

# „Introduction guide”

Arkadiusz Chrobot, PhD  
Karol Tomaszewski, PhD

February 24, 2024

# Contents

<b>Introduction</b>	<b>1</b>
<b>1. ssh</b>	<b>1</b>
<b>2. scp</b>	<b>1</b>
<b>3. Linux Cross Reference</b>	<b>2</b>

## Introduction

These introductory guide contains brief descriptions of the tools whose knowledge will be necessary for the efficient implementation of tasks within the laboratories of Operating Systems 2. Each team will be assigned a virtual machine that will need to be operated from the console, therefore it is necessary to **remember the basic commands of the system shell**. These commands will not be described in these guides. Section 1 describes how to use the **ssh** (*Secure Shell*) command, Section 2 presents the **scp** (*Secure Copy*) command, and Section 3 refers to the use of the Linux Cross Reference service.

### 1. ssh

The **ssh** command is used for working with the remote computer. This work is done using a system shell (console), and the network traffic is encrypted. In case of laboratory classes, the remote computer is a virtual machine. To connect to it, issue the following command in a local computer terminal or console:

```
ssh so2@81.26.20.32 -p X
```

The **so2** in the command is a username, **81.26.20.32** is an IP address of virtual machine server, and **X** is a port number. Every team will have a separate virtual machine assigned. The port numbers used by the machines will be given by the person conducting the laboratory classes. When the command is issued, the remote system will ask for the password. Enter the password provided by the person conducting the laboratory. While typing the password, no characters will appear on the screen until the **Enter** button is pressed. **Please note, that You can connect to the virtual machine only from within the Department of Information Systems internal network, provided the machine is on.**

The command allows You to log as the **so2** user. This account has limited privileges. To load or unload a kernel module, what is essential to accomplish tasks from laboratory guides, You need to log to the **root** account, using the following command:

```
su -
```

The system will ask for the **root** user password, which also will be given by the person conducting the laboratory. If the password is valid, then You will be granted access to an account that allows You to administrative operations, including those mentioned earlier.

It is also possible to log in as the **root** using another command:

```
sudo su -
```

This time, the system will ask for the **so2** user password, not the **root** one. If it is valid, You will also be granted access to the **root** account.

The **sudo** command has more applications. Generally, it is uses to perform privileged commands by an unprivileged user. However, these cannot be any commands. They should have been previously specified for that user by the **root**.

### 2. scp

The **scp** command belongs to the same package as **ssh**. It allows the user to copy files between a local and a remote computer. Copying of the file named **source.c** located in the current directory on the local machine, to the home directory of the **so2** user on the remote virtual machine on the **X** port, can be done using the following command:

```
scp -P X source.c so2@81.26.20.32:~
```

Please note the „ : ” character after the IP address followed by the directory name. In this case, it is the user’s home directory, which in Linux is denoted with a tilde character (~). Should the destination directory be `lab1`, located inside the home directory, then the command would have to be:

```
scp -P X source.c so2@81.26.20.32:~/lab1
```

The remote system will ask for the `so2` user password, just like in the case of `ssh`.

Similarly, files can be copied in the opposite direction. For example, to copy a file named `Makefile` from the home directory of the `so2` user at the virtual machine, to the local computer current directory, issue the following command:

```
scp -P X so2@81.26.20.32:~/Makefile .
```

The dot at the end of the command denotes the current directory. Please note, that the order of `scp` arguments is the same, as in the case of the `cp` command, used for copying files locally.

