Context Switching Accounting Mechanism

Toni Castillo Girona
Enginyeria Tècnica en Informàtica de Gestió

Miquel Angel Senar Rosell
June 8, 2012

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Preamble

Every single process that is running on a GNU/Linux computer can be either executing code in **Ring 0** or **Ring 3**. A running process is just some code that is being executed at one measurable time by some processor installed on the system. Every time a process needs to do some low-level tasks, the operating system must be called. That is so because the system must remain stable at any time, to avoid crashes. Therefore, the operating system is the only one capable of executing code in Ring 0, where some low-level instructions are permitted. Whenever a running process does not need to call the operating system, then its code is executed in Ring 3, where some very simple low-level instruction-set is available. It seems fairly obvious that a process cannot run forever, not even in those computers where there is more than one processor. This is called preemption. Preemption allows different processes to run on the same computer as if there was only one process being executed all the time. Whenever a preemption occurs, there is a **Context Switch**. A Context Switch saves the current task's context and loads another one's, so that a new task - a previous preempted one or just a new one being executed for the very first time - could be run. This is time-consuming. A Context Switch can possibly occur by means of calling an operating system function called *schedule()*. Normally, this function will be called directly by an operating system's component called **the scheduler** at certain events. This scheduler decides whether a running task must be preempted or not, and in order to do so it does use certain algorithms. In nowadays GNU/Linux kernels, this algorithm is known as the **Completely Fair Schedule Algorithm**, or **CFS** [1].

Next sections introduce some background concerning the GNU/Linux scheduler, the CFS, the concept of Context Switching and the different ways it can happen. To conclude this introduction, this first chapter describes the importance of the Context Switching and this project's goals.

1.2 Preemption

Let p be a non real-time process running on a GNU/Linux computer. Let's assume p's state is `TASK_RUNNING`. Let c_i be any available core on the system where p is running on. As long as p owns a given computed proportion of c_i, no context-switching occurs. Thus, we can infer from this that p is actually running. Let's assume the system's scheduler is `CFS`. Then, the total cpu-proportion assigned to any given process, including p, theoretically, would be $1/n$, where n is the total number of runnable processes.

At any moment, in that previous state, a Context Switch can possibly occur. Whether voluntary or involuntary, it is feasible to nail it down for future analysis. In the next section we will discuss briefly about the different ways this Context Switch can happen. So far, the concept of Context Switching is clearly remarkable because there is an important computational cost whenever it happens. Thus, we want p not to be preempted for the best part of its own c_i proportion. This way, we could ensure an actual fair time-slice for p. As the next section will discuss, not every time a Context Switch occurs is p-related.

1.2.1 Context Switching Mechanism

Our project will focus on either **voluntary** or **involuntary** context-switching [2]. Thus, as briefly explained in the Goals section, we will be able to detect otherwise unnoticed issues.

Voluntary Context Switching

Voluntary Context Switching occurs anytime p needs to do so, either whilst running in **Ring 3** or in **Ring 0**. Our project's main goal is to intercept and store any system call issued by p and which processor-core ran that call. However, p can preempt itself by issuing a call to `schedule()` explicitly, whilst running in **Ring 0**. This is not the usual case, where a certain process p will be programmed to run in **Ring 3** most of the time. Usually, the only voluntary Context Switch that can occur would be that one involving system calls. Below, a list of three different situations when p could explicitly demand a Context Switch:

- p issued a System Call, and the kernel is now returning from it. Because the Kernel is now returning to **Ring 3**, it assumes it is safe to preempt p if needed.
- p issues an explicit `schedule()` call - in `Ring 0` mode. -
- p exits - whether explicitly calling `exit()` or returning from the `main` function -. This happens in **Ring 3** as well.

Involuntary Context-Switching

Involuntary Context Switching will occur suddenly. Whenever this happens, p will be preempted. Thus, our project has to be capable of tracing and detecting such cases. This will be primordial in order to determine how many times p has been preempted involuntarily. This way, we would study the proportion of involuntary Context Switching and thus determine if we can infer there is an issue on our system. A large proportion of involuntary Context Switching could obviously reduce our process throughput considerably. Below, a list of situations when involuntary Context Switching can happen:

- p is preempted because of an Interrupt Handler.
- p has exhausted its processor time proportion.
- p has been preempted in favour of a recently waken up task with a higher priority.

1.3 Motivation

How many times a given process *p* preempts, either voluntarily or involuntarily, is an important threat to computer's processes throughput. Whenever running cpu-bound processes on a multi-core system without an actual system grid engine as commonly found on Grid Clusters, their performance and stability are directly related to their accurate implementation and the system reliability which is, to an extend, an important caveat most of the times so difficult to detect.

Context Switching is time-consuming. Thus, if we could develop a tool capable of detecting it and gather data from every single performed Context Switch, we would be able to study this data and present some results that should pin-point at whatever their main cause could be.

1.4 Goals

Being able to gather statistical information about any Context Switch for a given process *p* on a GNU/Linux box using the `CFS` scheduler, we will be able to determine whether our process is taking advantage of our processor's capabilities or just the opposite. Studying how many times *p* has been preempted, and splitting it among voluntary and involuntary preemption, we will possible be quite accurate in determining whether our computer is behaving normally or, on the contrary, there are certain issues we would have to explore and take care of. Particularly, our specific goals will be:

- To have a deep insight into the GNU/Linux Kernel CFS scheduler.
- To analyse existing tools dealing with per-task statistics, focusing on Context Switching accounting.
- To study different techniques in order to develop a new tool for dealing with this sort of statistical data, such as Loadable Kernel Modules (LKM) [3], Kernel TaskStats ABI [11], Linux KProbes [5], and so on.
- To develop a new tool, capable of gathering either voluntary or involuntary Context Switching data, so that some statistical information can be analysed and plotted.
- To learn about R language [6], a powerful OpenSource statistical programming language, deriving from S language, that we will use to analyse and infer some statistical data from our new tool.
- To determine, using statistical criteria, whether a given process *p* is working fine or, on the contrary, a defect seems to be affecting its execution.
- To frame this defect-to-be, in order to fathom if it exists because of an external issue or a bad *p*'s implementation.

1.5 Work Plan

Below, we will present a detailed work plan in order to complete our project. It is divided in five main tasks, all of them containing more specific sub-tasks. Every single task is designed so that it does focus on a precise point. A whole Data-Gathering task has also been included:

1. **PAC1**: *Project description and tasks.*
2. **PAC2**: *Available tools analysis.*
3. **PAC3**: *Design and Implementation.*
4. **PAC4**: *Data-Gathering.*
5. *Final Report and Video.*

1.5.1 *PAC1*: Project description and tasks.

The first task is to provide a coherent work plan scheme. It defines our project and describes some basic steps to be done before attempting any further one. According to the project's classroom dates, it is called **PAC1**, or more specifically **Work Plan**. It also includes a sub-task to setup a minimal development environment, relying mostly on Virtual Machines via **Virtual Box** [7].

PAC1: Work Plan	
Task	*Dates*
Project Description	29/02 - 09/03
Bibliography	09/03 - 09/03
Work Plan elaboration	10/03 - 16/03
Work Plan report	10/03 - 18/03
Framework setup	17/03 - 18/03
PAC1 submit	18/03 - 18/03

1.5.2 *PAC2*: Available tools analysis.

Before going any further, this task analyses existing tools concerning per-task accounting, focusing obviously on Context Switching accounting, like PAPI [8]. Moreover, it includes nowadays techniques and Linux Kernel ABIs to be considered as candidates for developing a new tool capable of gathering Context Switching statistical data. It is called **PAC2**.

PAC2: Tools analysis	
Task	*Dates*
Accounting tools analysis	19/03 - 25/03
Kernel ABIs analysis	26/03 - 31/03
PAC2 Report	21/03 - 01/04
PAC2 submit	01/04 - 01/04

1.5.3 *PAC3*: Design and Implementation.

The third task is in charge of designing and implementing the new piece of software. It applies all sorts of previous related analysed material and Kernel-programming techniques so that a reliable utility could be completed. During this process, a complete Voluntary Context Switching accounting system will be fully implemented. It is called **PAC3**.

PAC3: Tool Design and implementation	
Task	*Dates*
Design	02/04 - 10/04
Implementation Part I [1]	11/04 - 19/04
Testing & debugging	19/04 - 21/04
PAC3 Report	02/04 - 21/04
PAC3 submit	22/04 - 22/04

1.5.4 *PAC4*: Data-Gathering.

This project is related to statistical data up to a point. The main goal, in fact, is to gather an important amount of data from running processes, in order to analyse them using a statistical programming language, that is, `R`. Thus, this task involves running the new recently developed tool to gather and store data to be analysed. It is called **PAC4**.

PAC4: Data Gathering	
Task	*Dates*
Implementation Part II [2]	23/04 - 29/04
Testing & debugging	30/04 - 04/05
Data Gathering	05/05 - 15/05
R language study	23/04 - 15/05
Data Analysis	15/05 - 20/05
PAC4 Report	23/04 - 20/05
PAC4 submit	20/05 - 20/05

1.5.5 Final report

The last task concerns writing the final report and presenting some results and conclusions after analysing all the gathered data from the previous task. It adapts and corrects, whenever necessary, any setbacks. Setbacks are common when working on a project of any sort. This last task enforces coherency.

Final Report & Presentation Video	
Task	*Dates*
Final Report	21/05 - 08/06
Presentation Video	09/06 - 15/06
Final Report submit	08/06 - 08/06
Video submit	15/06 - 15/06

[1] Voluntary Context Switching accounting only.
[2] Involuntary Context Switching accounting.

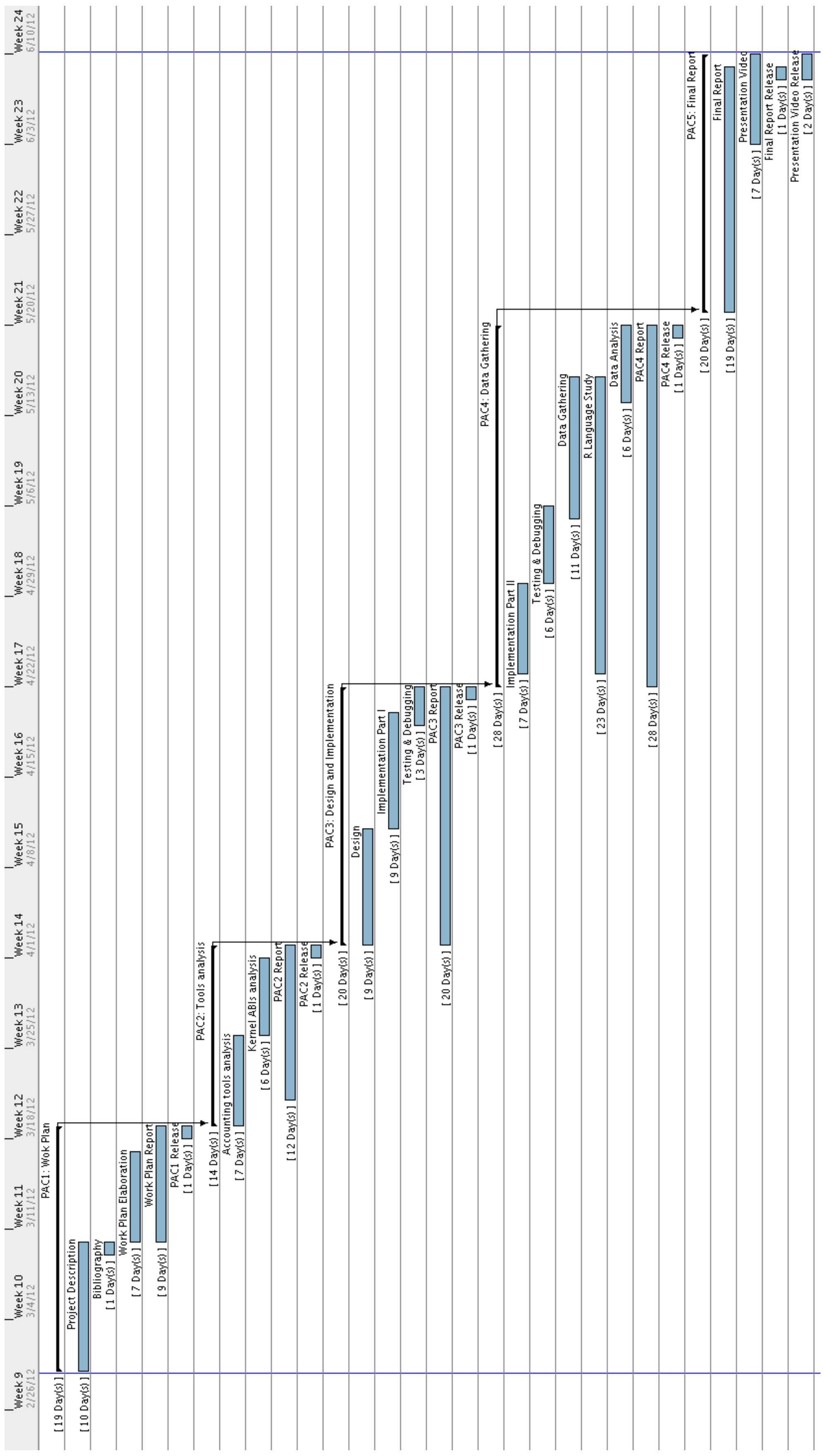


Figure 1.1: Context Switching Accounting: Gantt Chart

Chapter 2

Tools, APIs & ABIs analysis

2.1 Introduction

This chapter analyses different tools, APIs and Kernel ABIs originally designed for gathering statistical information concerning tasks, above all those ones involving Context Switching accounting. As reported by our work plan, this chapter is divided into six main sections. The first one deals primarily with tools, like `atsar` [9]. The second and third one focus on APIs and Kernel ABIs, that is, routines and frameworks we could use during the task design of our Context Switching Data Gathering tool, like `PAPI` and `Kernel taskstats` [11]. Whilst elaborating these sections, we will perform a trivial experiment to illustrate the way we can make use of the `taskstats` interface in order to fulfill our purposes: by extending it - see Section 2.4.2 -. Then, a section describing our framework, focusing on particular software-vendor versions, utilities, compilers, and so on has been included in order to frame our project. Finally, the last section provides some preliminary conclusions.

