Linux debugging, profiling, tracing and performance analysis training

Practical Labs

bootlin

https://bootlin.com

November 25, 2022

About this document

Updates to this document can be found on https://bootlin.com/doc/training/debugging.

This document was generated from LaTeX sources found on https://github.com/bootlin/training-materials.

More details about our training sessions can be found on https://bootlin.com/training.

Copying this document

Corrections, suggestions, contributions and translations are welcome!

Training setup

Download files and directories used in practical labs

Install lab data

For the different labs in this course, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Download and extract its tarball from a terminal:

```
$ cd
$ wget https://bootlin.com/doc/training/debugging/debugging-labs.tar.xz
$ tar xvf debugging-labs.tar.xz
```

Lab data are now available in an `debugging-labs` directory in your home directory. This directory contains directories and files used in the various practical labs. It will also be used as working space, in particular to keep generated files separate when needed.

Update your distribution

To avoid any issue installing packages during the practical labs, you should apply the latest updates to the packages in your distro:

```
$ sudo apt update
$ sudo apt dist-upgrade
```

You are now ready to start the real practical labs!

Install extra packages

Feel free to install other packages you may need for your development environment. In particular, we recommend to install your favorite text editor and configure it to your taste. The favorite text editors of embedded Linux developers are of course *Vim* and *Emacs*, but there are also plenty of other possibilities, such as Visual Studio Code[1], *GEdit*, *Qt Creator*, *CodeBlocks*, *Geany*, etc.

It is worth mentioning that by default, Ubuntu comes with a very limited version of the `vi` editor. So if you would like to use `vi`, we recommend to use the more featureful version by installing the `vim` package.

More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.

[1] This tool from Microsoft is Open Source! To try it on Ubuntu: `sudo snap install code --classic`

- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the `root` user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.
- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give the new files back to your regular user.
  Example: `$ sudo chown -R myuser.myuser linux/`

Preparing the system

Objectives:

- *Prepare the STM32MP157D board*

Building the image

We created a special image for the training. This image will contain everything we need (tools, configured kernel, etc). This image will be built with buildroot which allows to build a complete image for embedded systems.

```
$ cd /home/<user>/debugging-labs/
$ git clone https://github.com/bootlin/buildroot
$ cd buildroot
$ git checkout debugging-training/2022.08
$ make stm32mp157a_dk1_debugging_defconfig
$ make -j12
```

This will take a few minutes. At the end of the build, a images directory will contain the images that can be used on the board. During this course, we will use a kernel located on a sdcard and a rootfs via NFS. This will let us transfer data freely from and to the target board.

The rootfs should be extracted at `/home/<user>/debugging-labs/nfsroot` using this command:

```
$ tar xvf output/images/rootfs.tar -C /home/<user>/debugging-labs/nfsroot
```

We will also export the `CROSS_COMPILE` variable to set the toolchain as our cross compiling toolchain:

```
$ export CROSS_COMPILE=/home/<user>/debugging-labs/buildroot/output/host/bin/\
    arm-linux-
```

This export needs to be either done in each shell in which `CROSS_COMPILE` is going to be used or added to your shell configuration (`.bashrc` for instance)

Prepare the STM32MP1 Discovery Kit 1

The STM32M1 Discovery Kit 1 is powered via a USB-C cable, which you need to connect to the `CN6` (also labeled `PWR_IN`) connector.

In addition, to access the debug serial console, you need to use a micro-USB cable connected to the `CN11` (also labeled `ST-LINK`) connector.

Once your micro-USB cable is connected, a `/dev/ttyACM0` device will apear on your PC. You can see this device appear by looking at the output of `dmesg` on your workstation.

To communicate with the board through the serial port, install a serial communication program, such as `picocom`:


```
sudo apt install picocom
```

If you run `ls -l /dev/ttyACM0`, you can also see that only `root` and users belonging to the `dialout` group have read and write access to this file. Therefore, you need to add your user to the `dialout` group:

```
sudo adduser $USER dialout
```

Important: for the group change to be effective, in Ubuntu 18.04, you have to *completely reboot* the system [2]. A workaround is to run `newgrp dialout`, but it is not global. You have to run it in each terminal.

Now, you can run `picocom -b 115200 /dev/ttyACM0`, to start serial communication on `/dev/ttyACM0`, with a baudrate of `115200`. If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

There should be nothing on the serial line so far, as the board is not powered up yet.

Set up the Ethernet communication on the workstation

With a network cable, connect the Ethernet port of your board to the one of your computer. If your computer already has a wired connection to the network, your instructor will provide you with a USB Ethernet adapter. A new network interface should appear on your Linux system.