There are two ways to copy the entire directory to a remote system. Suppose this directory is called `sources` and is a subdirectory of the current directory. The first way is to use the `-r` option of the `scp` command, which recursively copies the directory:

```
scp -P X -r sources so2@81.26.20.32:~
```

In the second method, the directory should be first compressed, e.g. with the `tar` command. It can be done as follows:

```
tar jcvf sources.tar.bz2 sources,
```

where `sources.tar.bz2` is the name of the output file being a compressed archive of the `sources` directory. After copying it to a remote computer, its contents can be extracted with the command:

```
tar xvf sources.tar.bz2
```

### 3. Linux Cross Reference

There is documentation for the Linux kernel source code in the form of `man` pages, but unfortunately it is very limited. The best way to investigate what a piece of code does and how it is built is to browse kernel sources, which are very extensive. Fortunately, there is a tool that can generate HTML pages that allow to easily navigate this code. The pages created this way are available, among others, at <https://elixir.bootlin.com/linux/latest/source>. The following example describes how to use this site, searching for information about the `schedule()` function<sup>1</sup>. After visiting the given URL address, the page should appear like in Figure 1.

On this page, the version `4.15` of the kernel sources should be selected, as shown in Figures 2 and 3.

After selecting the kernel source code version, enter the name of function: `schedule` in the *Search Identifier* field, then click the search icon or press `Enter` (Figure 4).

The results page is divided into two parts. The first part, marked in yellow in Figure 5, contains references to the lines in the files in which the prototype or definition of the function is located. The second part, whose fragment is marked in green, is a list of links to the file lines in which this function is used.

After clicking the first link in the *Defined in 3 files:* section, the user will be taken to the page with the function declaration. Its prototype (header) is marked in yellow in Figure 6.

After clicking the second link in the aforementioned section, the user will be taken to the page with the definition of this function, which is marked in yellow in Figure 7.

---

<sup>1</sup>This function is an implementation of the processor scheduler.



