VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PLUGINS FOR GETTING INFORMATION ABOUT THE SYSTEM FOR BUSYBOX

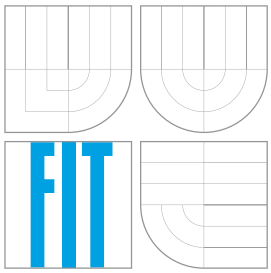
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MAREK POLÁČEK

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PLUGINY PRO ZÍSKÁVÁNÍ INFORMACÍ O SYSTÉMU PRO PROJEKT BUSYBOX

PLUGINS FOR GETTING INFORMATION ABOUT THE SYSTEM FOR BUSYBOX

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MAREK POLÁČEK

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2011

Abstrakt

Tato práce se zabývá implementací nástrojů pro získávání informací o operačním systému pro projekt Busybox. Diskutovány jsou souborové systémy sysfs a procfs v operačním systému Linux. Dále se práce zabývá tím, jak vytvářet co nejmenší programy v jazyce C. Také se věnuje struktuře programů `iostat`, `mpstat` a `powertop`. V rámci práce byly vytvořeny minimalistické implementace již existujících nástrojů, zejména z balíku sysstat, který obsahuje například utility `iostat` a `mpstat`.

Abstract

In this thesis, we discuss implementation of tools for getting information from the system. We examine file systems sysfs and procfs in the Linux operating system. Furthermore, we discourse how to write small programs in the C language. Eventually, we take a look at implementation of tools like `iostat`, `mpstat` and `powertop`. These tools were implemented in a minimalistic form suitable for Busybox within this thesis.

Klíčová slova

Busybox, mpstat, iostat, powertop, sysfs, procfs, informace o systému, gcc optimalizace, Linux, jazyk C, překladače, knihovny, UNIXové utility.

Keywords

Busybox, mpstat, iostat, powertop, sysfs, procfs, system stats, gcc optimizations, Linux, C language, compilers, libraries, UNIX utilities.

Citace

Marek Poláček: Plugins for Getting Information about the System for BusyBox, bakalářská práce, Brno, FIT VUT v Brně, 2011

Plugins for Getting Information about the System for BusyBox

Statement

I hereby state that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged. This thesis was elaborated under the supervision of doc. Ing. Tomáš Vojnar, Ph.D.

.....
Marek Poláček
May 13, 2011

Acknowledgements

Above all, I would like to thank my colleagues from Red Hat, especially Denys Vlasenko, Petr Müller, and Ivana Vařeková for their continuous professional and moral support. I would further like to thank very much doc. Ing. Tomáš Vojnar, Ph.D. for feedback. Without these people, I would have never finished this thesis. Thank you.

Contents

Chapter 1

Introduction

Today, embedded devices are ubiquitous. We use mobile phones, PDAs, music players, cameras, printers, etc. Computer networking would never work without routers and bridges. New microwave ovens, smart fridges, and washing machines also include embedded devices to provide additional features and convenience. There are embedded devices in critical environments like hospitals, plains, trains, and cars. Consequently, their failures may cause serious problems.

1.1 Motivation

Embedded devices of course run software. Developers write the code as they would write an ordinary application code. This is often not optimal. For embedded devices, some additional techniques must be applied to produce decent code. Some knowledge is necessary to write optimized code. Here, optimized mainly means "as minimal as possible". This is because of many microprocessors have only a small amount of available RAM memory, often have no hard disks to keep down the power consumption. However, there are other requirements too: The programs must also be stable and secure. Nobody would want to restart his router every other day. Often, it is desirable to not implement all the possible features but instead only pick the really needed ones. This means more straightforward and smaller code, which in turn decreases the number of bugs.

1.2 Goals

The goal of this thesis is to provide a minimalist set of utilities for determining information about the operating system for the Busybox project. Subgoals of this main goal are to get acquainted with the philosophy and the structure of the Busybox, get familiar with various techniques how to write as small programs as possible and to summarize these techniques for further use. This requires good knowledge of available C libraries, compilers, assemblers, and linkers. Since the tools for getting information about the Linux operating system to a large extent use the `sys/` and `proc/` file systems, we need to examine these as well. Last but not least we need to acquaint ourselves with the existing implementation of the `mpstat`, `iostat`, and `powertop` tools.

1.3 Structure of the Thesis

In this thesis, we will take a close look at one of the best-known open source projects called Busybox which is often used in various embedded devices and minimalist operating systems. In Chapter 2, we will examine this project more thoroughly.

Chapter 3 speaks about how to keep data small. As we said, the size of programs which are intended to run on embedded devices is very important. So we will discuss what the programmer should always bear in mind when writing this kind of applications.

Since, in this thesis, we are mainly interested in programs for getting information about the system, we need to examine the `proc` and `sys` file systems in detail because these are used heavily. Chapter 4 is devoted to them.

Chapter 5 focuses on implementation of three tools which were created as a part of this thesis. These are: `iostat` (Section 5.2), `mpstat` (Section 5.1) and `powertop` (Section 5.3).

In Chapter 6, we close this thesis by a general summary. The achieved goals and future work will be discussed therein.

Chapter 2

Busybox

This chapter discusses the Busybox project. First of all, we will introduce this project: what it actually is, what can it do for us and where it is used. Then we will speak about how to obtain, install, and configure Busybox. Furthermore, we will briefly talk about cross-compiling, which is a technique of creating executables for a different architecture than the one the compiler is currently run. This is often the one and only way for embedded platforms upon which it is not possible to run the compiler. At the end of the chapter, we will demonstrate how one can add a new applet into Busybox.

2.1 What It Is

Busybox is very frequently described as "The Swiss Army Knife of Embedded Linux" [3]. This phrase is not chosen accidentally: Busybox is in fact one application that provides a set of standard UNIX utilities. It is meant to be as small as possible, therefore the utilities in Busybox have fewer options than the utilities from GNU Coreutils [10]. Nevertheless, the behavior of various program options is the same like the behavior of their coreutils' counterparts.

Busybox can easily be customized to contain only desired utilities and not the unneeded ones which is what shrinks the size of the executable to the bare minimum.

It is even possible to create a working operating system consisting only of the Linux kernel and Busybox (plus some nodes in the `dev/` directory).

Busybox adheres to the Single Unix Specification version 3 standard, specifically the "Shell and Utilities" section [24].

There are quite many places where Busybox can be found. A list of products using Busybox can be found here: `http://www.busybox.net/products.html`. For example, `dd-wrt` router firmware, Debian, Slackware and Red Hat installers, the Android operating system, the Tiny Core Linux, and ZyXEL routers all use Busybox.

2.2 Design

Having all the utilities in just one binary, called `busybox`, brings certain advantages. For example, we need just one set of ELF[1] headers. Every ELF file has an ELF header which defines the structure of a file—whether 32bit or 64bit, the type of the file, the byte ordering used, the architecture, virtual address of entry point, and so on. This header also contains

[1] Executable and Linking Format [25].

magic bytes at the very start of the file. Furthermore, every stand-alone executable has to hold a program header table, and every relocatable ELF object must contain a section header table. Additionally, only one virtual address space is used. When all the utilities occupy one virtual address space altogether, it is then possible to share various resources between particular utilities—for example, instructions or global data.

We can split the inner structure of Busybox into two main parts:

- `libbb` and
- applets.

These two parts closely cooperate. Now we will shortly analyze these two components.

2.2.1 Applets

A Busybox applet is just a regular utility, such as `sort`, which nevertheless has a defined interface (examined in detail in Section 2.3). The applets are divided into different subdirectories of main Busybox directory according to their purpose and coverage. E.g., we have the directory `editors/` where we can found applets for editing and manipulating text, like `ed, vim, sed` are. The other notable directories are `coreutils`, `findutils`, `loginutils`, `mailutils`, `networking`, and `util-linux`. The complete directory tree can be viewed in Busybox git repository [2].

We will take a detailed look at a single applet in Section 2.3.

Beside applets, all directories also contain files `Config.in` and `Kbuild`. These are a part of the `Kbuild` system. `Config.in` defines entries for the configuration menu, which can be invoked by `make menuconfig` (more on this later). In `printutils/`, `Config.in` contains the name of current menu section—`''Print Utilities''` and for every applet there is information like help text, program short name, and a variable, which tells us if this particular applet should be incorporated into Busybox executable by default. The `Kbuild` file grants information on how to build applets.

There are more ways how to actually run an applet. One way is to run:

```
$ busybox <applet>
```

However, the usual way is to create a symbolic link which is named after the command we want to run and which points to the Busybox executable. The command is then run automatically. This works because the main Busybox executable will get the name of the symbolic link file—the pointer to this name is stored in `argv[0]`[2]. For example, we can do this:

```
$ ln -s busybox comm
$ ./comm
```

The process of creating symbolic links can be automated utilizing the `--list` option of Busybox executable. Then it could look like:

[2]This is the argument vector. The kernel pushes these addresses on the User Mode stack when executing the `execve()` system call. More specifically, it is done by calling `__put_user()` in the `create_elf_tables()` function. Interested readers might want to look at the `linux/fs/binfmt_elf.c` file.

```
$ mkdir -pv bb
$ for i in $(busybox --list); do
    ln -s busybox bb/$i
  done
```

To see the help text for a specific applet, one can use the `--help` option:

```
$ ./zcat --help
```

2.2.2 libbb

Busybox has its own library, called `libbb`, where all commonly used functions reside. Like we said earlier, it is desirable to share as much code as possible. This way, we reduce the code size, and it is also easier to maintain the code.

We can encounter quite a lot of functions in the `libbb` directory presently. Many of them start with the prefix "x". These functions are basically wrappers around system or library calls which are testing return values of underlying functions or are automatically retrying requests[3]. For illustration, here is a function for safely opening a file:

```
FILE* FAST_FUNC xfopen(const char *path, const char *mode)
{
  FILE *fp = fopen(path, mode);
  if (fp == NULL)
    bb_perror_msg_and_die(``can't open '%s''' path);
  return fp;
}
```

We can see that the `xfopen` function either returns a valid `FILE` pointer or crashes the whole program. Then there is no need to check the return value in the source code of applets, which makes code simpler and smaller. The `FAST_FUNC` macro will be discussed in Section 2.3

Apart from the wrappers, there are also other often used functions: in `getopt32()` for a convenient parsing of command-line parameters, linked list management functions, various string-to-number functions or, for example, the `get_cpu_count()` function for determining the number of CPUs contributed by the author of this thesis.

The only thing needed for using these functions is to include the `libbb.h` header to provide function declarations, constant symbols, and various macros.

2.2.3 NOFORK and NOEXEC Applets

By and large, when running a program from another program, the scheme is to call the `fork` system call, which creates an exact clone of the calling process. If this system call succeeds, there will be two processes, parent and child, which have the same memory image, the same environment strings, and the same open files [32]. After this, the child process executes the `execve` system call which replaces its memory image, and then the new program is run.

These system calls are not cheap. They use a lot of CPU time and occupy memory. Thence, they are slow. So UNIX shells try to avoid them by re-implementing some UNIX

[3] Most functions return `-1` or `NULL` when a failure occurrs. Some failures are, however, recoverable which is usually notified by setting the `errrno` value to `EINTR`. In such a situation, it is appropriate to repeat the call. In the `unistd.h` header file there is a handy macro `TEMP_FAILURE_RETRY` for handling such situations.

commands internally, e.g., `echo`. This way the combination of `fork` and `execve` is not needed. The same situation applies to some Busybox applets which are marked as `NOFORK` or `NOEXEC`.