Whilst analysing these tools, some in-deep breakdown has taken place whenever feasible or necessary, so that a perfect understanding of how they come to work was forfeit. This way, our chances of engineering a new tool will increase considerably.

A Gantt chart concerning this pre-analysis task is shown in Figure 2.1.

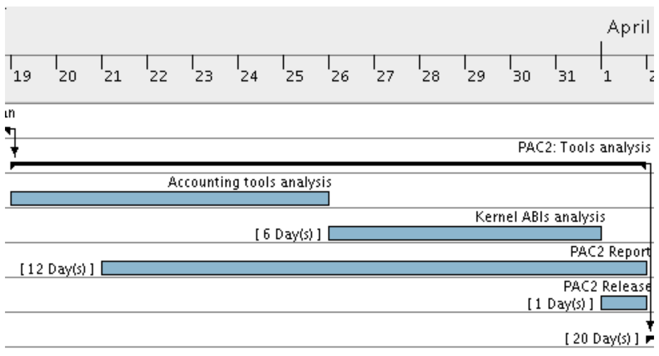


Figure 2.1: Context Switching Accounting: Tools & ABIs analysis Gantt Chart

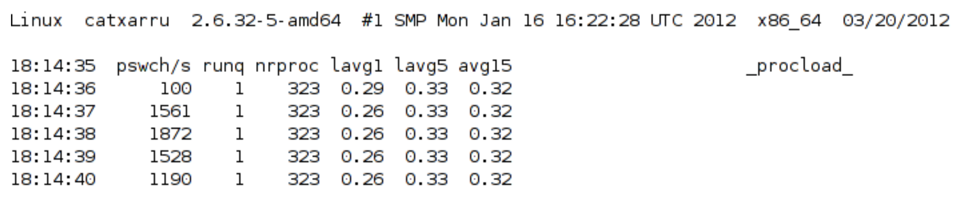
2.2 Tools

The GNU/Linux Kernel exports part of its internal data structures as user-readable files under the `/proc` directory. Thus, writing a user-space tool capable of reading those files and report statistical data is an easy task to do. Thus, a bunch of tools of the sort do exist. Below, this section provides a thorough analysis of a few of them.

2.2.1 `atsar`

`atsar` can be installed on any GNU/Linux distribution flavour. It is a simple tool written purely in C that reads data from the `/proc/stat` file, and presents it in a more human-like form. It can be used to gather cumulative data at n intervals of time t. According to its man page,

> The program `atsar(1)` can be used to deliver statistics. The design of this program can be compared with the standard `sadc(1)` and `sar(1)` programs being delivered for other UNIX-implementations.



```
Linux  catxarru  2.6.32-5-amd64  #1 SMP Mon Jan 16 16:22:28 UTC 2012  x86_64  03/20/2012

18:14:35  pswch/s runq nrproc lavg1 lavg5 avg15                    _procload_
18:14:36     100    1    323  0.29  0.33  0.32
18:14:37    1561    1    323  0.26  0.33  0.32
18:14:38    1872    1    323  0.26  0.33  0.32
18:14:39    1528    1    323  0.26  0.33  0.32
18:14:40    1190    1    323  0.26  0.33  0.32
```

Figure 2.2: `atsar`: Gathering cumulative Context Switching data; $n = 5, t = 1s$

Running `atsar` with the flag `-P` will show cumulative amount of Context Switches occurring at n intervals of t seconds, as shown in Figure 2.2. The second column in figure 2.2, `pswch/s`, involves all processors and runnable tasks on the system. Thus, it is far from being of service in our case. Its source is the field `ctxt`, which can be simply read by issuing:

```
$ cat /proc/stat|grep ctxt
```

A similar behaviour can be easily achieved by writing a trivial shell script, as shown in Figure 2.3. Its execution is shown in Figure 2.4.

2.2.2 `pidstat`

`pidstat` belongs to the `sysstat` [10] package. This tool deals with per-task statistics. Whereas `atsar` reads data from the `/proc/stat` file, `pidstat` uses the per-task statistical files located at `/proc/`*`PID`*`/`. Some of these files are exported by the GNU/Linux Kernel using the **Kernel TaskStats** interface [11], in consonance to [13]. Section 2.4.2 discusses about this GNU/Linux Kernel ABI widely.

By running `pidstat` with flags `-p` *`PID`* and `-w`, we can obtain the total amount of voluntary and involuntary context switches per second for that particular process. It simply reads statistical data, exported by the TaskStats infrastructure as briefly introduced earlier, from the `/proc/PID/status` file. A trivial example of `pidstat`'s execution is shown in Figure 2.5.

2.3 PAPI

PAPI can be used to access modern processors' counter registers from user-space programs. A counter register is architecture-dependent. Thus, PAPI provides abstraction, though it allows low-level calls in order to access counter registers not widely available for other platforms. Unluckily, as stated in its own reference:

```
STATS=/proc/stat            # This is the file "atsar" gets its stats from
TS=1s                       # 1 second by default   - atsar t parameter -
N=1                         # 1 iteration by default. - atsar n parameter -

if [ $# -ge 1 ]; then
    TS=${1}s                # Set the time in secs
    # Do we have the n parameter ? :
    test $# -eq 2 && N=$2
fi

echo "Context Switching cumulative stats at $N intervals of time $TS ... "

i=1
while [ $i -le $N ]; do

    # Read /proc/stat:
    a=`cat $STATS|grep ctxt|cut -d" " -f2`
    sleep $TS
    b=`cat $STATS|grep ctxt|cut -d" " -f2`

    # Show the computed value:
    r=`expr $b - $a`
    echo -ne "$i    \t  $r  \n"

    # Next iterarion
    i=`expr $i + 1`
done

# Show the total ones since the system had been booted-up:
echo "Total Context Switches: [`cat $STATS|grep ctxt|cut -d" " -f2`]"

exit 0
```

Figure 2.3: A trivial bash-script reading `/proc/stats`'s `ctxt` field

```
Context Switching cumulative stats at 5 intervals of time 1s ...
1                       1808
2                       1659
3                       1199
4                       1176
5                       1088
Total Context Switches: [9661197]
```

Figure 2.4: Gathering cumulative Context Switches by running the script in Figure 2.3; $n = 5, t = 1$

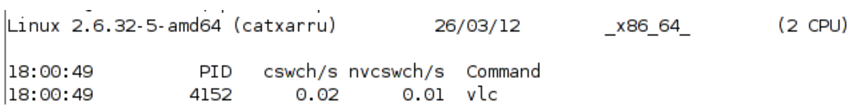
> PAPI only tracks 'hardware events', the occurrence of signals on-board the microprocessor. It does not count system calls, software interrupts or other software events. The user should remember that by default, PAPI only measures events that occur in User Space.

Our project deals precisely with System Calls - a particular type of Software Interrupt -. Therefore, we can not rely on PAPI. Moreover, there is no explicit Context-Switching per-processor's counter registers, as shown in [17]. Another approach is required.

2.4 Kernel ABIs

This section describes some important GNU/Linux Kernel ABIs involved in Context Switching accounting mechanisms. Any available user-space tool in charge of reading and presenting this data is heavily based on the `/proc` directory interface. As we have demonstrated in the previous sections, these tools read what the GNU/Linux Kernel provides through the `/proc` directory, albeit they do so by presenting this data in a more suitable human form.

First of all, this section focuses on an important GNU/Linux Kernel data structure, that is, the `task_struct` data structure. Then, it describes the Kernel TaskStats ABI



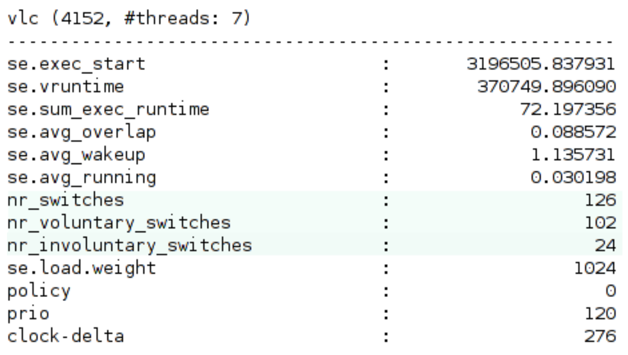
```
Linux 2.6.32-5-amd64 (catxarru)         26/03/12        _x86_64_        (2 CPU)

18:00:49          PID   cswch/s nvcswch/s  Command
18:00:49         4152      0.02      0.01  vlc
```

Figure 2.5: `pidstat`: Gathering voluntary and involuntary Context-Switches per second for a given task

and establishes a direct relationship between them.

2.4.1 `task_struct` data structure

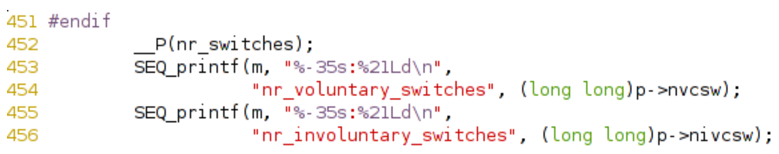
This data structure is defined in `include/linux/sched.h`. As far as the GNU/Linux Kernel is concerned, any process on the system is described using this data structure. Two fields are particularly interesting for our purposes: `nvcsw` and `nivcsw`. Both fields, being defined as `unsigned long`, store the number of voluntary and involuntary Context Switches any given task has underwent, respectively.



```
vlc (4152, #threads: 7)
---------------------------------------------------------
se.exec_start                      :        3196505.837931
se.vruntime                        :         370749.896090
se.sum_exec_runtime                :             72.197356
se.avg_overlap                     :              0.088572
se.avg_wakeup                      :              1.135731
se.avg_running                     :              0.030198
nr_switches                        :                   126
nr_voluntary_switches              :                   102
nr_involuntary_switches            :                    24
se.load.weight                     :                  1024
policy                             :                     0
prio                               :                   120
clock-delta                        :                   276
```

Figure 2.6: Reading the `/proc/PID/sched` file directly

Thus, the GNU/Linux Kernel does differentiate among voluntary and involuntary Context Switching accounting per task. This information can be read without using existing user-space tools but by reading the `/proc/PID/sched` file directly instead, as shown in Figure 2.6. In order to do so, the `CONFIG_SCHED_DEBUG` kernel option must be set. Even more statistical data can be obtained and presented by the GNU/Linux Kernel if the `CONFIG_SCHEDSTATS` kernel option is previously enabled, also. The code-snippet in Figure 2.7 shows how the Kernel exports the `nvcsw` and `nivcsw` accounting fields to the `/proc` directory interface.



```
#endif
        __P(nr_switches);
        SEQ_printf(m, "%-35s:%21Ld\n",
                   "nr_voluntary_switches", (long long)p->nvcsw);
        SEQ_printf(m, "%-35s:%21Ld\n",
                   "nr_involuntary_switches", (long long)p->nivcsw);
```

Figure 2.7: Code-Snippet: `kernel/sched_debug.c`

So, the GNU/Linux Kernel does offer a trivial way of gathering the total amount of Context Switches occurred to a particular process since its creation. It does account the number of voluntary and involuntary context switches and exports it to the `/proc` directory interface for any normal user to read. However, this approach is not fairly accurate and thus does not serve our purposes completely. Our project needs to determine the exact cause for any either voluntary or involuntary Context Switch, according to our previous discussion concerning the Context Switching mechanisms, as described briefly in [14].

2.4.2 Kernel TaskStats

As reported by the GNU/Linux Kernel Documentation, `Kernel TaskStats` is meant to be the best way to gather and report statistical data from any existing GNU/Linux task, even when this same task is exiting, that is, finishing its execution calling `exit()` or just returning from the `main()` function. Current `Kernel Taskstats`'s implementation accounts the number of voluntary and involuntary Context Switches per task, as described in [13].

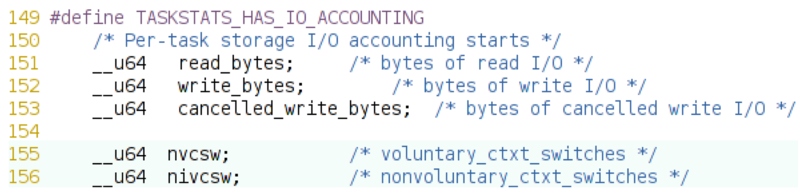
In order to `Kernel TaskStats` to be enabled, the `CONFIG_TASKSTATS` kernel option must be set. Additional statistics per task can be obtained by setting another `Kernel TaskStats`-related option: `CONFIG_TASK_DELAY_ACCT`, conforming to [11]. The GNU/Linux Kernel source contains a sample C program that uses the `Kernel TaskStats` ABI to gather some statistical information per task [12]. Next section presents the `Kernel TaskStats` interface, focusing only on those parts directly related to our project. It follows another section that provides an exhaustive analysis of `getdelays.c`, paying special attention to the **Netlink** ABI and the way it gets the `nvcsw` and `nivcsw` fields. To conclude, the last section establishes a relationship between the `task_struct` data structure, discussed in section 2.4.1 and the `Kernel TaskStats` interface.

The GNU/Linux Kernel TaskStats interface

As stated in its Documentation,

> Taskstats is a **netlink**-based [15] interface for sending per-task and per-process statistics from the kernel to user-space. Taskstats was designed for the following benefits:
>
> - Efficiently provide statistics during lifetime of a task and on its exit.
> - Unified interface for multiple accounting subsystems.
> - Extensibility for use by future accounting patches.

A **netlink**-based interface is another method of communicating with the GNU/Linux Kernel from user-space processes. It works both ways; that is, it allows the GNU/Linux Kernel to send messages to user-space processes as well. Other well-known one-way methods are: **ioctl**, **syscalls** and the `/proc` directory interface.



```
#define TASKSTATS_HAS_IO_ACCOUNTING
    /* Per-task storage I/O accounting starts */
    __u64   read_bytes;      /* bytes of read I/O */
    __u64   write_bytes;         /* bytes of write I/O */
    __u64   cancelled_write_bytes;  /* bytes of cancelled write I/O */

    __u64  nvcsw;            /* voluntary_ctxt_switches */
    __u64  nivcsw;           /* nonvoluntary_ctxt_switches */
```

Figure 2.8: `The GNU/Linux Kernel TaskStats`: `taskstat` data structure

Thus, `Kernel TaskStats` consists basically on a well-designed data-structure storing statistics for every single process on the system and its interface to be called from user-space processes in order to obtain them. As described in its documentation, this data structure can also be extended. This is the main point in our project.

