719: funcgraph_exit:       + 18.083 us  |      }
```

irqsoff tracer

- ftrace *irqsoff* tracer allows to trace the irqs latency due to interrupts being disabled for too long.
- Helpful to find why interrupts have high latencies on a system.
- This tracer will record the longest trace with interrupts being disabled.
- This tracer needs to be enabled with `IRQSOFF_TRACER=y`.
  - `preemptoff`, `premptirqsoff` tracers also exist to trace section of code were preemption is disabled.

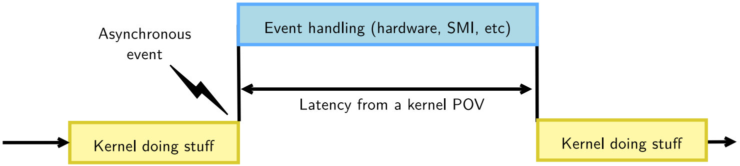


irqsoff tracer report example

```
# latency: 276 us, #104/104, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
#    -----------------
#    | task: stress-ng-114 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: __irq_usr
#  => ended at:   irq_exit
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
stress-n-114      0d...    2us : __irq_usr
stress-n-114      0d...    7us : gic_handle_irq <-__irq_usr
stress-n-114      0d...   10us : __handle_domain_irq <-gic_handle_irq
...
stress-n-114      0d...  270us : __local_bh_disable_ip <-__do_softirq
stress-n-114      0d.s.  275us : __do_softirq <-irq_exit
stress-n-114      0d.s.  279us+: tracer_hardirqs_on <-irq_exit
stress-n-114      0d.s.  290us : <stack trace>
```

Hardware latency detector

- ftrace *hwlat* tracer will help to find if the hardware generates latency.
  - Sytem Management interrupts for instance are non maskable and directly trigger some firmware support feature, suspending CPU execution.
  - Interrupts handled by secure monitor can also cause this kind of latency.
- If some latency is found with this tracer, the system is probably not suitable for real time usage.
- Uses a single core looping while interrupts are disabled and measuring the time elapsed between two consecutive time reads.
- Needs to be builtin the kernel with `CONFIG_HWLAT_TRACER=y`.



trace_printk()

- trace_printk() allows to emit strings in the trace buffer
- Useful to trace some specific conditions in your code and display it in the trace buffer

```
#include <linux/ftrace.h>
void read_hw()
{
  if (condition)
    trace_printk("Condition is true!\n");
}
```

- Will display the following in the trace buffer for function_graph tracer

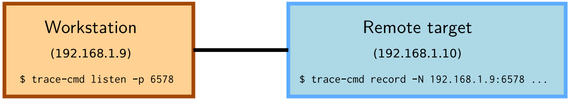
```
1)               |             read_hw() {
1)               |                /* Condition is true! */
1)   2.657 us    |             }
```

trace-cmd

- *trace-cmd* is a tool written by Steven Rostedt which allows interacting with *ftrace* (`man 1 trace-cmd`).
- The tracers supported by *trace-cmd* are those exposed by ftrace.
- *trace-cmd* offers multiple commands:
  - *list*: List available plugins/events that can be recorded.
  - *record*: Record a trace into the file `trace.dat`.
  - *report*: Display `trace.dat` acquisition results.
- At the end of recording, a `trace.dat` file will be generated.

Remote tracing with *trace-cmd*

- *trace-cmd* output can be quite big and thus difficult to store on an embedded platform with limited storage.
- For that purpose, a `listen` command is available and allows sending the acquisitions over the network:
  - Run `trace-cmd listen -p 6578` on the remote system that will be collecting the traces
  - On the target system, use `trace-cmd record -N <target_ip>:6578` to specify the remote system that will collect the traces



trace-cmd examples (1/3)

- List available tracers

```
$ trace-cmd list -t
blk mmiotrace function_graph function nop
```

- List available events

```
$ trace-cmd list -e
...
migrate:mm_migrate_pages_start
migrate:mm_migrate_pages
tlb:tlb_flush
syscalls:sys_exit_process_vm_writev
...
```

- List available functions for filtering with `function` and `function_graph` tracers

```
$ trace-cmd list -f
...
wait_for_initramfs
__ftrace_invalid_address___64
calibration_delay_done
calibrate_delay
...
```

trace-cmd examples (2/3)

- Start the function tracer and record data globally on the system

```
$ trace-cmd record -p function
```

- Use the function tracer but filter only `spi_*` functions

```
$ trace-cmd record -l spi_* -p function
```

- Trace the *dd* command using the function graph tracer:

```
$ trace-cmd record -p function_graph dd if=/dev/mmcblk0 of=out bs=512 count=10
```

- Visualize the data that have been acquired in `trace.dat`:

```
$ trace-cmd report
```

trace-cmd examples (3/3)

- Reset all the *ftrace* buffers and remove tracers

```
$ trace-cmd reset
```

- Run the *irqsoff* tracer on the system:

```
$ trace-cmd record -p irqsoff
```

- Record only `irq_handler_exit/irq_handler_entry` events on the system:

```
$ trace-cmd record -e irq:irq_handler_exit -e irq:irq_handler_entry
```

Adding ftrace tracepoints (1/2)

- For some custom needs, it might be needed to add custom tracepoints
- First, one needs to declare the tracepoint definition in a `.h` file

```
#undef TRACE_SYSTEM
#define TRACE_SYSTEM subsys

#if !defined(_TRACE_SUBSYS_H) || defined(TRACE_HEADER_MULTI_READ)
#define _TRACE_SUBSYS_H

#include <linux/tracepoint.h>

DECLARE_TRACE(subsys_eventname,
        TP_PROTO(int firstarg, struct task_struct *p),
        TP_ARGS(firstarg, p));

#endif /* _TRACE_SUBSYS_H */

/* This part must be outside protection */
#include <trace/define_trace.h>
```

Adding ftrace tracepoints (2/2)

- Then, emit tracepoint in a `.c` file using that header file

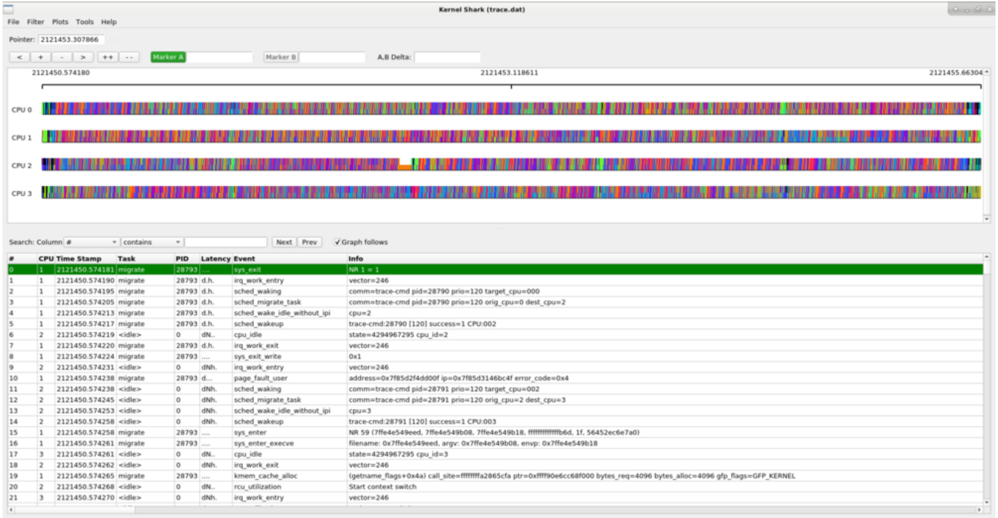
```
#include <trace/events/subsys.h>

#define CREATE_TRACE_POINTS
DEFINE_TRACE(subsys_eventname);

void any_func(void)
{
  ...
  trace_subsys_eventname(arg, task);
  ...
}
```

- See `trace/tracepoints` for more information

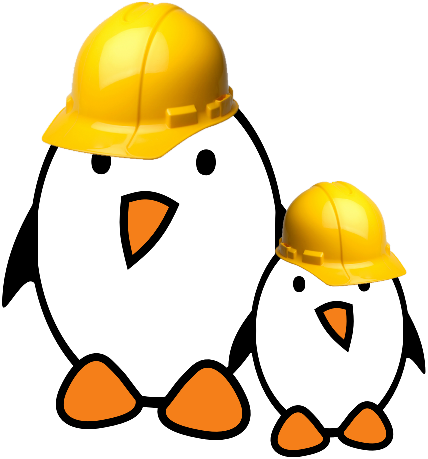
Kernelshark

- Kernelshark is a Qt-based graphical interface for processing *trace-cmd* trace.dat reports.
- Can also setup and acquire data using *trace-cmd*.
- Displays CPU and tasks as different colors along with the recorded events.
- Useful when a deep analysis is required for a specific bug.

kernelshark



Practical lab - System wide profiling



Profiling a system from userspace to kernel space

- Profiling with ftrace, uprobes and kernelshark
- Profiling with perf

LTTng

LTTng

- LTTng is an open source tracing framework for Linux maintained by the EfficiOS company.
- LTTng allows understanding the interactions between the kernel and applications (C, C++, Java, Python).
  - Also expose a `/dev/lttng-logger` that can be used from any application.
- Tracepoints are associated with a payload (data).
- LTTng is focused on low-overhead tracing.
- Uses the Common Trace Format (so traces are readable with other software like babeltrace or trace-compass)

Tracepoints with *LTTng*

- LTTng works with a session daemon that receive all events from kernel and userspace LTTng tracing components.
- LTTng can use and trace the following instrumentation points:
  - LTTng kernel tracepoints
  - kprobes and kretprobes
  - Linux kernel system calls
  - Linux user space probe
  - User space LTTng tracepoints

Creating userspace tracepoints with *LTTng*

- New userspace tracepoints can be defined using LTTng.
- Tracepoints have multiple characteristics:
  - A provider namespace
  - A name identifying the tracepoint
  - Parameters of various types (int, char *, etc)
  - Fields describing how to display the tracepoint parameters (decimal, hexadecimal, etc) (see LTTng-ust manpage for types)
- Developpers must perform multiple operations to use UST tracepoint: write a tracepoint provider (.h), write a tracepoint package (.c), build the package, call the tracepoint in the traced application, and finally build the application, linked with lttng-ust library and the package provider.
- LTTng provides the `lttng-gen-tp` to ease all those steps, allowing to only write a template (.tp) file.

Defining a *LTTng* tracepoint (1/2)

- Tracepoint template (`hello_world-tp.tp`):

