



Linux debugging, profiling, tracing and performance analysis training

On-site training, 3 days

Latest update: May 08, 2024

Title	Linux debugging, profiling, tracing and performance analysis training
Training objectives	<ul style="list-style-type: none">• Be able to understand the main concepts of Linux that are relevant for performance analysis: process, threads, memory management, virtual memory, execution contexts, etc.• Be able to analyze why a system is loaded and what are the elements that contributes to this load using common Linux observability tools.• Be able to debug an userspace application using <i>gdb</i>, either live or after a crash, and analyze the contents of ELF binaries.• Be able to trace and profile a complete userspace application and its interactions with the Linux kernel in order to fix bugs using <i>strace</i>, <i>ltrace</i>, <i>perf</i> or <i>Callgrind</i>.• Be able to understand classical memory issues and analyze them using <i>valgrind</i>, <i>libefence</i> or <i>Massif</i>.• Be able to trace and profile the entire Linux system, using <i>perf</i>, <i>ftrace</i>, <i>kprobes</i>, <i>eBPF</i> tools, <i>kernelshark</i> or <i>LTTng</i>• Be able to debug Linux kernel issues: debug kernel crashes live or post-mortem, analyze memory issues at the kernel level, analyze locking issues, use kernel-level debuggers.
Duration	Three days - 24 hours (8 hours per day)
Pedagogics	<ul style="list-style-type: none">• Lectures delivered by the trainer: 40% of the duration• Practical labs done by participants: 60% of the duration• Electronic copies of presentations, lab instructions and data files. They are freely available at https://bootlin.com/doc/training/debugging.
Trainer	One of the engineers listed on: https://bootlin.com/training/trainers/
Language	Oral lectures: English, French. Materials: English.
Audience	Companies and engineers interested in debugging, profiling and tracing Linux systems and applications, to analyze and address performance or latency problems.



Prerequisites	<ul style="list-style-type: none">• Knowledge and practice of UNIX or GNU/Linux commands: participants must be familiar with the Linux command line. Participants lacking experience on this topic should get trained by themselves, for example with our freely available on-line slides at bootlin.com/blog/command-line/.• Minimal experience in embedded Linux development: participants should have a minimal understanding of the architecture of embedded Linux systems: role of the Linux kernel vs. user-space, development of Linux user-space applications in C. Following Bootlin's <i>Embedded Linux</i> course at bootlin.com/training/embedded-linux/ allows to fulfill this pre-requisite.• Minimal English language level: B1, according to the <i>Common European Framework of References for Languages</i>, for our sessions in English. See bootlin.com/pub/training/cefr-grid.pdf for self-evaluation.
Required equipment	<ul style="list-style-type: none">• Video projector• One PC computer on each desk (for one or two persons) with at least 8 GB of RAM, and Ubuntu Linux 22.04 installed in a free partition of at least 30 GB• Distributions other than Ubuntu Linux 22.04 are not supported, and using Linux in a virtual machine is not supported.• Unfiltered and fast connection to Internet: at least 50 Mbit/s of download bandwidth, and no filtering of web sites or protocols.• PC computers with valuable data must be backed up before being used in our sessions.
Certificate	Only the participants who have attended all training sessions, and who have scored over 50% of correct answers at the final evaluation will receive a training certificate from Bootlin.
Disabilities	Participants with disabilities who have special needs are invited to contact us at training@bootlin.com to discuss adaptations to the training course.