The `NOEXEC` tag means that `execve` is not called when starting the given program: after `fork`ing a new process the applet's main entry function is called. However, one needs to be careful because we cannot assume the global data on the heap to be initialized properly, thus we may need to re-initialize them using, e.g., the `memset` function. Another pitfall is that the parent process might have set the buffering somehow differently than we would expect.

The `NOFORK` applets do not use the `fork` system call nor the `execve` system call—a new applet is run just by calling its main entry function. This significantly reduces the overhead, but brings a lot of limitations which the programmer must be aware of. Except the `NOEXEC` limitations, one cannot, for example, ever use the `exit` or `exec` functions, one must not leave malloced blocks unfreed, one cannot use `xmalloc` freely, and one must always close opened file descriptors, and restore various flags, terminal settings, and signal handlers.

It should be obvious now that applets marked as `NOEXEC`/`NOFORK` are quite hard to write, and it is not an easy thing to maintain them. Therefore, only simple and very often used applets should make use of this interesting feature.

2.3 A Closer Look at an Applet

Let us now examine one single applet more deeply. Almost every applet starts by including the `libbb` header:

```
# include ‘‘libbb.h’’
```

Now it is possible to use the Busybox library. The `libbb` header includes all the commonly used header files like `stdio.h`, `unistd.h`, `dirent.h`, `fcntl.h`, `stdarg.h`, and so on. However, sometimes this is not enough—if we needed the definition of, say, `struct utsname`, we would have to include `sys/utsname.h` as well.

Afterwards there is commented-out information for the `Kbuild` system. This defines the menu entries, and we have already talked about them in Section 2.2.1:

```
//kbuild:lib-$(CONFIG_IOSTAT) += iostat.o

//config:config IOSTAT
//config:   bool ‘‘iostat’’
//config:   default y
//config:   help
//config:     Report CPU and I/O statistics
```

Then there is information which ends up in `include/applets.h` file which basically stores a list of all available applets in Busybox. The line looks like:

```
//applet:IF_IOSTAT(APPLET(iostat, BB_DIR_BIN, BB_SUID_DROP))
```

Finally, there is also stored the usage text displayed when using the `--help` command-line option, both in the short and long variants. After processing, this text will be stored in the `include/usage.h` file.

```
//usage:#define iostat_trivial_usage
//usage:       ,,[-c] [-d] [-t] [-z] [-k|-m] [ALL|DEV...] [ITV [COUNT]]‘‘
//usage:#define iostat_full_usage ,,\n\n‘‘
//usage:       ,,Report CPU and I/O statistics\n‘‘
//usage:     ,,\nOptions:‘‘
//usage:     ,,\n    -c  Show CPU utilization‘‘
//usage:     ,,\n    -d  Show device utilization‘‘
//usage:     ,,\n    -t  Print current time‘‘
//usage:     ,,\n    -z  Omit devices with no activity‘‘
//usage:     ,,\n    -k  Use kb/s‘‘
//usage:     ,,\n    -m  Use Mb/s‘‘
```

All this commented-out information are automatically processed by the `gen_build_files.sh` shell script located in `scripts/` sub-directory.

2.3.1 Symbols Visibility

Let us now concentrate on the actual code of an applet. The essential function of every applet is a function declared as:

```
int foo_main(int argc, char **argv) MAIN_EXTERNALLY_VISIBLE;
```

This is the main entry point of an applet and the one and only function that is visible outside of the applet. All other functions should be declared with the `static` keyword. Adding the keyword `static` means that a variable or a function will not be exported[4]. This further brings certain advantages—we need fewer relocation[5] (we still might have a few relative relocations but this is not a big deal since relative relocation is handled by `ld`[6] within the linking phase) and fewer PLT[7] entries. This speeds up the execution of the program and also allows the compiler to optimize functions marked as `static` more aggressively. The `static` keyword is occasionally dropped because of the fact that compiler can auto-inline only `static` functions[8], which is sometimes unwanted because inlining has often negative impact on the size of the code. Nevertheless, this is not needed anymore since we can tell the compiler[9] to never inline a particular function using `__attribute__ ((noinline))`. So the rule is to always use the `static` keyword when defining/declaring a function except the `foo_main` function.

Attentive readers might have noticed the `MAIN_EXTERNALLY_VISIBLE` keyword in the declaration of the `foo_main` function. When the GCC version is 4.1 or newer, this macro expands to `__attribute__ ((visibility(,,default‘‘)))`[10]. To understand why this is

[4] The GNU `as` directive `.global` will disappear. The linker cannot see symbols without this directive. More on this in [30].

[5] Computing and assigning run-time addresses to symbols.

[6] The GNU Linker [31].

[7] Procedure Linkage Table.

[8] In fact, this is not entirely true. GCC may auto-inline even non-static functions when the `-finline-functions` option is used. This option is included in optimization levels `-Os`, `-O3`, and `-Ofast`. The `-Ofast` optimization level was introduced in GCC version 4.6.

[9] In this thesis we assume GCC, the GNU Compiler Collection [22].

[10] Before GCC 3.1 the programmer would have to use `asm` to add the information [27]: `int foo; asm (‘‘.hidden foo’’);`.

needed, we have to be thoroughly familiar with the ELF visibility notion and with the GCC's `-fwhole-program` optimization option.

The generic ELF ABI defines visibility of symbols [28]. ELF symbol visibility was invented to provide more control over which symbols were accessible outside a shared library [11]. There are four different types of visibility[11]:

- `STV_DEFAULT`:
  default symbol visibility rules, global symbols are visible everywhere.
- `STV_HIDDEN`:
  the symbol is unavailable in other modules, the symbol is not visible outside the current executable or shared library.
- `STV_INTERNAL`:
  a processor specific hidden class.
- `STV_PROTECTED`:
  not preemptible, not exported.

With GCC 4.0 and newer there is a command-line option `-fvisibility`. This option can be used to change default visibility of all symbols, e.g. `-fvisibility=hidden` will make sure that all symbols are defined with `STV_HIDDEN` unless specified otherwise [28]. One should be aware that the `-fvisibility` flag only affects definitions and not declarations. However, it is possible to define per-symbol visibility using GCC attributes[12]. This works also for declarations. And this is exactly what the `MAIN_EXTERNALLY_VISIBLE` macro does: it explicitly tells the compiler that `foo_main` should have bear the default visibility.

2.3.2 The `-fwhole-program` Option

Now let us briefly talk about the GCC's `-fwhole-program` optimization option. This option is available since GCC 4.1. This optimization basically tells the compiler to treat all public functions (except the `main` function) and variables as if they were be declared with the `static` keyword. This allows the compiler to perform some further optimization which would not be possible otherwise [9]. The problem is that in Busybox applet source codes there are no `main` functions! However, the `foo_main` functions ultimately must be exported for the reason that they have to be callable from the main Busybox executable. This is precisely the reason why one marks the applet's main functions with default visibility. Note that this is needed only when we are compiling with the `-fwhole-program` option. There is another way how to achieve this: use the `externally_visible` attribute, which nullifies the effect of the `-fwhole-program` option [9]. This attribute can be used on both functions and variables.

2.3.3 The `regparm` and `stdcall` Attributes

At times, we can see functions marked with `FAST_FUNC` in Busybox. This is especially true for the `libbb` library. This macro expands to nothing on all architectures but x86[13]. On x86, this expands to `__attribute__ ((regparm(3), stdcall))`. These attributes enforce

[11] Description taken from `elfutils/libelf/elf.h`.
[12] Another way is to use `#pragma gcc visibility push(default)` and `#pragma gcc visibility pop`.
[13] Also, GCC 3.1 or newer is needed.

specific calling conventions which possibly may make function calls faster and smaller[14] . This is usually used on non-`static` functions since the `static` functions are optimized automatically by the compiler[15].

The `stdcall` attribute tells compiler to assume that a called function will pop arguments from the stack. This ordinarily does the caller. This attribute has no effect on functions taking a variable number of arguments. Normally, the caller must adjust the stack using the instruction `addl`, which has three bytes. With the `stdcall` attribute, the stack should be corrected by the callee using the instruction `ret` which occupies two bytes [29]. Nonetheless, this attribute must be used with care because nowadays the compilers are more and more clever, and they actually do not have to clean up the stack after every function call, but instead they are eligible to do so after several function calls. Thus the size advantage may suddenly very well become a disadvantage.

The `regparm` attribute takes one parameter: an integer. It causes the compiler to pass the specified number of arguments in registers instead of on the stack [9]. The registers `eax`, `ecx`, and `edx` may be used[16]. This attribute is ignored for functions taking a variable number of arguments. With this attribute, we can get rid of unnecessary `movl` instructions, which normally be would used for moving the values from the stack to registers.

2.4 Obtaining Busybox

We now describe how to get, install, and configure Busybox.

Probably the simplest way is to just download pre-compiled binary for the desired architecture from Busybox Prebuilt Binaries [4]. This is a statically linked version, so there are no dependencies. After downloading it, it is only necessary to set the execution the bit, and then the Busybox executable is ready to run:

```
$ wget -q -O bbox busybox.net/downloads/binaries/latest/busybox-‘uname -m‘
$ chmod +xX bbox
$ file -b bbox
ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
stripped
$ ./bbox
```

Another possibility is to download the source codes from the git repository [2] of Busybox and then to manually configure, compile, and install Busybox. In this case, we first obtain the source codes using `git clone`:

```
$ git clone git://busybox.net/busybox.git
```

Now we have to configure Busybox. Configuring Busybox is essentially the same as configuring the Linux kernel. The easiest thing to do is to create a default config file, which includes almost all features but does not burden the executable with debug information:

[14]There are also similar attributes like `fastcall`, `sseregparm`, `cdecl`, and `thiscall`. These will not be described here.

[15]Here techniques like Value Range Propagation are used.

[16]People who are familiar with the kernel code might know this attribute already. In kernel there is a frequently used macro named `asmlinkage` which is just a synonym for `__attribute__((regparm(0)))` on x86. Its effect is that all arguments are always passed on the stack and not in registers. For example, all system calls are marked this way. See `include/linux/syscalls.h`.

```
$ make defconfig
```

Also, we can create a minimal config file with all the features turned off:

```
$ make allnoconfig
```

Then there is an option to enable absolutely everything:

```
$ make allyesconfig
```

For people who want to set everything manually there is an option which makes it possible to use a ncurses-based menu interface:

```
$ make menuconfig
```

After configuring, we can compile Busybox right off:

```
$ make
```

This produces a dynamically-linked executable. If we wanted a statically linked version of the Busybox executable, we would have to set the environment variable `LDFLAGS`:

```
$ LDFLAGS=''--static'' make
```

Last but not least, it is possible to cross-compile Busybox for another architecture than we are currently on:

```
$ make CROSS_COMPILE=armv4tl-
```

When everything is compiled, we can step up to the last part, installing Busybox:

```
# make install
```

There exist also another `make` options which we will not describe here. For displaying these options, just use:

```
$ make help
```

2.5 How To Add a New Applet

We will explain how to create and add a simple applet to Busybox. The way how to add an applet has recently changed. Previously, one had to create an applet and then modify other files too, such as `usage.h`, `applets.h`, `Kbuild`, and `Config.in`. These files were described in Section 2.3. Fortunately, this is no longer needed: all one has to do is just to write an applet, and the build system takes care of everything else.

Now we will write a very simple applet—a `getpid` program. Its functionality is to just print its own Process ID. In the Linux operating system, one can use the the `sys_getpid` for this purpose system call[17]. Figure 2.1 shows how the `getpid` applet can look like.

