Preempt RT Training

STM32MP1 variant

Practical Labs

bootlin

https://bootlin.com

May 07, 2024

About this document

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

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/preempt-rt/preempt-rt-labs.tar.xz
$ tar xvf preempt-rt-labs.tar.xz
```

Lab data are now available in an `preempt-rt-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.
- 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.

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

- 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/`

Lab1: Building and Booting a Preempt-RT Kernel

Download, Configure, Build and Boot

During this lab, you will:

- Configure the Buildroot Build-system to generate an image based on the upstream linux-rt repository
- Configure the kernel to enable full preemption
- Boot the system and check that it runs preempt-rt

Initial Setup

As specified in the Buildroot manual[2], Buildroot requires a few packages to be installed on your machine. Let's install them using Ubuntu's package manager:

```
sudo apt install sed make binutils gcc g++ bash patch \
  gzip bzip2 perl tar cpio python unzip rsync wget libncurses-dev
```

Download Buildroot

Since we're going to do Buildroot development, let's clone the Buildroot source code from its Git repository:

```
git clone https://git.busybox.net/buildroot
```

In case this is blocked on your network, you can download the Buildroot tarball `buildroot-2023.11.1.tar.bz2` from `https://buildroot.org/downloads/` and extract it. However in this case, you won't be able to use *Git* to visualize your changes and keep track of them.

Go into the newly created `buildroot` directory.

We're going to start a branch from the *2023.11.1* Buildroot release, with which this training has been tested.

```
git checkout -b bootlinlabs 2023.11.1
```

Configuring Buildroot

The buildroot configuration is provided in the lab materials. Copy the configuration in the "configs" folder of your buildroot installation:

```
cp ~/preempt-rt/preempt-rt-lab-data/stm32mp157a_dk1_rt_defconfig configs
```

We'll use that configuration as a basis for our setup:

```
make stm32mp157a_dk1_rt_defconfig
```

Kernel configuration

The standard way to configure the kernel is through the `make menuconfig` interface. Here, we're building everything using Buildroot, because it's an easy way to build a fully integrated image, with our custom kernel but also our custom applications.

[2] https://buildroot.org/downloads/manual/manual.html#requirement-mandatory

We'll use Buildroot's `make linux-menuconfig` to modify our kernel configuration, it's strictly equivalent to the `make menuconfig` command from the kernel's source tree.

```
make linux-menuconfig
```

The default kernel configuration file for this platform isn't made for a -RT kernel. Building an kernel with the Preempt-RT patch is not enough to benefit from the full kernel preemption, we also need to enable it. In the menuconfig interface, you'll find the `Preemption Model` setting under the `General Setup` category.

We want to enable the `Fully Preemptible Kernel` mode, but it's not proposed as an available choice by default. To enable it, we first need to enable the `expert` mode, by selecting the `CONFIG_EXPERT` option. Once enabled, we can select the Preempt-RT mode!

```
make
```

While the build is ongoing, don't hesitate to take a look at the latest version of the patchset:

```
wget https://cdn.kernel.org/pub/linux/kernel/projects/rt/6.2/patches-6.2-rc3-rt1.tar.gz
```

Look at the `series` file for more information about each individual patch.

Setting up serial communication with the board - STM32MP157

The STM32MP1 devkit's serial port can be accessed through the micro-USB connector.

Once the USB cable is plugged in, a new serial port should appear: `/dev/ttyACM0`. You can also see this device appear by looking at the output of `dmesg`.

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 [3]. 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.

Flash the image to the SDCard

Checking that we run a Patched kernel

To check that we are indeed running a kernel with the preempt-RT patch applied and full kernel preemption enabled, we have 2 main ways of checing.

First, use the `uname -a` command. Running an RT kernel should show the `PREEMPT_RT` version item.

The other way to check is to look at the file `/sys/kernel/realtime`. It's content is always '1', but the file only exists for RT kernels.

Take a look at the boot logs, can you see something that's not going right? Why?

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

Fixing our first kernel bug

You should see quite a lot of `BUG: scheduling while atomic` messages. The actual issue is non-trivial to fix, but it's a nice example to analyze.

Lab2: Testing and Benchmarking the system

During this lab, you will:

- Use benchmarking tools to measure the latencies of a system
- Use stressors to analyze various scenarios
- Determine suitable options to further improve latencies

Determine the maximum latency

First, we will need to install some benchmark and analysis tools on our system.

In the Buildroot `make menuconfig` interface, enable the following packages:

- rt-tests
- powertop
- perf
- stress-ng
- iperf3
- fping
- scheduling utilites from the util-linux package
- python3
- screen
- dropbear

You'll also need to enable a few features in the kernel. Run `make linux-menuconfig` and select the following options, located in "Kernel Hacking":

- Compile-time checks and compiler options -> Compile the kernel with debug info
- Tracers -> Function tracer, Interrupts-off tracer, Preemption off tracer, scheduling latency tracer...

Re-build the image, and boot it on the board: `make linux-rebuild all`

Let's first start by establishing the baseline latency by simply running `cyclictest`:

```
cyclictest -p
```

The goal of this lab is to try to lower the latency as much as possible while the system is under various types of loads. This will allow us to get the best running conditions for our applications.

Some tweaks will really be useful on SMP systems, so don't hesitate to test on your own machine!

Here's a few leads:

- Is the system SMP?
- Do we need to isolate our task?

- Try changing the scheduling policy
- Try changing the scheduling priority
- Investigate the various interrupts
- Take a look at the cpuidle and cpufreq configuration
- Are there any NMIs?

Stress the system using several tools:

- hackbench
- stress-ng
- iperf3
- fping

Some kernel options can also be useful:

- `CONFIG_TRACE_HWLAT`

First analysis

To get a first idea of the wakeup latencies you can expect on your system, launch `cyclictest` on the target, and take a look at the Max latency. The lower, the better. You shouldn't get big latency spikes.

By running `cyclictest` as is, you will run the benchmark with the default scheduling policy (`SCHED_OTHER`), and without any CPU pinning.

You may not notice huge latencies right away, since the system at that point isn't doing much. You can try to load the system and see how that affects the latencies

To run cyclictest with a real-time scheduling policy, use the `-p <prio>` option. Cyclictest doesn't play well with the `chrt` command, since it will itself re-set it's own scheduling policy.

Try running `cyclictest -p 40` and see if you get better latencies.

Network load

An easy way to introduce some load is to generate some network traffic. This will generate some interrupts, but also stress the kernel and create some context switches.

First, setup the board's network interface:

```
ip link set eth0 up
```

```
ip address add 192.168.0.2/24 dev eth0
```

Make sure that your computer has its network interface on the same subnet as your target.

On the board,run the `iperf3` server in the background:

```
iperf3 -s -D
```

Re-run your cyclictest benchmark, and start sending traffic to your target. From you host computer, run `iperf3 -c 192.168.0.2`. You should start seeing the latency rising up.

Try comparing the latencies you get between `cyclictest` and `cyclictest -p 40`. Do you still see high latencies while some network traffic is being received? If so, why and how could we fix this?

Scheduling load

Another way to stress the system without any external source is with the `hackbench` tool, which generates a lot of context switches by exchanging data back and forth between multiple processes and multiple threads. This benchmark is pretty intense and can bring an entire system down to a non-responsive state, so launch it with only 10 file-descriptors, running an infinite amout of loops:

```
hackbench -f 10 -l -1 &
```

With hackbench running in the background, compare the output of `cyclictest` and `cyclictest -p 40`, the difference should be pretty impressive.

Analyzing the system configuration

CPU Pinning

Take a look at how many CPUs are on your system. You can run "htop", or look in `/sys/devices/system/cpu/`. This place is useful since you'll find lots of ways to manage the CPUs:

- in `cpuX/cpufreq`, you'll find ways to inspect and control the CPU's frequency
- in `cpuX/cpuidle`, you'll find ways to inspect and control the CPU's idle states

If you have multiple CPU cores, it's a good idea to start by isolating:

- Your process
- Important interrupts

On the contrary, you might want to restrict interrupts to cores that won't affect you process.

To perform this, use the `taskset` command, both for running your process, but also to change the interrupt CPU affinity.

For cyclictest, you can either run cyclictest with the `-a <cpu_num>` option, or use `taskset -c <cpu_num> cyclictest ....`

Try running hackbench and cyclictest on the same CPU, and then on different CPU and compare the induced latencies.

Interrupt Pinning

It might be a good idea to make sure that no unexpected interrupts occur on the CPU you use for your realtime application. To know how many interrupts fire on each CPU core, take a look at the `/proc/interrupts` file:

`cat /proc/interrupts`.

After you identify the interrupts that fire (take a look at the `ethernet` interrupt when you generate traffic), you cat change it's CPU affinity by going into `/proc/irq/<num>/`.

You can then limit it by echo-ing an integer corresponding to the bitfield of enabled CPUS:

`echo 1 > smp_affinity` will limit the interrupt to the CPU 0

`echo 3 > smp_affinity` will limit the interrupt to the CPU 0 and 1

Try to limit the ethernet interrupt to one CPU core, and launch `cyclictest -p 40` on the other core. You can see that even when launching cyclictest with a priority lower than the threaded interrupt's, you don't see any impact of network traffic on the latencies.

CPU Isolation

To go even further, you can completely isolate one CPU from the scheduler's pool, abd have it only accessible through `taskset`. To do so, you need to change the kernel's commandline, passed by the bootloader. On the

STM32MP157 platform, this is done using the `extlinux` infrastructure. You can change the commandline by editing the `/boot/extlinux/extlinux.conf` file:

```
vi /boot/extlinux/extlinux.conf
```

Add the `isolcpus=1` to the `append` line to isolate CPU 1.

Reboot your target, and run cyclictest on CPU1. What can you notice?

Using the Tracing subsystem

In order to use Ftrace, we need to make sure it is enabled in the kernel configuration. Using `make linux-menuconfig`, go in the "Kernel hacking" section and enable the following options:

- Generic Kernel Debugging Instruments -> Debug Filesystem
- Tracers -> Kernel Function Tracer
- Tracers -> Kernel Function Profiler
- Tracers -> Interrupts-off Latency Tracer
- Tracers -> Preemption-off Latency Tracer
- Tracers -> Scheduling Latency Tracer
- Tracers -> Tracer to detect hardware latencies

You can now recompile your kernel and re-generate your full image:

```
make linux-rebuild
make
```

Once you've rebooted your board with ftrace enabled, you'll need to mount the Tracing Filesystem, that gets automatically mounted in debugfs:

```
mount -t debugfs debugfs /sys/kernel/debug
```

```
mount -t tracefs nodev /sys/kernel/tracing
```

You can now move into the tracing subsystem's main directory:

```
cd /sys/kernel/debug/tracing
```

You can now take a little tour of the files here:

- `available_tracers`: Lists all tracers you can use. Each tracer corresponds to what and how do you want the tracing to occur.
- `current_tracer`: Write the tracer you want to use in this file
- `tracing_on`: Write 1 to start tracing, 0 to stop tracing
- `trace`: The content of the trace buffer after tracing is finished

First, try the `preemptirqsoff` tracer and start tracing:

```
echo preemptirqsoff > current_tracer
echo 1 > tracing_on
sleep 10
echo 0 > tracing_on
cat trace
```

Give a try to other traces: `wakeup_rt`, `function`, `function_graph`, using various of the stressing methods proposed above. Can you get meaningful information?

Using trace-cmd and kernelshark

Trace-cmd is a wrapper over the raw silesystem-based interface of ftrace. There are various ways of using it, either standalone, or by recording the behaviour of the system during a command.

Recording

To record traces during a command, run trace-cmd with the `record` keyword:

```
# record a full function_graph trace for the duration of the cyclictest run
trace-cmd record -p function_graph cyclictest -p 80 -D 10

# Display the output
trace-cmd report
```

Running a recording session will automatically export the generated trace in a ”trace.dat” file, that can be displayed with `trace-cmd report` or `kernelshark`.

Manual tracing

Another approach is to simply start an ftrace session, without relying on a process's execution. This is useful in combination with the `wakeup_rt`, `wakeup`, `preemptirqsoff` tracers.

```
# Start a tracing session
trace-cmd start -p wakeup_rt

# Stop a tracing session
trace-cmd stop

# Display the generated trace
trace-cmd show

# Export and clear the trace buffer into trace.dat
trace-cmd extract

# Display the extraced trace
trace-cmd report
```

This approach also allows you to use `cyclictest` in conjunction with `ftrace`, since you can start at tracing session with `trace-cmd start` and have it stopped automatically when cyclictest detects a high latency:

```
# Start a tracing session
trace-cmd start -p function_graph

# Launch cyclictest with the breaktrace option
cyclictest -p 80 --breaktrace=2000 --tracemark

# When the high latency is hit, you can display the trace
trace-cmd show
```

Using the osnoise tracer

The following command is used to generate an histogram of os noises with the `osnoise` tracer:

```
osnoise hist -c 1 -P f:49 -d 1m
```

The output shoud show a maximum latency around 50 000 microseconds. Why is that?

Lab3: Optimizing an application for Real-Time

During this lab, you will:

- Better understand the page-faulting mechanism
- Debug interferences with a time-sensitive application

Compile and run the program

Compile the crc_test program: `make`

Test the various options, and analyze how the Stack and Heap are allocated and pre-faulted.

Use ftrace to visualise the page-fault occurings:

```
trace-cmd record -e *page_fault* ./crc_test
```