Figure 1: Main page of the Linux Cross Reference service

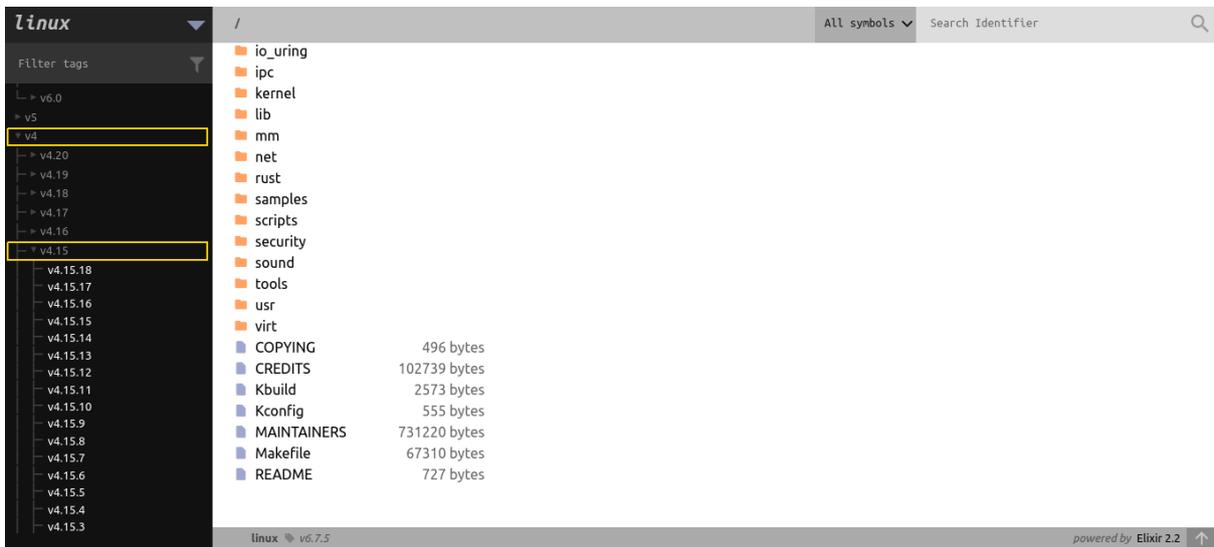


Figure 2: Kernel source version selection menu

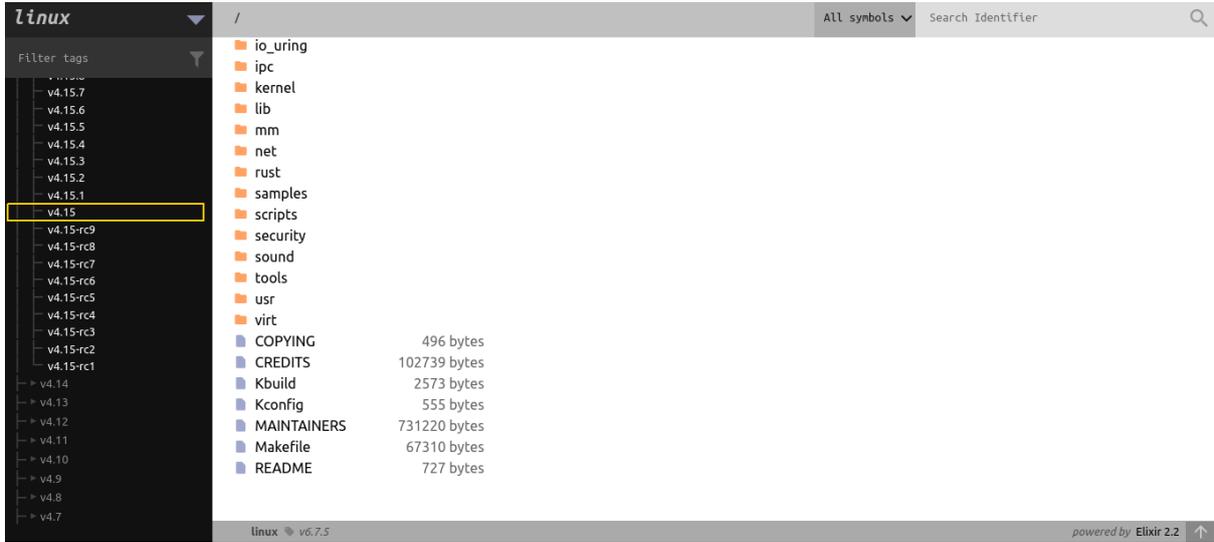


Figure 3: Detailed menu for choosing the kernel source version

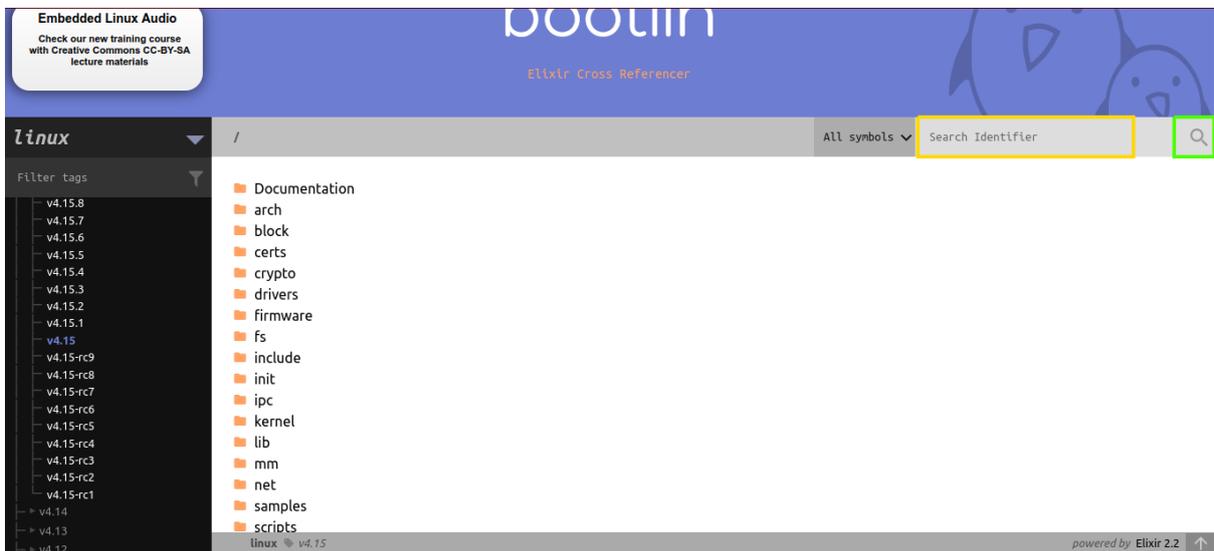


Figure 4: Name search

The screenshot shows the bootlin website interface. At the top, there is a navigation bar with links for HOME, ENGINEERING, TRAINING, DOCS, COMMUNITY, and COMPANY. The bootlin logo is prominently displayed in the center, with the tagline "ELUxR - Cross Referencer" below it. On the left side, there is a sidebar with a "linux" dropdown menu and a "Filter tags" section. The main content area displays search results for the term "schedule".

The search results are organized into three sections:

- Defined in 1 files as a prototype:** A box highlights the entry: `include/linux/sched.h` line 178 (as a prototype).
- Defined in 11 files as a member:** A list of files where `schedule` is used as a member, including:
  - `drivers/gpu/drm/915/intel_ringbuffer.h` line 407 (as a member)
  - `drivers/net/usb/rl152.c` line 715 (as a member)
  - `drivers/net/wireless/ath/wcn36xx/hal.h` line 2454 (as a member)
  - `drivers/net/wireless/ath/wcn36xx/hal.h` line 2464 (as a member)
  - `drivers/net/wireless/intel/iwlwifi/fw/api/scan.h` line 368 (as a member)
  - `drivers/net/wireless/intel/iwlwifi/fw/api/scan.h` line 623 (as a member)
  - `drivers/scsi/hcr53c8xx.c` line 1348 (as a member)
  - `drivers/usb/host/isp116x.h` line 324 (as a member)
  - `drivers/usb/host/isp1362.h` line 454 (as a member)
  - `drivers/usb/host/s1811.h` line 189 (as a member)
  - `include/net/ip_vs.h` line 719 (as a member)
- Defined in 1 files as a function:** A box highlights the entry: `kernel/sched/core.c` line 3427 (as a function).
- Referenced in 452 files:** A list of files that reference the `schedule` function, including:
  - `arch/alpha/kernel/entry.S` line 556
  - `arch/alpha/kernel/signal.c` line 539
  - `arch/arc/kernel/entry-compact.S` line 354
  - `arch/arc/kernel/entry.S` 2 times
  - `arch/arm/kernel/signal.c` line 642
  - `arch/arm64/kernel/signal.c` line 918
  - `arch/s390/kernel/entry.c` line 100