At this point, everything that is needed is to configure and compile Busybox. After this, the new applet can be run:

```
$ ./busybox getpid
Current PID: 2768
```

[17]This call in fact returns the thread group id of the current process. The `tid` and `pid` are usually identical.

```
//applet:IF_GETPID(APPLET(getpid, BB_DIR_BIN, BB_SUID_DROP))

//kbuild:lib-$(CONFIG_GETPID) += getpid.o

//config:config GETPID
//config:       bool ``getpid''
//config:       default y
//config:       help
//config:         Print the PID of current process.

//usage:#define getpid_trivial_usage
//usage:        ```
//usage:#define getpid_full_usage ``\n\n''
//usage:        ``Print the PID of current process''

#include ``libbb.h''

int getpid_main(int argc, char **argv) MAIN_EXTERNALLY_VISIBLE;
int getpid_main(int argc UNUSED_PARAM, char **argv UNUSED_PARAM)
{
    printf(``Current PID: %u\n'', getpid());

    return EXIT_SUCCESS;
}
```

Figure 2.1: An example of a Busybox plugin—the `getpid` program

Chapter 3

How to Keep Data Small

In this chapter we will discuss several techniques how to write smaller programs, based on the author's own experience. This by no means is a complete list. We will talk about available libraries, about how to write programs with the size aspects in mind, and last but not least, we will examine various compiler/linker options, which could help us to reduce the size of our programs. We will also tackle a question how to make already compiled programs even smaller. Some of these ideas will be documented using illustrative examples in the C programming language. Many of these tricks were used when the author was writing plugins for Busybox.

3.1 About Libraries

Almost every program depends on some library, that is, it uses functions which some library defines. The most important library is the C library. Since Busybox is always linked with some C library, be it statically or dynamically, we need to know which available library will serve our purpose best. The C library is essential to almost all user space programs[18] and must provide at least what the ISO C standard defines [12]. The C library also provides system call wrappers that enable user space applications to interface with the kernel. Nonetheless, the C library might be broadened to cover the system-dependent extensions as well[19]. There is not just one single C library, there are more of them. Now we will go into the merits of available C libraries.

3.1.1 The GNU C Library

Probably the most commonly used C library in the Linux world is the GNU C library [21]. This library provides stable, portable, high performance, and rich collection of functions. In addition, this library is also very well internationalized. The GNU C library moreover contains add-on packages[20] such as:

- `libidn`: a library for handling internationalized domain names,

[18] With GCC, it is, however, possible to bypass linking programs with default C library by using the `-nostdlib` or `-nodefaultlibs` command-line options.

[19] The extensions are guarded by the feature test macros. An example of using this macro is the code `#if defined __USE_POSIX` in the `stdio.h` header file.

[20] These add-ons can be enabled or disabled when configuring `glibc` using the `--enable-add-ons` command-line option.

- `nptl`: a native POSIX thread library,
- `nss`: a library for the name service switch,
- `resolv`: lightweight resolver in Berkeley Internet Name Daemon,
- `crypt`: DES cryptographic functions.

This broad scale of features has unfortunately also some negative impact on the size of this library. Usually, not all the functions and facilities are really needed. Especially small devices suffer from this fact: a big footprint makes the devices run slowly—functions, which are not used take up the virtual address space, tend to fragment the page set for the functions that are actually used, fragment the icaches[21] and they are also causing more TLB[22] misses. This restraint contributed to the fact that new C libraries were developed. We now explore some of alternate C libraries that are nowadays available for the Linux operating system.

3.1.2 Alternate C Libraries

These libraries emphasize an approach based on reducing the size of the C library by removing unnecessary functions and add-ons. This approach is more embedded devices-friendly because with these C libraries the user applications are smaller and the amount of memory that they require to execute is not so big.

Here is an overview of various GNU C library alternatives:

- *eglibc*
  is a fork of `glibc` which is designed to work well on embedded systems [7]. Eglibc means "embedded glibc". Eglibc should be binary compatible with `glibc`. Furthermore, eglibc lays emphasis on easy cross-compiling.
- *uClibc* is a C library for embedded Linux [26]. This library is probably the most complete alternative to `glibc`. The main goal is nevertheless the size of the library. uClibc was originally developed for the uClinux project. uClibc can also run on MMU-less systems. This C library is very often used with Busybox and is closely related to the Buildroot project [1].
- *Newlib* is a C library intended for use on embedded systems [23]. It consists of several parts. This library can be used on systems without any operating system. A remarkable thing is that this library also contains a small math library. It is easy to port this library to another operating system [17].
- *dietlib* is another C library optimized for size [5]. This library comes with its own driver which runs gcc and links the program with the dietlibc automatically. However, this library contains quite a lot of bugs which make it uneasy to use this library on embedded devices.
- *klibc* this library is mainly intended to run during the Linux boot process, but it can also be used on embedded devices [14].

[21] These are the instruction caches.
[22] Translation Lookaside Buffer.

Which of these libraries should we use is an open question. When using Busybox on a desktop PC, it is sensible to just use the GNU C library because on desktop PC, we usually have a lot of memory available. However, probably the most proven library which is intended to cooperate with Busybox on embedded devices is the uClibc library.

3.2 Using Proper Data Types

To save space, it is possible to use smaller data types. So, for example, we try to avoid using data types like `long`, `intmax_t`, `double`, etc. We might spare some space by using types like `short int`, `bool` (C99 only), or `char` in places where we need to use small numbers only. However, sometimes, using the smallest possible type is not as benefical as it may seem. There are situations where using `char` over `int` does not bring any space advantage, for example, a byte access may in fact lead to a larger code on some architectures! Fortunately, this is not true for widely used architectures such as i386, x86_64, and MIPS.

The structure type (the `struct` keyword in C) is often left out. However, we can very easily save here some bytes as well—we have to reorganize the structure layout so there will be no gaps between structure's elements. The structure elements are normally aligned to the 4byte boundary. This does not apply when the structure is defined with the `__attribute__ ((packed))` flag.

We can use the felicitous tool named `pahole` [6]. This software is used for displaying structure layouts. It makes use of the DWARF debugging format. This tool requires that the binary to examine must have been compiled with the GCC's `-g` flag. Then `pahole` can analyze the `.debug_info` section.

Let us have the following not very well organized structure:

```
struct foo
  {
    char c;
    _Complex int i;
    char a[2];
    float f;
    const void *p;
  };
```

Pahole tells us:

```
struct foo {
        char                    c;              /*     0     1 */

        /* XXX 3 bytes hole, try to pack */

        complex int             i;              /*     4     8 */
        char                    a[2];           /*    12     2 */

        /* XXX 2 bytes hole, try to pack */

        float                   f;              /*    16     4 */
```

```
        /* XXX 4 bytes hole, try to pack */

        const void  *              p;                    /*    24     8 */

        /* size: 32, cachelines: 1, members: 5 */
        /* sum members: 23, holes: 3, sum holes: 9 */
        /* last cacheline: 32 bytes */
};
```

Note that the considered structure is 32 bytes big. Now, we try to place the structure elements in a better way:

```
struct foo
  {
    _Complex int i;
    const void *p;
    float f;
    char c;
    char a[2];
  };
```

After this reordering, the size of this structure will be 24 bytes only !

3.3 Eliminate Unnecessary Strings

Strings are used in various warning and error messages. It is wise to use predefined functions of this kind. In context of Busybox this means that one should always use the already defined strings. So, instead of:

```
bb_error_msg_and_die(‘‘memory exhausted’’);
bb_error_msg_and_die(‘‘virtual memory exhausted’’);
bb_error_msg_and_die(‘‘out of memory’’);
...
```

write this:

```
bb_error_msg_and_die(bb_msg_memory_exhausted);
```

These predefined messages can be found in `messages.c` and are in the following form:

```
const char bb_msg_memory_exhausted[] ALIGN1 = ‘‘out of memory’’;
const char bb_msg_invalid_date[] ALIGN1 = ‘‘invalid date ’%s’’’;
...
```

This way, it suffices to have only one set of strings in the executable. Sometimes, however, we need to create our own strings which are unique to an applet. When the strings will not be modified, they should be marked as const. Also, it is often desirable to have exactly one copy of the string. Thus, for constant strings, we should write something like this:

```
static const char str[] = ‘‘I am a string!’’;
```

This string will be saved in a read-only memory. In addition, with the `static` keyword, the `str` is not a variable anymore. It is merely a label for a location in memory. The string will be stored in the section `.rodata`.

However, there are other techniques how to decrease the number of strings needed. GCC and the linker can cooperate and perform a *string merging*. This way, it is possible to share some parts of strings. Currently, this works only on suffixes of the strings.

Furthermore, strings which appear in more object files appear only once in final executable [28].

The linker's string merger then generates data sections marked with the flags `SHF_MERGE` and `SHF_STRINGS`. Below we show a quick example of string merging.

We start with a file `a.c`:

```
#include <stdio.h>

int
main (void)
{
  puts (‘‘hello kids’’);
  puts (‘‘kids’’);
  foo();
}
```

Now, consider a file `b.c`:

```
#include <stdio.h>

void
foo (void)
{
  puts (‘‘hello kids’’);
}
```

We compile this together:

```
$ gcc -fmerge-all-constants a.c b.c
```

The strings are stored in the `.rodata` section:

```
$ eu-readelf --strings=.rodata a.out
String section [15] ’.rodata’ contains 27 bytes at offset 0x5c8:
...
[     f]
[    10]  hello kids
```

From the above output, we can see that there are no unnecessary duplicates of strings, which will save us some bytes.

3.4 Reduce Stack Usage

Stack is a part of the virtual address space where local (auto) variables and function parameters are stored. Also, chunks of the memory allocated using the library call `alloca` take place on the stack. Stack is, especially in context of the embedded devices, a quite precious piece of memory. Allocating too much data on the stack may lead to immediate SIGSEGV[23]. Due to this, the programmer must be very careful.

3.4.1 Passing Parameters to Functions

As we said, stack is heavily used when passing function parameters. Of course, it is possible to use GCC attributes to change this calling convention, but this works only on x86 and does not work for a variable length parameter list (we spoke about this in Subsection 2.3.3). Sometimes we may want to completely avoid passing arguments to functions. This will save us some move instructions, which means a smaller `.text` segment, and thus a smaller executable. We illustrate this by an example

```
#include <stdint.h>

static int
foo (const void *p, intmax_t m, const char *s)
{
  /* ...  */
  return 0;
}

int
main (void)
{
  return foo (main, 1 << 16, __func__);
}
```

We compile the code into assembler as follows[24]:

```
$ gcc -O0 -fverbose-asm -dA -S stack1.c -o -
```

The resulting assembler code given below shows us calling the `foo` function[25] (only the important instructions are displayed):

```
foo:
    pushq   %rbp    #
    movq    %rsp, %rbp      #,
    movq    %rdi, -8(%rbp)  # p, p
    movq    %rsi, -16(%rbp) # m, m
    movq    %rdx, -24(%rbp) # s, s
```

[23] We are able to manipulate the maximum size of stack segment by using `ulimit -s` command or system call `setrlimit (RLIMIT_STACK, &rlim)`.

[24] Note that, with GCC optimizations turned on, the parameters would be optimized away. In this example, we do not want this to happen.

[25] It should be clear that this output was produced on x86_64 architecture with frame pointers turned on.

```
    movl    $0, %eax        #, D.1624

main:
    movl $__func__.1628, %edx #,
    movl $65536, %esi #,
    movl $main, %edi #,
    call foo #
```

If we omit all the parameters that the function `foo` takes, we end up with this assembler output:

```
foo:
    pushq   %rbp    #
    movq    %rsp, %rbp      #,
    movl    $0, %eax        #, D.1594

main:
    call foo #
```

It is apparent, that passing the parameters increases the code size quite a lot.