```
LTTNG_UST_TRACEPOINT_EVENT(
  // Tracepoint provider name
  hello_world,

  // Tracepoint/event name
  first_tp,

  // Tracepoint arguments (input)
  LTTNG_UST_TP_ARGS(
      char *, text
  ),

  // Tracepoint/event fields (output)
  LTTNG_UST_TP_FIELDS(
      lttng_ust_field_string(message, text)
  )
)
```

- `lttng-gen-tp` will take this template file and generate/build all needed files (.h, .c and .o files)

Defining a *LTTng* tracepoint (2/2)

- Build tracepoint provider:

```
$ lttng-gen-tp hello_world-tp.tp
```

- Tracepoint usage (`hello_world.c`):

```c
#include <stdio.h>
#include "hello-tp.h"

int main(int argc, char *argv[])
{
    lttng_ust_tracepoint(hello_world, my_first_tracepoint, 23, "hi there!");
    return 0;
}
```

- Compilation:

```
$ gcc hello_world.c hello_world-tp.o -llttng-ust -o hello_world
```

Using *LTTng*

```
$ lttng create my-tracing-session --output=./my_traces
$ lttng list --kernel
$ lttng list --userspace
$ lttng enable-event --userspace hello_world:my_first_tracepoint
$ lttng enable-event --kernel --syscall open,close,write
$ lttng start
$ /* Run your application or do something */
$ lttng destroy
$ babeltrace2 ./my_traces
```

- You can also use trace-compass to display the traces in a GUI

Remote tracing with *LTTng*

- LTTng allows to record traces over the network.
- Useful for embedded systems with limited storage capabilities.
- On the remote computer, run `lttng-relayd` command

```
$ lttng-relayd --output=${PWD}/traces
```

- Then on the target, at session creation, use the `--set-url`

```
$ lttng create my-session --set-url=net://remote-system
```

- Traces will then be recorded directly on the remote computer.

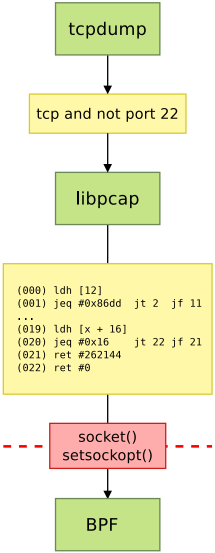
eBPF

The ancestor: Berkeley Packet filter

- BPF stands for Berkeley Packet Filter and was initially used for network packet filtering
- BPF is implemented and used in Linux to perform Linux Socket Filtering (see `networking/filter`)
- tcpdump and Wireshark heavily rely on BPF (through libpcap) for packet capture

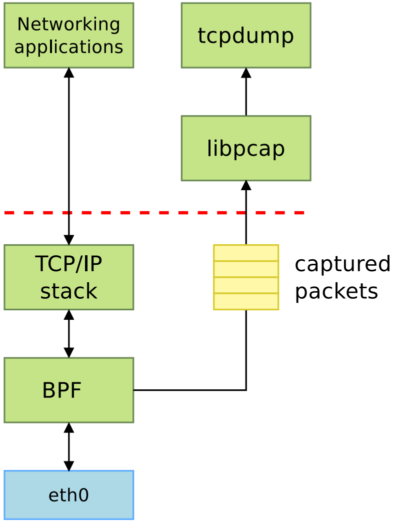
BPF in libpcap: setup

- tcpdump passes the capture filter string from the user to libpcap
- libpcap translates the capture filter into a binary program
  - This program uses the instruction set of an abstract machine (the “BPF instruction set”)
- libpcap sends the binary program to the kernel via the `setsockopt()` syscall



BPF in libpcap: capture

- The kernel implements the BPF "virtual machine"
- The BPF virtual machine executes the program for every packet
- The program inspects the packet data and returns a non-zero value if the packet must be captured
- If the return value is non-zero, the packet is captured in addition to regular packet processing



eBPF (1/2)

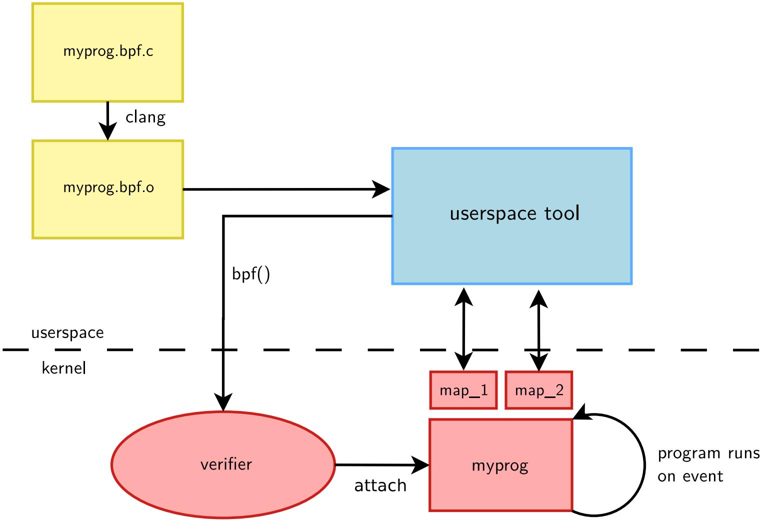
- eBPF is a new framework allowing to run small user programs directly in the kernel, in a safe and efficient way. It has been added in kernel 3.18 but it is still evolving and receiving updates frequently.
- eBPF programs can capture and expose kernel data to userspace, and also alter kernel behavior based on some user-defined rules.
- eBPF is event-driven: an eBPF program is triggered and executed on a specific kernel event
- A major benefit from eBPF is the possibility to reprogram the kernel behavior, without performing kernel development:
  - no risk of crashing the kernel because of bugs
  - faster development cycles to get a new feature ready

Image credits: https://ebpf.io/

eBPF (2/2)

- The most notable eBPF features are:
  - A new instruction set, interpreter and verifier
  - A wide variety of "attach" locations, allowing to hook programs almost anywhere in the kernel
  - dedicated data structures called "maps", to exchange data between multiple eBPF programs or between programs and userspace
  - A dedicated `bpf()` syscall to manipulate eBPF programs and data
  - plenty of (kernel) helper functions accessible from eBPF programs.

eBPF program lifecycle



Kernel configuration for eBPF

- CONFIG_NET to enable eBPF subsystem
- CONFIG_BPF_SYSCALL to enable the bpf() syscall
- CONFIG_BPF_JIT to enable JIT on programs and so increase performance
- CONFIG_BPF_JIT_ALWAYS_ON to force JIT
- CONFIG_BPF_UNPRIV_DEFAULT_OFF=n in **development** to allow eBPF usage without root
- You may then want to enable more general features to "unlock" specific hooking locations:
  - CONFIG_KPROBES to allow hooking programs on kprobes
  - CONFIG_TRACING to allow hooking programs on kernel tracepoints
  - CONFIG_NET_CLS_BPF to write packets classifiers
  - CONFIG_CGROUP_BPF to attach programs on cgroups hooks

eBPF ISA

- eBPF is a ”virtual” ISA, defining its own set of instructions: load and store instruction, arithmetic instructions, jump instructions,etc
- It also defines a set of 10 64-bits wide registers as well as a calling convention:
  - `R0`: return value from functions and BPF program
  - `R1, R2, R3, R4, R5`: function arguments
  - `R6, R7, R8, R9`: callee-saved registers
  - `R10`: stack pointer