Find the name of this interface by typing:

```
$ ip a
```

The network interface name is likely to be `enxxx`[3]. If you have a pluggable Ethernet device, it's easy to identify as it's the one that shows up after pluging in the device.

Then, instead of configuring the host IP address from NetWork Manager's graphical interface, let's do it through its command line interface, which is so much easier to use:

```
$ nmcli con add type ethernet ifname en<xxx> ip4 192.168.0.1/24
```

Setting up the NFS server

Install the NFS server by installing the `nfs-kernel-server` package. Once installed, edit the `/etc/exports` file as `root` to add the following lines, assuming that the IP address of your board will be `192.168.0.100`:

```
/home/<user>/debugging-labs/nfsroot 192.168.0.100(rw,no_root_squash,no_subtree_check)
```

Of course, replace `<user>` by your actual user name.

Make sure that the path and the options are on the same line. Also make sure that there is no space between the IP address and the NFS options, otherwise default options will be used for this IP address, causing your root filesystem to be read-only.

Then, restart the NFS server:

[2] As explained on https://askubuntu.com/questions/1045993/after-adding-a-group-logoutlogin-is-not-enough-in-18-04/.

[3] Following the *Predictable Network Interface Names* convention: https://www.freedesktop.org/wiki/Software/systemd/PredictableNetworkInterfaceNames/


```
$ sudo exportfs -r
```

If there is any error message, this usually means that there was a syntax error in the `/etc/exports` file. Don't proceed until these errors disappear.

Flashing the sdcard

In order to get a working board, you will need to flash a sdcard with the `output/images/sdcard.img` file. Plug your sdcard on your computer and check on which `/dev/sdX` it has been mounted (you can use the `dmesg` command to check that). For instance, if the sdcard has been mounted on `/dev/sde`, use the following command:

```
$ sudo dd if=output/images/sdcard.img of=/dev/sde
$ sync
```

Once flashed, plug the sdcard onto the STM32MP157D board and reboot the board.

U-Boot setup

In order to use a rootfs on NFS, we will use an external rootfs. This can be specified by passing bootargs to the kernel. To do so, we are going to set the `bootargs` U-Boot variable and save the environment.

```
STM32MP1> env set bootargs root=/dev/nfs ip=192.168.0.100:::::eth0
  nfsroot=192.168.0.1:/home/<user>/debugging-labs/nfsroot/,nfsvers=3,tcp rw
STM32MP1> saveenv
```

System status

Objectives:

- *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*.*

Observe system status

In order to examine the platform, you can either execute commands through the `picocom` terminal or open a SSH connection.

Now that the board is up and running, let's try to understand what is running on this system. The provided image includes numerous tools to analyze the system. Try to answer the following questions using the commands that were presented during the course:

- How many CPU does this processor have ?
- What are the memory maps used by the `dropbear` process ?
- How much PSS memory is used by `dropbear` ?
- What is the amount of memory available for applications on the system ?
- How much unused memory is left on our system ?
- Is there swapped memory ?
- Is there an application using to much CPU ?
- How much time is spent by CPU0 in system mode (kernel) ?
- Is there some IOs ongoing with storage devices ?
- How much Mbytes/s are transferred to the MMC card ?
- What is the process generating transfers to the MMC ?
- Which processor receive most of the interrupts ?
- How many interrupts were received from the MMC controller ?
- How many TCP sockets are currently in ESTABLISHED mode ? and in LISTEN mode ?
- Which foreign port is used by the host for an incoming SSH connection ?

Once found, you can remove the files `/etc/init.d/S25stress-ng` and `/etc/init.d/S26mmc-reader` and reboot to have a cleaner system.

Solving an application crash

Objectives:

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

Compiler explorer

Go to `https://godbolt.org/` and paste the content of `swap_bytes.c`. Select the correct compiler for armv7 and observe the generated assembly. Try to modify the compiler options to optimize the generation (-O3). Observe the result.