3.4.2 Local Arrays

Local arrays are proper when one wants to use them only in the scope of a function or a block. Often, we do not know how big the array will be, this will be known only at run-time. There are basically two options[26] how to allocate an array on the stack.

The first option is to make use of a nice feature which the C99 standard brought and which is called a *variable lenght array* (VLA) [12]. VLA's size is determined at run-time instead of at compile time. These arrays cannot be initialized. The space used by the VLA is recyclable. An example shows a VLA which takes up to `N` bytes, depending on the iteration of the loops in which an array is allocated and initialized:

```
for (int i = 0; i < N; ++i)
  {
    char arr[i + 1];
    memset (arr, i, i + 1);
  }
```

The second option how to allocate space on the stack is to use the `alloca` function. This function just moves the stack pointer, which is very cheap since this can be done using one instruction. The space allocated by `alloca` will be automatically freed when the function returns. An important fact is, that the space is not recycled! Thus, the example piece of code shown in Figure **??** will occupy $n(n+1)/2$ bytes! This can lead to strange crashes[27], if one allocates a lot of space because then it is possible to accidentaly write to memory which does not belong to the stack. One must be aware of potential risks using `alloca`.

[26] Note that we are talking about arrays on the stack, not on the heap or somewhere else. Thus, the functions like `malloc`, `brk`, and `mmap` are not of our interest right now.

[27] Note that we ought not to check the return value of `alloca` because this call never fails.

```
for (int i = 0; i < N; ++i)
  {
    char *arr = alloca (i + 1);
    memset (arr, i, i + 1);
  }
```

3.5 Reduce Global Data Usage

We have already mentioned that the main Busybox executable consists of many small object files, applets. Every applet's object file contains, among other the `.bss` and `.rodata` sections. These sections will be concatenated into the main Busybox executable, so we wind up with one "big" `.bss` and `.rodata` section. When we run the Busybox executable, all the sections in a segment marked as `PT_LOAD` must be loaded. Below we show the segments (`PhysAddr` column was removed).

```
$ eu-readelf -l busybox
Program Headers:
  Type           Offset   VirtAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000400040 0x0001c0 0x0001c0 R E 0x8
  INTERP         0x000200 0x0000000000400200 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000400000 0x0e8aac 0x0e8aac R E 0x200000
  LOAD           0x0e9000 0x00000000006e9000 0x00100a 0x003580 RW  0x200000
  DYNAMIC        0x0e9028 0x00000000006e9028 0x0001a0 0x0001a0 RW  0x8
  NOTE           0x00021c 0x000000000040021c 0x000024 0x000024 R   0x4
  GNU_EH_FRAME   0x0c6c90 0x00000000004c6c90 0x004f14 0x004f14 R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x000000 0x000000 RW  0x8

Section to Segment mapping:
 Segment Sections...
  00
  01      [RO: .interp]
  02      [RO: .interp .note.gnu.build-id .gnu.hash .dynsym .dynstr
               .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt
       .text .fini .rodata .eh_frame_hdr .eh_frame]
  03      .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
  04      .dynamic
  05      [RO: .note.gnu.build-id]
  06      [RO: .eh_frame_hdr]
  07
```

This remark implies, that we have to avoid using large global data in all applets. This is really crucial point. Now we will present a sub-optimal example which really should not be incorporated into Busybox:

```
#include <string.h>

static char arr[1000];
static int cnt;

int
main (void)
{
  memset (arr, 0x55, sizeof arr);
  cnt = 5;

  return cnt ^ arr[52];
}
```

The program[28] above has the following sizes of the `.bss` and `.rodata` sections.

```
$ eu-size -dA | grep -e bss -e rodata
section                          size                 addr
.rodata                            16              4195816
.bss                             1040              6293664
```

The size could be optimized: we present in the following how the ordinary Busybox applets solve the global variables issue.

We first define our structure `struct globals`. Then we define a constant pointer to this structure. Because we defined this pointer as a `const` pointer, the compiler will know that this pointer will never change. To assign to this pointer, we need to use the `SET_PTR_TO_GLOBALS` macro. After all these modifications, we get:

```
#include <stdlib.h>
#include <string.h>

struct globals
{
  char arr[1000];
  int cnt;
};

struct globals *const ptr_to_globals;

#define G (*ptr_to_globals)
#define SET_PTR_TO_GLOBALS(x) do { \
  (*(struct globals **) &ptr_to_globals) = (void *) (x); \
} while (0)

int
main (void)
{
  SET_PTR_TO_GLOBALS(calloc(1, sizeof(G)));
```

[28] No optimizations were applied.

```
  G.cnt = 5;

  return G.cnt ^ G.arr[52];
}
```

Let us now check the sizes of the sections now:

```
$ eu-size -dA | grep -e bss -e rodata
section                          size               addr
.rodata                            16            4195848
.bss                               24            6293688
```

As we can see, the results are dramatically different.

3.6 Factor Out Functions

All functions that may be useful for more applets should be factored out and moved to `libbb`. Sometimes it is not that easy, and one must untangle the function from original source code. The factored out functions must be as general as possible to make it possible to use them in other applets. This means that the function should not depend on some global variable or have side-effects.

An example of a factored out function is `safe_write` from `libbb`.

3.7 Use Your Compiler and Linker Well

Aside tuning the code, it is more than desirable to know various compiler and linker options that can reduce the code size. With these options, we can maximize our space-saving effort. Even when we write an optimal code, there are many ways how the compiler will interpret this code. The code can be performance-oriented, or space-oriented. We will here describe some useful GCC command-line options which can significantly help us to reduce the size of executable.

3.7.1 GCC Optimization Options

GCC allows us to set the optimization level by specifying one of the `-O` options. These are: `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, and since version 4.6 also `-Ofast`. The level `-Os` means optimizing the code with the size in mind. The `-Os` level enables the `-O2` optimizations that do not increase the code size [9]. Here we are interested only in the `-Os` level. In Figure 3.1 there is a list of all optimization flags which are turned on by the `-Os` level[29]. We will not describe all these options here. Instead, we will devote our attention only to some of these options.

3.7.2 The `-ffunction-sections` Option

The first option which we will talk about is widely used in Busybox as well as in the Linux kernel. Most of the time, this option is used together with the `-fdata-sections` option. The `-ffunction-sections` causes that the GCC will put each function (even the static ones) in its own section. The resulting section will be named `.text.function`. Normally,

[29] This list was generated using `gcc -Q -Os --help=optimizers | grep enable | colrm 40`. An interested reader might also want to take a look at `default_options_table[]` in the `gcc/opts.c` file.

all the functions are in one `.text` section. This gives the linker a better opportunity to do dead code removal and further optimize locality of reference in the instruction space [9]. Similarly, `-fdata-sections` instructs GCC to put the data elements into its own sections.

These two options are best used in combination with the linker option `--gc-sections`[30]. With this option, the linker tries to get rid of unused sections. It works by analyzing the symbol table and relocations. It can, however, discard only entire sections, which is the reason why it makes sense to use `-ffunction-sections` and `-fdata-sections` as well.

One must use these optimizations with caution. Because of the way the linker determines which section should disappear, there might arise problems, for instance, when the symbol in a section is used only via a `dlsym` call. The linker cannot determine this and thus can happily discard a section which, in fact, is used. The GCC attribute `used` may help in a situation like this.

Let us now demonstrate the `-ffunction-sections` option by a short example. We have a static library which contains two functions: `foo` and `bar`. When compiling the source code of this library, we used the `-ffunction-sections` option, so the object file in this library will contain, among other, sections `.text.foo` and `.text.bar`. Now, we will use this library, but we only use the `foo` function. Then, when linking, we use the `--print-gc-sections`. The linker then prints all the sections it is going to discard:

```
$ gcc m.c -Wl,--gc-sections -Wl,--print-gc-sections libfoo.a
/usr/bin/ld: Removing unused section '.text.bar' in file 'libfoo.a(f.o)'
```

This is precisely what we expect: the linker discarded the unused `.text.bar` section.

3.7.3 Data Alignment

Every variable and function is aligned to a certain byte boundary. However, sometimes, GCC uses a too big value to align to, which consumes extra space and generally increases code size. It is true especially for structures. A bigger alignment is by and large beneficial for the performance, which is the reason why the `-O2` optimization level uses higher values to align controlling options, such as `-falign-functions`, `-falign-jumps`, `-falign-loops`, and `-falign-labels`. The gaps are filled with the NOP instructions. Lowering these values might decrease the code size. For instance, Busybox uses the value of 1. When we have a very limited stack space, we can use `-mprefered-stack-boundary=2` to save some space.

We can set the functions, variables and structure members to be aligned to a desired value by using the GCC attribute `aligned`. We will show it in an example where we make use of the GCC keyword `__BIGGEST_ALIGNMENT__`. This is a target dependent macro which, in fact, defines the largest alignment possible for a particular machine. Thus, for instance, the following program has the size of 6275 bytes:

```
int __attribute__ ((aligned (__BIGGEST_ALIGNMENT__)))
main (void)
{
}
```

Nevertheless, when we align the main function to a 2 byte boundary, the executable size decreases to 6259 bytes.

[30] This option can be passed to the linker through GCC using `-Wl,--gc-sections`

3.7.4 Inline Functions

Inlining is a technique with which the body of a function is put on the place where the function is called. This usually has a positive impact on performance because:

- It removes the need of a function call (`call`, `leave`). Also, function prologues/epilogues are no longer needed.
- The code to be executed is closer to itself in memory, which improves the locality of reference, the code is usually held in the instruction cache.
- Additional techniques like value range propagation or constant range propagation can be used more extensively.

Although inlining sometimes might decrease the code size, usually it does the opposite. Thus, we generally try to avoid inlining when writing programs for embedded devices. GCC has a few attributes which allow us to control inlining more precisely. The first one is the `noinline` attribute which prevents a function from being ever inlined. The `always_inline` attribute, on the contrary, tells GCC to always inline that particular function. Furthermore, there is one more interesting attribute: `flatten`, which means that all function calls in the flat function will be inlined.

An inliner is implemented as a GCC pass which tries to inline functions. Whether a function will be inlined depends on several parameters: *unit growth*, *function growth*, and *stack frame growth*. These factors are useful for computing the so called *badness* value. The function can be inlined only until certain badness limit is hit. The tendency is to inline the so called *hot* functions (these are called often) and to inline *cold* functions (unlikely executed) only if it is expected that this decreases the code size. Functions called just once, such as static constructors and destructors are inlined if the stack frame growth limit is not reached. More on this can be found in [13].

The user can to a great extent control what will be inlined. We have already talked about attributes, now we will take a look at some useful GCC options. We can limit the size of functions that should be inlined. This is commonly used in Busybox and it is done using the `finline-limit=N` option. Here the `N` is a number of pseudo instructions. These pseudo instructions do not correspond with the assembly instructions. Pseudo instructions are an abstract measurement of function's size [9]. The `-finline-small-functions` option is quite useful because it does not make the code size bigger. This option will inline functions when they are smaller than the function call code. Probably the most important option is the `-finline-functions` option which causes that the compiler will use its heuristic to decide which functions to inline. The `-finline-functions-called-once` tries to inline all functions that are called only once. The `-fearly-inlining` option will inline functions marked as `always_inline` before the proper inlining pass. This makes the later profiling cheaper.