Figure 5: Results of the name search

The screenshot shows the bootlin website interface displaying the declaration of the `schedule()` function. The search path is set to `/ include / linux / sched.h`. The search results show the function declaration and its associated structures and macros.

```

172 extern long schedule_timeout(long timeout);
173 extern long schedule_timeout_interruptible(long timeout);
174 extern long schedule_timeout_killable(long timeout);
175 extern long schedule_timeout_uninterruptible(long timeout);
176 extern long schedule_timeout_idle(long timeout);
177
178 asmlinkage void schedule(void);
179 extern void schedule_preempt_disable(void);
180
181 extern int __must_check lo_schedule_prepare(void);
182 extern void lo_schedule_finish(int token);
183 extern long lo_schedule_timeout(long timeout);
184 extern void lo_schedule(void);
185
186
187 /*
188  * struct prev_cputime - snapshot of system and user cputime
189  * @utime: time spent in user mode
190  * @stime: time spent in system mode
191  * @block: protects the above two fields
192  *
193  * Stores previous user/system time values such that we can guarantee
194  * nonatomicity.
195  */
196 #ifdefined CONFIG_VIRT_CPU_ACCOUNTING_NATIVE
197 struct prev_cputime {
198     u64 utime;
199     u64 stime;
200     raw_spinlock_t lock;
201 };
202 #endif
203
204 /*
205  * struct task_cputime - collected CPU time counts
206  * @utime: time spent in user mode, in nanoseconds
207  * @stime: time spent in kernel mode, in nanoseconds
208  * @sum_exec_runtime: total time spent on the CPU, in nanoseconds
209  *
210  * This structure groups together three kinds of CPU time that are tracked for
211  * threads and thread groups. Most things considering CPU time want to group
212  * these counts together and treat all three of them in parallel.
213  */
214 struct task_cputime {
215     u64 utime;
216     u64 stime;
217     unsigned long long sum_exec_runtime;
218 };
219
220 /* Alternate field names when used on cache exiprations: */
221 #define VIRT_EXP utime
222 #define PROF_EXP stime
223 #define SCHED_EXP sum_exec_runtime
224
225 enum vtine_state {
226     /* Task is sleeping or running in a CPU with VTINE inactive: */
227     VTINE_INACTIVE = 0,
228 };
  