```
; bpf_printk("Hello %s\n", "World");
      0:   r1 = 0x0 ll
      2:   r2 = 0xa
      3:   r3 = 0x0 ll
      5:   call 0x6
; return 0;
      6:   r0 = 0x0
      7:   exit
```

The eBPF verifier

- When loaded into the kernel, a program must first be validated by the eBPF verifier.
- The verifier is a complex piece of software which checks eBPF programs against a set of rules to ensure that running those may not compromise the whole kernel. For example:
  - a program must always return and so not contain paths which could make them "infinite" (e.g: no infinite loop)
  - a program must make sure that a pointer is valid before dereferencing it
  - a program can not access arbitrary memory addresses, it must use passed context and available helpers
- If a program violates one of the verifier rules, it will be rejected.
- Despite the presence of the verifier, you still need to be careful when writing programs ! eBPF programs run with preemption enabled (but CPU migration disabled), so they can still suffer from concurrency issues
  - Hopefully there are some mechanisms and helpers to avoid those isses, like per-cpu maps types.

Program types and attach points

- There are different "types" of places to which a program can be hooked
  - an arbitrary kprobe
  - a kernel-defined static tracepoint
  - a specific perf event
  - throughout the network stack
  - and a lot more, see `bpf_attach_type`
- A specific attach-point type can only be hooked with a set of specific program types, see `bpf_prog_type` and `bpf/libbpf/program_types`.
- The program type then defines the data passed to an eBPF program as input when it is invoked. For example:
  - A `BPF_PROG_TYPE_TRACEPOINT` program will receive a structure containing all data returned to userspace by the targeted tracepoint.
  - A `BPF_PROG_TYPE_SCHED_CLS` program (used to implement packets classifiers) will receive a `struct __sk_buff`, the kernel representation of a socket buffer.
  - You can learn about the context passed to any program type by checking `include/linux/bpf_types.h`

eBPF maps

- eBPF programs exchange data with userspace or other programs through maps of different nature:
  - `BPF_MAP_TYPE_ARRAY`: generic array storage. Can be differentiated per cpu
  - `BPF_MAP_TYPE_HASH`: a storage composed of key-value pairs. Keys can be of different types: `__u32`, a device type, an ip address...
  - `BPF_MAP_TYPE_QUEUE`: a FIFO-type queue
  - `BPF_MAP_TYPE_CGROUP_STORAGE`: a specific hash map keyed by a cgroup id. There are other types of maps specific to other object types (inodes, tasks, sockets, etc)
  - etc...
- For basic data, it is easier and more efficient to directly use eBPF global variables (no syscalls involved, contrary to maps)

The bpf() syscall

- The kernel exposes a bpf() syscall to allow interacting with the eBPF subsystem
- The syscall takes a set of subcommands, and depending on the subcommand, some specific data:
  - BPF_PROG_LOAD to load a bpf program
  - BPF_MAP_CREATE to allocate maps to be used by a program
  - BPF_MAP_LOOKUP_ELEM to search for an entry in a map
  - BPF_MAP_UPDATE_ELEM to update an entry in a map
  - etc
- The syscall works with file descriptors pointing to eBPF resources. Those resources (program, maps, links, etc) remain valid while there is at least one program holding a valid file descriptor to it. Those are automatically cleaned once there are no user left.
- For more details, see man 2 bpf

Writing eBPF programs

- eBPF programs can either be written directly in raw eBPF assembly or in higher level languages (e.g: C or rust), and are compiled using the clang compiler.
- The kernel provides some helpers that can be called from an eBPF program:
  - `bpf_trace_printk` Emits a log to the trace buffer
  - `bpf_map_{lookup,update,delete}_elem` Manipulates maps
  - `bpf_probe_{read,write}[_user]` Safely read/write data from/to kernel or userspace
  - `bpf_get_current_pid_tgid` Returns current Process ID and Thread group ID
  - `bpf_get_current_uid_gid` Returns current User ID and Group ID
  - `bpf_get_current_comm` Returns the name of the executable running in the current task
  - `bpf_get_current_task` Returns the current `struct task_struct`
  - Many other helpers are available, see `man 7 bpf-helpers`
- Kernel also exposes kfuncs (see `bpf/kfuncs`), but contrary to bpf-helpers, those do not belong to the kernel stable interface.

Manipulating eBPF program

- There are different ways to build, load and manipulate eBPF programs:
  - One way is to write an eBPF program, build it with clang, and then load it, attach it and read data from it with bare `bpf()` calls in a custom userspace program
  - One can also use `bpftool` on the built ebpf program to manipulate it (load, attach, read maps, etc), without writing any userspace tool
  - Or we can write our own eBPF tool thanks to some intermediate libraries which handle most of the hard work, like libbpf
  - We can also use specialized frameworks like BCC or bpftrace to really get all operations (bpf program build included) handled

BCC

- BPF Compiler Collection (BCC) is (as its name suggests) a collection of BPF based tools.
- BCC provides a large number of ready-to-use tools written in BPF.
- Also provides an interface to write, load and hook BPF programs more easily than using "raw" BPF language.
- Available on a large number of architecture (Unfortunately, not ARM32).
  - On debian, when installed, all tools are named `<tool>-bpfcc`.
- BCC requires a kernel version >= 4.1.
- BCC evolves quickly, many distributions have old versions: you may need to compile from the latest sources

Image credits:
https://github.com/iovisor/bcc

BCC tools

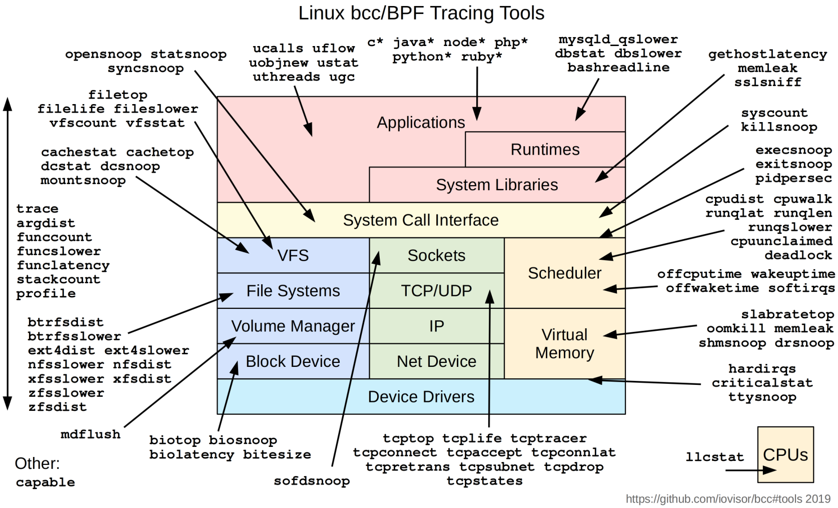


Image credits: https://www.brendangregg.com/ebpf.html

BCC Tools example

- `profile.py` is a CPU profiler allowing to capture stack traces of current execution. Its output can be used for flamegraph generation:

```
$ git clone https://github.com/brendangregg/FlameGraph.git
$ profile.py -df -F 99 10 | ./FlameGraph/flamegraph.pl > flamegraph.svg
```

- `tcpconnect.py` script displays all new TCP connection live

```
$ tcpconnect
PID     COMM         IP SADDR                            DADDR                 DPORT
220321 ssh          6  ::1                              ::1                   22
220321 ssh          4  127.0.0.1                        127.0.0.1             22
17676  Chrome_Child 6  2a01:cb15:81e4:8100:37cf:d45b:d87d:d97d 2606:50c0:8003::154 443
[...]
```

- And much more to discover at https://github.com/iovisor/bcc

Using BCC with python

- BCC exposes a bcc module, and especially a BPF class
- eBPF programs are written in C and stored either in external files or directly in a python string.
- When an instance of the BPF class is created and fed with the program (either as string or file), it automatically builds, loads, and possibly attaches the program
- There are multiple ways to attach a program:
  - By using a proper program name prefix, depending on the targeted attach point (and so the attach step is performed automatically)
  - By explicitely calling the relevant attach method on the BPF instance created earlier

Using BCC with python

- Hook with a *kprobe* on the `clone()` system call and display `"Hello, World!"` each time it is called

```
from bcc import BPF

# define BPF program
prog = """
int hello(void *ctx) {
    bpf_trace_printk("Hello, World!\\n");
    return 0;
}
"""
# load BPF program
b = BPF(text=prog)
b.attach_kprobe(event=b.get_syscall_fnname("clone"), fn_name="hello")
```

libbpf

- Instead of using a high level framework like BCC, one can use libbpf to build custom tools with a finer control on every aspect of the program.
- libbpf is a C-based library that aims to ease eBPF programming thanks to the following features:
  - userspace APIs to handle open/load/attach/teardown of bpf programs
  - userspace APIs to interact with attached programs
  - eBPF APIs to ease eBPF program writing
- Packaged in many distributions and build systems (e.g.: Buildroot)
- Learn more at https://libbpf.readthedocs.io/en/latest/

eBPF programming with libbpf (1/2)

my_prog.bpf.c

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

#define TASK_COMM_LEN 16
struct {
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __type(key, __u32);
  __type(value, __u64);
  __uint(max_entries, 1);
} counter_map SEC(".maps");

struct sched_switch_args {
  unsigned long long pad;
  char prev_comm[TASK_COMM_LEN];
  int prev_pid;
  int prev_prio;
  long long prev_state;
  char next_comm[TASK_COMM_LEN];
  int next_pid;
  int next_prio;
};
```

eBPF programming with libbpf (2/2)

my_prog.bpf.c

```
SEC("tracepoint/sched/sched_switch")
int sched_tracer(struct sched_switch_args *ctx)
{
  __u32 key = 0;
  __u64 *counter;
  char *file;

  char fmt[] = "Old task was %s, new task is %s\n";
  bpf_trace_printk(fmt, sizeof(fmt), ctx->prev_comm, ctx->next_comm);

  counter = bpf_map_lookup_elem(&counter_map, &key);
  if(counter) {
          *counter += 1;
          bpf_map_update_elem(&counter_map, &key, counter, 0);
  }

  return 0;
}

char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

Building eBPF programs

- An eBPF program written in C can be built into a loadable object thanks to clang:

```
$ clang -target bpf -O2 -g -c my_prog.bpf.c -o my_prog.bpf.o
```

  - The `-g` option allows to add debug information as well as BTF information
- GCC can be used too with recent versions
  - the toolchain can be installed with the `gcc-bpf` package in Debian/Ubuntu
  - it exposes the `bpf-unknown-none` target
- To easily manipulate this program with a userspace program based on libbpf, we need "skeleton" APIs, which can be generated with to `bpftool`

bpftool

- `bpftool` is a command line tool allowing to interact with bpf object files and the kernel to manipulate bpf programs:
  - Load programs into the kernel
  - List loaded programs
  - Dump program instructions, either as BPF code or JIT code
  - List loaded maps
  - Dump map content
  - Attach programs to hooks (so they can run)
  - etc
- You may need to mount the bpf filesystem to be able to pin program (needed to keep a program loaded after bpftool has finished running):

```
$ mount -t bpf none /sys/fs/bpf
```

bpftool

- List loaded programs

```
$ bpftool prog
348: tracepoint  name sched_tracer  tag 3051de4551f07909  gpl
loaded_at 2024-08-06T15:43:11+0200  uid 0
xlated 376B  jited 215B  memlock 4096B  map_ids 146,148
btf_id 545
```

- Load and attach a program

```
$ mkdir /sys/fs/bpf/myprog
$ bpftool prog loadall trace_execve.bpf.o /sys/fs/bpf/myprog autoattach
```

- Unload a program