3.8 Getting Rid of Debug Information

This section suggests how to make an already compiled executable even smaller by removing the debug information.

GCC supports many options which control how much debugging information we get and in which format. There are more debugging formats such as STABS or COFF, but in scope of this thesis, we assume the DWARF Debugging Format [20]. These debugging

information comes in handy when we want to debug our program, e.g., using `gdb`[31], because there will be available the symbol table or line numbers information. However, there are also other tools which rely on debugging information. These tools include `pahole` (see Section 3.2) or `addr2line` (which converts addresses into file names and line numbers).

The most used GCC debug option is certainly the `-g` switch. This switch causes that the compiler will emit debug information. Not many people know that this switch has various levels; so, for example, with `-g3` the compiler will also produce some additional information (macro definitions). Then there are switches specific for the GNU Debugger: `-ggdbN`. There is no point in describing all these options. An interested reader might want to take a look at [8]. The debug data are stored in sections. The names of these sections always start with the `.debug_` prefix. Here is an excerpt of a section dump of an executable with all the debug information, which are split into various sections:

```
[26] .comment              PROGBITS     00000874 0000002d  1 MS      0   0  1
[27] .debug_aranges        PROGBITS     000008a1 00000030  0         0   0  1
[28] .debug_pubnames       PROGBITS     000008d1 0000001b  0         0   0  1
[29] .debug_info           PROGBITS     000008ec 00000090  0         0   0  1
[30] .debug_abbrev         PROGBITS     0000097c 00000043  0         0   0  1
[31] .debug_line           PROGBITS     000009bf 00000169  0         0   0  1
[32] .debug_str            PROGBITS     00000b28 00000086  1 MS      0   0  1
[33] .debug_macinfo        PROGBITS     00000bae 000034f6  0         0   0  1
[34] .debug_pubtypes       PROGBITS     000040a4 00000012  0         0   0  1
```

These debug data of course occupy some space which is often unwanted. Fortunately there is a way how to discard all unnecessary data from an executable—using the `strip` utility. This utility discards also the `.strtab` (which holds the string table) and `.symtab` (which holds the symbol table) sections. The same effect could be achieved using GCC's `-s` option when compiling[32].

In the following we will show how this works. We have a simple Hello World program:

```
#include <stdio.h>

int
main (void)
{
  printf (``Hello World\n'');
}
```

Now we compile it with a lot of debug information (according to C99, we do not need to write the `return 0;` statement at the end of the `main` function):

```
$ gcc -std=gnu99 -g3 x.c -o hello
```

This file is not stripped yet:

```
$ file -b hello
ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), dynamically linked
    (uses shared libs), for GNU/Linux 2.6.18, not stripped
```

[31] This is the GNU Debugger.

[32] More precisely, when linking. If we used the `-c` option, which supresses the linking phase,`-s` would have no effect.

We print out the size[33]:

```
$ stat --format=,,Size: %s Bytes‘‘ hello
Size: 21701 Bytes
```

This is just too much. So we will strip this file and then again print out the size:

```
$ eu-strip hello
$ file -b hello
ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), dynamically linked
    (uses shared libs), for GNU/Linux 2.6.18, stripped
$ stat --format=,,Size: %s Bytes‘‘ hello
Size: 4240 Bytes
```

With this scoured executable we saved around 17 kilobytes. This means that stripping the binaries is definitely something one should consider.

[33]Please note that we do not use the `eu-size` utility to print out the size, because this utility counts only sections marked as `SHF_ALLOC`, which means all sections not used at run-time are ignored. And debug sections are not used at run-time.

```
-falign-functions
-falign-jumps
-falign-labels
-fargument-alias
-fasynchronous-unwind-tables
-fbranch-count-reg
-fcaller-saves
-fcommon
-fcprop-registers
-fcrossjumping
-fcse-follow-jumps
-fdce
-fdefer-pop
-fdelete-null-pointer-checks
-fdse
-fearly-inlining
-fexpensive-optimizations
-fforward-propagate
-fgcse
-fgcse-lm
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions
-finline-functions-called-once
-finline-small-functions
-fipa-cp
-fipa-pure-const
-fipa-reference
-fivopts
-fjump-tables
-fmath-errno
-fmerge-constants
-fmove-loop-invariants
-fomit-frame-pointer
-foptimize-register-move
-foptimize-sibling-calls
-fpeephole
-fpeephole2
-fregmove
-frename-registers
-fweb
-freorder-blocks
-freorder-functions
-frerun-cse-after-loop
-fsched-interblock
-fsched-spec
-fsched-stalled-insns-dep
-fschedule-insns2
-fsigned-zeros
-fsplit-ivs-in-unroller
-fsplit-wide-types
-fstrict-aliasing
-fthread-jumps
-ftoplevel-reorder
-ftrapping-math
-ftree-builtin-call-dce
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-copyrename
-ftree-cselim
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-fre
-ftree-loop-im
-ftree-loop-ivcanon
-ftree-loop-optimize
-ftree-pre
-ftree-reassoc
-ftree-scev-cprop
-ftree-sink
-ftree-sra
-ftree-switch-conversion
-ftree-ter
-ftree-vect-loop-version
-ftree-vrp
-funit-at-a-time
-fvar-tracking
-fvar-tracking-assignments
-fvect-cost-model
```

Figure 3.1: GCC `-Os` optimization flags

Chapter 4

The procfs and sysfs File Systems

We now turn our attention to two virtual file systems in Linux. These two file systems, `procfs` and `sysfs`, are essential for the tools we set out to write. Basically, the tools for getting information about the system only read certain system files, do some computation over data obtained from there, and preferably also format the output to be more user-friendly.

We begin with a general description of `procfs` and `sysfs`. Afterwards, we will take a look at specific files that appear in the `procfs` and `sysfs` file systems. This look, however, will be very confined since we need to examine only a fraction of all the available files. We will also present a few examples of how the `procfs` and `sysfs` files might be useful for everyone running the Linux kernel.

4.1 The `procfs` File System

Right from the beginning, we need to know that the `procfs` file system (i.e., the process file system) is not Linux specific. The process file system originally comes from the 8th version of UNIX [32]. Presently, we can find the process file system in various UNIX-like operating systems, such as BSD, IRIX, AIX, Solaris, and QNX.

The process file system does not contain real files and directories. Almost all the files in the `procfs` have the file size of 0. It merely contains process-related information. When these pseudo-files are read, the system retrieves the information from the actual processes or the system [32]. Traditionally, this file system is located under the `/proc` mount point. For instance, we can obtain all the exported symbols from the kernel by reading the `/proc/kallsyms` file. The `/proc` directory mainly contains numbered directories. For every running process in the kernel there is created a directory —the name of this directory is actually the process ID. In this directory, one can find various information about the process, such as memory maps, status information, OOM[34] score, etc. Also, there is a symbolic link called *self*. This symbolic link points to the current process.

If we want to take a look at a memory map of current process (`cat`, in this example), we can run:

```
$ cat /proc/self/maps
```

It is interesting that some of the process file system files may be written to in order to change system parameters [32]. Hence, one can, e.g., free the page caches by using:

[34] OOM means Out Of Memory. When the system is OOM, it can kill some process based on its OOM score to gain additional memory.

```
$ echo 1 > /proc/sys/vm/drop_caches
```

Many of the already existing system utilities are based purely on reading certain files from the process file system. A few examples of utilities using the `procfs` to a great extent are `top`, `vmstat`, `ps`, `slabtop`, `pmap`, `lsof`, and `free`.

Now we are to describe some `procfs` files more thoroughly. Since there are so many files in there, we will inquire only those files, which we made use of when writing tools `mpstat`, `iostat`, and `powertop`.

The `/proc/uptime` file contains just two numbers: system up time and the time spent in the idle mode. Both are given in seconds.

The `/proc/diskstats` file contains disk input and output statistics, such as the number of reads, the number of sectors read, the number of milliseconds spent writing, and the number of operations currently active. Here is and example contents of the file:

```
8       16 sdb 296 1216 2723 5244 0 0 0 0 0 4808 5244
8       17 sdb1 199 1214 1931 3739 0 0 0 0 0 3411 3739
```

The `/proc/stat` stores various statistics about kernel activities. The statistics here are collected since the system boot. The numbers express what the CPUs do, in `USER_HZ` units[35]. For example, we can determine, how much time the CPU spends on executing the user processes, the kernel processes, waiting for I/O operations to finish, servicing interrupts, or being idle. Also, context switches are to be found here.

In the `/proc/interrupts` are the interrupt statistics saved. They are nicely readable because there are also descriptions of each column. Every IRQ has its own number in the IRQ vector. It is apparent that the IRQ number zero is reserved for the timer interrupts, which are the most common:

The `/proc/softirqs` concieves the counts of soft IRQ handlers serviced [15].

```
            CPU0       CPU1       CPU2       CPU3
    HI:          0          0          0          0
 TIMER:  148980950  148991855          0          0
NET_TX:    1418599     265482          0          0
NET_RX:     114753    3116129          0          0
 BLOCK:     592191       6034          0          0
```

The `/proc/timer_stats` is in fact a debugging facility that allows one to collect information about the timer events. We can determine which process issued how many timer interrupts. The usage is trivial. First, we must start the timer by writing '1' to the `/proc/timer_stats` file:

```
# echo 1 > /proc/timer_stats
```

After a certain period of time, we stop the timer:

```
# echo 0 > /proc/timer_stats
```

Now, we can read the collected data. Below is an example of reading the `/proc/timer_stats`. Here we can see the number of events, the PID of the process, which initialized the timer, the name of the process, then two functions—the first one is the function that initialized the timer, the second one is the callback function.

[35] This is most of the time 1/100ths of a second.

```
# cat /proc/timer_stats
Timer Stats Version: v0.2
Sample period: 12.150 s
 12150,     0 swapper          hrtimer_start_range_ns (tick_sched_timer)
 12150,     0 swapper          hrtimer_start_range_ns (tick_sched_timer)
 32, 27458 bash             queue_delayed_work (delayed_work_timer_fn)
 9429,     9 events/0         queue_delayed_work (delayed_work_timer_fn)
 261, 21155 firefox          hrtimer_start_range_ns (hrtimer_wakeup)
```

Note that doing the above as is requires the superuser privileges. The timer stats facility can be turned on by enabling the `CONFIG_TIMER_STATS` kernel configuration option.

4.2 The `sysfs` File System

Another virtual file system in the Linux operating system[36] is the `sysfs`. This file system is used to access information about the kernel objects, i.e. devices and drivers. The kernel objects are represented as directories, the kernel object attributes as regular files, and the relationships between objects as symbolic links [16]. It is also possible to change the system's setting by writing into files in this file system.

The `sysfs` directory structure is hierarchical. It is best viewed by using the program `tree(1)`:

```
$ tree -FCL 1 /sys/
/sys/
|-- block/
|-- bus/
|-- class/
|-- dev/
|-- devices/
|-- firmware/
|-- fs/
|-- hypervisor/
|-- kernel/
|-- module/
|-- power/

11 directories, 0 files
```

In this thesis, we care mainly about the `devices/` sub-directory. This directory contains information about each device detected by the operating system. We can differentiate between platform devices and system devices. For example, the CPUs and timers are among the system devices while devices that have I/O ports are among the platform devices.

