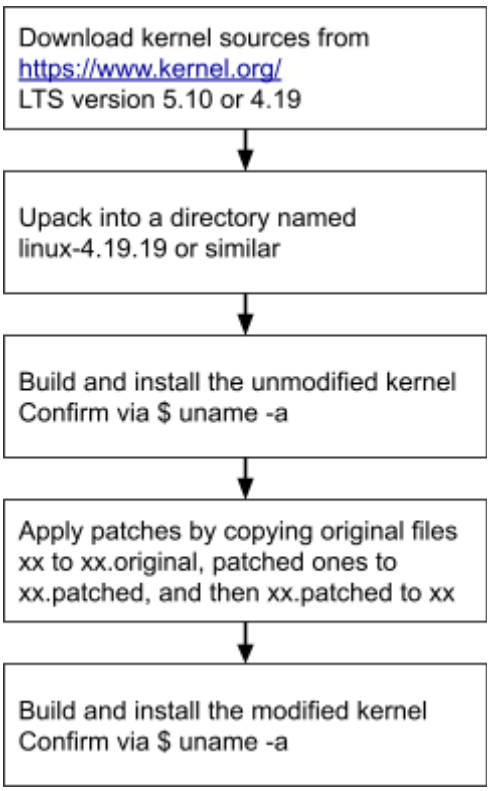
Linux patches installation guide
Dick Sites
2021.10.28

Installing Linux patches
This section is a quick walk-through of building a patched kernel for KUtrace.



Some Linux kernel versions have committed long term support (LTS) for five years. The book text is all based on x86 version 4.19.19 (the last 19 indicates minor-change sub-version number). The Raspberry Pi-4B patches are for the beta 64-bit version of RaspOS. They run on the 8GB version of the Pi-4B hardware; the 4GB and smaller versions only run the 32-bit OS. More precisely, the 64-bit RaspianOS can run on a 4GB machine, but it may have excessive paging that wears out an SD card too quickly. The patches are based on Linux version 5.10.46-v8+ (for ARM 64-bit).

Download kernel sources
Go to kernel.org and pick which x86 version you want. Download the tarball. Use
`git clone --depth=1 https://github.com/raspberrypi/linux`
for the Raspberry Pi.

If you have an AMD or Intel x86 processor and want the exact kernel used in the book, use 4.19.19, for which matching patches are posted. Or instead use 5.10 for x86 or RPi4. If you want to use the kernel version currently running on your machine, download the sources for that one, but then you will have to manually put the KUtrace patches into the right places in the source files.

Unpack

Unpack the tarball into a directory whose name resembles `linux-4.19.19` (for the version in the book). You may want to move the unpacked directory to someplace other than the default place, which might be in Downloads. This command does the default unpack:

```
$ tar xvf linux-4.19.19.tar.xz
```

The unpacked directory contains in the `arch` subdirectory many sets of machine-specific source code, including `arm, arm64, x86, and x86_64` (for both AMD and Intel). The rest of the code in other subdirectories is shared across all machine types. I will use "linux-4.19.19" in the examples below. Substitute your chosen directory name instead.

Build and install unmodified kernel, x86 (Raspberry Pi is below)

The Linux build system is fairly straightforward to use, but documentation varies from too sparse to too detailed. https://phoenixnap.com/kb/build-linux-kernel (currently) has a good description of the build process.

The initial step is to install a number of software packages that are used by the build, as described in the phoenixnap article. You only need to do this once.

```
$ sudo apt-get install git fakeroot build-essential ncurses-dev xz-utils \
  libssl-dev bc flex libelf-dev bison
```

Take a moment to do

```
$ uname -a
```

to see the current (original) kernel version and build date. Add this to the notes that you are keeping to document what you are doing. I just do this in a running text file that I go back to frequently when making changes later.

The next step is to create an appropriate kernel configuration file. This is an extensive text file named .config, but it does not exist in the download. You can copy the current configuration file on your running Linux machine. This can be found in various places:

/boot/config-4.19.19
/proc/config.gz
/lib/modules/$(uname -r)/build/.config

The last of these expands into

/lib/modules/4.19.19/build/.config

for example.

If none of these files exist, the command

```
$ sudo modprobe configs
```

will create /proc/config.gz, and zcat /proc/config.gz > some_config_filename will expand it.

Once you have found the current kernel configuration file, copy it to linux-4.19.19/.config. For fallback, copy it to a file named myconfig.original or somesuch.