```

Figure 6: Declaration of the schedule() function

```

linux / kernel / sched / core.c All symbols Search Identifier
Filter tags
v4.20
v4.19
v4.18
v4.17
v4.16
v4.15
v4.15.18
v4.15.17
v4.15.16
v4.15.15
v4.15.14
v4.15.13
v4.15.12
v4.15.11
v4.15.10
v4.15.9
v4.15.8
v4.15.7
v4.15.6
v4.15.5
v4.15.4
v4.15.3
v4.15.2
v4.15.1
v4.15
v4.15rc9
v4.15rc8
v4.15rc7
v4.15rc6
v4.15rc5
v4.15rc4
v4.15rc3
v4.15rc2
v4.15rc1
v4.14
v4.13
v4.12
v4.11
v4.10
v4.9
v4.8
v4.7

3421      * make sure to submit it to avoid deadlocks.
3422      */
3423      if (blk_needs_flush_plug(task))
3424          blk_schedule_flush_plug(task);
3425      }
3426
3427  asmlinkage __visible void __sched schedule(void)
3428  {
3429      struct task_struct *task = current;
3430
3431      sched_submit_work(task);
3432      do {
3433          preempt_disable();
3434          __schedule(false);
3435          sched_preempt_enable_no_resched();
3436      } while (need_resched());
3437  }
3438  EXPORT_SYMBOL(schedule);
3439
3440  /*
3441   * synchronize_rcu_tasks() makes sure that no task is stuck in preempted
3442   * state (have scheduled out non-voluntarily) by making sure that all
3443   * tasks have either left the run queue or have gone into user space.
3444   * As idle tasks do not do either, they must not ever be preempted
3445   * (schedule out non-voluntarily).
3446   */
3447   * schedule_idle() is similar to schedule_preempt_disable() except that it
3448   * never enables preemption because it does not call sched_submit_work().
3449   */
3450  void __sched schedule_idle(void)
3451  {
3452      /*
3453       * As this skips calling sched_submit_work(), which the idle task does
3454       * regardless because that function is a nop when the task is in a
3455       * TASK_RUNNING state, make sure this isn't used someplace that the
3456       * current task can be in any other state. Note, idle is always in the
3457       * TASK_RUNNING state.
3458       */
3459      WARN_ON_ONCE(current->state);
3460      do {
3461          __schedule(false);
3462      } while (need_resched());
3463  }
3464
3465  #ifdef CONFIG_CONTEXT_TRACKING
3466  asmlinkage __visible void __sched schedule_user(void)
3467  {
3468      /*
3469       * If we come here after a random call to set_need_resched(),
3470       * or we have been woken up remotely but the IPI has not yet arrived,
3471       * we haven't yet exited the RCU idle mode. Do it here manually until
3472       * we find a better solution.
3473       */
3474      /* NB: There are buggy callers of this function. Ideally we
3475       * should warn if prev_state is CONTEXT_USER, but that will trigger
3476       */
3477  }
3478  #endif

```

Figure 7: Definition of the schedule() function