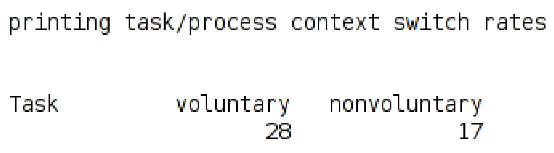
In accordance with [16], fields `nvcsw` and `nivcsw` contain the total amount of voluntary and involuntary Context Switches per task, respectively. These are exactly the same fields as described earlier, in section 2.4.1. This is shown in Figure 2.8. The way they are related to each other is described in Section 2.4.2.

Analysing `getdelays.c`

`getdelays.c` is a sample C program written to illustrate the `netlink` and `Kernel TaskStats` interfaces. It can easily be compiled by issuing the command:

```
$ gcc getdelays.c -o getdelays
```

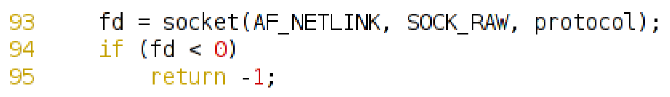
By running `getdelays` with flags `-p` *`PID`* and `-q`, we can read the `nvcsw` and `nivcsw` fields, as described in Section 2.4.2. This is shown in Figure 2.9.



```
printing task/process context switch rates


Task            voluntary   nonvoluntary
                       28             17
```

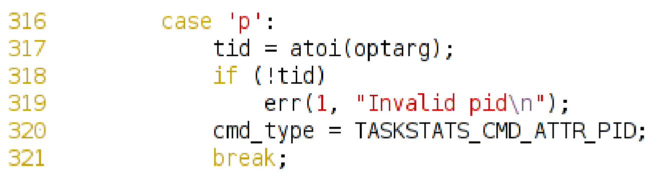
Figure 2.9: `The GNU/Linux Kernel TaskStats`: Reading `nvcsw` and `nivcsw` with `getstats` sample code

This code uses the **netlink** interface to communicate with the GNU/Linux Kernel. It creates an `AF_NETLINK` socket - Figure 2.10 -, sends a `TASKSTATS_CMD_GET` message to the GNU/Linux Kernel - Figure 2.12 - and waits until it receives a Kernel's response - Figure 2.13 -. Then, it simply prints the process's statistics - Figure 2.14 -. Before sending the actual message to the kernel, `getdelays` gets the process's pid from the command line and sets the variable `cmd_type` to `TASKSTATS_CMD_ATTR_PID`, as shown in Figure 2.11. In consonance with [11], this is the usual way of sending a message from user-space processes to the GNU/Linux Kernel, in order to get statistics for a given process.



```
    fd = socket(AF_NETLINK, SOCK_RAW, protocol);
    if (fd < 0)
        return -1;
```

Figure 2.10: `getdelays`: Creating an `AF_NETLINK` socket



```
        case 'p':
            tid = atoi(optarg);
            if (!tid)
                err(1, "Invalid pid\n");
            cmd_type = TASKSTATS_CMD_ATTR_PID;
            break;
```

Figure 2.11: `getdelays`: Obtaining the process's pid from the cli and setting up the command to be sent: `TASKSTATS_CMD_ATTR_PID`

Altering `getdelays.c`'s behaviour is easy. First of all, we don't need to deal with all the sort of Netlink or TaskStats messages received from the Kernel. Our project needs to focus only on Context Switching accounting. In order to achieve that, `getdelays.c` has been easily converted into another simple user-space tool, that is, `getcsw.c`. We have removed any pointless piece of code, and made some small changes on it. Its execution is

```
    if (tid) {
        rc = send_cmd(nl_sd, id, mypid, TASKSTATS_CMD_GET,
                      cmd_type, &tid, sizeof(__u32));
        PRINTF("Sent pid/tgid, retval %d\n", rc);
        if (rc < 0) {
            fprintf(stderr, "error sending tid/tgid cmd\n");
            goto done;
        }
    }
```

Figure 2.12: `getdelays`: Sending a netlink message to the GNU/Linux Kernel

```
    do {
        int i;

        rep_len = recv(nl_sd, &msg, sizeof(msg), 0);
        PRINTF("received %d bytes\n", rep_len);
```

Figure 2.13: `getdelays`: Receiving a netlink message from the GNU/Linux Kernel

shown in Figure 2.15. We can get the total amount of voluntary and involuntary context switches per-task at intervals of t seconds by running **`getcsw`** this way:

```
$ ./getcsw -p PID -v -l -d t
```

where t is the delay in seconds among subsequent Netlink messages sent to the kernel.

Extending TaskStats

Let $*s$ be a pointer to a `taskstats` data structure. Let's consider a certain netlink message, m, being of type `TASKSTATS_CMD_ATTR_PID`, that has been sent to the kernel. In line with [11], a response will contain the whole data structure, that is, $*s$, as long as its version is paired with that one specified by m. Bearing this is mind, it is feasible to extend the `taskstats` data structure by two different ways, as stated in [11]. Our project uses the first approach:

> Adding more fields to the end of the existing struct taskstats. Backward compatibility is ensured by the version number within the structure. Userspace will use only the fields of the struct that correspond to the version its using.

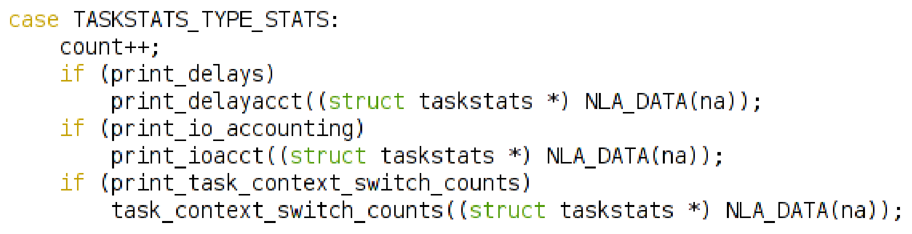
This first experiment deals with this idea. A certain `dummy` field, of type `__u64`, has been added to the `taskstats` data structure, defined in `linux/taskstats.h`. This field has been added at the end of the data structure, accordingly. The `__u16 version` field has been incremented by one, pursuant to:

> To add new fields:
>
> - Bump up `TASKSTATS_VERSION`.
> - Add comment indicating new version number at end of struct.
> - Add new fields after version comment; maintain 64-bit alignment.

This is shown in Listing 2.1.

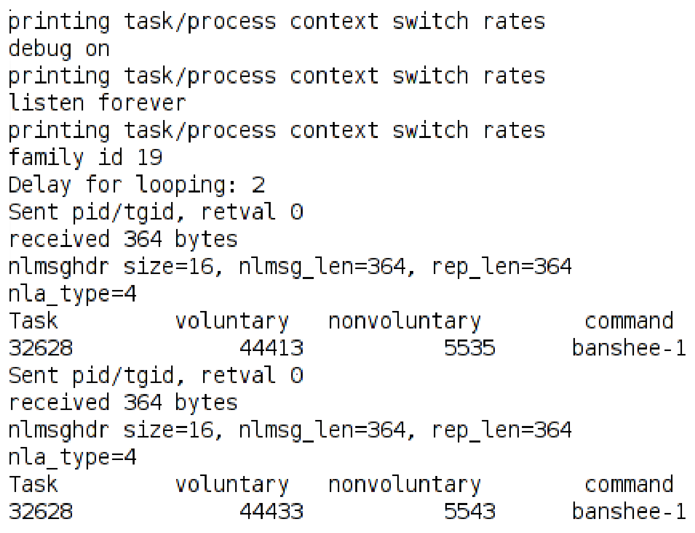
Listing 2.1: Extending the `taskstats` data structure

```
#define TASKSTATS_VERSION    8
...
```



```
case TASKSTATS_TYPE_STATS:
    count++;
    if (print_delays)
        print_delayacct((struct taskstats *) NLA_DATA(na));
    if (print_io_accounting)
        print_ioacct((struct taskstats *) NLA_DATA(na));
    if (print_task_context_switch_counts)
        task_context_switch_counts((struct taskstats *) NLA_DATA(na));
```

Figure 2.14: **getdelays**: Printing per-task statistics



```
printing task/process context switch rates
debug on
printing task/process context switch rates
listen forever
printing task/process context switch rates
family id 19
Delay for looping: 2
Sent pid/tgid, retval 0
received 364 bytes
nlmsghdr size=16, nlmsg_len=364, rep_len=364
nla_type=4
Task          voluntary   nonvoluntary        command
32628              44413            5535      banshee-1
Sent pid/tgid, retval 0
received 364 bytes
nlmsghdr size=16, nlmsg_len=364, rep_len=364
nla_type=4
Task          voluntary   nonvoluntary        command
32628              44433            5543      banshee-1
```

Figure 2.15: **getcsw**: Getting per-task Context Switches

```
    struct taskstats {
      ...
        /* New version: 8*/
        __u64    dummy;
    }
```

As discussed in Section 2.4.2, the function `fill_pid()` has also been modified in order to initialize `dummy` to a default numeric value of 666, as shown in Listing 2.2.

Listing 2.2: Setting `dummy`'s default value to 666

```
    /* Let's fill "dummy" : */
    stats->dummy = 666;
```

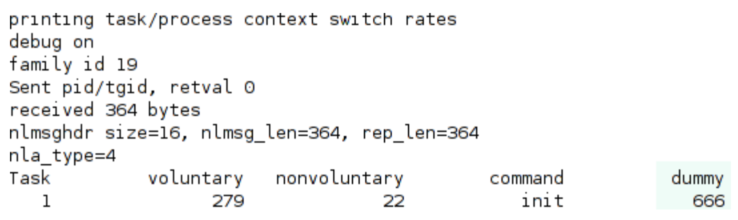
Finally, `getcsw.c` has been slightly altered so that this new field can be read and printed, as shown in Listing 2.3. Its execution is shown in Figure 2.16.

Listing 2.3: Reading `dummy`'s value selectively

```
    #define MIN_VERSION 8
    ...
    /* Get dummy if we do have the right version ... */
        (t->version>=MIN_VERSION)?t->dummy:0
    );
    ...
```

`task_struct` and `Kernel TaskStats` relationship

As shown in Figure 2.17, their relationship is truly simplistic. The GNU/Linux Kernel does account the total amount of voluntary and involuntary Context Switches per-task as



```
printing task/process context switch rates
debug on
family id 19
Sent pid/tgid, retval 0
received 364 bytes
nlmsghdr size=16, nlmsg_len=364, rep_len=364
nla_type=4
Task          voluntary   nonvoluntary        command          dummy
   1                279             22           init            666
```

Figure 2.16: `getcsw.c`: Reading `dummy`'s value from the Kernel

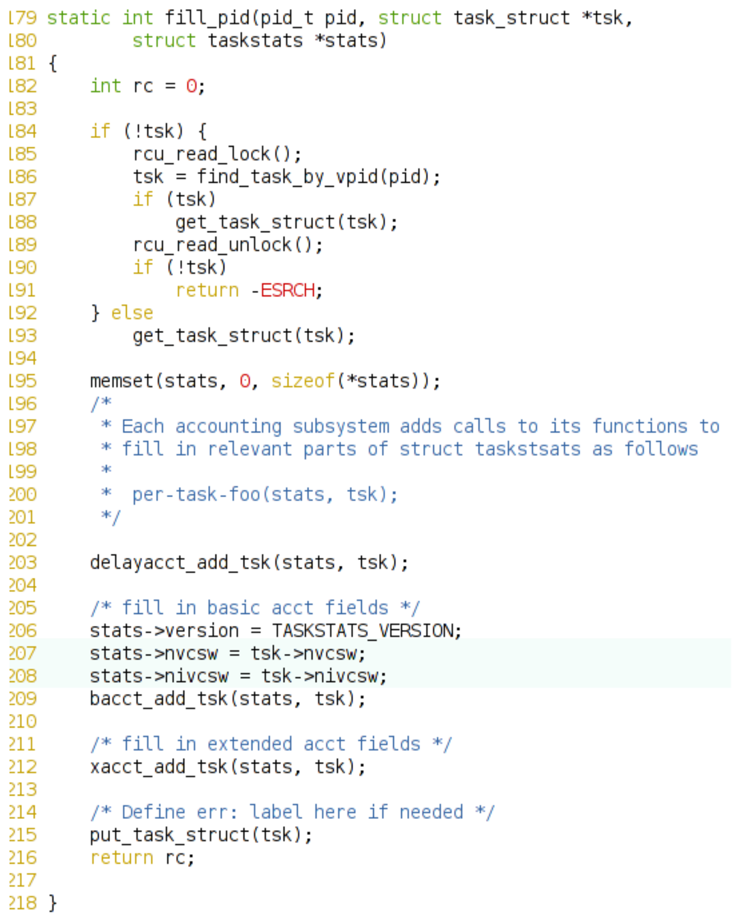
previously introduced in Section 2.4.1. In line with Figure 2.17, the `Kernel TaskStats` infrastructure copies both `task_struct` `nvscw` and `nivcsw` fields into the correspondent ones pointed by `*stats`. The function `fill_pid()` applies whenever statistics for a single task are required. This function is implemented in the `kernel/taskstats.c` source file.

2.5 Setup framework

Dealing with the GNU/Linux Kernel in an experimental manner implies risks. Any sort of misconducted step can possibly end up triggering a **Kernel OOPS** or even a **Kernel Panic**. Thus, framing it inside a **Virtual Machine** is mandatory. This project will design and implement a tool capable of differentiating the type of Context Switch a certain process has just endeavoured. Thus, our first approach will focus just on that, without paying too much attention to other aspects like performance issues which are, to an extend, of a paramount importance on a real environment.