You could also copy a kernel .config file from some other similar running Linux machine.

Next, use

```
$ cd linux-4.19.19
```

to set the current directory at the top of the source tree. You always want to be here when building the kernel.

The command

```
$ make oldconfig
```

reads the existing .config file that was used for an old kernel and prompts the user for options in the current kernel source tree that are not found in the file. This is useful when taking an existing configuration and moving it to a new kernel. The options in the current kernel source are in text files named Kconfig scattered throughout the source directories. At this stage, you generally want to simply accept any defaults offered.

Next run

```
$ make menuconfig
```

It lets you modify the configuration, and will highlight any new options that do not yet have values assigned. It uses the ncurses text package. At this stage, you generally want to simply accept any defaults offered. But take a little time and browse around the options to get somewhat familiar with them.

When you are done, you have an up-to-date .config file in linux-4.19.19. Copy it to .config.original.

The actual unmodified build follows. Keep an eye on the messages, so you have some idea of the normal warnings and complaints in a vanilla build. This will help if you need to spot any differences in the modified build(s).

Time to run the

```
$ sudo make -j4
```

command. It will build the entire kernel using 4 parallel threads, which is good on a four-core processor. If you have a different number of CPU cores, change the 4 to match the number you have. The first build will take a while, perhaps 15-75 minutes. It will produce a kernel binary in arch/x86/boot or a similar place for your kernel version and CPU type. Later builds will only do changed files, so will be faster.

We are not quite done yet. In addition to the kernel image itself, there are kernel loadable modules that will be linked into the kernel at runtime. These modules must be built against the just-built kernel's header files so that the linking knows the proper offsets of kernel image variables that the modules use. The command

```
$ sudo make modules_install
```

builds all the modules, including drivers for all sorts of I/O devices. Be patient for a few minutes.

Finally, do

```
$ sudo make install
```

to package everything up and move the new kernel to the /boot directory and point the bootloader to it.

Reboot, perhaps via

```
$ sudo reboot
```

When the system comes back up, check the kernel version and build time via

```
$ uname -a
```

It should show the kernel you just built. Add this to the notes that you are keeping to document what you are doing. Take a break and celebrate.

Build and install unmodified kernel, Raspberry Pi-4B 64-bit version

Build directions from https://www.raspberrypi.org/documentation/computers/linux_kernel.html

```
$ sudo apt install git bc bison flex libssl-dev make
$ git clone --depth=1 https://github.com/raspberrypi/linux
```

Build:

```
$ sudo make bcm2711_defconfig
```

This makes the default RPi 64-bit .config file. Only do this once. We will modify it later.

```
$ KERNEL=kernel8
$ sudo make -j4 Image modules dtbs
```

This produces a kernel image 5.10.46-v8+ of about 21MB bytes uncompressed. The shipped original kernel at about 7MB is compressed.

After

```
$ sudo make modules_install
$ sudo cp /boot/$KERNEL.img /boot/$KERNEL-backup.img
$ sudo cp arch/arm64/boot/Image /boot/$KERNEL.img
$ sudo cp arch/arm64/boot/dts/broadcom/*.dtb /boot/
$ sudo cp arch/arm64/boot/dts/overlays/*.dtb* /boot/overlays/
$ sudo cp arch/arm64/boot/dts/overlays/README /boot/overlays/
$ sudo reboot
```

The 5.10.46 kernel boots. Do

```
$ uname -a
```

To confirm the new build date.

Apply KUtrace patches

All the source code associated with the book is located at
https://www.informit.com/store/understanding-software-dynamics-9780137589739
under the Extras tab. Patch files for x86 Linux versions 4.19.19 and 5.10.66, plus ARM (Raspberry Pi-4B 8GB) version 5.10.46-v8+ are bundled there.

The style I have used is to have three source files for each patch, for example:
arch/x86/mm/fault.c
arch/x86/mm/fault.c.original

arch/x86/mm/fault.c.patched

The first is the one used by the build system. It is either foo.original or foo.patched. You can tell which it is by looking at the file sizes. Once these three files are set up, copying foo.patched to foo and building will use the patched file, while copying foo.original to foo and building will use the original file. The command

```
$ diff foo.original foo.patched
```

will show exactly what is different in the patched version.

The supplied patch files have *.original and *.patched for each changed file, organized into their respective parts of the kernel source tree. Move these into your source pool. If you have the exact matching source pool, your unmodified arch/x86/mm/fault.c and the supplied arch/x86/mm/fault.c.original, for example, will be identical.