Hardware in practical labs

The hardware platform used for the practical labs of this training session is the **STMicroelectronics STM32MP157D-DK1 Discovery board** board, which features:

- STM32MP157D (dual Cortex-A7) CPU from STMicroelectronics
- USB powered
- 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
- Arduino Uno v3-compatible headers
- Audio codec
- Misc: buttons, LEDs



Day 1 - Morning

Lecture - Linux application stack

- Global picture: understanding the general architecture of a Linux system, overview of the major components.
- What is the difference between a process and a thread, how applications run concurrently.
- ELF files and associated analysis tools.
- Userspace application memory layout (heap, stack, shared libraries mappings, etc).
- MMU and memory management: physical/virtual address spaces.
- Kernel context switching and scheduling
- Kernel execution contexts: kernel threads, workqueues, interrupt, threaded interrupts, softirq



Lecture - Common analysis & observability tools

- Analyzing an ELF file with GNU binary utilities (*objdump*, *addr2line*).
- Tools to monitor a Linux system: processes, memory usage and mapping, resources.
- Using *vmstat*, *iostat*, *ps*, *top*, *iotop*, *free* and understanding the metrics they provide.
- Pseudo filesystems: *procfs*, *sysfs* and *debugfs*.

Day 1 - Afternoon

Lab - Check what is running on a system and its load

- Observe running processes using *ps* and *top*.
- Check memory allocation and mapping with *procfs* and *pmap*.
- Monitor other resources usage using *iostat*, *vmstat* and *netstat*.

Lecture - Debugging an application

- Using *gdb* on a live process.
- Understanding compiler optimizations impact on debuggability.
- Postmortem diagnostic using core files.
- Remote debugging with *gdbserver*.
- Extending *gdb* capabilities using python scripting

Lab - Solving an application crash

- Analysis of compiled C code with compiler-explorer to understand optimizations.
- Managing *gdb* from the command line, then from an IDE.
- Using *gdb* Python scripting capabilities.
- Debugging a crashed application using a core dump with *gdb*.



Day 2 - Morning

Lecture - Tracing an application

- Tracing system calls with *strace*.
- Tracing library calls with *ltrace*.
- Overloading library functions using *LD_PRELOAD*.

Lab – Debugging application issues

- Analyze dynamic library calls from an application using *ltrace*.
- Overloading library functions using *LD_PRELOAD*.
- Analyzing an application system calls using *strace*.

Lecture - Memory issues

- Usual memory issues: buffer overflow, segmentation fault, memory leaks, heap-stack collision.
- Memory corruption tooling, *valgrind*, *libefence*, etc.
- Heap profiling using *Massif* and *heaptrack*

Lab – Debugging memory issues

- Memory leak and misbehavior detection with *valgrind* and *vgdb*.
- Visualizing application heap using *Massif*.

Day 2 - Afternoon

Lecture – Application profiling

- Performances issues.
- Gathering profiling data with *perf*.
- Analyzing an application callgraph using *Callgrind* and *KCachegrind*.
- Interpreting the data recorded by *perf*.

Lab - Application profiling

- Profiling an application with *Callgrind/KCachegrind*.
- Analyzing application performance with *perf*.
- Generating a flamegraph using *FlameGraph*.



Day 3 - Morning

Lecture - System wide profiling and tracing

- System wide profiling using *perf*.
- Using *kprobes* to hook on kernel code without recompiling.
- *eBPF* tools (*bcctools*, *bpfftrace*, etc) for complex tracing scenarios.
- Application and kernel tracing and visualization using *ftrace*, *kernelshark* or *LTTng*

Lab - System wide profiling and tracing

- System profiling with *perf*.
- IRQ latencies using *ftrace*.
- Tracing and visualizing system activity using *kernelshark* or *LTTng*

Day 3 - Afternoon

Lecture - Kernel debugging

- Kernel compilation results (*vmlinux*, *System.map*).
- Understanding and configuring kernel *oops* behavior.
- Post mortem analysis using kernel crash dump with *crash*.
- Memory issues (*KASAN*, *UBSAN*, *Kmemleak*).
- Debugging the kernel using *KGDB* and *KDB*.
- Kernel locking debug configuration options (*lockdep*).
- Other kernel configuration options that are useful for debug.

Lab - Kernel debugging

- Analyzing an *oops* after using a faulty module with *objdump* and *addr2line*.
- Debugging a deadlock problem using *PROVE_LOCKING* options.
- Detecting undefined behavior with *UBSAN* in kernel code.
- Find a module memory leak using *kmemleak*.
- Debugging a module with *KGDB*.