Using GDB

Take our `linked_list.c` program. It uses the `<sys/queue.h>` header which provides multiple linked-list implementations. This program creates and fill a linked list with the names read from a file. Compile it using the following command:

```
$ cd /home/<user>/debugging-labs/nfsroot/root/gdb/
$ make
```

By default, it will look for a `word_list` file located in the current directory. This program should display the list of words that were read from the file.

```
$ ./linked_list
```

From what you can see, it actually crashes ! So we will use GDB to debug that program. We will do that remotely since our target does not embed a full gdb, only a gdbserver, a lightweight gdb server that allows connecting with a remote full feature GDB. Start our program using gdbserver in multi mode:

```
$ gdbserver --multi :2000 ./linked_list
```

On the host side install gdb-multiarch if not already done and attach to this process using gdb-multiarch:

```
$ sudo apt install gdb-multiarch
$ gdb-multiarch ./linked_list
(gdb) target extended-remote 192.168.0.100:2000
(gdb) set sysroot /home/<user>/debugging-labs/buildroot/output/staging/
```


Then continue the execution and try to find the error using GDB. There are multiple ways to debug such program. We will track down up to the error in order to understand

TIP: you can execute command automatically at GDB startup by putting them in a `~/.gdbinit` *file. For instance, history can be enabled with* `set history save on` *and pretty printing of structure with* `set print pretty on`.

TIP: GDB features a TUI which can be spawned using Ctrl + X + A. You can switch from the command line to the TUI view using Ctrl X + O.

TIP: in gdb, not only values can be displayed using `p` *command but functions can also be called directly from gdb ! Try to call* `display_linked_list()`.

NOTE: you can exit gdbserver from the connected gdb process using the `monitor exit` *command.*

Using a coredump with GDB

Sometimes, the problems only arise in production and you can only gather data once the application crashed. This is also something that can be used if the crash is not reproducible but crashes only once in a while. If so, we can use the kernel coredump support to generate a core dump of the faulty application and do a post-mortem analysis.

First of all, we need to enable kernel coredumping support of programs:

```
$ ulimit -c unlimited
```

Then, run the program normally:

```
$ ./linked_list
Segmentation fault (core dumped)
```

When crashing, a `core` file will be generated. Copy this file from the NFS directory on you desktop computer using `gdb-multiarch`:

```
$ gdb-multiarch <program_binary> <coredump_file>
```

You can then inspect the program state (memory, registers, etc) at the time it crashed. While less dynamic, it allows to pinpoint the place that triggered the crash.

GDB Python support

When developping and debugging applications, sometimes we often uses the same set of commands over and over under GDB. Rather than doing so, we can create python scripts that are integrated with GDB.

In order to display our program list from GDB, we provide a python GDB script named `linked_list.py` that displays this list. You will need to fill two parts of this script to display a complete list correctly. This python script takes the list head name and the next field name as parameters.

The part to be filled in are the pretty printer struct formatting and the iteration on the list. We would like to display each `struct name` as `index: name`. In order to access a struct field in gdb python, you can use `self.val['field_name']`.

Once done, you can use the following commands to test your script:


```
(gdb) source linked_list.py
(gdb) printslist name_list next
```

Debugging application issues

Objectives:

- *Analyze dynamic library calls from an application using* ltrace*.*
- *Using* strace *to analyze program syscalls.*

ltrace

On your computer, go into the `ltrace` lab folder:

```
$ cd /home/<user>/debugging-labs/nfsroot/root/ltrace/
$ make
```

From there, run the `traceme` application on the target:

```
$ cd /root/ltrace
$ export LD_LIBRARY_PATH=$PWD
$ ./authent
Error: failed to authenticate the user !
```

As you can see, it seems our application is failing to correctly authenticate the system. Using *ltrace*, trace the application in order to understand what is going on.

```
$ ltrace ./authent
```

From that trace, try to find which function fails.

In order to overload this check, we can use a `LD_PRELOAD` a library. We'll override the `al_authent_user()` based on the `authent_library.h` definitions. Create a file `overload.c` which override the `al_authent_user()`, prints the user, password and returns 0. Compile it using the following command line:

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

Finally, run your application and preload the new library using the following command:

```
$ LD_PRELOAD=./overload.so ./authent
```

strace

`strace` is useful to debug an application when you don't have the source. For that example, use the `strace_me` binary that is present in on the target in `/root/strace` and run it with strace:

```
$ cd /root/strace
$ strace ./strace_me
```

Based on the output and running strace with other options, try to answer the following questions:

- What are the files that are opened by this binary ?
- How many time is `read()` called ?
- Which `openat` system call failed ?
- How many system calls are issued by the program ?

Debugging memory issues

Objectives:

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

valgrind & vgdb

Go into the `valgrind folder` and compile `valgrind.c` with debugging information using:

```
$ cd /home/<user>/debugging-labs/nfsroot/root/valgrind
$ make
```

Then run it on the target. Do you notice any problem ? Does it run correctly ? Even though there is no segfault, an application might have some memory leaks or even out-of-bounds accesses, uninitialized memory, etc.