Since there is a lot of information in the files in the `sysfs`, we will pick up only few. For example, if we are to determine why and how long the particular CPU is idle, in the `/sys/devices/system/cpu/cpu0/cpuidle/state0/` directory we might find useful following files:

The `desc` file is the text description of the idle state, i.e., the CPU does not do anything useful, the `latency` file keeps the latency of the idle state in microseconds, the `name` holds

[36] We assume kernel version 2.6.

the names of the idle states. These are the so-called *C-states*. The `power` file shows how much power was consumed while the CPU was in the idle state. The value is in milliwatts. In the `usage` file is stored how many times the CPU was in this idle state. In the `time` file can one determine how long the CPU was in the idle state. The value is described in microseconds.

Chapter 5

Busybox Plugins

In this chapter, we describe the Busybox plugins which were created as a part of this thesis. We describe the main concept of how these tools work. Interested readers are advised to look at the source codes of these plugins. The source codes are freely available in the Busybox git repository [2]. Moreover, we discuss various changes of the `mpstat`, `iostat`, and `powertop` tools that have been made by the author of this thesis in order to adjust the tools to the form suitable for incorporating them into the Busybox project.

5.1 The `mpstat` Plugin

Mpstat is a tool to report CPU-related statistics. It originally comes from the Sysstat package [19]. The Busybox mpstat retained all the command-line options that the original mpstat has. We will now shortly describe what mpstat can do.

When run without parameters, the output of the mpstat will be just an overview of global CPU statistics:

```
$ ./busybox mpstat
Linux 2.6.32-71.el6.x86_64 (dhcp-25-89.brq.redhat.com) 04/29/11 _x86_64_
20:58:29     CPU    %usr   %nice    %sys %iowait    %irq   %soft    %idle
20:58:29     all    3.69    0.00    2.00    1.19    0.08    0.13    92.91
```

The above given command produces a global overview in what states most of the time the CPU is. We get this output also when we specify the `-u` command line argument.

Normally, mpstat will just display numbers like these and then exit. It can, however, run infinitely with various delays. To achieve this, we just supply a number as an argument. It is also possible for mpstat to output just a certain number of statistics which happens when we supply another number as an argument. So, for generating statistics ten times every two seconds, we would write `$ mpstat 2 10`.

If we are on a machine with more CPUs, we can select a particular CPU by using the `-P` option. Thus, to display information about the first CPU only, one would use `$ mpstat -P 0`; to display information about all the CPUs, we can use `mpstat -P ALL`.

It is possible to produce a different output of mpstat than the default one. If we are interested in seeing interrupts, we can specify the `-I` option together with the `SUM` keyword. This way we get information about how many interrupts per second there occurr. A much more detailed output will be generated when we use `ALL` in place of the `SUM` keyword.

We will now shortly delve into the implementation of mpstat. The source code can be found in the `procps` directory in the Busybox directory. The mpstat program consists mostly of collecting various pieces of data from the process file system. We described the process file system in Section 4. The pieces of data from the `procfs` are stored in different structures holding statistics. First of the structures holding statistics is the `struct stats_irqcpu` structure. The purpose of the `struct stats_irqcpu` structure is to store information about the CPU interrupts. Here we can found the number of the interrupts and the interrupt name:

```
struct stats_irqcpu {
    unsigned interrupts;
    char irq_name[MAX_IRQNAME_LEN];
};
```

Next, the `struct stats_cpu` structure holds information about one CPU, i.e. the counts of various activities. The `data_t` type is an unsigned integral type which can hold even very high numbers. Usually, it is a synonym for the `unsigned long long int` type:

```
struct stats_cpu {
    data_t cpu_user;
    data_t cpu_nice;
    data_t cpu_system;
    data_t cpu_idle;
    data_t cpu_iowait;
    data_t cpu_steal;
    data_t cpu_irq;
    data_t cpu_softirq;
    data_t cpu_guest;
};
```

The another important structure is the `struct stats_irq` structure. It has just one element which stores the number of all the interrupts that have occurred since the system boot:

```
struct stats_irq {
    data_t irq_nr;
};
```

At the beginning of the program, mpstat gets the interrupt counts. This is done just by reading the `/proc/interrupt` and `/proc/softirqs` files. Afterwards, the function `main_loop` is called. This is the workhorse of the mpstat. The function `main_loop` always updates the `stats_cpu` data structure by reading the `/proc/stat` file. What happens next depends on the command line arguments specified. If the `-I SUM` option was specified, the main loop will update the `stats_irq` structure. If the user wanted to see a more detailed interrupts statistics, the `stats_irqcpu` structure is updated at this point. This is done by reading the `/proc/interrupt` or `/proc/softirqs` file. If the user did not specify any interval, the collected statistics are printed. If, however, the interval is set, we now get into a loop, which is terminated when the user kills the mpstat or when the *count* parameter is hit. After this, the average statistics are also printed to the standard output. The timing is implemented by using the `alarm` library function, which sends the `SIGALRM` signal to the

calling process after a specified number of seconds. The real implementation is somewhat more complicated, but here we are trying to cover only the very basics of how this program works.

5.1.1 Modifications

Almost complete functionality of the original mpstat was retained. Hence, Busybox mpstat accepts all the command line options like the original mpstat, except the `-V` option, which causes mpstat to display version and exit immediately. Printing the version and help is handled by Busybox core—the user can use the `--help` option. However, there is one difference in that the Busybox mpstat does not honour the `S_TIME_FORMAT` environment variable, which is used to set a different locale. Ignoring the `S_TIME_FORMAT` environment variable saves a `getenv` and a `strcmp` call. Busybox applets in general do not support the National Language Support, thus Busybox mpstat does not support it neither. This saved a few `setlocale` calls, the `bindtextdomain` call, and the `textdomain` call.

Both mpstats need to determine the number of CPUs. In original mpstat this is done interating over the files residing in the `/sys/devices/system/cpu` sub-directory using functions `opendir` and `readdir`. Every file is compared by calling the `strncmp` function with a string `‘‘cpu’’`. Then the functions such as `snprintf`, `stat`, and `isdigit` are called to determine the number of CPUs. All these functions are big, thus, in Busybox, we use a different scheme. To determine the number of CPUs, we only read the `/proc/stat` file by using the `fgets` function. In addition, we do not even use the `strcmp` function, instead, to determine if a string starts with the `‘‘cpu’’`, we use this kind of optimization for size:

```
int FAST_FUNC starts_with_cpu(const char *str)
{
    return ((str[0] - 'c') | (str[1] - 'p') | (str[2] - 'u')) == 0;
}
```

To significantly reduce the `.bss` segment usage, all global variables were moved to the `struct globals` structure. We described this technique in Section 3.5. Furthermore, some space was saved by using the `xzalloc` function prom the `libbb`, instead of using the `malloc` and `memset`.

Another great opportunity to save space was the parsing of the command line arguments. Original mpstat does not use any function like the `getopt`, instead the parsing is done using a long chain of `if`s together with a lot of the `strcmp` calls. In Busybox, we took a better approach: parsing is done using the `getopt32` function and the `strcmp` functions were replaced by just a single call of function `index_in_strings`. Both the `getopt32` and the `index_in_strings` functions reside in the `libbb`. We have talked about using the `libbb` functions in Subsection 2.2.2.

The function for reading the CPU statistics was optimized. In original mpstat is the function unnecessarily long since it duplicates its code. Below is given the original function in pseudo-code:

```
if (strncmp ())
  {
    memset ();
    scanf ();
    A = sum;
  }
```

```
else if (strncmp ())
  {
    memset ();
    scanf ();
    B = sum;
  }
```

Duplicating the code is never a good idea. Hence, we end up with a better variant, where only a single call of `memset` and `scanf` is needed:

```
if (starts_with_cpu ())
  continue;

memset ();
scanf ();
if (strncmp ()
   A = sum;
else
   B = sum;
```

At some places in the original mpstat we may find a `TEST_STDOUT` macro, which tries to write an empty string to the standard output. This `TEST_STDOUT` was removed since it does not bring anything very useful.

One of the biggest functions in original mpstat is the function for printing the statistics called `write_stats_core`. In original mpstat, the `write_stats_core` function ill-advisedly uses much more `printf` calls than is really needed. Consider this example:

```
if (cond1)
  printf (A);
for ()
  {
    if (cond2)
      printf (B);
    if (cond3)
      printf (B);
    else
      printf (A);
  }
```

All those `printf` calls are not necessary. The `printf` function must not be underestimated, since it is a big function, and also because the `printf` takes a string as an argument, which must be stored somewhere, most likely in the `.rodata` section. Thus, above given code was reorganized a bit in the form that only two `printf` calls are needed:

```
for ()
  {
    if (! cond1)
      continue;
    if (cond2)
      goto zeros;
```

```
    if (cond3)
      {
zeros:
      printf (B);
      continue;
      }
    printf (A);
  }
```

In the function for printing the statistics was originally a variable, which determined, if the header should be printed (called unintuitively `dis_hdr`). In Busybox mpstat we print the header always—thus, variable like `dis_hdr` is no longer needed. Removing the `dis_hdr` variable saved some space, e.g., it is no longer needed to pass the variable to functions (see Subsection 3.4.1), also various `if (dis_hdr)` are gone now. In addition to the removed `dis_hdr` variable, in Busybox mpstat we do not determine the size of the terminal windows. Thus, there is no expensive `ioctl` call in Busybox mpstat.

Furthermore, some completely redundant code was removed. Consider the following example, taken from original mpstat:

```
void write_stats_avg(int curr, int dis)
{
    char string[16];

    strncpy(string, _(''Average:''), 16);
    string[15] = '\0';
    write_stats_core(2, curr, dis, string, string);
}
```

In the example above, the variable `string` and the `strncpy` call are not needed. It completely suffices to pass the `''Average:''` string as a parameter to the `write_stats_core`.

At various places were used functions from the `libbb`. For instance, it is not needed to check the return value of `fopen`, since in Busybox mpstat is used the `fopen_for_read` function which already handles the case when a file cannot be opened. Also, in Busybox mpstat is used the `xmalloc` function, thus the necessity to check the return value vanishes. Using the functions for the `libbb` much simplifies the code. Furthermore, the `libbb` function `safe_strncpy` is used, thus it is no longer necessary to write the NULL byte at the end of a buffer.

Another change is that a lot of variables were moved into the smallest scope needed. Described another way, if a variable is needed only in a certain scope, it is better to move the variable into this scope, because with clever compilers, we can save a register space. Also, moving the definitions of variables closer to the place, where they are used, has a good impact on readibility of the code. Better readibility paves the way for any future improvements.

Last but not least, several badly named functions and variables were renamed to better reflect their meaning. For example, in original mpstat, one can find the `ll_sp_value` function. It arguably is not clear, what a function with a name like this should in fact do, thus, it was renamed to `overflow_safe_sub` to represent, that it serves to subtracting two numbers with respect to overflows. Many variables with names such as `itv`, `ic_nr`, `scc`, and `scp`, that baffled many, so they were renamed to more meaningful names.