Below, an enumeration of our setup framework is provided:

- ***VirtualBox 4.1***. Our tests will be conducted all the time inside a Virtual Machine. This vm does have all sorts of utilities, compilers and debuggers. In favour of pragmatism, we access this vm through a secure shell connection from our actual GNU/Linux box.
- ***Debian GNU/Linux Squeeze 6.0.4 X86_64***. This is the GNU/Linux distribution installed on the vm. We chose a 64 bit architecture because most of nowadays GNU/Linux computers run this kind of GNU/Linux flavour, due to modern processors. We forcibly avoided 32 bit architectures, albeit this project will use, whenever feasible, the GNU/Linux Kernel ABIs facilities so as to guarantee architecture-independence.
- ***GNU/Linux Kernel 2.6.32.5 X86_64***. The stable GNU/Linux Kernel release distributed by default with Debian GNU/Linux Squeeze 6.0.4. Its sources have been installed using the `apt-get` package manager under the `/usr/src/linux-source-2.6.32`. Our project will study this GNU/Linux kernel sources and will make use of its ABI intensely. Thus, whenever a code snippet is provided or a certain data structure discussed, it will always refer to this particular version.
- ***Vim 7.2.445***. Our text editor program.
- ***gcc 4.4.5***. All the code will be compiled using this version of the GNU C Compiler. Portability is an important feature of this project. Thus, our code will be implemented mainly in C.
- ***cscope 15.7a***. Browsing the entire GNU/Linux Kernel source is tiresome. In order to find a particular data structure or function quickly, `cscope` will be used.



```
static int fill_pid(pid_t pid, struct task_struct *tsk,
        struct taskstats *stats)
{
    int rc = 0;

    if (!tsk) {
        rcu_read_lock();
        tsk = find_task_by_vpid(pid);
        if (tsk)
            get_task_struct(tsk);
        rcu_read_unlock();
        if (!tsk)
            return -ESRCH;
    } else
        get_task_struct(tsk);

    memset(stats, 0, sizeof(*stats));
    /*
     * Each accounting subsystem adds calls to its functions to
     * fill in relevant parts of struct taskstsats as follows
     *
     *  per-task-foo(stats, tsk);
     */

    delayacct_add_tsk(stats, tsk);

    /* fill in basic acct fields */
    stats->version = TASKSTATS_VERSION;
    stats->nvcsw = tsk->nvcsw;
    stats->nivcsw = tsk->nivcsw;
    bacct_add_tsk(stats, tsk);

    /* fill in extended acct fields */
    xacct_add_tsk(stats, tsk);

    /* Define err: label here if needed */
    put_task_struct(tsk);
    return rc;

}
```

Figure 2.17: `The GNU/Linux Kernel TaskStats`: filling per-task statistics

Right after decompressing the GNU/Linux Kernel source files, the command `make cscope` was executed to create `cscope`'s database and cross-reference information.

- ***GNU binutils 2.20.1-16***.
- ***gdb 7.0.1-debian***. Whenever a debugging session is required, `gdb` will be used. Debugging the GNU/Linux Kernel is tricky. We will use `gdb` only for user-space programs debugging purposes.
- ***R-language 2.11.1***. In order to analyse and plot all the data gathered from the execution of our new Context Switching Accounting utility, we will use the R-language, as presented in [14].

Chapter 3

Design & Implementation

3.1 Introduction

The GNU/Linux Kernel does account Voluntary and non-Voluntary Context Switches per-task, as we have demonstrated in [18]. However, we have also demonstrated that there are no actual mechanisms to split these context switches in further categories. This chapter puts forward a stable yet experimental method to achieve precisely that. This technique and implementation does apply to Voluntary and Involuntary Context Switching.

This chapter is structured as follows: Section 3.2 provides background information on Voluntary and Involuntary Context Switching, respectively. Section 3.3 conducts a deep insight into the GNU/Linux Kernel so that a clear picture of the way it does increment both counters, that is, `nvcsw` and `nivcsw` [19], can be achieved. Section 3.4 describes in depth our Context Switching Accounting mechanism's implementation. Finally, Section 3.5 demonstrates the way our tool can be applied to the GNU/Linux Kernel mainstream.

This task has been conducted according to the gantt chart shown in Figure 3.1.

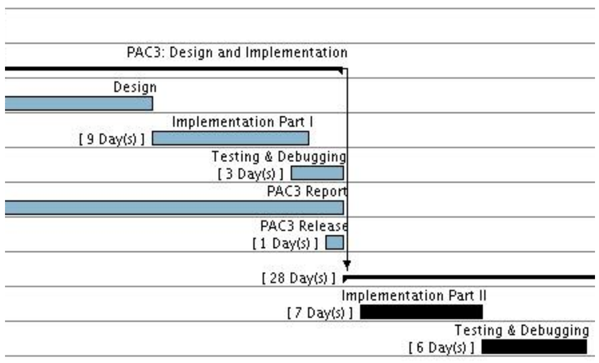


Figure 3.1: Context Switching Accounting: Design & Implementation Gantt Chart

3.2 Background

Voluntary Context Switching

Let p be a process running on a GNU/Linux box, so that $(p \rightarrow state) \longleftarrow TASK\_RUN NING$. In such scenario, Voluntary Context Switching can only happen whenever p is waiting for some event to occur or some hardware to be ready. In this particular case, p removes itself from the active **run queue** and puts itself into the proper **wait queue**.

Thus, we have stated that Voluntary Context Switching implies, for any given process p_i in a processor's **run queue**, and provided that $(p_i \rightarrow state) \longleftarrow TASK\_RUNNING$ applies, yielding the processor by self-calling the `schedule()` function.

Let's assume p is a **user-space** application. p can relinquish the processor voluntarily only whilst the GNU/Linux Kernel is in **process context** or whenever returning from a System Call. Whenever a user-space program issues a System Call, the GNU/Linux kernel is in process context. That is so because the GNU/Linux Kernel is running on behalf of p. So, Voluntary Context Switching can happen either in **Ring 0** or in returning to **Ring 3** from **Ring 0**. p cannot relinquish the processor where it is running on otherwise; that would be Involuntary Context Switching. Therefore, p could only be preempted itself **due to issued system calls**. Table 3.1 summarizes these situations.

Mode	*Resource*	*Cause*
User-Mode	-	Returning from a System Call
User-Mode	-	An explicit call to `sched_yield()`
Process Context	Any	An issued System Call that can block

Table 3.1: Voluntary Context Switching events

Our project takes into account some particularities from Table 3.1. Let vc be the total amount of Voluntary Context Switches during p's life cycle; let rs be the total amount of Context Switches taking place whilst returning from a system call; let vb be the calls to `schedule()` in **process context**, that is, **voluntary blocking**; let ve be the total amount of calls to `schedule()` during p's exit. Then, it immediately follows that:

$$vc = rs + vb + ve \tag{3.1}$$

Equation 3.1 applies if we do consider p to relinquish the processor by itself whilst returning from a System Call, as long as `schedule()` is called. That is so because any probable call to `schedule()` will be performed by the `current` task, that is, p. As next sections will prove, it can be stated that $rs \ll vb$ and that $ve = 1$ as long as p is not **io-bounded** or, if it is, it is not issuing a huge amount of system calls. Thus, we can rewrite 3.1 as follows:

$$vc \approx vb \tag{3.2}$$

given any p_i which is not **io-bounded** or is not issuing a large number of System Calls.

As described in section 3.4.5, our GNU/Linux Kernel patch does add some code so that rs can be accounted. Whenever p exits, our patch can also determine if the subsequent call to **schedule()** has been issued by p itself or not. If p has issued a call to `exit()`, it is considered as Voluntary Context Switching- p wants to exit, so it does yield the processor by calling the `exit()` function -.

Involuntary Context Switching

Let's consider a certain process, that is, p_2, whose state is either `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`. This task is in a **wait queue**, therefore it is waiting for some event to occur. Whenever p_2 awakes, whether this is so due to the fact that the awaited event has finally happened or p_2 has been signaled, the consequences are quite the same: the `current` task, let's say p_1, can possibly be preempted, thus allowing p_2 to be executed in a near or immediate future. If p_2 is meant to be executed immediately, p_1 is said to relinquish the processor involuntarily as long as p_2 had a higher priority. Otherwise, p_2 would still be removed from its wait queue and put in a **run queue**, but could not possibly be executed immediately, thus allowing p_1 to exhaust its **processor's time proportion**.

Apart from exhausting its time slice or being preempted because of another awaking task with a higher priority, p_1 can still yield the processor because of an **Interrupt Handler**. An Interrupt Handler is triggered whenever a piece of hardware requires processor's attention. This is clearly an Involuntary Context Switch, because p_1 has nothing to do with the fired interrupt. However, let's suppose this scenario: a certain process, p, calls `read()`. Somehow or other, this call to the `read()` function can provoke a hardware interrupt, because the hard disk has to tell the processor it does have that data ready to be read. Then it immediately follows that p can possibly be preempted because of the execution of an interrupt handler that has been fired as a direct consequence of p's behaviour. Our project does track every single hardware interrupt. Whenever a piece of hardware issues an interrupt, the GNU/Linux Kernel has to deal with it. Thus, p can yield the processor if that same processor is the one in charge of running the Interrupt Handler associated with the interrupt line that has just been triggered. The GNU/Linux Kernel is said to be in **Interrupt Context** whilst processing a hardware interrupt. Table 3.2 summarizes these situations.

Mode	*Cause*
Process Context or User Mode	A higher priority task has been awakened
Process Context or User-Mode	Time-Slice exhaustion
Interrupt Context	A piece of hardware has triggered an interrupt

Table 3.2: Involuntary Context Switching events

3.3 The GNU/Linux `schedule()` implementation

The GNU/Linux Kernel scheduler is invoked directly or in a deferred way. In consonance with the `cscope` command, there are about **561** direct calls to the `schedule()` function for the 2.6.32 GNU/Linux Kernel version. A direct call to the scheduler is easy to understand: whenever a part of the GNU/Linux Kernel needs to block, typically inside drivers, an explicit call to the `schedule()` function is made, thus invoking the GNU/Linux scheduler in doing so.

Deferred calls to the GNU/Linux scheduler are consequence of setting the `TIF_NEED_RESCHED` flag of the `thread_info` data structure for a given process, that is, `thread_info->flags`. The GNU/Linux Kernel always checks this flag whilst trying to resume user-processes' execution, so whenever it is set, the GNU/Linux Kernel Scheduler will be invoked in a certainly near future.

As far as Voluntary Context Switching is concerned, a given process p can voluntarily relinquish the processor where it is running on by calling the GNU/Linux scheduler in a direct way. However, as this section will prove, we do consider Voluntary Context Switching any explicit issued call to `schedule()` by the `current` task whereas the GNU/Linux Kernel does not. Thus, whilst returning from a System Call, `current` always refers to the actively running process. Then it immediately follows that `current` checks its own `TIF_NEED_RESCHED` flag, and if it is set, it does self-call the `schedule()` function.

The GNU/Linux scheduler is also in charge of incrementing the `nvcsw` and `nivcsw` fields. According to Listing 3.1, it does increment either `nvcsw` or `nivcsw` depending on what condition the `current` task meets. Line 7 shows that, by default and whenever executing this call, a primary supposition is made: that is, that the Context Switch that is about to happen is due to Involuntary Context Switching. This is done by making the `switch_count` pointer point at the `prev->nivcsw`'s address.

Listing 3.1: The GNU/Linux `schedule()`'s implementation.

```
1  need_resched:
2    preempt_disable();
3    cpu = smp_processor_id();
4    rq = cpu_rq(cpu);
5    rcu_sched_qs(cpu);
6    prev = rq->curr;
7    switch_count = &prev->nivcsw;
8  ...
9  if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
10     if (unlikely(signal_pending_state(prev->state, prev)))
11          prev->state = TASK_RUNNING;
12       else
13          deactivate_task(rq, prev, 1);
14     switch_count = &prev->nvcsw;
15 }
16 ...
17 if (likely(prev != next)) {
18   ...
19   rq->nr_switches++;
20   rq->curr = next;
21   ++*switch_count;
22   ...
23 }
```

Lines 9-15 are the ones related to Voluntary Context Switching. The GNU/Linux Kernel is preemptive, that means not only user-space tasks can be preempted but the kernel itself. That can only happen whenever a particular task that is running in kernel-space does not hold any lock. In listing 3.1, line 9, only when the task is not runnable and it has not been preempted in kernel-space, does the Kernel suppose it is a Voluntary Context Switch, thus pointing the `switch_count` at the `nvcsw` field - as shown in Line 14 -. Whenever the task has been preempted in Kernel Space, the scheduler does consider it an Involuntary Context Switch. According to Listing 3.1, the bit `PREEMPT_ACTIVE` is checked by a trivial bit-wise **and** operation over the `thread_info->preempt_count` task field, by means of calling the function `preempt_count()`. Whenever this bit is set, this task has been preempted whilst being in **kernel-space**, thus the kernel does infer there is an Involuntary Context Switch involved. `TASK_RUNNING` is defined as 0, according to the `include/linux/sched.h` header file. Thus, whenever `prev->state` matches 0, the conditional branch in line 9 does not resolve to true, that is, ≥ 1. Only a state that complies to > 0, that is, `prev->state!=TASK_RUNNING`, and provided that `PREEMPT_ACTIVE` has not been set, can provoke a Voluntary Context Switch. Previous versions of the GNU/Linux Kernel tended to write this branched piece of code as shown in Listing 3.2. Both conditional branches are equivalent.

When a certain process p voluntarily relinquishes the processor where it is running on, it deactivates itself from the run queue and puts itself on the proper wait queue. In doing so, it does update its own state so that it can be inferred that `p->state != TASK_RUNNING`. That is how the code-snippet shown in Listing 3.1 comes to work - Lines 9-15 -. Despite this reasonable approach, our project does consider any explicit call to the `schedule()` function issued by the `current` task as a Voluntary one. Thus, our `nvcsw_ext[0]` counter is incremented whilst returning from a system call and it is attached to the Voluntary Context Switching accounting.

Finally, as it is shown in Listing 3.1, lines 17 and 21, if the next task to be executed is different from the previous one, the GNU/Linux Scheduler increments the number of Context Switches by using the previously referenced pointer, that is, `switch_count`. This is quite remarkable in itself, because it does prove that not all the calls to the `schedule()` function end up in an actual Context Switch. Therefore, our `nvcsw_ext[0]` counter has to be considered as **the total probable amount of Context Switches whilst returning from a System Call**. Not all of them, although **most of them**, will be performed.

Listing 3.2: The GNU/Linux `schedule()` Voluntary Context Switch branch's old way implementation.

```
if (prev->state != TASK_RUNNING && !(preempt_count() & PREEMPT_ACTIVE)) {
        if (unlikely(signal_pending_state(prev->state, prev)))
            prev->state = TASK_RUNNING;
        else
            deactivate_task(rq, prev, 1);
        switch_count = &prev->nvcsw;
}
...
```

3.4 Extending the counters

Our main goal is to extend the `nvcsw` and `nivcsw` counters so that a more accurate accounting of every single Context Switch can be performed. This section describes in depth the way we have altered the GNU/Linux Kernel sources in order to do so.