Now, run the command again with valgrind using the following command:

```
$ valgrind --leak-check=full ./valgrind
```

You'll see various errors found by valgrind

- Invalid memory write
- Uninitialized memory
- Memory leaks

In order to pinpoint exactly each error and be able to disable with gdb, vgdb can be used. We will do that remotely on the host using gdb-multiarch. First, we need to run valgrind with vgdb enabled on the target:

```
$ cd /root/valgrind
$ valgrind --vgdb-error=0 --leak-check=full ./valgrind
```

Then, in order to do remote debugging, we also need to run vgdb in listen mode. Start another terminal in SSH on the target and run the following command:

```
$ vgdb --port=1234
```

On the computer side, start gdb-multiarch and give it the `valgrind` binary which will allow to detect the architecture and read symbols:

```
$ gdb-multiarch ./valgrind
```

Finally, we'll need to connect to vgdb using the following gdb command:

```
(gdb) target remote <target_ip>:1234
```


You will then be able to debug each error using gdb and valgrind will interrupt the program each time it detects an error. Try to solve all the problems that were encountered by valgrind.

NOTE: The backtrace for leaks is not shown on the target because all libraries are stripped and thus do not have any debugging symbols anymore. This leads to the impossibility to use the dwarf information for backtracing.

Application profiling

Objectives:

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

Massif

Massif is really helpful to understand what is going on the memory allocation side of an application. Compile the `heap_profile` example that we did provide using the following command:

```
$ cd /home/<user>/debugging-labs/nfsroot/root/heap_profile
$ make
```

Once compile, on the target run it under massif using the following command:

```
$ cd /root/heap_profile
$ valgrind --tool=massif --time-unit=B ./heap_profile
```

NOTE: we use `--time-unit=B` to set the X axis to be based on the allocated size.

Once done, a `massif.out.<pid>` file will be created. This file can be displayed with `ms_print`. Based on the result, can you answer the following questions:

- What is the peak allocation size of this program ?
- How much memory was allocated during the program lifetime ?
- Do we have memory leaks at the end of execution ?

Note: `heaptrack` is not available on buildroot but is available on debian. You can try the same experience using heaptrack on your computer and visualizing the results with `heaptrack_gui`.

Cachegrind & Callgrind

Cachegrind and Callgrind allows to profile a userspace application by simulating the processor that will run it. In order to analyze our application and understand were the time is spent, we are going to profile it with both tools.

In order to profile the application using the `callgrind` tool. Our program takes two parameters, an input png and an output one. We provided a `tux_small.png` which can be used as an input file. First let's compile it using the following commands:

```
$ cd /home/<user>/debugging-labs/nfsroot/root/app_profiling
$ make
```


We are going to profile cache usage using Cachegrind with the following command:

```
$ valgrind --tool=cachegrind ./png_convert tux_small.png out.png
```

The execution will take some times and a `cachegrind.out.<pid>` will be generated. Analyze the results with `Kcachegrind` in order to understand the function that generates most of the D cache miss time.

Based on that result, modify the program to be more cache efficient. Run again the cachegrind analysis to check that the modifications were actually effective.

We also profile the execution time using callgrind with

```
$ valgrind --tool=callgrind ./png_convert tux_small.png out.png
```

Again, analyze the results using `Kcachegrind`. This time, the view is different and allow to display all the call graphs

Looking at the results, it seems like our conversion function is actually taking a negligible time. However, valgrind simulate the program with an "ideal" cache. In real life, the processor is often used by other applications and the kernel also takes some time to execute which leads to cache disturbance. Hence, callgrind is a good tool to optimize applications based on CPU time

Perf

In order to have a better view of the performances of our program in a real system, we will use `perf`. First of all, we will record our program execution using the `perf record` command.

```
$ perf record ./png_convert tux_small.png out.png
```

Once recorded, a `perf.data` file will be generated. This file will contain the traces that have been recorded. These traces can be analyzed using `perf report`. You will quickly notice that the output is not the same as valgrind because it displays a time spent per function (excluding function calls inside them). This allows to find which function takes most of the execution time. In order to compare this output to the valgrind one, we can run perf and also record the callgraph using the `--call-graph` option.

```
$ perf record --call-graph dwarf ./png_convert tux_small.png out.png
```

We specify that we want to record the call graph using the DWARF information that are contained in ELF file (compiled with `-g`). Once recorded, display the results with `perf report` and compare them with callgrind ones:

```
$ perf report --symfs=/home/<user>/debugging-labs/buildroot/output/staging/
  -k /home/<user>/debugging-labs/nfsroot/root/vmlinux ./png_convert
  tux_small.png out.png
```

System wide profiling and tracing

Objectives:

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

IRQ latencies using *ftrace*

ftrace features a specific tracer for irq latency which is named irqsoff. Using this tracer, analyze the system irqs latency. Run the following command for a few seconds and hit `Ctrl + [C]` to stop it.