```
$ rm -rf /sys/fs/bpf/myprog
```

bpftool

- Dump a loaded program

```
$ bpftool prog dump xlated id 348
int sched_tracer(struct sched_switch_args * ctx):
; int sched_tracer(struct sched_switch_args *ctx)
  0: (bf) r4 = r1
  1: (b7) r1 = 0
; __u32 key = 0;
  2: (63) *(u32 *)(r10 -4) = r1
; char fmt[] = "Old task was %s, new task is %s\n";
  3: (73) *(u8 *)(r10 -8) = r1
  4: (18) r1 = 0xa7325207369206b
  6: (7b) *(u64 *)(r10 -16) = r1
  7: (18) r1 = 0x7361742077656e20
[...]
```

- Dump eBPF program logs

```
$ bpftool prog tracelog
kworker/u80:0-11  [013] d..41  1796.003605: bpf_trace_printk: Old task was kworker/u80:0, new task is swapper/13
<idle>-0          [013] d..41  1796.003609: bpf_trace_printk: Old task was swapper/13, new task is kworker/u80:0
sudo-18640        [010] d..41  1796.003613: bpf_trace_printk: Old task was sudo, new task is swapper/10
<idle>-0          [010] d..41  1796.003617: bpf_trace_printk: Old task was swapper/10, new task is sudo
[...]
```

bpftool

- List created maps

```
$ bpftool map
80: array  name counter_map  flags 0x0
    key 4B  value 8B  max_entries 1  memlock 256B
    btf_id 421
82: array  name .rodata.str1.1  flags 0x80
    key 4B  value 33B  max_entries 1  memlock 288B
    frozen
96: array  name libbpf_global  flags 0x0
    key 4B  value 32B  max_entries 1  memlock 280B
[...]
```

- Show a map content

```
$ sudo bpftool map dump id 80
[{
  "key": 0,
  "value": 4877514
  }
]
```

- Generate libbpf APIs to manipulate a program

```
$ bpftool gen skeleton trace_execve.bpf.o name trace_execve > trace_execve.skel.h
```

- We can then write our userspace program and benefit from high level APIs to manipulate our eBPF program:
  - instantiation of a global context object which will have references to all of our programs, maps, links, etc
  - loading/attaching/unloading of our programs
  - eBPF program directly embedded in the generated header as a byte array

Userspace code with libbpf

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "trace_sched_switch.skel.h"

int main(int argc, char *argv[])
{
    struct trace_sched_switch *skel;
    int key = 0;
    long counter = 0;

    skel = trace_sched_switch__open_and_load();
    if(!skel)
        exit(EXIT_FAILURE);
    if (trace_sched_switch__attach(skel)) {
        trace_sched_switch__destroy(skel);
        exit(EXIT_FAILURE);
    }

    while(true) {
        bpf_map__lookup_elem(skel->maps.counter_map, &key, sizeof(key), &counter, sizeof(counter), 0);
        fprintf(stderr, "Scheduling switch count: %d\n", counter);
        sleep(1);
    }

    return 0;
}
```

eBPF programs portability (1/2)

- Kernel internals, contrary to userspace APIs, do not expose stable APIs. This means that an eBPF program manipulating some kernel data may not work with another kernel version
- The CO-RE (Compile Once - Run Everywhere) approach aims to solve this issue and make programs portable between **kernel versions**. It relies on the following features:
  - your kernel must be built with `CONFIG_DEBUG_INFO_BTF=y` to have BTF data embedded. BTF is a format similar to dwarf which encodes data layout and functions signatures in an efficient way.
  - your eBPF compiler must be able to emit BTF relocations (both clang and GCC are capable of this on recent versions, with the `-g` argument)
  - you need a BPF loader capable of processing BPF programs based on BTF data and adjust accordingly data accesses: `libbpf` is the de-facto standard bpf loader
  - you then need eBPF APIs to read/write to CO-RE relocatable variables. libbpf provides such helpers, like `bpf_core_read`
- To learn more, take a look at Andrii Nakryiko's CO-RE guide

eBPF programs portability (2/2)

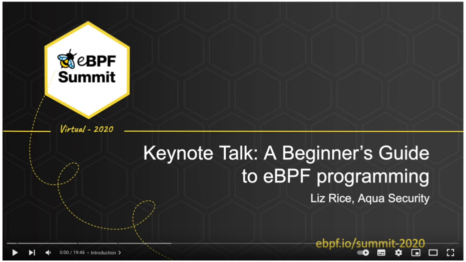
- Despite CO-RE, you may still face different constraints on different kernel versions, because of major features introduction or change, since the eBPF subsystem keeps receiving frequent updates:
  - eBPF tail calls (which allow a program to call a function) have been added in version 4.2, and allow to call another program only since version 5.10
  - eBPF spin locks have been added in version 5.1 to prevent concurrent accesses to maps shared between cpus.
  - Different attach types keep being added, but possibly on different kernel versions when it depends on the architecture: fentry/fexit attach points have been added in kernel 5.5 for x86 but in 6.0 for arm32.
  - Any kind of loop (even bounded) was forbidden until version 5.3
  - `CAP_BPF` capability, allowing a process to perform eBPF tasks, has been added in version 5.8

eBPF for tracing/profiling

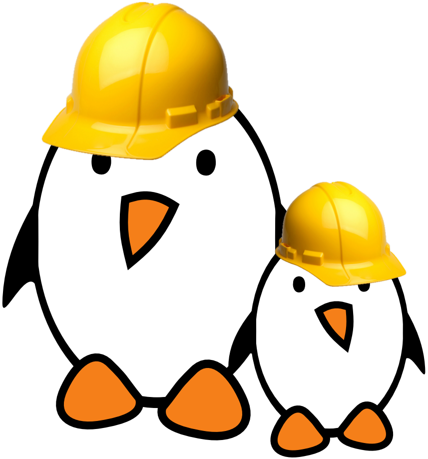
- eBPF is a very powerful framework to spy on kernel internals: thanks to the wide variety of attach point, you can expose almost any kernel code path and data.
- In the mean time, eBPF programs remain isolated from kernel code, which makes it safe (compared to kernel development) and easy to use.
- Thanks to the in-kernel interpreter and optimizations like JIT compilation, eBPF is very well suited for tracing or profiling with low overhead, even in production environments, while being very flexible.
- This is why eBPF adoption level keeps growing for debugging, tracing and profiling in the Linux ecosystem. As a few examples, we find eBPF usage in:
  - tracing frameworks like BCC and bpftrace
  - network infrastructure components, like Cilium or Calico
  - network packet tracers, like pwru or dropwatch
  - And many more, check ebpf.io for more examples

eBPF: resources

- BCC tutorial: https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md
- libbpf-bootsrap: https://github.com/libbpf/libbpf-bootstrap
- A Beginner's Guide to eBPF Programming - Liz Rice, 2020
  - Video: https://www.youtube.com/watch?v=lrSExTfS-iQ
  - Resources: https://github.com/lizrice/ebpf-beginners



Practical lab - System wide profiling



Creating custom tracing tools with eBPF

- Tracing with BCC
- Converting a BCC script to libbpf
- Bringing advanced features to the tool

Choosing the right tool

Choosing the right tool

- Before starting to profile or trace, one should know which type of tool to use.
- This choice is guided by the level of profiling
- Often start by analyzing/optimizing the application level using application tracing/profiling tools (valgrind, perf, etc).
- Then analyze user space + kernel performance
- Finally, trace or profile the whole system if the performance problems happens only when running under a loaded system.
  - For "constant" load problems, snapshot tools works fine.
  - For sporadic problems, record traces and analyze them.
- If you happen to have a complex setup that you often have to bring up, it is likely a sign that you want to ease this setup with some custom tooling: scripting, custom traces, eBPF, etc

Kernel Debugging

Preventing bugs

Static code analysis

- Static analysis can be run with the *sparse* tool
- *sparse* works with annotation and can detect various errors at compile time
  - Locking issues (unbalanced locking)
  - Address space issues, such as accessing user space pointer directly
- Analysis can be run using `make C=2` to run only on files that are recompiled
- Or with `make C=1` to run on all files
- Example of an unbalanced locking scheme:

```
rzn1_a5psw.c:81:13: warning: context imbalance in 'a5psw_reg_rmw' - wrong count
  at exit
```

Good practices in kernel development (1/2)

- When writing driver code, never expect the user to provide correct values. Always check these values.
- Use the `WARN_ON()` macro if you want to display a stacktrace when a specific condition did happen.
  - `dump_stack()` can also be used during debugging to show the current call stack.

```
static bool check_flags(u32 flags)
{
  if (WARN_ON(flags & STATE_INVALID))
    return -EINVAL;
  return 0;
}
```

Good practices in kernel development (2/2)

- If the values can be checked at compile time (configuration input, `sizeof()` structure fields), use the `BUILD_BUG_ON()` macro to ensure the condition is true.

```
BUILD_BUG_ON(sizeof(ctx->__reserved) != sizeof(reserved));
```

- If during compilation you have some warnings about unused variables/parameters, they must be fixed.
- Apply `checkpatch.pl --strict` when possible which might find some potential problems in your code.

Linux Kernel Debugging

Linux Kernel Debugging

- The Linux Kernel features some very useful tools for debugging.
- These tools are builtin the kernel since their activation often selects instrumentation code for debugging
  - Erroneous memory accesses debugging tools (*KASAN*, *Kmemleak*, *KFENCE*)
  - Undefined behavior code debugging (*UBSAN*)
  - Locking errors analysis (*lockdep*)
- All the debug features are located under the `Kernel hacking -> Kernel debugging` menuconfig entry.
  - `CONFIG_DEBUG_KERNEL` should be set to "y" to enable other debug options.