3.4.1 Extending some Kernel Data Structures

As previously introduced in [20], it is feasible to extend the `taskstats` data structure in order to access a series of new counters per-task. However, the `task_struct` structure, described briefly in [21], has to be extended accordingly. That is so because all the statistics from which the GNU/Linux Kernel TaskStats interface fills its own fields in come directly from the `task_struct` data structure for any given process. Therefore, we have extended both data structures, as shown in Listings 3.3 and 3.4. Table 3.3 summarizes the meaning and purpose of these new extended fields.

Listing 3.3: Extending the `task_struct` data structure, `include/linux/sched.h`

```
struct task_struct {
...
  unsigned long nsyscalls;
  unsigned long nvcsw_ext[3];
  atomic64_t nivcsw_ext[2];
  unsigned long nivcsw_ext_exit;
  unsigned long nsyscalls_schedule[__NR_syscall_max+1];
...
}
```

Listing 3.4: Extending the `taskstats` data structure, `include/linux/taskstats.h`

```
#define TASKSTATS_VERSION     8

struct taskstats {
  ...
```

```
        __u64    nsyscalls;
        __u64    nvcsw_ext[3];
        __u64    nivcsw_ext[2];
        __u64    nivcsw_ext_exit;
    }
```

Counter	*Index*	*Description*
`nsyscalls`	-	Total amount of issued syscalls
	0	Calls to `schedule()` at `ret_from_sys_call`.
`nvcsw_ext[]`	1	Voluntary task's exit. [1]
	2	Calls to the `sched_yield()` system call.
`nsyscalls_schedule[]`	-	Number of calls to `schedule()` per syscall.
	0	Calls to `schedule()` during `try_to_wake_up()`.
`nivcsw_ext[]`	1	Calls to `schedule()` during `do_irq()`.
`nivcsw_ext_exit`	-	Involuntary task's exit. [1]

Table 3.3: Per-task's extended counters

Some of the functions and macros to be used all along this section can be located in the `include/kernel/tcg.h` header file, shown in Listing 3.5.

Listing 3.5: The `include/linux/tcg.h` header file containing macros and function prototypes

```
#define TCG_NSYSCALLS             tcg_nsyscalls
#define TCG_NSYSCALLS_SCHEDULE    tcg_nsyscalls_schedule
#define TCG_SCHED_YIELD           tcg_sched_yield
#define TCG_NVCSW_0               tcg_nvcsw_0

#ifndef __NR_syscall_max
    #define __NR_syscall_max      298
#endif
```

3.4.2 Initializing the counters

Whenever a new process is created, a call to `do_fork()` is made. Thus, it seems quite feasible to initialize our new counters precisely in the `do_fork()` function. In agreement with the `kernel/fork.c` source file, the `nvcsw` and `nivcsw` counters are initialized to 0 inside the function `copy_mm()`, called from the `do_fork()` one, as shown in Listing 3.6, line 8.

Listing 3.6: Initializing the `nvcsw` and `nivcsw` counters

```
 1  static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
 2  {
 3      struct mm_struct * mm, *oldmm;
 4      int retval, i;
 5
 6      tsk->min_flt = tsk->maj_flt = 0;
 7
 8      tsk->nvcsw = tsk->nivcsw = 0;
 9    ...
10  }
```

[1]This is not a proper counter; as described in section 3.2, there is typically one Context Switch whenever a task is ending its execution. Maybe it could be changed by a bit field this way: `nvcsw_exit:1`.

We have modified line 8 in Listing 3.6 in order to initialize our `nsyscalls` counter as well. Besides, we have inserted a pair of trivial calls to `memset()` in order to initialize the rest of our counters right after this code line, as it is clearly shown in Listing 3.7, lines 8 and 10-14.

Listing 3.7: Initializing the rest of our extended counters

```
1    static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
2    {
3        struct mm_struct * mm, *oldmm;
4        int retval, i;
5
6        tsk->min_flt = tsk->maj_flt = 0;
7
8        tsk->nvcsw = tsk->nivcsw = tsk->nsyscalls = 0;
9
10       memset(&tsk->nvcsw_ext,0,sizeof(tsk->nvcsw_ext));
11       atomic64_set(&tsk->nivcsw_ext[0],0);
12       atomic64_set(&tsk->nivcsw_ext[1],0);
13       tsk->nivcsw_ext_exit = 0;
14       memset(&tsk->nsyscalls_schedule,0,sizeof(tsk->nsyscalls_schedule));
15     ...
16   }
```

3.4.3 Accounting the total number of System Calls per-task

Whenever a certain process *p* issues a System Call, it does so by using the `libc` wrapper routines. This way, there is no need to know in advance the exact system calls' addresses. Moreover, the wrapper library is in charge of calling a system call either by using the old-way `int $0x80` instruction or the more recent `sysenter`, widely available on most of nowadays X86-based processors. All the code dealing with the System Call interface resides in the `arch/x86/kernel/entry_64.S` file, written purely in assembler. There are two entry points particularly interesting for our purposes; the first one applies to the `sysenter` instruction and the other one is for the old `int $0x80` instruction. Then, it immediately follows that we needed to add some code in both of them, apparently. Because the GNU/Linux Kernel has been developed to spare redundant code whenever feasible, the entry point for the `sysenter` ends up calling part of the low-level instructions executed in the old-days by the `int $0x80` instruction, that is, the `system_call` entry point. As shown in Listing 3.8, lines 4-5 check for a valid issued system call. Our counter is called a few lines later, thus it does not account any erroneous call to an unknown or not implemented system call. Right before executing the system call in line 12, the `rax` register is saved before calling our high-level implemented counter `TCG_NSYSCALLS`, and then it is immediately restored, as clearly shown in lines 8-11. This register has to be saved before calling our high-level C-routine in order to prevent `rax` from being thrashed. The value stored in `rax` is the system call number to be executed in line 12; should `rax` be thrashed, the entire system would probably collapse.

Our high-level C-routine, `tcg_nsyscalls()`, is implemented in a new file added to the GNU/Linux Kernel mainstream, that is, `arch/x86/kernel/tcg.c`. It is shown in Listing 3.9. As discussed in section 3.4.8, modular arithmetic has been used to avoid overflows. `TCG_NSYSCALLS` has been defined as a macro in the `include/linux/tcg.h` header file, as shown in Listing 3.10.

Listing 3.8: Calling the `tcg_nsyscalls()` high-level function

```
1    ENTRY(system_call)
2     ...
```

```
system_call_fastpath:
  cmpq $__NR_syscall_max,%rax
  ja badsys
...
  movq %r10,%rcx
  push %rax
  mov %rax, syscallid
  call TCG_NSYSCALLS
  pop %rax
  call *sys_call_table(,%rax,8)  # XXX:    rip relative
...
```

Listing 3.9: Accounting the total amount of issued System Calls per-task, C-function

```
asmlinkage void tcg_nsyscalls (void){
    current->nsyscalls += ( 1 % ULONG_MAX );
}
```

Listing 3.10: The `TCG_NSYSCALLS` macro's definition

```
#define TCG_NSYSCALLS   tcg_nsyscalls
```

3.4.4 Accounting the total amount of calls to the `sched_yield()` system call

The system call `sched_yield()` is implemented in the `kernel/sched.c` source file. It is the only way a user-space program can explicitly ask for relinquishing the processor without waiting for a particular hardware to be ready or for an event to occur. That means this is still Voluntary Context Switching, but not due to **voluntary blocking**, and it is always issued whilst the process is running in user-mode. Our project does account any call to this particular system call and stores them in the `nvcsw_ext[2]` field, as summarized in Table 3.3. Listing 3.11 shows the `sched_yield()`'s implementation. It is quite obvious any call to this function ends up in an actual call to the GNU/Linux Scheduler - line 10 -. Our GNU/Linux Kernel patch adds some code to the `system_call` entry in order to determine whether the issued system call is `__NR_sched_yield` by means of comparing the value stored in the `rax` register or not. If the value of `rax` is equal to the one shown in Listing 3.12, that is, `__NR_sched_yield`, then we save `rax`'s value before calling our high-level C-routine `TCG_SCHED_YIELD` counter. Finally, we restore its value and the entry point continues normally - lines 4-8 - . As it is clearly shown in Listing 3.13, this call to `sched_yield` is also accounted calling our `TCG_NSYSCALLS` routine, so that we can have an accurate accounted number of issued system calls, including `sched_yield()` - lines 9-13 -. The `tcg_sched_yield()` C-routine is shown in Listing 3.15.

`TCG_SCHED_YIELD` has been defined as a macro in the `include/linux/tcg.h` header file, as shown in Listing 3.14.

Listing 3.11: The `sched_yield` system call's implementation, `kernel/sched.c`

```
1   SYSCALL_DEFINE0(sched_yield)
2   {
3       struct rq *rq = this_rq_lock();
4       schedstat_inc(rq, yld_count);
5       current->sched_class->yield_task(rq);
6       __release(rq->lock);
7       spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
8       _raw_spin_unlock(&rq->lock);
9       preempt_enable_no_resched();
10      schedule();
11      return 0;
12  }
```

Listing 3.12: The `sched_yield` system call's number, `arch/x86/include/asm/unistd_64.h`

```
#define __NR_sched_yield            24
__SYSCALL(__NR_sched_yield, sys_sched_yield)
```

Listing 3.13: Accounting the total amount of calls to the `sched_yield()` system call

```
system_call_fastpath:
        cmpq $__NR_syscall_max,%rax
        ja badsys
        cmp $__NR_sched_yield,%rax
        jnz tcg_normal_account
        push %rax
        call TCG_SCHED_YIELD
        pop %rax
tcg_normal_account:
        movq %r10,%rcx
        push %rax
        call TCG_NSYSCALLS
        pop %rax
        call *sys_call_table(,%rax,8)  # XXX:    rip relative
```

Listing 3.14: The `TCG_SCHED_YIELD` macro's definition

```
#define TCG_SCHED_YIELD    tcg_sched_yield
```

Listing 3.15: The `tcg_sched_yield()` high-level routine

```
asmlinkage void tcg_sched_yield (void){
        current->nvcsw_ext[2]+=(1%ULONG_MAX);
}
```

3.4.5 Accounting the total amount of Context Switches whilst returning from a System Call

As stated in section 3.2, the GNU/Linux Kernel can preempt a process whilst returning from a system call. Our approach is slightly different than the one taken by the GNU/Linux Kernel. Instead of considering any call made to the GNU/Linux scheduler at `ret_from_sys_call` as Involuntary Context Switching, we do assume it is Voluntary Context Switching because it is the own task that has issued the system call the one in charge of checking its own `TIF_NEED_RESCHED` flag, as shown in Listing 3.16, lines 9-11. If there are not any flags in the `flags` field for the `current` task already set, that is, if line 10 evaluates to 0, then the task can be resumed in user-space and continue its execution normally. Otherwise, a jump to the `sysret_careful` label is made. Our high-level C-counter is placed right after checking for pending signals to be attended, lines 14-15. If there are some, a jump to the `sysret_signal` label is made. If there aren't any, our high-level counter `TCG_NVCSW_0` will be called, lines 23-25. The actual call to the GNU/Linux Scheduler is made in line 26. Our high-level C-routine counter is shown in Listing 3.17. `TCG_NVCSW_0` has been defined as a macro in the `include/linux/tcg.h` header file, as shown in Listing 3.18.

Listing 3.16: Calling the `tcg_nvcsw_0()` high-level function

```
1  ret_from_sys_call:
2      movl $_TIF_ALLWORK_MASK,%edi
3      /* edi: flagmask */
4  sysret_check:
5          LOCKDEP_SYS_EXIT
```

```
        GET_THREAD_INFO(%rcx)
        DISABLE_INTERRUPTS(CLBR_NONE)
        TRACE_IRQS_OFF
        movl  TI_flags(%rcx),%edx
        andl  %edi,%edx
        jnz   sysret_careful
    ...
    sysret_careful:
        bt  $TIF_NEED_RESCHED,%edx
        jnc  sysret_signal
        TRACE_IRQS_ON
        ENABLE_INTERRUPTS(CLBR_NONE)
        pushq %rdi
        CFI_ADJUST_CFA_OFFSET  8
        push %rax
        call  TCG_NSYSCALLS_SCHEDULE
        pop %rax
        push %rax
        call  TCG_NVCSW_0
        pop %rax
        call  schedule
    ...
```

Listing 3.17: accounting the total calls to the scheduler whilst returning from a System Call

```
    asmlinkage void tcg_nvcsw_0  (void){
        current->nvcsw_ext[0]  += ( 1 % ULONG_MAX  );
    }
```

Listing 3.18: The `TCG_NVCSW_0` macro's definition

```
    #define TCG_NVCSW_0    tcg_nvcsw_0
```

Building the per-syscall table

As shown in Table 3.3, our patch can also account the total amount of calls to `schedule()` issued per System Call, thanks to the `nsyscalls_schedule[]` counter. This counter is just an immediate index matching any valid system call to its accounting information. This matching is implemented as follows: let s_i be a particular System Call, then it is quite obvious that `nsyscalls_schedule[`s_i`]` will contain the total amount of calls to the scheduler whilst returning to user-space from s_i's process context. According to Listing 3.8, line 9, we store the system call id in the high-level C-variable `syscallid`, so that it can be used later on. This variable is defined in the `arch/x86/kernel/tcg.c` source file. When a probable call to the GNU/Linux scheduler is about to happen, as is shown in Listing 3.16, our high-level C-routine `tcg_nsyscalls_schedule()` is called, lines 20-22. This routine is shown in Listing 3.19 and it is truly simplistic: it does use the previously stored `syscallid` variable as an index inside the `nsyscalls_schedule[]` array in order to increment the right slot by 1.