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

Once done, you can visualize which section of code generated too much latency by using:

```
$ trace-cmd report
```

This will display a trace of the longest chain of calls while interrupts were disabled. Based on this report, can you find the code that generates this latency ?

ftrace & uprobes

First of all, we will start a small program using the following command:

```
$ mystery_program 1000 200 2 &
```

In order to trace a full system, we can use ftrace. However, if we want to trace the userspace, we'll need to add new tracepoints using uprobes. This can be done manually with the uprobe sysfs interface or using `perf probe`.

Before starting to profile, we will compile our program to be instrumented:

```
$ cd /home/<user>/debugging-labs/nfsroot/root/system_profiling
$ make
```

On the target, we will create a uprobe in the main function of the `crc_random` program each time a crc is computed. First, let list the lines number that are recognized by perf to add a uprobe:

```
$ perf probe --source=./ -x ./crc_random -L main
```


Note: in order to be able to add such userspace probe, perf needs to access symbols and source file

Then, we can create a uprobe and capture the crc value using:

```
$ perf probe --source=./ -x ./crc_random main:35 crc
```

We can finally trace the application using trace-cmd with this event

```
$ trace-cmd record -e probe_crc_random:main_L35 ./crc_random
^C
```

Then, using kernelshark tool on the host, analyze the trace:

```
$ sudo apt install kernelshark
$ kernelshark
```

We can see that something is wrong, our process does not seems to compute crc at a fixed rate. Let's trace the `sched_switch` *events to see whats happening:*

```
$ trace-cmd record -e sched_switch -e probe_crc_random:main_L35 ./crc_random
^C
```

Reanalyze the traces with kernel shark and try to understand what is going on.

LTTng

In order to observe our program performances, we want to instrument it with tracepoints. We would like to know how much times it takes to compute the crc32 of a specific buffer.

In order to do so, add tracepoints to your program which will allows to measure this. We'll add 2 tracepoints:

- *One for the start of crc32 computation (*`compute_crc_start`*) without any arguments.*
- *Another for the end of crc32 computation (*`compute_crc_end`*) with a crc argument that will be displayed as an hexadecimal integer.*

For that, create a tracepoint provider header file template named `crc_random-tp.h` *and another one for the tracepoint provider named* `crc_random-tp.c`*. These tracepoints should belong to the* `crc_random` *provider namespace.*

You can then use the new tracepoints in your program to trace specific points of execution. Once added, you can compile your application using the following command:

```
$ $(CROSS_COMPILE)-gcc -I. crc_random-tp.c crc_random.c -llttng-ust -o crc_random
```

Finally, on the target, enable the program tracepoints, run it and collect tracepoints. We are going to do that remotely using the `lttng-relayd` *tool on the remote computer:*

```
$ sudo apt install lttng-tools
$ lttng-relayd --output=$PWD/traces
```

Then on the target, start the trace acquisition using:

```
$ lttng-sessiond --daemonize
```

```
$ lttng create crc_session --set-url=net://192.168.0.1
$ lttng enable-event --userspace crc_random:compute_crc_start
$ lttng enable-event --userspace crc_random:compute_crc_end
$ lttng enable-event --kernel sched_switch
$ lttng start
$ ./crc_random
$ lttng destroy
```

Once finished, the traces will be visible in `$PWD/traces/<hostname>/<session>` *on the remtoe computer. In our case, the hostname is buildroot so traces will be located in* `$PWD/traces/buildroot/<session>`

Using `babeltrace2`*, you can display the raw traces that were acquired:*

```
$ babeltrace2 $PWD/traces/buildroot/<session>/
```

In order to analyze our traces more visually, we are going to use tracecompass. Download `tracecompass` *latest version and extract it using:*

```
$ wget https://ftp.fau.de/eclipse/tracecompass/releases/8.1.0/rcp/\
    trace-compass-8.1.0-20220919-0815-linux.gtk.x86_64.tar.gz
$ tar -xvf trace-compass-*.tar.gz
```