Original mpstat has around 1300 lines of code, while Busybox mpstat has around 900 lines of code. In Figure 5.1 are shown statistics of Busybox after adding the mpstat plugin.

```
function                                             old     new   delta
mpstat_main                                            -    1334   +1334
write_stats_core                                       -    1326   +1326
write_irqcpu_stats                                     -     771    +771
get_irqs_from_interrupts                               -     607    +607
.rodata                                           148189  148709    +520
get_cpu_statistics                                     -     370    +370
get_uptime                                             -     139    +139
static.get_irqcpu_nr                                   -     131    +131
get_irqs_from_stat                                     -     124    +124
print_header                                           -     120    +120
write_stats                                            -     119    +119
packed_usage                                       28406   28498     +92
get_per_cpu_interval                                   -      63     +63
is_cpu_in_bitmap                                       -      37     +37
applet_main                                         2768    2776      +8
applet_names                                        2372    2379      +7
static.v                                               -       4      +4
applet_nameofs                                       692     694      +2
applet_install_loc                                   173     174      +1
alarm_handler                                         53      37     -16
------------------------------------------------------------------------------
(add/remove: 14/0 grow/shrink: 6/1 up/down: 5775/-16)         Total: 5759 bytes
   text    data     bss     dec     hex filename
 951935    4106    9568  965609   ebbe9 busybox_old
 958757    4114    9568  972439   ed697 busybox_unstripped
```

Figure 5.1: Busybox after adding mpstat

5.2 The `iostat` Plugin

Iostat is another tool coming from the Sysstat package [19]. Its original purpose is to report CPU statistics as well as I/O and NFS[37] statistics for devices and partitions. Nevertheless, in Busybox, we dropped the NFS support. This way we achieved a much smaller size of this applet. Iostat from Busybox does not come with a lot of command line options. By default, the output looks like:

```
$ ./busybox iostat
Linux 2.6.32-71.el6.x86_64 (dhcp-25-89.brq.redhat.com)  01/00/00  _x86_64_

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           6.69    0.00    3.09    1.34    0.00   88.87

   Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
   sda               5.03        72.30       132.72    2075391    3809962
   dm-0             18.73        72.05       132.72    2068420    3809944
```

[37] NFS is a network file system protocol. The newest version is NFSv4.

```
dm-1              3.58        36.49        20.80     1047426      596976
dm-2              0.82         0.86         5.71       24616      163904
dm-3             13.93        34.68       106.21      995442     3049064
```

In the first part of the output, we can see an overall CPU statistics. If we want to see solely this part, we can use the `-c` command line option. In the second part, we see various numbers expressing how many blocks were written and/or read, at which speed, and how many transfers per second (*tps*) occurred.

Analogously to mpstat, the user can specify the interval and count parameters too. Also, the user can specify about which partitions and devices the statistics will be printed. Thus, for instance, we can select some partitions only:

```
$ ./busybox iostat -cm /dev/sda /dev/sda1 2 10
```

The `-m` option causes the output to be in megabytes and `-k` in kilobytes. The `-t` option will print the time and, the `-z` option will not display inactive devices. Inactive devices are those which are neither read from nor written to.

In iostat, two structures are essential. The first one is the `struct stats_cpu` structure. This structure is in fact equivalent to the `struct stats_cpu` structure from mpstat. Another one is the `struct stats_dev` structure, which holds information about devices: their name, sectors read/written, and operations count:

```
struct stats_dev {
    char dname[MAX_DEVICE_NAME];
    unsigned long long rd_sectors;
    unsigned long long wr_sectors;
    unsigned long rd_ops;
    unsigned long wr_ops;
};
```

Basically, the course of events is happening in the `iostat_main` function. There is an infinite loop where the program always prints the header and then, according to the command line options, also specific data. The iostat applet may print out the time and the CPU stats. If the user did not specified the `-c` option, the function `dev_report` and in turn the `do_disk_statistics` is called. Here we read the `/proc/diskstats` file and then we print relevant information. On the other hand, if no options are specified or the `-c` option is on, we call the `cpu_report` function. This function prints the `stats_cpu` structure. The `stats_cpu` structure is filled by the function `get_cpu_statistics`, which, essentially, reads `/proc/stat`. The whole loop is terminated if the limit count is hit or if the interval value was not specified.

5.2.1 Modifications

Probably the most dramatic change is that Busybox iostat does not support some option as original mpstat does. The removed command line options are the `-n` for displaying NFS report, `-N` for viewing LVM2 statistics, and the `-x` for displaying the extended statistics. Also, Busybox iostat does not know the `-p` command line option for displaying statistics for block devices and all their partitions. The `-x` and `-n` options require the 2.6 Linux kernel, while some embedded devices still run the 2.4 Linux kernel. The `-N` command line option was find

to not to be too useful. The code size of Busybox iostat decreased quite a lot, since it was not needed to re-implement some big function from original iostat. For instance, functions such as the `write_ext_stat`, the `get_devmap_major`, and the `transform_devmapname` were completely omitted.

Other changes to iostat are quite similar to the changes made to mpstat. That said, the National Language Support was removed, the parsing of command line arguments is done by using the `getop32` function from the `libbb`, and all the global variables were moved to the `struct globals` structure (see Section 3.5).

It is possible to specify certain devices—iostat will then print information about the specified devices only. In original iostat this is solved by saving all the specified devices into an array of strings. In Busybox iostat, however, this is solved differently—the llist interface, which is the linked list interface, is used. All the functions for handling a linked list are a part of the `libbb`. It works by having a `llist_t` variable. Adding an item into the `llist_t` variable is very easy—it is handled by the `llist_add_to` function. To be more concrete, below is given a piece of code from the Busybox iostat, which handles the device list:

```
char *dev_name = skip_dev_pfx (*argv);
if (!llist_find_str (G.dev_list, dev_name))
  {
    llist_add_to (&G.dev_list, dev_name);
    dev_num++;
  }
```

At the end of the iostat, this list is freed by calling the function `llist_free`. Using the llist Busybox interface saved a lot of code and is also much more readable.

Next, the `alarm` and `pause` functions were removed, instead, only the single `sleep` is used. Using the `sleep` function brought further simplification: the core of Busybox iostat now looks roughly like the following:

```
for (;;)
  {
   get_cpu_statistics ();

   if (cond1)
     dev_report ();

   if (cond2)
     cpu_report ();

    if (count > 0)
      if (--count == 0)
        break;

    sleep (interval);
  }
```

In Busybox iostat, the function for getting the CPU statistics is way simpler. In original iostat, getting the CPU statistics is basically done as shown below:

```
while (fgets ())
  {
    if (strncmp ())
      {
        memset ();
        sscanf ();
        up1 = sum;
      }
    else if (strncmp ())
      {
        memset ();
        sscanf ();
        up2 = sum;
      }
  }
```

This can be simplified into the form given below:

```
while (fgets ())
  {
    if (starts_with_cpu ())
      continue;
    for ()
      {
        sscanf ();
        sum += value;
      }
    break;
  }
```

We saved the `memset` and the `sscanf` calls which leads to a smaller code size.

Another opportunity to save the size was to review the data types. It came out that at some places we could save some bytes by using a smaller data type. Thus, the data type of the `curr` variable which holds either 0 or 1 was changed to `smallint`. We talked about the data types in Section 3.2.

Among above given changes, some variables and functions were renamed to more meaningful names, for instance, it is hard to guess what the `it` variable actually means, so its new name is the `interval`.

Also, not surprisingly, various functions from the `libbb` were used. E.g., for setting a memory block to zero, is the `xzalloc` function used.

Original iostat has around 2000 lines of code, while the Busybox iostat consists of around 500 lines of code. This is a very positive improvement. In the Figure 5.2 are shown Busybox statistics after adding the iostat applet.

5.3 The `powertop` Utility

Powertop, unlike mpstat and iostat, does not come from the Sysstat package, but instead it is a standalone software. This program serves to analyze power consumption on Intel-based laptops [18]. Powertop from Busybox does not come with any command line parameters.

```
function                                             old     new   delta
iostat_main                                            -    2105   +2105
.rodata                                           148260  148709    +449
print_stats_dev_struct                                 -     263    +263
packed_usage                                       28429   28498     +69
is_partition                                           -      35     +35
applet_main                                         2768    2776      +8
applet_names                                        2372    2379      +7
applet_nameofs                                       692     694      +2
applet_install_loc                                   173     174      +1
------------------------------------------------------------------------------
(add/remove: 4/0 grow/shrink: 6/0 up/down: 2939/0)          Total: 2939 bytes
   text    data     bss     dec     hex filename
 955445    4114    9568  969127   ec9a7 busybox_old
 958757    4114    9568  972439   ed697 busybox_unstripped
```

Figure 5.2: Busybox after adding iostat

Powertop needs to be run with superuser privileges. This is due to the fact that we need to be able to write into the `/proc/timer_stats` file. We spoke about this file in Section 4.1.

The basic goal of Powertop is to determine which applications are causing most *wakeups*. To know what a wakeup is, we need to get to know the CPU *power states*. With regard to power consumption, a CPU has three power states:

- *C-states*, the idle states.
- *P-states*, the performance states.
- *T-states*, the thermal states.

If we want to accomplish better savings, we should adjust these states to get better results as far as power consumption concerns. We can differentiate the C-states into four levels, C0 to C4. Basically, when the CPU is in the C0 state, it is executing some instructions. This state is the most expensive. The C1 and higher states mean, that the CPU is idle. The higher level, the more power savings. Having that said, if we want to really save the power consumption, then the CPU should be in the C3 or C4[38] state most of the time. Of course, being in a higher C-state has also some downsides. These downsides are above all worse latency and performance. The crucial point is that changing the C-states involves quite a lot of power consumption. It is more expensive to change state from C4 to C0 than from C1 to C0. These C-state changes are the so-called wakeups. To see the C-states of a particular system, one could issue:

```
$ cat /proc/acpi/processor/*/power
```

Obviously to save more power, we should lower the number of the wakeups. This is when Powertop comes in handy. The output of Powertop could look like the following (it refreshes this output automatically every 10 seconds), where we, in the first part, can see available C-states together with the information how long in average the CPU spends in a particular C-state and a number, which says, in what C-state the CPU most of the time is, and in the second part we can see which applications are causing most wakeups:

[38] Note that the C4 state is often not available when the laptop runs on AC power.

```
Cn   Avg residency
C0 (cpu running)        ( 5.4%)
C1   0.0ms ( 0.0%)
C2   0.6ms ( 3.0%)
C3   2.0ms (91.0%)

Wakeups-from-idle in 10 seconds: 10281

Top causes for wakeups:
  14.2% ( 1417)     <kernel core> : hrtimer_start_range_ns (tick_sched_timer)
  11.0% ( 1099)      npviewer.bin : hrtimer_start_range_ns (hrtimer_wakeup)
   6.7% (  664)       <interrupt> : iwlagn
   5.5% (  545)           firefox : hrtimer_start_range_ns (hrtimer_wakeup)
   2.5% (  250)             totem : hrtimer_start_range_ns (hrtimer_wakeup)
   1.4% (  138)     <kernel core> : hrtimer_start (tick_sched_timer)
   1.3% (  130)        pulseaudio : hrtimer_start_range_ns (hrtimer_wakeup)
   1.1% (  113)      <kernel IPI> : Rescheduling interrupts
   1.0% (   98)   thunderbird-bin : hrtimer_start_range_ns (hrtimer_wakeup)
   0.8% (   80)     <kernel core> : usb_hcd_poll_rh_status (rh_timer_func)
   0.7% (   73)     <kernel core> : sk_reset_timer (tcp_delack_timer)
   0.5% (   54)       <interrupt> : hda_intel
   0.4% (   40)     <kernel core> : hrtimer_start (nmi_watchdog_timer_fn)
   0.4% (   35)       <interrupt> : ahci
   0.1% (   13)             irssi : hrtimer_start_range_ns (hrtimer_wakeup)
```

It is apparent that most of the time the CPU is in the C3 state, which is good. However, if we want to further lower the wakeups count, we can investigate the table—this is a list with running applications on the system causing most wakeups. In this very example, Firefox and Totem are generating a fair amount of wakeups. Thus, if the user wanted to reduce the number of wakeups, s/he should shut Firefox and/or Totem down. Below, we give an overview of the implementation of Powertop.