Debugging using messages

Debugging using messages (1/3)

Three APIs are available

- The old printk(), no longer recommended for new debugging messages
- The pr_*() family of functions: pr_emerg(), pr_alert(), pr_crit(), pr_err(), pr_warn(), pr_notice(), pr_info(), pr_cont()
  and the special pr_debug() (see next pages)
  - Defined in include/linux/printk.h
  - They take a classic format string with arguments
  - Example:

    ```
    pr_info("Booting CPU %d\n", cpu);
    ```
  - Here's what you get in the kernel log:

    ```
    [  202.350064] Booting CPU 1
    ```
- print_hex_dump_debug(): useful to dump a buffer with hexdump like display

Debugging using messages (2/3)

- The `dev_*()` family of functions: `dev_emerg()`, `dev_alert()`, `dev_crit()`, `dev_err()`, `dev_warn()`, `dev_notice()`, `dev_info()` and the special `dev_dbg()` (see next page)
  - They take a pointer to `struct device` as first argument, and then a format string with arguments
  - Defined in `include/linux/dev_printk.h`
  - To be used in drivers integrated with the Linux device model
  - Example:

    ```
    dev_info(&pdev->dev, "in probe\n");
    ```

  - Here's what you get in the kernel log:

    ```
    [   25.878382] serial 48024000.serial: in probe
    [   25.884873] serial 481a8000.serial: in probe
    ```

- `*_ratelimited()` version exists which limits the amount of print if called too much based on `/proc/sys/kernel/printk_ratelimit{_burst}` values

Debugging using messages (3/3)

- The kernel defines many more format specifiers than the standard `printf()` existing ones.
  - `%p`: Display the hashed value of pointer by default.
  - `%px`: Always display the address of a pointer (use carefully on non-sensitive addresses).
  - `%pK`: Display hashed pointer value, zeros or the pointer address depending on `kptr_restrict` sysctl value.
  - `%pOF`: Device-tree node format specifier.
  - `%pr`: Resource structure format specifier.
  - `%pa`: Physical address display (work on all architectures 32/64 bits)
  - `%pe`: Error pointer (displays the string corresponding to the error number)
- `/proc/sys/kernel/kptr_restrict` should be set to `1` in order to display pointers which uses `%pK`
- See `core-api/printk-formats` for an exhaustive list of supported format specifiers

pr_debug() and dev_dbg()

- When the driver is compiled with `DEBUG` defined, all these messages are compiled and printed at the debug level. `DEBUG` can be defined by `#define DEBUG` at the beginning of the driver, or using `ccflags-$(CONFIG_DRIVER) += -DDEBUG` in the `Makefile`
- When the kernel is compiled with `CONFIG_DYNAMIC_DEBUG`, then these messages can dynamically be enabled on a per-file, per-module or per-message basis, by writing commands to `/proc/dynamic_debug/control`. Note that messages are not enabled by default.
  - Details in `admin-guide/dynamic-debug-howto`
  - Very powerful feature to only get the debug messages you're interested in.
- When neither `DEBUG` nor `CONFIG_DYNAMIC_DEBUG` are used, these messages are not compiled in.

pr_debug() and dev_dbg() usage

- Debug prints can be enabled using the `/proc/dynamic_debug/control` file.
  - `cat /proc/dynamic_debug/control` will display all lines that can be enabled in the kernel
  - Example: `init/main.c:1427 [main]run_init_process =p " \%s\012"`
- A syntax allows to enable individual print using lines, files or modules
  - `echo "file drivers/pinctrl/core.c +p" > /proc/dynamic_debug/control` will enable all debug prints in `drivers/pinctrl/core.c`
  - `echo "module pciehp +p" > /proc/dynamic_debug/control` will enable the debug print located in the `pciehp` module
  - `echo "file init/main.c line 1427 +p" > /proc/dynamic_debug/control` will enable the debug print located at line 1247 of file `init/main.c`
  - Replace `+p` with `-p` to disable the debug print

Debug logs troubleshooting

- When using dynamic debug, make sure that your debug call is enabled: it must be visible in `control` file in debugfs **and** be actived (`=p`)
- Is your log output only in kernel log buffer?
  - You can see it thanks to `dmesg`
  - You can lower the `loglevel` to output it to the console directly
  - You can also set `ignore_loglevel` in the kernel command line to force all kernel logs to console
- If you are working on an out-of-tree module, you may prefer to define `DEBUG` in your module source or Makefile instead of using dynamic debug
- If configuration is done through kernel command line, is it properly interpreted?
  - Starting from 5.14, kernel will let you know about faulty command line: `Unknown kernel command line parameters foo, will be passed to user space.`
  - You may need to take care of special characters escaping (e.g: quotes)
- Be aware that a few subsystems bring their own logging infrastructure, with specific configuration/controls, eg: `drm.debug=0x1ff`

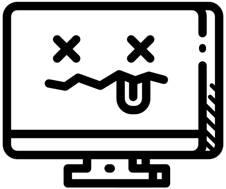
Kernel early debug

- When booting, the kernel sometimes crashes even before displaying the system messages
- On ARM, if your kernel doesn't boot or hangs without any message, you can activate early debugging options
  - `CONFIG_DEBUG_LL=y` to enable ARM early serial output capabilities
  - `CONFIG_EARLYPRINTK=y` will allow printk to output the prints earlier
- `earlyprintk` command line parameter should be given to enable early printk output

Kernel crashes and oops

Kernel crashes

- The kernel is not immune to crash, many errors can be done and lead to crashes
  - Memory access error (NULL pointer, out of bounds access, etc)
  - Voluntarily panicking on error detection (using `panic()`)
  - Kernel incorrect execution mode (sleeping in atomic context)
  - Deadlocks detected by the kernel (Soft lockup/locking problem)
- On error, the kernel will display a message on the console that is called a "Kernel oops"



Icon by Peter van Driel, TheNounProject.com

Kernel oops (1/2)

- The content of this message depends on the architecture that is used.
- Almost all architectures display at least the following information:
  - CPU state when the oops happened
  - Registers content with potential interpretation
  - Backtrace of function calls that led to the crash
  - Stack content (last X bytes)
- Depending on the architecture, the crash location can be identified using the content of the PC registers (sometimes named IP, EIP, etc).
- To have a meaningful backtrace with symbol names use `CONFIG_KALLSYMS=y` which will embed the symbol names in the kernel image.

Kernel oops (2/2)

- Symbols are displayed in the backtrace using the following format:
  - `<symbol_name>+<hex_offset>/<symbol_size>`
- If the oops is not critical (taken in process context), then the kernel will kill process and continue its execution
  - The kernel stability might be compromised!
- Tasks that are taking too much time to execute and that are hung can also generate an oops (`CONFIG_DETECT_HUNG_TASK`)
- If KGDB support is present and configured, on oops, the kernel will switch to KGDB mode.

Oops example (1/2)

```
1.635909] 8<--- cut here ---
1.639034] Unable to handle kernel NULL pointer dereference at virtual address 00000050
1.647304] [00000050] *pgd=00000000
1.650959] Internal error: Oops: 5 [#1] SMP ARM
1.655635] Modules linked in:
1.658736] CPU: 0 PID: 28 Comm: kworker/u4:1 Not tainted 5.18.0-rc6-01642-g59dcbe02d20b #48
1.667261] Hardware name: Generic DT based system
1.672104] Workqueue: events_unbound deferred_probe_work_func
1.678029] PC is at miic_create+0xac/0x134
1.682280] LR is at _raw_spin_unlock_irqrestore+0x2c/0x34
1.687841] pc : [<804ba6e4>]    lr : [<806811f0>]    psr: 20000053
1.694172] sp : 90a99cf8  ip : 90a99cc8  fp : 90a99d1c
1.699453] r10: 8089339e  r9 : 808a5a18  r8 : 8fdb4284
1.704733] r7 : 82173000  r6 : 8fdb47d8  r5 : 82173010  r4 : 8fdb3ad8
1.711327] r3 : 00000000  r2 : 00000000  r1 : a0000053  r0 : 8fdb3ad8
1.717921] Flags: nzCv  IRQs on  FIQs off  Mode SVC_32  ISA ARM  Segment none
1.725220] Control: 10c5387d  Table: 8000406a  DAC: 00000051
1.731022] Register r0 information: non-slab/vmalloc memory
1.736754] Register r1 information: non-paged memory
1.741867] Register r2 information: NULL pointer
1.746629] Register r3 information: NULL pointer
1.751389] Register r4 information: non-slab/vmalloc memory
1.757113] Register r5 information: slab kmalloc-512 start 82173000 pointer offset 16 size 512
1.765943] Register r6 information: non-slab/vmalloc memory
1.771668] Register r7 information: slab kmalloc-512 start 82173000 pointer offset 0 size 512
1.780401] Register r8 information: non-slab/vmalloc memory
1.786125] Register r9 information: non-slab/vmalloc memory
1.791848] Register r10 information: non-slab/vmalloc memory
1.797661] Register r11 information: 2-page vmalloc region starting at 0x90a98000 allocated at kernel_clone+0xb4/0x2a4
1.808567] Register r12 information: 2-page vmalloc region starting at 0x90a98000 allocated at kernel_clone+0xb4/0x2a4
1.819468] Process kworker/u4:1 (pid: 28, stack limit = 0x(ptrval))
1.825889] Stack: (0x90a99cf8 to 0x90a9a000)
1.830304] 9ce0:                                                       00000005 ddf3fe37
1.838570] 9d00: 8259c840 8259c840 82173010 8fdb47d8 90a99dac 90a99d20 804bb7b8 804ba644
1.846834] 9d20: 00000000 90a99d34 00000002 8fdb46bc 00000000 8fdb4284 00000000 804f1c80
1.855099] 9d40: ffffffff 00000001 8089ad70 ffffffff 8fdb43f8 ff8fb728 ff8fb728 ddf3fe37
1.863364] 9d60: 82173010 82173010 81383ea0 82173010 00000002 8219cac0 90a99d9c ddf3fe37
1.871627] 9d80: 80458940 00000000 82173010 81383ea0 82173010 00000002 8219cac0 8200f00d
1.879892] 9da0: 90a99dcc 90a99db0 80449f44 804bb664 00000000 82173010 81383ea0 82173010
1.888155] 9dc0: 90a99dec 90a99dd0 80447cb8 80449ee8 82173010 81383ea0 81383ea0 82173010
1.896420] 9de0: 90a99e04 90a99df0 80447ef0 80447b9c 815b034c 815b0350 90a99e2c 90a99e08
1.904684] 9e00: 80447f54 80447e28 81383ea0 90a99e84 82173010 00000001 00000000 8219cac0
1.912948] 9e20: 90a99e4c 90a99e30 80448320 80447f10 00000000 90a99e84 8044827c 00000001
1.921213] 9e40: 90a99e7c 90a99e50 80445dac 80448288 82173010 82098e6c 82575ab8 ddf3fe37
1.929478] 9e60: 90a99e7c 82173010 8137c268 82173054 90a99eac 90a99e80 80448158 80445d00
1.937742] 9e80: 90a99eac 82173010 00000001 ddf3fe37 82173010 8137c268 82173010 8200f000
```