Run it

```
$ cd trace-compass*
$ ./tracecompass
```

We are going to merge the kernel and the user traces in tracecompass. To do so, open the traces using the `File -> Open Trace...` *menu and open the traces by loading both the* `kernel` *(*`kernel/channel0_0`*) and the* `ust` *(*`ust/uid/0/32-bit/channel0_0`*) folders.*

Once opened, in the left pane, expand the `Tracing` *item and right click on* `Experiments[0]`*. Select* `New`*, name the new experiment* `debugging_lab`*. Then expand the* `Experiments[1]` *item and right click on the* `debugging_lab` *one and click on* `Select traces...`*, In the new window, check the 32-bit box and the kernel one, then click on* `Finish`*. Finally, double click on the* `debugging_lab[2]` *item to display the merged trace. Explore the interface, and try to follow the task execution on both the Resources view and in the Control flow one.*

System profiling with *perf*

In order to profile the whole system, we are going to use perf and try to find the function that takes most of the time executing.

First of all, we will run a global recording of functions and their backtrace on (all CPUs) during 10 seconds using the following command:

```
$ perf record -F 99 -g -- sleep 10
```

This command will generate a `perf.data` *file that can be used on a remote computer for further analysis. Copy that file and fix the permissions using* `chown`*:*

```
$ sudo cp /home/<user>/debugging-labs/nfsroot/root/system_profiling/perf.data .
$ sudo chown <user>:<user> perf.data
```

We will then use perf report to visualize the aquired data:


```
$ perf report --symfs=/home/<user>/debugging-labs/buildroot/output/staging/
  -k /home/<user>/debugging-labs/nfsroot/root/vmlinux
```

Another useful tool for performance analysis is flamegraphs. Latest perf version includes a builtin support for flamegraphs but the template is not available on debian so we will use another support provided by Brendan Gregg scripts.

```
$ git clone https://github.com/brendangregg/FlameGraph.git
$ perf script | ./FlameGraph/stackcollapse-perf.pl | ./FlameGraph/flamegraph.pl >\
     flame.html
```

Using this flamegraph, analyze the system load.

Kernel debugging

Objectives:

- *Debugging a deadlock problem using* PROVE_LOCKING *options.*
- *Find a module memory leak using* kmemleak*.*
- *Analyzing an* oops *with* addr2line*.*
- *Debugging with* KGDB*.*
- *Setting up* Kexec & kdump*.*

Locking problems

CONFIG_PROVE_LOCKING *has been enabled in the provided kernel image. First, compile the module using the following command line:*

```
$ cd /home/<user>/debugging-labs/nfsroot/root/locking
$ export CROSS_COMPILE=/home/<user>/debugging-labs/buildroot/output/host/bin/\
    arm-linux-
$ export ARCH=arm
$ export KDIR=/home/<user>/debugging-labs/buildroot/output/build/linux-5.13/
$ make
```

Load the locking.ko *module and look at the output in dmesg:*

```
# cd /root/locking
# insmod locking_test.ko
# dmesg
```

Once analyzed, unload the module. Try to understand and fix all the problems that have been reported by the lockdep *system.*

Kmemleak

The provided kernel image contains kmemleak but it is disabled by default to avoid having a large overhead. In order to enable it, reboot and enable it by adding kmemleak=on *on the command line. Interrupt U-Boot at reboot and modify the* bootargs *variable:*

```
STM32MP> env edit bootargs
STM32MP> <existing bootargs> kmemleak=on
STM32MP> boot
```

Then compile the kmemleak test module:


```
$ cd /home/<user>/debugging-labs/nfsroot/root/kmemleak
$ make
```

Once done, use the `boot` *command to actually boot the kernel. Once booted, load the* `kmemleak_test.ko` *and trigger an immediate kmemleak scan using:*

```
# cd /root/kmemleak
# insmod kmemleak_test.ko
# echo scan > /sys/kernel/debug/kmemleak
```

Soon after that, the kernel will report that some leaks have been identified. Display them and analyze them using:

```
# cat /sys/kernel/debug/kmemleak
```

You can use `addr2line` *to identify the location in source code of the lines that did cause the reports. You will also notice other memory leaks that are actually some real memory leaks that did exist in the 5.13 kernel version !*

OOPS analysis

We noticed that the watchdog command generated a crash on the kernel. In order to reproduce the crash, run the following command:

```
$ watchdog -T 10 -t 5 /dev/watchdog0
```

Immediatly after executing this commands, you'll see that the kernel reported an OOPS !