The main part of Powertop is an infinite for loop. Here are always the C-state counts updated by calling the `read_cstate_counts` function. This function merely examines the `/proc/acpi/processor/*/power` files. After this, wakeups in 10 seconds are displayed. The function `process_irq_counts` is handling the total number of wakeups. It works primarily by reading the `/proc/interrupts` file. Finally, the timer stats are displayed. Lines to be printed are all saved in the `struct line *lines` array. Every element of this array comprises a name and the corresponding wakeup count. These lines were collected by the `process_timer_stats` function by reading the `/proc/timer_stats`. After gathering all of them, they are sorted using the `qsort` library function and finally they are displayed to the standard output. When all of this is done, another iteration starts.

An interesting thing related to Powertop worth mentioning is how can we obtain additional information about the CPU. One possible way is to use the `cpuid` assembly instruction. In Powertop, running on x86, this is used to get the information about available C-states. The EAX register is used to specify which information we are interested in (e.g., if EAX is equal 1, we can get the vendor identification):

The inline assembly code looks like[39]:

[39] The `'='` constraint modifier means, that the variable is write-only.

```
__asm__ (
    ,,   pushl %%ebx\n‘‘     /* Save EBX */
    ,,   cpuid\n‘‘
    ,,   movl %%ebx, %1\n‘‘  /* Save content of EBX */
    ,,   popl %%ebx\n‘‘      /* Restore EBX */
    : ,,=a‘‘(*eax), /* Output */
      ,,=r‘‘(*ebx),
      ,,=c‘‘(*ecx),
      ,,=d‘‘(*edx)
    : ,,0‘‘(*eax),  /* Input */
      ,,1‘‘(*ebx),
      ,,2‘‘(*ecx),
      ,,3‘‘(*edx)
);
```

This way, we can get all the supported C-states by bit-shifting the `edx` variable.

5.3.1 Modifications

Powertop is a plugin that underwent the most dramatic changes. The first of this striking changes is that original powertop uses the ncurses library for displaying various colors and shapes, while the Busybox powertop does not use the ncurses library. In Busybox, this is unthinkable, because it would be a tremendous code bloat. Not using the ncurses library is less user-friendly, but it also saved several hundreds lines of code. Another benefit is that no additional dependency is needed. However, since powertop is an interactive tool, it was needed, e.g., to turn on the unbuffered input. The unbuffered input is useful in situation, when we want that the user input does not have to be confirmed by Enter key. For this, we use the `tcsetattr` function. Similar approach is taken in the `top` Busybox applet.

A very crucial modification is that in Busybox powertop were omitted the so-called *suggestions*. Suggestion is a helpful text which advises, that the user should change some system value or, e.g., turn on or off some kernel option. The powertop itself can change some of the system values. However, the suggestions are taking up too much space, suggestions are virtually hundreds lines of code. Except this, suggestions also contain lots of long strings. The necessity to hold a lot of long strings would make the `.rodata` section much more bigger. In addition, there is one more major drawback. In the future, the suggestions would need to be constantly updated to reflect current system options. The need to update the suggestions is somewhat unwelcome.

Busybox powertop also does not print the battery status. Not displaying the battery status allowed to remove some of the original powertop functions. Gathering the information about battery was handled by the `print_battery_proc_pmu`, `print_battery_proc_acpi`, and `print_battery_sysfs` functions. The functions for determining information about battery were quite big, and contained enormous number of functions such as the `readdir`, the `sprintf`, the `strtoull`, and the `fgets`. Thus, a huge savings of code were achieved.

Reading and parsing the `/proc/timer_stats` is done in both original and Busybox powertop. Nevertheless, in Busybox was taken a more size-oriented approach. In original powertop the reading and parsing is done in the main loop of the program. In Busybox, there is a specialized function, that is also more optimized. For instance, instead of a long chain of the `strchr` functions together with while loops for skipping whitespaces, in Busybox powertop there is the following piece of code:

```
get_func_name:
  p = strchr (p, ' ');
  if (! p)
    continue;
  *p++ = '\0';
  p = skip_whitespace (p);
  if (process == NULL)
    {
      process = p;
      goto get_func_name;
    }
```

Note, that the `skip_whitespace` function from the `libbb` is used in place of a while loop.

The function for parsing the `/proc/interrupts` file was also further optimized. A sequence of the `strcmp` functions was removed; instead, one single call of the `index_in_strings` is used. Consider following example from original powertop:

```
if (strncmp(line,``NMI:'', 4)==0)
  nr=20000;
if (strncmp(line,``RES:'', 4)==0)
  nr=20001;
if (strncmp(line,``CAL:'', 4)==0)
  nr=20002;
...
```

In Busybox it is better, with respect to size, write the following:

```
nr = index_in_strings(,,NMI\0RES\0CAL\0TLB\0TRM\0THR\0SPU\0``, buf);
```

Moreover, consider below given code from original powertop:

```
c = strchr(name, '\n');
if (c)
  *c = 0;
```

To save additional space, in Busybox powertop, the following variant is used:

```
strchrnul(name, '\n')[0] = '\0';
```

Moreover, all global variables were, to save the space in the `.bss` segment, moved to the `struct globals` structure. This structure was described in Section 3.5.

It is possible to save some more space by writing more generic functions. Consider the below given example from original powertop:

```
void stop_timerstats(void)
{
  FILE *file;
  file =
    fopen(``/proc/timer_stats'', ``w'');
  if (!file)
    {
      nostats = 1;
      return;
    }
  fprintf(file, ``0\n'');
  fclose(file);
}
```

```
void start_timerstats(void)
{
  FILE *file;
  file =
      fopen(``/proc/timer_stats'', ``w'');
  if (!file)
    {
      nostats = 1;
      return;
    }
  fprintf(file, ``1\n'');
  fclose(file);
}
```

Above given functions differ in the string that will be written into the file. This is completely unnecessary. A better approach is to use just one more general function and pass the string to be written as an argument:

```
static int
write_str_to_file(const char *fname, const char *str)
{
    FILE *fp = fopen_for_write(fname);
    if (!fp)
        return 1;
    fputs(str, fp);
    fclose(fp);
    return 0;
}
```

Another change is that the Busybox powertop does not try to load the `cpufreq_stats` kernel module. Original powertop does this using the `system` library function. However, using the `system` function together with superuser privileges is not safe and thus should be avoided.

Some names of functions and variables were amended. For instance, it is not very clear that the `linehead` variable is a number of lines saved. The `lines_cnt` variable is more readable.

Original powertop consists of around 4 000 lines of code, while the Busybox powertop consists of around 850 lines of code. Below is presented the result of adding powertop to Busybox:

```
function                                             old     new   delta
iostat_main                                            -    2105   +2105
.rodata                                           148260  148709    +449
print_stats_dev_struct                                 -     263    +263
packed_usage                                       28429   28498     +69
is_partition                                           -      35     +35
applet_main                                         2768    2776      +8
applet_names                                        2372    2379      +7
applet_nameofs                                       692     694      +2
applet_install_loc                                   173     174      +1
------------------------------------------------------------------------------
(add/remove: 4/0 grow/shrink: 6/0 up/down: 2939/0)           Total: 2939 bytes
   text    data     bss     dec     hex filename
 955445    4114    9568  969127   ec9a7 busybox_old
 958757    4114    9568  972439   ed697 busybox_unstripped
```

Figure 5.3: Busybox after adding powertop

Chapter 6

Conclusion

Let us briefly summarize the contents of this thesis. We have acquainted ourselves with the Busybox project. We have investigated how this project is structured, we have also taken a look at the Busybox library. Since the mpstat, iostat, and powertop applets are an essential part of the Busybox, we have examined a single applet to the great extent. We have shown how to obtain, configure and run the Busybox and its applets. We have also created and added a simple applet to the Busybox.

We have payed a lot of attention to the area of how to make our programs smaller. We have discussed several programming techniques on how to better utilize our programs to save space. We have described available C libraries. We have proposed and described quite a lot of GCC command line options, which can further help to reduce the size of the final executable. Apart from studying how to write more compact code, we have also introduced several tools which are useful when tuning code.

Then, we have discussed the Sysfs and Procfs file system. We have assumed that we are using the Linux operating system. These file systems are crucial for tools used to getting information about the system. We have picked up only several files from the procps and the sysfs, and we have described these in more detail.

In the last chapter we debated the implemented applets—`mpstat`, `iostat`, and `powertop`. We have presented examples on how to use these tools and what information they can bring us. As a part of our contribution these applets were written and are now a part of stable version of Busybox. They have been added since the version 1.18.0. With this work we attempted to describe on a higher level, how these tools actually work.

Finally, we have also studied the modifications of the implemented tools, which were made to keep the code size at minimum. We have always presumed that the implementation language is the C language.

Bibliography

[1] Buildroot Homepage. http://buildroot.uclibc.org/.

[2] Busybox Git Repository. http://git.busybox.net/.

[3] Busybox Homepage. http://www.busybox.net/.

[4] Busybox Prebuilt Binaries. http://busybox.net/downloads/binaries/latest/.

[5] diet libc - a libc optimized for small size. http://www.fefe.de/dietlibc/.

[6] DWARF debugging tools. http://git.kernel.org/?p=linux/kernel/git/acme/pahole.git;a=summary.

[7] Embedded GLIBC Homepage. http://www.eglibc.org/home.

[8] GCC Debugging Options. http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html.

[9] GCC Optimize Options. http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[10] GNU Coreutils. http://www.gnu.org/software/coreutils/.

[11] Ian Lance Taylor, *Linkers*, 2007. http://www.airs.com/blog/archives/42.

[12] ISO/IEC 9899:1999 C standard. http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899.

[13] Jan Hubicka, Taras Glek, *Optimizing real world applications with GCC Link Time Optimization*, 2010. http://arxiv.org/abs/1010.2196.

[14] kLIBC Homepage. http://svn.netlabs.org/libc.

[15] Linux Kernel Documentation, 2011. http://www.kernel.org/doc/.

[16] Patrick Mochel, *The sysfs Filesystem*, 2005. http://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf.

[17] Porting Newlib. http://wiki.osdev.org/Porting_Newlib.

[18] Powertop Homepage. http://www.lesswatts.org/projects/powertop/.

[19] Sysstat Homepage. http://sebastien.godard.pagesperso-orange.fr/.

[20] The DWARF Debugging Standard. http://www.dwarfstd.org/.

[21] The GNU C Library. http://www.gnu.org/s/libc/.

[22] The GNU Compiler Collection. http://gcc.gnu.org.

[23] The Newlib Homepage. http://sourceware.org/newlib/.

[24] The Open Group Base Specifications Issue 6. http://pubs.opengroup.org/onlinepubs/009695399/.

[25] Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, version 1.2. http://refspecs.freestandards.org/elf/elf.pdf.

[26] uClibc Homepage. http://www.uclibc.org/about.html.

[27] Ulrich Drepper, Red Hat, Inc., *Good Practices in Library Design, Implementation, and Maintenance*, 2002. www.akkadia.org/drepper/goodpractice.pdf.

[28] Ulrich Drepper, Red Hat, Inc., *How To Write Shared Libraries*, 2006. www.akkadia.org/drepper/dsohowto.pdf.

[29] Ulrich Drepper, Red Hat, Inc., *Optimizing Applications with gcc & glibc*, 1999. www.akkadia.org/drepper/optimtut1.ps.gz.

[30] Using The GNU Assembler. http://sourceware.org/binutils/docs/as/.

[31] Using The GNU Linker. http://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html.

[32] Andrew Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2008.