Cause and generic information

Register content and CPU state

Stack content

Oops example (2/2)

```
 2.036894] Backtrace:
 2.039380]  miic_create from a5psw_probe+0x160/0x3e0
 2.044530]  r6:8fdb47d8 r5:82173010 r4:8259c840
 2.049194]  a5psw_probe from platform_probe+0x68/0xb8
 2.054432]  r10:8200f00d r9:8219cac0 r8:00000002 r7:82173010 r6:81383ea0 r5:82173010
 2.062332]  r4:00000000
 2.064896]  platform_probe from really_probe+0x128/0x28c
 2.070383]  r7:82173010 r6:81383ea0 r5:82173010 r4:00000000
 2.076095]  really_probe from __driver_probe_device+0xd4/0xe8
 2.082018]  r7:82173010 r6:81383ea0 r5:81383ea0 r4:82173010
 2.087730]  __driver_probe_device from driver_probe_device+0x50/0xd0
 2.094260]  r5:815b0350 r4:815b034c
 2.097874]  driver_probe_device from __device_attach_driver+0xa4/0xc4
 2.104499]  r9:8219cac0 r8:00000000 r7:00000001 r6:82173010 r5:90a99e84 r4:81383ea0
 2.112309]  __device_attach_driver from bus_for_each_drv+0xb8/0xd0
 2.118677]  r7:00000001 r6:8044827c r5:90a99e84 r4:00000000
 2.124388]  bus_for_each_drv from __device_attach+0xe0/0x158
 2.130226]  r6:82173054 r5:8137c268 r4:82173010
 2.134881]  __device_attach from device_initial_probe+0x1c/0x20
 2.140977]  r7:8200f000 r6:82173010 r5:8137c268 r4:82173010
 2.146689]  device_initial_probe from bus_probe_device+0x38/0x90
 2.152869]  bus_probe_device from deferred_probe_work_func+0x90/0xa8
 2.159408]  r7:8200f000 r6:8137c1ec r5:8137c188 r4:82173010
 2.165121]  deferred_probe_work_func from process_one_work+0x1ac/0x278
 2.171839]  r7:8200f000 r6:82006800 r5:8137c1b0 r4:82180000
 2.177550]  process_one_work from process_scheduled_works+0x38/0x3c
 2.184011]  r10:90841e94 r9:81399b20 r8:8200681c r7:80d03d00 r6:82006800 r5:82180018
 2.191913]  r4:82180000
 2.194478]  process_scheduled_works from worker_thread+0x23c/0x2d4
 2.200840]  r5:82180018 r4:82180000
 2.204455]  worker_thread from kthread+0xd4/0xdc
 2.209254]  r9:82180000 r8:80131078 r7:821e2150 r6:821e2080 r5:821e2140 r4:8219cac0
 2.217065]  kthread from ret_from_fork+0x14/0x2c
 2.221842] Exception stack(0x90a99fb0 to 0x90a99ff8)
 2.226958] 9fa0:                                     00000000 00000000 00000000 00000000
 2.235220] 9fc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
 2.243479] 9fe0: 00000000 00000000 00000000 00000000 00000013 00000000
 2.250170]  r10:00000000 r9:00000000 r8:00000000 r7:00000000 r6:00000000 r5:801373c4
 2.258074]  r4:821e2140 r3:00000001
 2.261704] Code: e3500000 0affffe6 e3a03000 e1a00004 (e5936050)
 2.267944] ---[ end trace 0000000000000000 ]---
```

Backtrace

Exception stack content

Kernel oops debugging: addr2line

- In order to convert addresses/symbol name from this display to source code lines, one can use addr2line
  - `addr2line -e vmlinux <address>`
- GNU binutils >= 2.39 takes the symbol+offset notation too:
  - `addr2line -e vmlinux <symbol_name>+<off>`
- The symbol+offset notation can be used with older binutils versions via the `faddr2line` script in the kernel sources:
  - `scripts/faddr2line vmlinux <symbol_name>+<off>`
- The kernel must have been compiled with `CONFIG_DEBUG_INFO=y` to embed the debugging information into the vmlinux file.

Kernel oops debugging: decode_stacktrace.sh

- addr2line decoding of oopses can be automated using decode_stacktrace.sh script which is provided in the kernel sources.
- This script will translate all symbol names/addresses to the matching file/lines and will display the assembly code where the crash did trigger.
- `./scripts/decode_stacktrace.sh vmlinux linux_source_path/ < oops_report.txt > decoded_oops.txt`
- NOTE: CROSS_COMPILE and ARCH env var should be set to obtain the correct disassembly dump.

Oops behavior configuration

- Sometimes, crash might be so bad that the kernel will panic and halt its execution entirely by stopping scheduling application and staying in a busy loop.
- Automatic reboot on panic can be enabled via `CONFIG_PANIC_TIMEOUT`
  - 0: never reboots
  - Negative value: reboot immediately
  - Positive value: seconds to wait before rebooting
- OOPS can be configured to always panic:
  - at boot time, adding `oops=panic` to the command line
  - at build time, setting `CONFIG_PANIC_ON_OOPS=y`

The Magic SysRq

The Magic SysRq

Functionality provided by serial drivers

- Allows to run multiple debug/rescue commands even when the kernel seems to be in deep trouble
  - On embedded: in the console, send a break character (Picocom: press `[Ctrl]` + `a` followed by `[Ctrl]` + `\`), then press `<character>`
  - By echoing `<character>` in `/proc/sysrq-trigger`
- Example commands:
  - `h`: show available commands
  - `s`: sync all mounted filesystems
  - `b`: reboot the system
  - `w`: shows the kernel stack of all sleeping processes
  - `t`: shows the kernel stack of all running processes
  - `g`: enter kgdb mode
  - `z`: flush trace buffer
  - `c`: triggers a crash (kernel panic)
  - You can even register your own!
- Detailed in `admin-guide/sysrq`

Built-in kernel self tests

Kernel memory issue debugging

- The same kind of memory issues that can happen in user space can be triggered while writing kernel code
  - Out of bounds accesses
  - Use-after-free errors (dereferencing a pointer after `kfree()`)
  - Out of memory due to missing `kfree()`
- Various tools are present in the kernel to catch these issues
  - *KASAN* to find use-after-free and out-of-bound memory accesses
  - *KFENCE* to find use-after-free and out-of-bound in production systems
  - *Kmemleak* to find memory leak due to missing free of memory

KASAN

- Kernel Address Space Sanitizer
- Allows to find use-after-free and out-of-bounds memory accesses
- Uses GCC to instrument the kernel at compile-time
- Supported by almost all architectures (ARM, ARM64, PowerPC, RISC-V, S390, Xtensa and X86)
- Needs to be enabled at kernel configuration with `CONFIG_KASAN`
- Can then be enabled for files by modifying Makefile
  - `KASAN_SANITIZE_file.o := y` for a specific file
  - `KASAN_SANITIZE := y` for all files in the Makefile folder

Kmemleak

- Kmemleak allows to find memory leaks for dynamically allocated objects with `kmalloc()`
  - Works by scanning the memory to detect if allocated address are not referenced anymore anywhere (large overhead).
- Once enabled with `CONFIG_DEBUG_KMEMLEAK`, kmemleak control files will be visible in *debugfs*
- Memory leaks is scanned every 10 minutes
  - can be disabled via `CONFIG_DEBUG_KMEMLEAK_AUTO_SCAN`
- An immediate scan can be triggered using
  - `# echo scan > /sys/kernel/debug/kmemleak`
- Results are displayed in debugfs
  - `# cat /sys/kernel/debug/kmemleak`
- See `dev-tools/kmemleak` for more information

Kmemleak report

```
# cat /sys/kernel/debug/kmemleak
unreferenced object 0x82d43100 (size 64):
  comm "insmod", pid 140, jiffies 4294943424 (age 270.420s)
  hex dump (first 32 bytes):
    b4 bb e1 8f c8 a4 e1 8f 8c ce e1 8f 88 c6 e1 8f  ................
    10 a5 e1 8f 18 e2 e1 8f ac c6 e1 8f 0c c1 e1 8f  ................
  backtrace:
    [<c31f5b59>] slab_post_alloc_hook+0xa8/0x1b8
    [<c8200adb>] kmem_cache_alloc_trace+0xb8/0x104
    [<1836406b>] 0x7f005038
    [<89fff56d>] do_one_initcall+0x80/0x1a8
    [<31d908e3>] do_init_module+0x50/0x210
    [<2658dd55>] load_module+0x208c/0x211c
    [<e1d48f15>] sys_finit_module+0xe4/0xf4
    [<1de12529>] ret_fast_syscall+0x0/0x54
    [<7ee81f34>] 0x7eca8c80
```