Analyzing the crash message

Analyze the crash message carefully. Knowing that on ARM, the `PC` *register contains the location of the instruction being executed, find in which function does the crash happen, and what the function call stack is.*

Using Elixir (`https://elixir.bootlin.com/linux/latest/source`*) or the kernel source code, have a look at the definition of this function. In most cases, a careful review of the driver source code is enough to understand the issue. But not in that case !*

Locating the exact line where the error happens

Even if you already found out which instruction caused the crash, it's useful to use information in the crash report.

If you look again, the report tells you at what offset in the function this happens. We will disassemble the code for this function to understand exactly where the issue happened.

That is where we need a kernel compiled with `CONFIG_DEBUG_INFO` *as we did at the beginning of this lab. This way, the kernel vmlinux file is compiled with* `-g` *compiler flag, which adds a lot of debugging information (matching between source code lines and assembly for instance).*

Using `addr2line`*, find the exact source code line were the crash happened. For that, you can use the following command:*

```
$ addr2line -e vmlinux -a <crash_address>
```

We can even go a step further and use `gdb-multiarch` *to open vmlinux and locate the function and corresponding offset in assembly*

```
$ gdb-multiarch /home/<user>/debugging-labs/nfsroot/root/vmlinux
(gdb) disassemble <function>
```

KGDB debugging

In order to debug this OOPS, we'll use KGDB which is an in-kernel debugger. The provided image already contains the necessary KGDB support and the watchdog has been disabled to avoid rebooting while debugging. In order to use KGDB and the console simultaneously, compile and run kdmx:

```
$ git clone https://git.kernel.org/pub/scm/utils/kernel/kgdb/agent-proxy.git
$ cd agent-proxy/kdmx
$ make
$ ./kdmx -n -d -p/dev/ttyACM0 -b115200
serial port: /dev/ttyACM0
Initalizing the serial port to 115200 8n1
/dev/pts/7 is slave pty for terminal emulator
/dev/pts/8 is slave pty for gdb

Use <ctrl>C to terminate program
```

Note: the slave ports number will depend on the run.

Important: before using `/dev/pts/7` ***and*** `/dev/pts/8`***, the picocom process that did opened*** `/dev/ttyACM0` ***must be closed !***

On the target, setup KGDB by setting the console to be used for that purpose in kgdboc module parameters:

```
$ echo ttySTM0 > /sys/module/kgdboc/parameters/kgdboc
```

Once done, trigger the crash by running the watchdog command, the system will automatically wait for a debugger to be attached. Run `gdb-multiarch` *and attach a gdb process to KGDB with the following command:*

```
$ gdb-multiarch /home/<user>/debugging-labs/buildroot/output/build/linux-5.13/\
    vmlinux
(gdb) target remote /dev/pts/8
```

TIP: in order to allow auto-loading of python scripts, you can add `set auto-load safe-path /` in your .gdbinit file

First of all, confirm the previous information that were obtain post crash using GDB. This will allow you to also display variables values. Starting from that point, we will add a breakpoint on the `watchdog_set_drvdata()` *function. However, this function is called early in boot so we will need to actually attach with KGDB at boot time. To do so, we'll modify the bootargs to specify that. In U-Boot, add the following arguments to bootargs using* `env edit`*:*


```
STM32MP> env edit bootargs
STM32MP> <existing bootargs> kgdboc=ttySTM0,115200 kgdbwait
STM32MP> boot
```

Then the kernel will halt during boot waiting for a GDB process to be attached. Attached using the same command that was previously used:

```
$ gdb-multiarch /home/<user>/debugging-labs/buildroot/output/build/linux-5.13/\
    vmlinux
(gdb) target remote /dev/pts/8
```

Note: if you do not specify a file to be used, gdb-multiarch won't be able to detect the architecture automatically and the target command will fail. In that case, you can set the architecture using:

```
(gdb) set arch arm
(gdb) set gnutarget elf32-littlearm
```

Before continuing the execution, add a breakpoint on `watchdog_set_drvdata()` *using the* `break` *GDB command and then continue the execution using the continue* `command`

```
(gdb) break watchdog_set_drvdata
(gdb) continue
```

Analyze the subsequent calls and find the place where the driver data are clobbered.

TIP: you can fix the problem in "live" by modifying the content of the `wdd->driver_data` *variable directly using the following command:*

```
(gdb) p/x var=hex_value
```

Use it to set the variable with the previous value that was used before getting clobbered with NULL. Once done, continue the execution and verify that you fixed the problem using the `watchdog` *command.*

Note: In theory, we could have add a watchpoint to watch the address that was modified but the arm32 platforms do not provide watchpoints support with KGDB.