Listing 3.19: Constructing the per-syscall calls to the GNU/Linux Scheduler table

```
    ...
    unsigned int syscallid;
    ...
    asmlinkage void tcg_nsyscalls_schedule  (void) {
        if(syscallid<=__NR_syscall_max)
            current->nsyscalls_schedule[syscallid]+=(1%ULONG_MAX);
        else
```

```
                printk(KERN_ERR "TCG: %d, syscallid has been thrashed: %d\n",
                 current->pid, syscallid);
    }
```

3.4.6 Accounting involuntary preemption due to interrupts

When a hardware interrupt is triggered, the processor stops whatever code could be executing and jumps to a predefined memory address where the entry point for handling interrupts is located. This particular entry point is set by the GNU/Linux Kernel, and it is in charge of saving the registers' values and the interrupt line number just before calling `do_IRQ()`. Our main concern is to deal directly with the `do_IRQ()` function. As previously introduced, the GNU/Linux Kernel is running in **Interrupt Context**, and therefore any piece of code is not allowed to sleep. Any possible operation must be performed atomically. Thus, we cannot increase a variable which is not atomic. We do know that any single call to `do_IRQ()` does interrupt the task that was previously running. Then it immediately follows that the `current` macro does point at the previous running task. So, it is safe to suppose that we can increase its counter by one, as long as we do that atomically. That is why we have implemented this counter as an `atomic64_t`. This is shown in Listing 3.20, lines 13-14.

Listing 3.20: Accounting preemption due to interrupts atomically

```
1   unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
2   {
3       struct pt_regs *old_regs = set_irq_regs(regs);
4
5       unsigned vector = ~regs->orig_ax;
6       unsigned irq;
7
8       exit_idle();
9       irq_enter();
10
11      irq = __0get_cpu_var(vector_irq)[vector];
12
13      atomic64_set(&current->nivcsw_ext[1],
14          (atomic64_read(&current->nivcsw_ext[1])+1)%ULONG_MAX);
15    ...
16  }
```

3.4.7 Accounting involuntary preemption due to `try_to_wake_up()`

This project focus explicitly on the **CFS** GNU/Linux scheduler algorithm. Thus, when a process that has been sleeping in a wait queue does awake, the CFS can determine that it does have a higher priority than the one that is running. This ends up in an involuntary context switch because the current task is forced to yield the processor and the recently awakened one is put to run instead. Not all the awakened tasks are suitable for running immediately because of their priority. In these cases, they are inserted in the **red-black tree** for the active run queue, if feasible, so that they could be executed in a near future. The CFS algorithm is in charge of doing precisely so, and all the code can be found in the C-source file `kernel/sched_fair.c`. The function dealing with tasks' priorities is called `check_preempt_wakeup()`. This function does call `resched_task()` whenever necessary, thus setting the `TIF_NEED_RESCHED` flag for the `current` macro. Our extended counter is incremented right here, as long as a call to `resched_task()` is made. For example, let's consider that the just recently awakened task p is real-time. According to the Listing 3.21, this new task is meant to be executed always, thus preempting the `current` one.

Therefore, we can increase our extended counter here, as clearly shown in lines 3-4. In consonance with the `check_preempt_wakeup` function, there are other parts in its code where the awakened task can provoke a re-schedule of the current one. Our counter is incremented there too, accordingly. Because a re-schedule can occur concurrently, we have to implement our extended counter as an `atomic64_t`.

Listing 3.21: Accounting re-schedules due to awakened tasks having a higher priority

```
1       ...
2       if (unlikely(rt_prio(p->prio))) {
3             atomic64_set(&curr->nivcsw_ext[0],
4                   (atomic64_read(&curr->nivcsw_ext[0])+1)%ULONG_MAX);
5             resched_task(curr);
6             return;
7       }
8           ...
```

3.4.8 Avoiding overflows

Our new counters are defined as `unsigned long` numeric values. In two cases these counters have been defined as **atomic64_t** in order to increase their values atomically. This applies to our Involuntary Context Switching counters, that is, `nivcsw_ext[0]` and `nivcsw_ext[1]`. That is so because both fields are read and altered in functions that can certainly run concurrently: `do_IRQ()` and `check_preempt_wakeup()`. Whatever the case, incrementing a counter in kernel-space is tricky and dangerous, as long as there is no mechanism to detect and avoid overflows. An overflow in kernel-space can trigger a Kernel OOPS or even a Kernel PANIC. Stability is mandatory; thus, we have to keep track of overflows-to-be and avoid them. As defined in the `include/linux/kernel.h` header file, we do know how huge an `unsigned long` numeric value could be, that is, `ULONG_MAX`. The exact value depends on the processor's architecture. Therefore, the way we increment any single counter is by applying modular operations:

$$counter \longleftarrow ((counter + 1) \mod \texttt{ULONG\_MAX}) \tag{3.3}$$

This is shown in Listings 3.9, 3.15, 3.17 , 3.19, 3.20 and 3.21 respectively.

3.4.9 Extending the `/proc/`*PID*`/sched` file

Whenever the `CONFIG_SCHED_DEBUG` kernel option is set, the GNU/Linux Kernel exports some per-task statistical information on `/proc/`*PID*`/sched`. Our kernel-patch adds the `nsyscalls`, `nvcsw_ext[0]`, `nvcsw_ext[2]`, **nivcsw_ext[0]** and `nivcsw_ext[1]` counters to these stats, so that they can be easily read by issuing a simple `cat` command, as described in section 3.5.4.

As shown in Listing 3.22, we have added some lines in order to export our counters' values whenever a read over `/proc/`*PID*`/sched` is required. As analyzed in [19], the GNU/Linux Kernel performs these operations in the `linux/kernel/sched_debug.c` source file, in the `proc_sched_show_task()` function. According to Listing 3.22, our counters are added in lines 4-7 and 10-21.

Listing 3.22: Adding the `nsyscalls`, `nvcsw_ext[0]` and `nvcsw_ext[2]` counters to the Debug Sched interface

```
1        __P(nr_switches);
2        SEQ_printf(m, "%-35s:%21Ld\n",
3              "nr_voluntary_switches", (long long)p->nvcsw);
4        SEQ_printf(m, "%-35s:%21Ld\n",
            "nr_vol_switches_due_to_syscalls", (long long)p->nvcsw_ext[0]);
        SEQ_printf(m, "%-35s:%21Ld\n",
             "nr_vol_switches_sched_yield", (long long)p->nvcsw_ext[2]);
        SEQ_printf(m, "%-35s:%21Ld\n",
             "nr_involuntary_switches", (long long)p->nivcsw);
        SEQ_printf(m, "%-35s:%21Ld\n",
             "nr_involuntary_sws_try_to_wakeup",
          (long long)atomic64_read(&p->nivcsw_ext [0]));
        SEQ_printf(m, "%-35s:%21Ld\n",
             "nr_involuntary_sws_do_IRQ",
          (long long)atomic64_read(&p->nivcsw_ext [1]));
           "nr_issued_system_calls", (long long)p->nsyscalls);
        for(i=0;i<=__NR_syscall_max;
          seq_printf(m, "      TCG.table.%-21d:%21Ld\n",
                       i, (long long)p->nsyscalls_schedule[i]),
          ++i
        );
```

The GNU/Linux Scheduler Debugging information does have another function, that is, `proc_sched_set_task()`, to initialize all the values for most of the `task_struct` fields related to the scheduling mechanism. Whenever the `CONFIG_SCHEDSTATS` kernel option is set, as previously discussed, this function sets all the fields to initial values of 0. Thus, our patch adds three trivial calls to the `memset()` function so that our counters could be initialized properly, as shown in Listing 3.23.

Listing 3.23: Initializing our counters in order to extend the `Kernel Scheduler Debugging Information` accordingly

```
        p->nsyscalls             = 0;
        memset(&p->nvcsw_ext, 0, sizeof(p->nvcsw_ext));
        atomic64_set(&p->nivcsw_ext[0],0);
        atomic64_set(&p->nivcsw_ext[1],0);
        p->nivcsw_ext_exit = 0;
        memset(&p->nsyscalls_schedule, 0, sizeof(p->nsyscalls_schedule));
```

3.4.10 Determining whenever a process ends its own execution voluntarily

Let p be a process that is ending its execution. The GNU/Linux Kernel facilities allow us to determine whether this exit is caused by a normal operation or not. The exit status code for a given task is always passed to the `do_exit()` function during p's termination as a parameter. The GNU/Linux Kernel adds an error value to this integer parameter whenever an unusual event has forced the task to end its execution. That uncommon event could be a signal sent by another process, a segmentation fault or whatever makes p exit abnormally. In fact, whenever a process is due to cease its own existence, there is always a signal involved. Thus, we can determine whether p has just been killed because of a signal by checking if the `__WIFSIGNALED` flag has been set to the `code` parameter. This is shown in Listing 3.24, line 692. In line with `include/bits/waitstatus.h`, `__WIFSIGNALED` is a macro defined as shown in Listing 3.25. Therefore, whenever `code & 0x7` resolves to 0, p has ended its own execution in an ordered, thus voluntary manner, and so the `nvcsw_ext[1]` is set to 1.

Listing 3.24: Accounting p's exit event; either voluntary or not

```
        if (unlikely(!tsk->pid))
             panic("Attempted to kill the idle task!");
        if((code & 0x7f) == 0)
            tsk->nvcsw_ext[1]=1;
```

```
    else
        tsk->=nivcsw_ext_exit=1;
```

Listing 3.25: Accounting *p*'s exit event; `__WIFSIGNALED`'s definition

```
#define __WIFSIGNALED(status) \
    (((signed char) (((status) & 0x7f) + 1) >> 1) > 0)
```

3.4.11 Altering the GNU/Linux Taskstats interface

Our project uses the GNU/Linux Taskstats interface in order to gather these new counters from user-space applications. Our main user-space tool is `getcswc`, a program written in C derived from `getdelays.c`, as widely discussed in [22]. To allow this program to access these new counters, the function `fill_pid()` in `kernel/taskstats.c` has been updated accordingly, as shown in Listing 3.26. It is truly simplistic: we simply copy all the counters from the `task_struct` data structure into the `taskstats` one for the desired task by means of calling the `memcpy()` function.

Listing 3.26: Filling *p*'s stats accordingly in the `fill_pid()` function

```
...
stats->nsyscalls = tsk->nsyscalls;

memcpy(&stats->nvcsw_ext,&tsk->nvcsw_ext, sizeof(tsk->nvcsw_ext));
stats->nivcsw_ext[0] = atomic64_read(&tsk->nivcsw_ext[0]);
stats->nivcsw_ext[1] = atomic64_read(&tsk->nivcsw_ext[1]);
stats->nivcsw_ext_exit = tsk->nivcsw_ext_exit;
...
```

3.4.12 Side-effects of altering the `task_struct` and `taskstats` data structures

This is an experimental approach. That means this sort of altered GNU/Linux Kernel mainstream is not meant to be used as a regular one. This entire project tries to show the way the GNU/Linux scheduler works. As previously described, we do not care about performance at all. Thus, altering certain kernel data structures immediately provokes changing the way the whole system works plus injecting some setbacks. One obvious setback is the total amount of memory the kernel needs to allocate when creating a new process, because the `task_struct` data structure now is larger. Listing 3.27 shows a simple call to the `printk()` function, placed in the `init/main.c` source file, so that a precise new size for both structures can be determined.

Listing 3.27: Showing both modified structures' new size

```
static int __init kernel_init(void * unused)
{
 ...
 printk (KERN_INFO "%s\nTask_struct size: %li, taskstats size: %li\n",
       __TCG__MESSAGE ,
       sizeof(struct task_struct),
       sizeof(struct taskstats)
 );
 ...
}
```

Table 3.4 summarizes these sizes for a patched and a non-patched GNU/Linux Kernel, respectively. It is obvious that both GNU/Linux Kernels does have the same configuration file. That is extremely important, due to the fact some additional per-structure fields are dynamically compiled depending on the number of previously set options.

	non-patched	*patched*	Δ
`task_struct`	1808	4248	2432
`taskstats`	336	384	48

Table 3.4: GNU/Linux Kernel data structures sizes comparison, in bytes

`task_struct`'s size does affect allocated buffers everywhere in the GNU/Linux Kernel sources, obviously. Let's consider, for example, the call to `fill_tgid()` in the TaskStats interface. In order to be able to get a valid netlink message when a certain process p ends its execution, we need to instruct Taskstats not to allocate memory for gathering per-tgid stats. Otherwise, we would get an `ENOBUFF` errno code by running our `getcsw` program with flags `-m` *`CPUMASK`*.

Whenever our routines are executed, they can possibly affect p's time-slice. Thus, p could relinquish the processor whilst executing our high-level C-routines, due to the fact that they do not hold any lock. That means $p \rightarrow (preempt\_count() \& PRREMPT\_ACTIVE)$ would resolve to 1. As previously stated, the GNU/Linux Kernel is preemptive, so every single process can be preempted even when it is running in kernel mode. That includes our routines. However, this situation can not affect our Voluntary Context Switching counters. If any, it would be our Involuntary Context Switching counters the ones being affected. Even if p has been preempted due to time-slice exhaustion whilst executing any of our high-level C-counters routines, *before that* a certain call to the GNU/Linux Kernel Scheduler would have been performed. Therefore, this call to `schedule()` is still to be accounted.

3.5 Deploying & Using our Context Switching Accounting tool

Our project is kernel-related. Thus, all the code involved in extending the `nvcsw` and `nivcsw` counters is meant to be mainly found in the GNU/Linux Kernel source mainstream. Thus, the best way to deploy all the code-changes inside the GNU/Linux kernel is by writing a **patch**-file. This patch could be then easily applied later on by running the `patch` command over an unaltered GNU/Linux kernel source tree.

3.5.1 Creating the patch-file

After modifying the GNU/Linux Kernel sources, and assuming that `/usr/src/linux` is our altered tree, to create our patch-file we ran the `diff` utility, as shown in Listing 3.28.

Listing 3.28: Creating the patch-file using the `diff` command

```
# cd /usr/src
# diff -urN linux-source-2.6.32/ linux/ > patch_file
```

3.5.2 Applying the patch

The patch has to be applied to an unaltered GNU/Linux Kernel source tree in order to get access to our counters. As introduced in [21], our changes can only be applied to a 2.6.32.5 GNU/Linux Kernel version. More specifically, this patch-file will only work for a 2.6.32 41 GNU/Linux Kernel version, see `linux-source-2.6.32/version.Debian`.

The way to patch a GNU/Linux Kernel source tree is well-documented. See listing 3.5.3 for details.