UBSAN

- UBSAN is a runtime checker for code with undefined behavior
  - Shifting with a value larger than the type
  - Overflow of integers (signed and unsigned)
  - Misaligned pointer access
  - Out of bound access to static arrays
  - https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
- It uses compile-time instrumentation to insert checks that will be executed at runtime
- Must be enabled using `CONFIG_UBSAN=y`
- Then, can be enabled for specific files by modifying Makefile
  - `UBSAN_SANITIZE_file.o := y` for a specific file
  - `UBSAN_SANITIZE := y` for all files in the Makefile folder

UBSAN: example of UBSAN report

- Report for an undefined behavior due to a shift with a value > 32.

```
UBSAN: Undefined behaviour in mm/page_alloc.c:3117:19
shift exponent 51 is too large for 32-bit type 'int'
CPU: 0 PID: 6520 Comm: syz-executor1 Not tainted 4.19.0-rc2 #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Bochs 01/01/2011
Call Trace:
__dump_stack lib/dump_stack.c:77 [inline]
dump_stack+0xd2/0x148 lib/dump_stack.c:113
ubsan_epilogue+0x12/0x94 lib/ubsan.c:159
__ubsan_handle_shift_out_of_bounds+0x2b6/0x30b lib/ubsan.c:425
...
RIP: 0033:0x4497b9
Code: e8 8c 9f 02 00 48 83 c4 18 c3 0f 1f 80 00 00 00 00 48 89 f8 48
89 f7 48 89 d6 48 89 ca 4d 89 c2 4d 89 c8 4c 8b 4c 24 08 0f 05 <48> 3d
01 f0 ff ff 0f 83 9b 6b fc ff c3 66 2e 0f 1f 84 00 00 00 00
RSP: 002b:00007fb5ef0e2c68 EFLAGS: 00000246 ORIG_RAX: 0000000000000010
RAX: ffffffffffffffda RBX: 00007fb5ef0e36cc RCX: 00000000004497b9
RDX: 0000000020000040 RSI: 0000000000000258 RDI: 0000000000000014
RBP: 000000000071bea0 R08: 0000000000000000 R09: 0000000000000000
R10: 0000000000000000 R11: 0000000000000246 R12: 00000000ffffffff
R13: 0000000000005490 R14: 00000000006ed530 R15: 00007fb5ef0e3700
```

Debugging locking

- Lock debugging: prove locking correctness
  - CONFIG_PROVE_LOCKING
  - Adds instrumentation to kernel locking code
  - Detect violations of locking rules during system life, such as:
    - Locks acquired in different order (keeps track of locking sequences and compares them).
    - Spinlocks acquired in interrupt handlers and also in process context when interrupts are enabled.
  - Not suitable for production systems but acceptable overhead in development.
  - See locking/lockdep-design for details
- CONFIG_DEBUG_ATOMIC_SLEEP allows to detect code that incorrectly sleeps in atomic section (while holding lock typically).
  - Warning displayed in dmesg in case of such violation.

Concurrency issues

- Kernel Concurrency SANitizer framework
- CONFIG_KCSAN, introduced in Linux 5.8.
- Dynamic race detector relying on compile time instrumentation.
- Can find concurrency issues (mainly data races) in your system.
- See dev-tools/kcsan and https://lwn.net/Articles/816850/ for details.

KGDB

kgdb - A kernel debugger

- CONFIG_KGDB in *Kernel hacking*.
- The execution of the kernel is fully controlled by gdb from another machine, connected through a serial line.
- Can do almost everything, including inserting breakpoints in interrupt handlers.
- Feature supported for the most popular CPU architectures
- CONFIG_GDB_SCRIPTS allows to build GDB python scripts that are provided by the kernel.
  - See dev-tools/gdb-kernel-debugging for more information

kgdb kernel config

- CONFIG_DEBUG_KERNEL=y to make KGDB support visible
- CONFIG_KGDB=y to enable KGDB support
- CONFIG_DEBUG_INFO=y to compile the kernel with debug info (-g)
- CONFIG_FRAME_POINTER=y to have more reliable stacktraces
- CONFIG_KGDB_SERIAL_CONSOLE=y to enable KGDB support over serial
- CONFIG_GDB_SCRIPTS=y to enable kernel GDB python scripts
- CONFIG_RANDOMIZE_BASE=n to disable KASLR
- CONFIG_WATCHDOG=n to disable watchdog
- CONFIG_MAGIC_SYSRQ=y to enable Magic SysReq support
- CONFIG_STRICT_KERNEL_RWX=n to disable memory protection on code section, thus allowing to put breakpoints

kgdb pitfalls

- KASLR should be disabled to avoid confusing gdb with randomized kernel addresses
  - Disable *kaslr mode using* `nokaslr` *command line parameter if enabled in your kernel.*
- Disable the platform watchdog to avoid rebooting while debugging.
  - When interrupted by KGDB, all interrupts are disabled thus, the watchdog is not serviced.
  - Sometimes, watchdog is enabled by upper boot levels. Make sure to disable the watchdog there too.
- Can not interrupt kernel execution from gdb using `interrupt` command or `Ctrl + C`.
- Not possible to break everywhere (see `CONFIG_KGDB_HONOUR_BLOCKLIST`).
- Need a console driver with polling support.
- Some architecture lacks functionalities (No watchpoints on arm32 for instance) and some instabilities might happen!

Using kgdb (1/2)

- Details available in the kernel documentation: dev-tools/kgdb
- You must include a kgdb I/O driver. One of them is kgdb over serial console (kgdboc: kgdb over console, enabled by CONFIG_KGDB_SERIAL_CONSOLE)
- Configure kgdboc at boot time by passing to the kernel:
  - `kgdboc=<tty-device>,<bauds>`.
  - For example: `kgdboc=ttyS0,115200`
- Or at runtime using sysfs:
  - `echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc`
  - If the console does not have polling support, this command will yield an error.

Using kgdb (2/2)

- Then also pass `kgdbwait` to the kernel: it makes `kgdb` wait for a debugger connection.
- Boot your kernel, and when the console is initialized, interrupt the kernel with a break character and then `g` in the serial console (see our *Magic SysRq* explanations).
- On your workstation, start `gdb` as follows:
  - `arm-linux-gdb ./vmlinux`
  - `(gdb) set remotebaud 115200`
  - `(gdb) target remote /dev/ttyS0`
- Once connected, you can debug a kernel the way you would debug an application program.
- On GDB side, the first threads represent the CPU context (ShadowCPU<x>), then all the other threads represents a task.

Kernel *GDB* scripts

- `CONFIG_GDB_SCRIPTS` allows to build a set of python script which ease the kernel debugging by adding new commands and functions.
- When using `gdb vmlinux`, the scripts present in vmlinux-gdb.py file at the root of build dir will be loaded automatically.
  - `lx-symbols`: (Re)load symbols for vmlinux and modules
  - `lx-dmesg`: display kernel dmesg
  - `lx-lsmod`: display loaded modules
  - `lx-device-{bus|class|tree}`: display device bus, classes and tree
  - `lx-ps`: `ps` like view of tasks
  - `$lx_current()` contains the current `task_struct`
  - `$lx_per_cpu(var, cpu)` returns a per-cpu variable
  - `apropos lx` To display all available functions.
- `dev-tools/gdb-kernel-debugging`

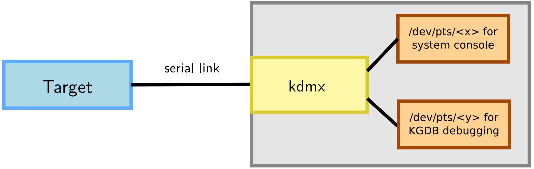
KDB

- CONFIG_KGDB_KDB includes a kgdb frontend name "KDB"
- This frontend exposes a debug prompt on the serial console which allows debugging the kernel without the need for an external gdb.
- KDB can be entered using the same mechanism used for entering kgdb mode.
- *KDB* and *KGDB* can coexist and be used at the same time.
  - Use the `kgdb` command in KDB to enter kgdb mode.
  - Send a maintenance packet from gdb using `maintenance packet 3` to switch from kgdb to KDB mode.

kdmx

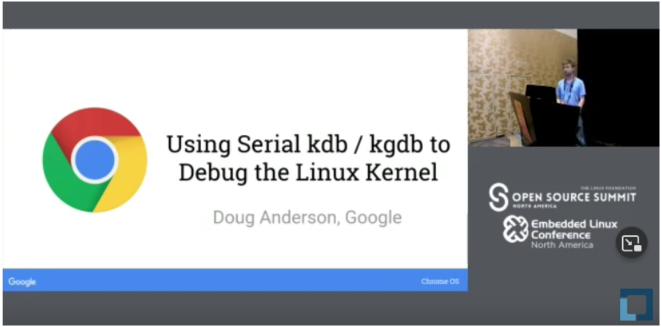
- When the system has only a single serial port, it is not possible to use both KGDB and the serial line as an output terminal since only one program can access that port.
- Fortunately, the *kdmx* tool allows to use both KGDB and serial output by splitting GDB messages and standard console from a single port to 2 slave pty (`/dev/pts/x`)
- https://git.kernel.org/pub/scm/utils/kernel/kgdb/agent-proxy.git
  - Located in the subdirectory `kdmx`

```
$ kdmx -n -d -p/dev/ttyACM0 -b115200
serial port: /dev/ttyACM0
Initalizing the serial port to 115200 8n1
/dev/pts/6 is slave pty for terminal emulator
/dev/pts/7 is slave pty for gdb

Use <ctrl>C to terminate program
```