Manually editing in patches

If you are using some other version of the kernel sources, you will need to manually edit or create the *.patched files, using the supplied original/patched differences as a guide.

I did this recently by opening one command-line window in the previously-patched linux-4.19.19 directory and one in the unmodified linux-5.10.66 directory, then proceeding like this:

4.19 directory:

```
$ find . -name "*.patched"
```

to find all the patched files

For each of those other than .config, in the 5.10 directory do

```
$ ls -l foo*
```

to verify that the base file exists, then do

```
$ cp foo foo.original
$ cp foo foo.patched
```

to get ready for manually applying edits to foo.patched.

For each patched file, manually copy patched lines in a similar version to the corresponding place in your version. Moving from 4.19 to 5.10, I found 5.10 has a new file arch/x86/kernel/apic/ipi.c that contains some of the code previously in smp.c, but it was straightforward to find the matching places to insert patches.

Save the modified version, and then copy it to the base name:

```
$ cp foo.patched foo
```

In moving from 419 to 5.10 I patched the files in a particular order, so I could build and test tracing incrementally.

Step 0.

./.config.patched

DON'T use this file. Use the one for your machine as described above in the unmodified build. Use this for reference if things get off track.

Step 1.

./include/linux/kutrace.h.patched
./arch/x86/Kconfig.patched
./kernel/Makefile.patched
./kernel/kutrace/Makefile.patched
./kernel/kutrace/kutrace.c.patched
These are the underpinning for including any KUtrace code at all. Apply patched lines to all of them.

As described below, run
```
$ make menuconfig
```
to pick up the new KUTRACE configuration variable and set it to "y" to enable it.

Building here via
```
$ sudo make -j4
```
confirms no compilation errors. You should see CC kutrace.o go by.

Step 2.
./arch/x86/entry/common.c.patched
./arch/x86/kernel/apic/apic.c.patched
KUtrace cannot be turned on without the control hook in common.c and the raw trace file timestamps will not expand correctly without timer interrupt events at least every 10 msec on every CPU core. The common.c patches trace all system calls. The apic.c patches also sample PC addresses and CPU frequency at each timer interrupt.

Building and rebooting here, building the kutrace_mod module, inserting it, and doing kutrace_control commands go then `stop` and postprocessing will give an incomplete but error-free trace.

Step 3.
./kernel/sched/core.c.patched
The scheduler entry/exit/switch and make-runnable patches are the underpinning for tracking context switches (and the sometimes surprisingly-large time spent in the scheduler itself).

Step 4.
./arch/x86/kernel/irq_work.c.patched
./arch/x86/kernel/smp.c.patched
./arch/x86/kernel/irq.c.patched
./kernel/softirq.c.patched
These trace interrupts.

Step 5.
./arch/x86/mm/fault.c.patched
This traces page faults.

Building and rebooting here, building the kutrace_mod module, inserting it, and doing
```
$ ./kutrace_control
```
go then `stop` and postprocessing will give a nearly-complete meaningful trace.

Step 6.
./fs/exec.c.patched
This adds to a trace the command-line name of any new execve target file.

Step 7.
/drivers/idle/intel_idle.c.patched
./drivers/acpi/acpi_pad.c.patched
./drivers/acpi/processor_idle.c.patched
These add x86-specific tracing of switching to low-power idle via mwait instructions, for both AMD and Intel processors.

Step 8.
./arch/x86/kernel/acpi/cstate.c.patched
This attempts to add x86-specific tracing of frequency changes. Depending on power configuration, this is unused and the timer-interrupt sampling is the fallback.

Step 9.
./net/ipv4/tcp_input.c.patched
./net/ipv4/tcp_output.c.patched
./net/ipv4/udp.c.patched
These trace quick hashes of incoming and outgoing packets.
Building and rebooting now gives you a complete running KUtrace system!

Building the modified kernel
Before you build the modified kernel the first time, you need to make a couple of changes in the .config file.

Make sure you have done

```
$ cd linux-4.19.19
```

and have copied .config to .config.original. Next we will modify .config and copy it to .config.patched. You will then have three files
.config
.config.original
.config.patched
with .config identical to .config.patched.

Run

```
$ make menuconfig
```

It should notice the new KUtrace configuration option. Enable it.