Listing 3.29: Applying the patch-file and installing the new kernel and modules

```
# cd /usr/src/linux-2.6.32/
# cp /boot/config-2.6.32-5-amd64 .config
# make oldconfig
# patch -p1 < ../patch_file
# make -jn && make -jn modules
# make modules_install
# cp arch/x86_64/boot/bzImage /boot/vmlinuz-2.6.32
# cp System.map /boot/System.map-2.6.32
# cp .config /boot/config-2.6.32
# mkinitramfs -o /boot/initrd.img-2.6.32 2.6.32
# update-grub
```

After rebooting the computer and loading into the new kernel, access to these counters is guaranteed.

3.5.3 Compiling and using `getcsw.c`

`getcsw.c` is the user-space tool in charge of communicating with the GNU/Linux Kernel via **NetLink** sockets, taking advantage of the GNU/Linux TaskStats ABI. To compile it, just run:

```
$ gcc -I/usr/src/linux-source-2.6.32/include getcsw.c -o getcsw
```

This command can be used to:

- Gather statistical information about a running process.
- Gather statistical information about a process that is ending its execution.

As previously discussed, whenever a process exits, we need a way to obtain its statistics and, more importantly, to be able to read its `nvcsw_ext[1]` field. The GNU/Linux Kernel TaskStats interface allows us to do precisely so. Thus, `getcsw` can be instructed to listen to a particular process's exit. Table 3.5 summarizes all the flags and options that can be applied to the `getcsw` program.

Flag	*Description*	*Sample*
`-p` *PID*	Mandatory. Gets process *PID*'s stats.	`./getcsw -p 2345`
`-i`	Shows Involuntary Context Switches extended counters.	`./getcsw -p 2345 -i`
`-l`	Gets process *PID*'s stats at infinite intervals of 1s.	`./getcsw -p 1 -l`
`-d` *t*	Alters the default `-l`'s delay of 1s to *t* seconds.	`./getcsw -p 45 -l -d 10`
`-m` *mask*	Sets which cpu(s) we are listening to finishing tasks.	`./getcsw -p 1 -m "0,3"`
`-v`	Enables verbosity.	`./getcsw -p 2345 -l -v`

Table 3.5: `getcsw`'s program flags

3.5.4 Execution examples

This section shows some trivial examples of data output obtained by running the `getcsw` command on a patched GNU/Linux Kernel. In order to run the command, `root` privileges are required. It also presents another way of gathering such data without needing to become `root`: by reading the file `/proc/PID/sched`, that has been updated with our new per-task counters, as described in section 3.4.9.

Having a peak at the process's counters

At any time, `getcsw` can be executed to have a quick look at any desired process's counters' values. In this example, we ran `getcsw` in order to determine the total amount of calls to the `schedule()` function during `ret_from_sys_call` for the `scp` command, whilst copying 1GB of data over the Internet. This is shown in Listing 3.30.

Listing 3.30: Obtaining the current `scp`'s counter values during a file-copy over the Internet

```
./getcsw -p `ps -C scp|tail -1|cut -d" " -f2`
Task  voluntary  nonvoluntary   syscalls  ret.syscalls  vol.sched  vol.exit
2880       1035            32       9266            19       1016         -
```

Reading the Involuntary Context Switches extended counters

By default, `getcsw` does not show the extended Involuntary Context Switches counters. In order to do so, `getcsw` has to be executed with the `-i` flag. This way, it is feasible to read how many times a particular task has been preempted due to hardware interrupts or high-priority tasks recently awakened. It is of the utmost importance to bear in mind that, whenever the second case does apply, we have to consider these figures as **possible preemption due to re-schedule calls**, for not all the calls to the GNU/Linux Scheduler, that is, `schedule()`, end up in an actual context switch, as previously demonstrated. This is shown in Listing 3.31.

Listing 3.31: `init`'s involuntary switches due to IRQs and awakened tasks after 4 days.

```
./getcsw  -p 1 -i
Task         voluntary    nonvoluntary        wake_up          IRQS
  1              71586             142            139            64
```

Reading the process's counters at infinite intervals of time t =10 seconds.

Whenever a constant tracing of the counters' values is required for a given process, `getcsw` can be executed with flags `-l` and `-d` t, as shown in Table 3.5. This time, we traced `ping`'s execution at intervals of t =10 seconds. The results are shown in Listing 3.32.

Listing 3.32: Tracing `ping`'s counters at intervals of t =10 seconds

```
./getcsw -p `ps -C ping|tail -1|cut -d" " -f2` -l -d 10
Task  voluntary  nonvoluntary   syscalls  ret.syscalls  vol.sched  vol.exit
1417         42             3        473             2         40         -
1417        191             5       1667             4        187         -
1417        281             5       2221             4        277         -
...
```

Waiting for a task to end

Our project does account whenever a process has been killed or has ended its execution in an ordered manner by calling the `exit()` function on its own or whilst returning from the `main()` function. This counter can be read neither using the `getcsw` command in the usual way nor by reading the `/proc/`*PID*`/sched` file because the process is ending its execution. To solve this problem, `getcsw` can wait for a particular task to end. As we know, a task can be executed on any processor available. Thus, we can set the `-m` flag in order to listen to any given process to terminate that is running on a particular cpu. In this example, we ran the `getcsw` command in order to wait for the `scp` command to complete the file-copy process on any processor available. Because our computer had four cores, we used the mask: `"0-3"`, as shown in Listing 3.33.

Listing 3.33: Waiting for the `scp`'s command to end

```
./getcsw -p ‘ps -C ping|tail -1|cut -d" " -f2‘ -m "0-3"
Task   voluntary   nonvoluntary    syscalls   ret.syscalls   vol.sched   vol.exit
3315          96              2         901              1          95          y
```

Obtaining process's stats by reading /proc/*PID*/sched.

We can obtain the process's counters at any time without executing the `getcsw` command. Our GNU/Linux Kernel patch does add these counters to the scheduler debugging information, exported by the kernel through the `/proc` directory facility as long as the `CONFIG_SCHED_DEBUG` option is set. In this example, we used a simple `cat` command to get the total amount of Epiphany Internet Browser's issued system calls, as clearly shown in Listing 3.34.

Listing 3.34: Getting the total amount of issued system calls made by Epiphany Web Browser

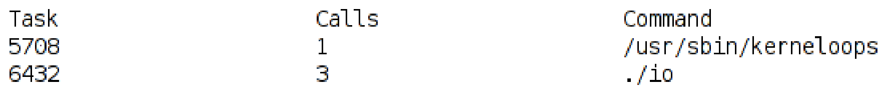
```
cat /proc/‘ps -C epiphany-browser|tail -1|cut -d" " -f2‘/sched|grep issued
nr_issued_system_calls                 :                 84799
```

Looking for tasks calling `sched_yield`

We can write a trivial shell-script in order to look for tasks issuing `sched_yield()` system calls. All we need is to read /proc/*PID*/sched file and get the `nvcsw_ext[2]` counter that has been exported using our patch, as described in section 3.4.9. A sample script file is shown in Listing 3.35, and its output is shown in figure 3.2.

Listing 3.35: Looking for tasks executing `sched_yield()`, sample bash-script

```bash
#!/bin/bash
echo -ne "Task  \t  Calls   \t  Command \n"

for p in /proc/*;
do
 # TCG: take care not to look at our own pid!
    if  [ -d $p -a "$p" != "/proc/$$" ]; then
        if [ -r $p/sched ]; then
            yield=‘cat $p/sched |grep yield|cut -d":" -f2|tr -d ' ' 2>/dev/null‘
            # We do have some sched yields, show them:
            if [ ! -z "$yield" -a "$yield" != "0" ]; then
                echo -ne "‘basename $p‘ \t  $yield  \t  ‘cat $p/cmdline‘    \n"
            fi
        fi
    fi
done
```



```
Task                    Calls                   Command
5708                    1                       /usr/sbin/kerneloops
6432                    3                       ./io
```

Figure 3.2: Two programs have been detected calling the `sched_yield()` system call

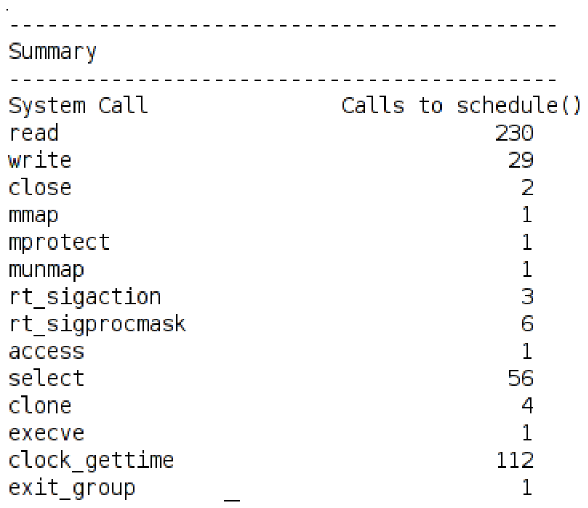
Summarizing per-System Call calls to the GNU/Linux Kernel Scheduler

This is another example of our kernel patch possibilities. A trivial bash-script has been developed in order to summarize every single call to the GNU/Linux Scheduler whilst returning to user-space mode per system call. The entire script is shown in Listing 3.36. Its execution is shown in figure 3.3.

Listing 3.36: Running `fcalltable.sh` to summarize `wget` calls to `schedule()` per syscall

```
#!/bin/bash
#
# fcalltable.sh
#
#  Shows the per-syscall calls to the schedule() function
#  getting its data from the /proc/PID/sched file. In
#  order to get the system calls' names, it does need to
#  read arch/x86/ / header file.
#
# 2012 by Toni Castillo Girona
# <toni.castillo@fa.upc.edu>
#
#
#set -x

LINUXSOURCE="/usr/src/linux-source-2.6.32"      # Kernel source tree
SYSCALLNAMES="arch/x86/include/asm/unistd_64.h"     # Syscall defines, C-header file
NSYSCALL_MAX=299              # nsyscall_max + 1

#-----------------------------------------------------------------------------
# get_syscall_name
#  Return the name for a particular system call id.
#-----------------------------------------------------------------------------
get_syscall_name () {
 if [ -r $LINUXSOURCE/$SYSCALLNAMES ]; then
  # Get all the syscall names, ordered by index, so the first one is index=0,
  # the next one index=1, and so on
  index=`expr $1 + 1`   # first 0 + 1 = 1 ...
  sname=`cat $LINUXSOURCE/$SYSCALLNAMES|grep -E "^#define __NR_"|tr  \
          '\t' ' '|cut -d" " -f2|head -$index|tail -1`
  echo "${sname:5}"
 else
  echo ""  # Empty string
  return 1 # error
 fi
}

#
# We need the pid to get the call table for
#
if [ $# -lt 1 ]; then
 echo "Usage: `basename $0` PID [-s]\n"
 exit 1
fi

# Be careful, the pid must exist:
if [ -r /proc/$1/sched ]; then

 clear
 DATA=""

 # print task's pid and name:
 echo -ne "Task: $1\nCommand: `cat /proc/$1/cmdline`\nTotal calls: \
    `cat /proc/$1/sched|grep nr_vol_switches_due_to_syscalls |tr -d ' ' \
   |cut -d":" -f2`\n"
 echo ""

 #Header:
 test "$2" != "-s" &&  printf "%-25s %15s \n" "System Call" "Calls to schedule()"

 # Get table's values:
 table=`cat /proc/$1/sched|grep -E "TCG.table."|tr -d ' '| \
   cut -d":" -f2|tr '\n' ' '`
 itbl=0
 for row in $table;
 do
  test "$2" != "-s" && printf '%-25s %15d \n'  "`get_syscall_name $itbl`" "$row"
  # Get rows with >0
  if [ $row -ge 1 ]; then
   rs=`printf '%-25s %15d \n' "\`get_syscall_name $itbl\`" "$row"`
```

```
    DATA="$DATA\n$rs"
   fi
   itbl=`expr $itbl + 1`
  done
  # Print the summary:
  if [ ! -z "$DATA" ]; then
   echo "--------------------------------------------"
   echo "Summary"
   echo "--------------------------------------------"
   printf "%-25s %15s" "System Call" "Calls to schedule()"
   echo -ne "$DATA\n"
  fi
  # Print the calls actually issuing calls to the schedule() function:
else
  echo "Error: /proc/$1/sched does not exist."
  exit 1
fi
```



```
-------------------------------------------
Summary
-------------------------------------------
System Call                Calls to schedule()
read                                   230
write                                   29
close                                    2
mmap                                     1
mprotect                                 1
munmap                                   1
rt_sigaction                             3
rt_sigprocmask                           6
access                                   1
select                                  56
clone                                    4
execve                                   1
clock_gettime                          112
exit_group                               1
```

Figure 3.3: Running our `fcalltable.sh` script to summarize per-call calls to `schedule`

Chapter 4

Generating Hardware Interrupts at will

4.1 Introduction

Hardware Interrupts happen any time during the lifetime of a system. As we have described earlier, a certain process that is running can be preempted as soon as a hardware interrupt is raised. Then, as previously introduced, a call to the `do_IRQ()` function is made to deal with that particular interrupt request. Some of those interrupts can be masked so that they will be ignored. Our main concern is to determine whenever a certain running task, say *p*, yields the processor due to a hardware interrupt request. Sometimes these interrupts will be caused by normal hardware operations: the hard disk has the data it was previously asked for; a network packet has just come in; etcetera. However, not all the interrupt requests will follow this pattern: sometimes a **Non Maskable Interrupt** or **NMI** will be issued, generally pinpointing at hardware errors, such as memory parity error or an unknown hardware malfunction. This chapter describes how the GNU/Linux Kernel deals with Hardware Interrupts, focusing principally on the way we can force **NMI** interrupts to simulate hardware errors. This technique will allow us to gather some Involuntary Context Switching statistics.

4.2 The GNU/Linux Kernel `IDT`

In order to process interrupts, the GNU/Linux kernel keeps a table called **The Interrupt Descriptor Table**, IDT. This table maps an IRQ line to its handler. In our particular case, we need to focus on Hardware Interrupts. A hardware interrupt is handled by the kernel as long as there is an IRQ handler in charge of doing so. As previously stated, a call to the `do_IRQ()` function is always performed whenever an IRQ is requested. This function adopts that role of a hub; it calls the right IRQ handler according to the **IDT** table so that the last interrupt can be handled properly. It seems fairly coherent that only a certain piece of hardware, say an ECC memory module, can issue an interrupt. And, more specifically, that only a certain hardware that has a malfunction can raise a **NMI** interrupt. During the GNU/Linux Kernel start up process, the **IDT** is initialized right before enabling interrupts. In the **X86_64** architecture, that is left to the `native_init_IRQ()` function, as shown in Listing 4.1.