Going further with KGDB

- Good presentation from Doug Anderson with a lot of demos and explanations
  - Video: https://www.youtube.com/watch?v=HBOwoSyRmys
  - Slides: https://elinux.org/images/1/1b/ELC19_Serial_kdb_kgdb.pdf



crash

- *crash* is a CLI tool allowing to investigate kernel (dead or alive!)
  - Uses /dev/mem or /proc/kcore on live systems
  - Requires `CONFIG_STRICT_DEVMEM=n`
- Can use a coredump generated using kdump, kvmdump, etc.
- Based on `gdb` and provides many specific commands to inspect the kernel state.
  - Stack traces, dmesg (`log`), memory maps of the processes, irqs, virtual memory areas, etc.
- Allows examining all the tasks that are running on the system.
- Hosted at https://github.com/crash-utility/crash

crash example

```
$ crash vmlinux vmcore
[...]
    TASKS: 75
NODENAME: buildroot
  RELEASE: 5.13.0
  VERSION: #1 SMP PREEMPT Tue Nov 15 14:42:25 CET 2022
  MACHINE: armv7l  (unknown Mhz)
  MEMORY: 512 MB
    PANIC: "Unable to handle kernel NULL pointer dereference at virtual address 00000070"
      PID: 127
  COMMAND: "watchdog"
    TASK: c3f163c0  [THREAD_INFO: c3f00000]
      CPU: 1
    STATE: TASK_RUNNING (PANIC)

crash> mach
    MACHINE TYPE: armv7l
     MEMORY SIZE: 512 MB
            CPUS: 1
 PROCESSOR SPEED: (unknown)
              HZ: 100
       PAGE SIZE: 4096
KERNEL VIRTUAL BASE: c0000000
KERNEL MODULES BASE: bf000000
KERNEL VMALLOC BASE: e0000000
KERNEL STACK SIZE: 8192
```

Post-mortem analysis

Kernel crash post-mortem analysis

- Sometimes, accessing the crashed system is not possible or the system can't stay offline while waiting to be debugged
- Kernel can generate crash dumps (a *vmcore* file) to a remote location, allowing to quickly restart the system while still be able to perform post-mortem analysis with GDB.
- This feature relies on *kexec* and *kdump* which will boot another kernel as soon as the crash occurs right after dumping the *vmcore* file.
  - The *vmcore* file can be saved on local storage, via SSH, FTP etc.

kexec & kdump (1/2)

- On panic, the kernel kexec support will execute a ”dump-capture kernel” directly from the kernel that crashed
  - Most of the time, a specific dump-capture kernel is compiled for that task (minimal config with specific initramfs/initrd)
- *kexec* system works by saving some RAM for the kdump kernel execution at startup
  - `crashkernel` parameter should be set to specify the crash kernel dedicated physical memory region
- *kexec-tools* are then used to load dump-capture kernel into this memory zone using the `kexec` command
  - Internally uses the `kexec_load` system call `man 2 kexec_load`

kexec & kdump (2/2)

- Finally, on panic, the kernel will reboot into the "dump-capture" kernel allowing the user to dump the kernel coredump (`/proc/vmcore`) onto whatever media
- Additional command line options depends on the architecture
- See admin-guide/kdump/kdump for more comprehensive explanations on how to setup the kdump kernel with `kexec`.
- Additional user-space services and tools allow to automatically collect and dump the vmcore file to a remote location.
  - See kdump systemd service and the `makedumpfile` tool which can also compress the vmcore file into a smaller file (Only for x86, PPC, IA64, S390).
  - https://github.com/makedumpfile/makedumpfile

kdump

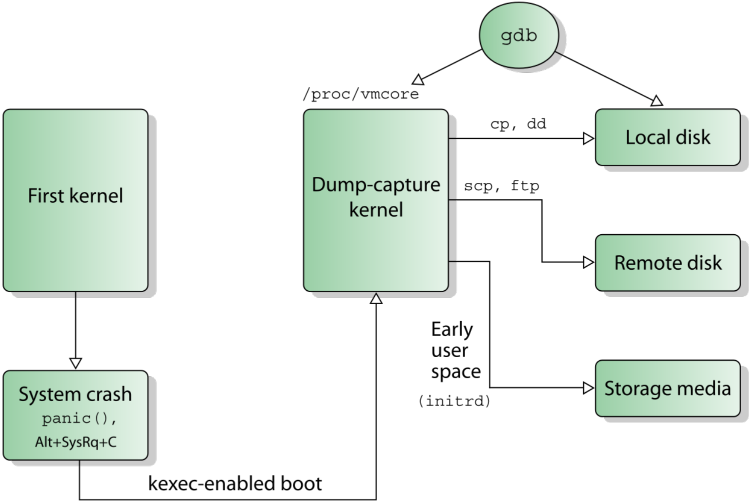


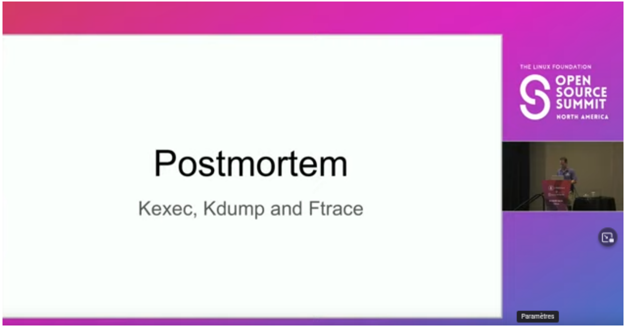
Image credits: Wikipedia

kexec config and setup

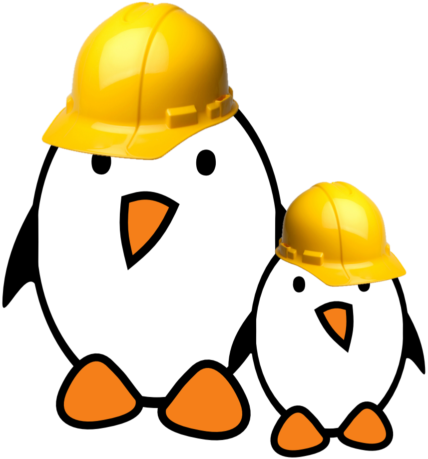
- On the standard kernel:
  - `CONFIG_KEXEC=y` to enable KEXEC support
  - `kexec-tools` to provide the `kexec` command
  - A kernel and a DTB accessible by `kexec`
- On the dump-capture kernel:
  - `CONFIG_CRASH_DUMP=y` to enable dumping a crashed kernel
  - `CONFIG_PROC_VMCORE=y` to enable `/proc/vmcore` support
  - `CONFIG_AUTO_ZRELADDR=y` on ARM32 platforms
- Set the correct `crashkernel` command line option:
  - `crashkernel=size[KMG][@offset[KMG]]`
- Load a dump-capture kernel on the first kernel with `kexec`:
  - `kexec --type zImage -p my_zImage --dtb=my_dtb.dtb --initrd=my_initrd --append="command line option"`
- Then simply wait for a crash to happen!

Going further with kexec & kdump

- Presentation from Steven Rostedt about using kexec, kdump and ftrace with lot of tips and tricks about using kexec/kdump
  - Video: https://www.youtube.com/watch?v=aUGNDJPpUUg
  - Slides: https://static.sched.com/hosted_files/ossna2022/c0/Postmortem_%20Kexec%2C%20Kdump%20and%20Ftrace.pdf



Practical lab - Kernel debugging



Debugging kernel crashes and driver problems

- Debug locking issues using lockdep
- Use kmemleak to detect memory leaks on the system
- Analyze an OOPS message
- Debug a crash with KGDB
- Setup kexec, kdump and extract a kernel coredump

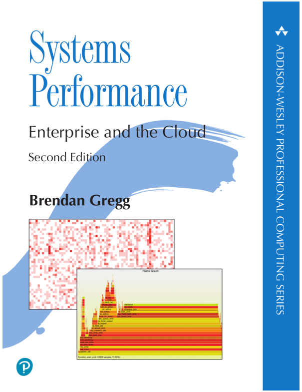
Going further

Debugging resources

- Brendan Gregg Systems performance book
- Brendan Gregg Linux Performance page
- *Tools and Techniques to Debug an Embedded Linux System*, talk from Sergio Prado, video, slides
- *Tracing with Ftrace: Critical Tooling for Linux Development*, talk from Steven Rostedt, video
- *Tutorial: Debugging Embedded Devices using GDB*, tutorial from Chris Simmonds, video

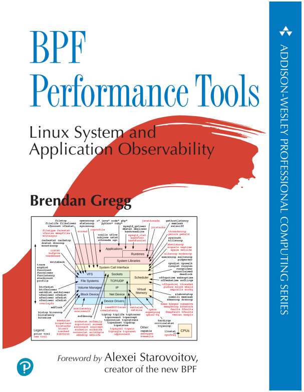
Going further (Tracing & Profiling)

- Great book from Brendan Gregg, an expert in tracing and profiling
- https://www.brendangregg.com/blog/2020-07-15/systems-performance-2nd-edition.html
- Covers concepts, strategy, tools, and tuning for Linux kernel and applications.

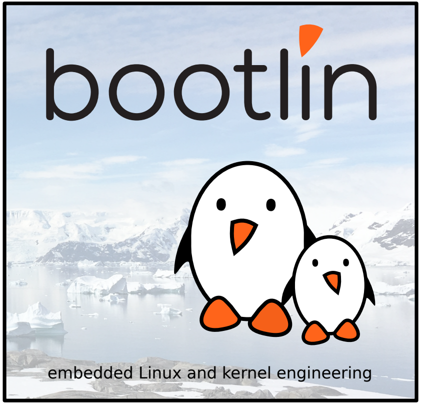


Going further (BPF)

- Still from Brendan Gregg!
- Covers more than 150 tools that uses BPF.
- Explains how to analyze the results from these tools to optimize your system.
- https://www.brendangregg.com/bpf-performance-tools-book.html



Last slides



Corrections, suggestions, contributions and translations are welcome!

Thank you!

And may the Source be with you

Rights to copy

Document sources: https://github.com/bootlin/training-materials/