Also check that the timer interrupts are set to periodic, with a constant rate of 100, 250, 1000 or whatever interrupts per second. KUtrace depends on periodic timer interrupts to each CPU at least 100 times per second. The location of the menuconfig choices for this vary. You can check the currently-booted configuration via

```
$ grep 'HZ' /boot/config-$(uname -r)
```

which should give something like

```
CONFIG_HZ_PERIODIC=y
# CONFIG_NO_HZ_IDLE is not set
# CONFIG_NO_HZ_FULL is not set
# CONFIG_NO_HZ is not set
# CONFIG_HZ_100 is not set
CONFIG_HZ_250=y
# CONFIG_HZ_300 is not set
# CONFIG_HZ_1000 is not set
CONFIG_HZ=250
# CONFIG_MACHZ_WDT is not set
```

You want to be sure that HZ_PERIODIC is set and that the various NO_HZ choices are not set.

Save the updated configuration. Copy it to .config.patched. Double-check your three files:

.config
.config.original
.config.patched

Here we go... (for x86, see recipe above for RPi-4)

```
$ sudo make -j4
$ sudo make modules_install
$ sudo make install
```

You can check at this point that the top-level text file System.map contains about 15-20 "kutrace" lines.

```
$ sudo reboot
$ uname -a
```

This should show the just-built, patched, kernel version and build date. We are almost done.

Most of KUtrace is implemented in a loadable module, whose source is kutrace_mod.c. Move this into your home directory or a subdirectory somewhere, along with its Makefile. Then `cd` to that directory and do

```
$ sudo make
```

to compile the KUtrace module. This compilation will use the headers just created for the patched kernel and will produce kutrace_mod.ko.

When you are ready to allow tracing, insert the loadable module into the kernel via:

```
$ sudo insmod kutrace_mod.ko tracemb=20
```

where the 20 sets aside 20MB of trace buffer. You can pick other values. At 4 bytes each, 20MB holds 5M event entries, which is fine for class work. All the examples in the book used 20MB.

When you want to disable tracing, use

```
$ sudo rmmod kutrace_mod.ko
```

With the module installed, run the kutrace_control program. It will print the prompt

```
control>
```

Typing `go` traces; then `stop` ends tracing and writes a raw binary trace file. Postprocess that and you will get an HTML file showing the CPU activity. Congratulations. Go celebrate.

When things look stable, you could start `$ ./kutrace_control` in one command-line window, up to the `control>` prompt.
In a second command-line window get ready to run the supplied hello_world_trace program, by typing but do NOT hit <cr>: `$ ./hello_world_trace`
Then do these three steps:
first window: goipc <cr>
second window: <cr> (running hello_world_trace)
first window: stop <cr>

This gives you a few-second raw trace file that contains the complete execution of hello world. The first time you do this, there will be a 15-20 msec gap during execve to fetch the executable from disk. Subsequent runs will have a minimal gap for fetching it from the in-RAM file cache. Compare the postprocessed trace to the supplied hello_world_trace.html file.

Troubleshooting
Calling on someone who has built kernels before helps. Or even going through the directions with a second person using a second set of eyes can often spot where you got off track.

Trouble with configuration. Diff between the original .config configuration and the troubled one can help. Diff between good and bad /boot/config-* can help.

Trouble building a modified kernel. Read the error messages carefully. Keep an eye out for failing to use "sudo" on some of the commands. Selectively remove some patched files by copying foo.original back to foo. Removing all of them and copying .config.original to .config should exactly build the unmodified kernel. You can also use `menuconfig` to remove KUtrace entirely from a build.

Sometimes
```
$ make clean
```
can help if there are build errors that don't make sense.

Trouble booting a newly-built kernel. You can reboot holding down left shift, right shift, or esc (depending on your machine) to get to the GRUB bootloader (Ubuntu options), which will give you a choice of kernels. Pick the last good one.

Trouble actively tracing. If the operating system crashes while tracing but is otherwise stable, selectively remove some patched files by copying foo.original back to foo, rebuild and reboot. This will eventually identify which particular patch is the problem. With luck, that will suggest what is going wrong. To reinstall some patches, copy foo.patched to foo.

The command
```
$ cat /proc/kallsyms |grep kutrace
```
will produce about 20 matches with a patched kernel, and about 40 more matches for a loaded kutrace_mod module. It will produce no matches for an unpatched kernel. Used with sudo, it will provide actual memory addresses instead of zeros.