Debugging a module

KGDB also allows to debug modules and thanks to the GDB python scripts (`lx-symbols`*) mainly, it is as easy as debugging kernel core code. In order to test that feature, we are going to compile a test module and break on it.*

```
$ cd /root/kgdb
$ make
```

Then on the target, insert the module using insmod:

```
# cd /root/kgdb_test
# insmod kgdb_test.ko
```

If KGDB was connected and the lx scripts were loaded, then it will be detected automatically and the symbols will be loaded:


```
# scanning for modules in /home/<user>/debugging-labs/nfsroot/root
# loading @0xbf000000: /home/<user>/debugging-labs/nfsroot/root/kgdb_test/kgdb_\
    test.ko
```

If you attach KGDB after module loading, then you will need to execute the `lx-symbols` *command in GDB:*

```
(gdb) lx-symbols
loading vmlinux
# scanning for modules in /home/<user>/debugging-labs/nfsroot/root
# loading @0xbf000000: /home/<user>/debugging-labs/nfsroot/root/kgdb_test/kgdb_\
    test.ko
```

Finally, add a breakpoint right after the `pr_debug()` *call continue the execution to trigger it.*

Note: Due to a GDB bug, the execution after the breakpoint will crash. You can use a temporary breakpoint using `tbreak` *command to workaround this problem.*

Note: a side quest you can also try to enable the `pr_debug()` *call using the dynamic debug feature of the kernel.*

kdump & kexec

As presented in the course, kdump/kexec allows to boot a new kernel and dump a perfect copy of the crashed kernel (memory, registers, etc) which can be then debugged using gdb or crash.

Configuring kexec

First of all we need to setup a kexec suitable memory zone for our crash kernel image. This is achieved via the linux command line. Reboot, interrupt U-Boot and add the `crashkernel=80M` *parameter. This will tell the kernel to reserve 80M of memory to load a "rescue" kernel that will be booted on panic. We will also add an option which will panic the kernel on oops to allow executing the kexec kernel.*

```
STM32MP> env edit bootargs
STM32MP> <existing bootargs> crashkernel=80M oops=panic
STM32MP> boot
```

Once done, we'll need to configure the kernel to be booted on crash using kexec. We will use the same kernel image for both the currently running one and the one booted on crash. In order to do that, we will need to mount the sdcard boot partition that contains the zImage and dtb:

```
mount -t ext4 /dev/mmcblk0p4 /mnt/
```

Note: normally, one would recompile a custom slim kernel to be as lightweight as possible instead of reusing a full feature kernel. A specific initrd/rootfs would also be provided to avoid any more risks of crashing.

To load the crash kernel into the previously reserved memory zone, run the following command:

```
$ kexec --type zImage -p /mnt/boot/zImage --dtb=/mnt/boot/stm32mp157a-dk1.dtb
  --command-line="root=/dev/nfs ip=192.168.0.100:::::eth0
  nfsroot=192.168.0.1:/srv/nfs/stm32_debug,nfsvers=3,tcp rw console=ttySTM0
```



```
maxcpus=1 reset_devices"
```

Once done, you can trigger a crash using the previously mentioned watchdog command:

```
$ watchdog -T 10 -t 5 /dev/watchdog0
```

At this moment, the kernel will reboot into a new kernel using the specified kernel after displaying the backtrace and a message:

```
[ 1181.987971] Loading crashdump kernel...
[ 1181.990839] Bye!
```

For some reason, the console is not working after rebooting. But the kernel is actually booting correctly. Wait a bit, connect with ssh and copy the coredump:

```
$ ssh root@192.168.0.100
$ cp /proc/vmcore /root/vmcore
```

Finally, we will be able to debug that kernel coredump using crash.

Compiling crash

`crash` *utility that is available on your computer does not support ARM so we will need to recompile it for the ARM target. This can be done using the following commands:*

```
$ sudo apt install gcc-multilib g++-multilib lib32z1-dev lib32ncurses5-dev \
    texinfo
$ cd /home/<user>/debugging-labs/
$ git clone https://github.com/crash-utility/crash.git
$ cd crash
$ make target=ARM
```

Once done, you can open the vmcore file with crash using

```
$ sudo chown <user>:<user> /home/<user>/debugging-labs/nfsroot/root/vmcore
$ ./crash /home/<user>/debugging-labs/nfsroot/root/vmlinux
  /home/<user>/debugging-labs/nfsroot/root/vmcore
```

Take some times to analyze the content of the dump using the commands that are offered by `crash`*.*