Listing 4.1: Initializing the The GNU/Linux `IDT`

```
for (i = FIRST_EXTERNAL_VECTOR; i < NR_VECTORS; i++) {
        /* IA32_SYSCALL_VECTOR could be used in trap_init already. */
        if (!test_bit(i, used_vectors))
4             set_intr_gate(i, interrupt[i-FIRST_EXTERNAL_VECTOR]);
5     }
```

`set_intr_gate()` inserts an Interrupt Gate at the nth **IDT** position. Every single hardware interrupt is handled by the GNU/Linux Kernel using Interrupt Gates. They are not accessible from user-land processes.

4.3 Allowing user-land processes to send Hardware Interrupts

Our project needs to send Hardware Interrupts. That is so because we don't want to provoke real hardware failures in order to prevent the computer from being damaged. Moreover, stability is mandatory: we cannot gather statistics if the system crashes. As previously introduced in section 4.2, a user-land process cannot issue interrupts. The only exception is `INT 0x80`, the old way of issuing a system call. The **IDT** at the 128th position is handled by a special handler, and it is considered as a **Software Interrupt**. Because we do know that every single interrupt is going to call **do_IRQ()** and preempting the current process on the same processor handling that very same interrupt in doing so, we have to find a suitable way to force them. Then, the gathering of statistical information concerning Involuntary Context Switching due to IRQs would be easy to reproduce.

Every single Interrupt Gate has a special field called `DPL`, that is two bit long. This field defines the priority level for that particular Interrupt Gate. User-land processes run in **Ring 3** mode, whereas Kernel-space processes run in **Ring 0** mode. Because Hardware Interrupts cannot be issued from user-land, the GNU/Linux Kernel initializes all the Interrupt Gates in the **IDT** to $DPL = 0$, but the $128th$ entry, which DPL is 3. The **Current Privilege Level** for that particular process is compare to the **DPL**, according to Equation 4.1, thus granting or denying access to that particular interrupt. As shown in Listing 4.1, the function setting the `DPL` for a particular Interrupt Gate is `set_intr_gate()`, line 4. This function, in turn, call `_set_gate()`, where the DPL for that given Interrupt Gate is actually set, as shown in Listing 4.2, fourth parameter. Thus, a small change is required in order to allow user-land processes to send Hardware Interrupts. In our particular case, we need to simulate hardware errors; that involves **NMI** interrupts, that are located at **IDT[2]**. So, the call to `pack_gate()` has been modified slightly, as shown in Listing 4.3, to accomplish that.

$$CPL \geq DPL\ ? \longrightarrow OK : SEGFAULT \tag{4.1}$$

Listing 4.2: Setting the *DPL* field for every single Interrupt Gate

```
static inline void set_intr_gate(unsigned int n, void *addr)
{
    BUG_ON((unsigned)n > 0xFF);
    _set_gate(n, GATE_INTERRUPT, addr, 0, 0, __KERNEL_CS);
}
```

Listing 4.3: Allowing user-land processes to issue **NMI** interrupts

```
static inline void _set_gate(int gate, unsigned type, void *addr,
                unsigned dpl, unsigned ist, unsigned seg)
{
    gate_desc s;
    pack_gate(&s, type, (unsigned long)addr, (gate==2)?3:dpl, ist, seg);
    write_idt_entry(idt_table, gate, &s);
}
```

4.3.1 Generating hardware errors at will

A trivial C-code has been written in order to issue an **INT 0x2** interrupt request. Because this IRQ line is the one associated with **NMI** interrupts, the GNU/Linux Kernel will process every single call to **INT 0x2**, thus preempting the current process in doing so. This interrupt can be balanced among all the available cores on the system, thus at any given time t_i, a certain process p_i could relinquish processor c_i, as long as p_i was originally running on c_i at time t_i. This user-land program is shown in Listing 4.4.

Listing 4.4: Allowing user-land processes to issue **NMI** interrupts

```c
/*
 int_nmi.c
 2012 by Toni Castillo Girona
 <toni.castillo@fa.upc.edu>

 This trivial C-program issues a INT $0x2 interrupt, that is,
 it does simulate there is a NMI interrupt requesting OS attention.
 NMI stands for "Non Maskable Interrupts", and they are used to
 take care of hardware-related problems, like memory parity errors and
 the sort.

 Our main purpose is to provoke a call to do_IRQ() and thus making
 current to yield the processor (as long as p is running on the SAME
 processor taking care of the unknwon NMI interrupt).

 NOTE: This interrupt cannot be issued from user-space unless the
 IDT has been previously altered so its DPL=3.

 USAGE: ./int_nmi -n NUMBER

 If NUMBER = -1, it loops forever sending NMI interrupts.

*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void _send_nmi (void);

int main (int argc, char **argv){

 int c, number, i;

 /* Get the argument from the cli: */
 c = getopt(argc, argv,"n:");
 if(c<0||(c>0 && c!='n')){
  printf("Usage: int_nmi -n number.\n");
  exit(-1);
 }

 number = atoi(optarg);

 /* Send some NMI interrupt requests to the GNU/Linux Kernel */
 for(i=0;
    (number>0)?i<number:i==i;
  _send_nmi(), /* This is the actual NMI */
  ((number==-1)?0:++i)
 );

 /* This is not going to be executed, this process is killed */
 return 0;

}

/*
 This function does send a INT 0x2 NMI interrupt
*/
void _send_nmi () {
 __asm__ __volatile__(
    "int $0x2\n"
        );
}
```

Because our NMI does not alter any sort of kernel data structure or write a certain value into a hardware general-purpose register, it is safe to run the previous program at any time during the lifetime of the system. Listing 4.5 shows how the GNU/Linux Kernel handles a NMI interrupt after executing the program shown in listing 4.4.

Listing 4.5: Forcing a **NMI** interrupt from user-land

```
/int_nmi -n 1
root@tfc:~#
Message from syslogd@tfc at May  9 12:19:41 ...
 kernel:[14650.672336] Uhhuh. NMI received for unknown reason 00 on CPU 2.

Message from syslogd@tfc at May  9 12:19:41 ...
 kernel:[14650.672339] Do you have a strange power saving mode enabled?

Message from syslogd@tfc at May  9 12:19:41 ...
 kernel:[14650.672341] Dazed and confused, but trying to continue
```

It is proved that, whenever $n \longrightarrow \infty$, and provided that the **nmi watchdog** is enabled, the GNU/Linux Kernel does kill the process issuing the NMI interrupts.

4.4 IRQ affinity among SMP systems

The GNU/Linux Kernel associates all the IRQ lines to the processor that has booted up the system, that is, **processor 0**. That is clearly shown in Listing 4.6. This output is taken from a workstation running Debian Squeeze with four processors. Some software has been implemented to aid the kernel so that interrupts can be balanced among all the available processors in the system [23]. Yet, it is feasible to redirect any single interrupt to a certain processor by means of using the `/proc` interface, in consonance with [24]. Our main goal is to assign all the IRQ lines to one processor, thus being able to gather some Involuntary Context Switching statistics.

Listing 4.6: IRQ lines per-processor distribution

```
            CPU0       CPU1       CPU2       CPU3
   0:         47          0          0          0   IO-APIC-edge      timer
   1:          7          0          0          0   IO-APIC-edge      i8042
   8:          0          0          0          0   IO-APIC-edge      rtc0
   9:          0          0          0          0   IO-APIC-fasteoi   acpi
  12:        105          0          0          0   IO-APIC-edge      i8042
  14:         70          0          0          0   IO-APIC-edge      ata_piix
  15:          0          0          0          0   IO-APIC-edge      ata_piix
  16:      68465          0          0          0   IO-APIC-fasteoi   eth1
  19:      69469          0          0          0   IO-APIC-fasteoi   ehci_hcd:usb1
  21:      19705          0          0          0   IO-APIC-fasteoi   ahci
  22:        186          0          0          0   IO-APIC-fasteoi   ohci_hcd:usb2
NMI:           0          0          0          0   Non-maskable interrupts
```

In order to divert all the interrupts to another processor at will, we have written a trivial shell script in bash, as shown in Listing 4.7.

Listing 4.7: A trivial shell script to divert interrupts

```
#!/bin/bash
#
# irq_divert.sh
# 2012 by Toni Castillo Girona
# <toni.castillo@fa.upc.edu>
#
# This script divert all IRQ lines according to processor_mask
# NOTE: not all the IRQS can be easily diverted. In this case,
# the old affinity wouldn't change.
#
# USAGE: ./irq_divert.sh <new_cpu_mask>
#
#
IRQS="/proc/irq"

# Mask from the CLI
if [ ! $# -eq 1 ]; then
 echo "Usage: `basename $0` processor_mask"
 exit 1
fi

curdir=`pwd`

cd $IRQS || exit 2
echo "Trying to set new affinity: $1"
for irq in *; do
 # Avoid "default...":
 if [ "$irq" != "default_smp_affinity" ]; then
  # Divert this irq:
  oaf=`cat $irq/smp_affinity` # old affinity
  echo "$1" > $irq/smp_affinity 2>/dev/null
  echo -ne "IRQ line:[$irq]\t" \
   "Old affinity:[$oaf]\t"  \
   "New affinity:[`cat $irq/smp_affinity`]\n"
 fi
done

cd $curdir
exit 0
```

Listing 4.8 shows the way we ran our script to divert all the interrupts to processor $CPU = 2$. The argument passed to the script via the CLI acts as a processor-mask. After executing the script, processor 2 was the one handling all the interrupts but interrupt line 0, as clearly shown in Listing 4.9. Not all the interrupt lines can be easily diverted once the system is up.

Listing 4.8: Diverting all the interrupts to processor 2

```
./irq_divert.sh 4
IRQ line:[0]   Old affinity:[f]   New affinity:[f]
IRQ line:[1]   Old affinity:[2]   New affinity:[4]
IRQ line:[10]  Old affinity:[2]   New affinity:[4]
IRQ line:[11]  Old affinity:[2]   New affinity:[4]
...
```

Listing 4.9: All the interrupts have been diverted to processor 2, but $IRQ = 0$

```
  0:        47         0         0         0   IO-APIC-edge      timer
  1:         7         0         0         0   IO-APIC-edge      i8042
  8:         0         0         0         0   IO-APIC-edge      rtc0
  9:         0         0         0         0   IO-APIC-fasteoi   acpi
 12:       105         0         0         0   IO-APIC-edge      i8042
 14:        70         0         0         0   IO-APIC-edge      ata_piix
 15:         0         0         0         0   IO-APIC-edge      ata_piix
 16:     69790       180       314         0   IO-APIC-fasteoi   eth1
 19:     70104        11       188         0   IO-APIC-fasteoi   ehci_hcd:usb1
 21:     19784        13         0         0   IO-APIC-fasteoi   ahci
 22:       186         0         0         0   IO-APIC-fasteoi   ohci_hcd:usb2
NMI:         0         0         0         0   Non-maskable interrupts
```

Chapter 5

Data Gathering

5.1 Introduction

This chapter provides some statistical information previously gathered by means of using `getcsw` or the `/proc/`PID`/sched` interface over a sort of tools and programs. The main goal is to demonstrate how an unexpected amount of interrupt requests can affect any task's performance. Firstly, some statistics have been gathered concerning Voluntary Context Switching. This first section proves that a **cpu-bound** process does not yield the processor on purpose, as well as that the proportion of calls to the GNU/Linux Kernel Scheduler during `ret_from_sys_call` is really small. Secondly, a section discussing how Involuntary Context Switching alters a process's throughput has also been provided.

The procedures applied to this task are shown in Figure 5.1.

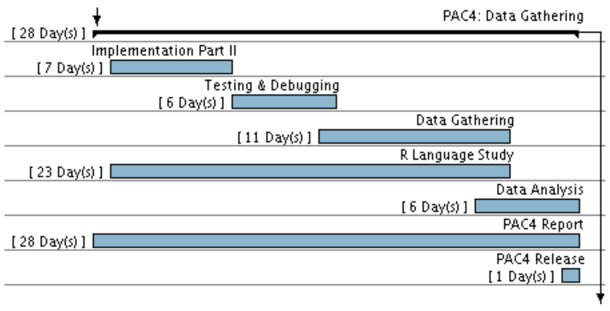


Figure 5.1: Context Switching Accounting: Data Gathering Gantt Chart

5.2 Voluntary Context Switching stats

`getcsw` has been executed over different sort of programs in order to gather some preliminary data at n intervals of time t. Voluntary Context Switching does occur, as demonstrated in section 3.2, by issuing system calls. Thus, a certain process p that is not issuing system calls at all is not likely to be preempted voluntarily. In order to prove that, we ran **cpu-bound** [2] and **io-bound** programs. Tables 5.1 to 5.4 present some results gathered with `getcsw`, in line with equation 3.2.

Table 5.2 shows a really small number for rs. Apparently, the proportion of Context Switches occurring during `ret_from_sys_call` after $t = 5s$, that is, $rs/syscalls$, is really inappreciable: 3.93×10^{-5} %. To determine whether this is a common situation or not,

[2] `burnMMX`, available by issuing `# apt-get install cpuburn`.

t_i	*vc*	*non-voluntary*	*syscalls*	*rs*	*vb*
1	0	231	43	0	0
2	0	236	43	0	0
3	0	241	43	0	0
4	0	246	43	0	0
5	0	251	43	0	0

Table 5.1: Voluntary Context Switching stats for a **cpu-bound** process; t=1s, n=5

t_i	*vc*	*non-voluntary*	*syscalls*	*rs*	*vb*
1	773	171	90663589	41	732
2	775	176	94704425	41	734
3	776	182	98770112	41	735
4	778	187	102819031	42	736
5	780	192	106866597	42	738

Table 5.2: Voluntary Context Switching stats for an **io-bound** process; t=1s, n=5

we ran two more programs: `wget` and `ping`. The former was executed in order to get an iso-image file from the Internet; the latter was run infinitely times pinging a reachable Internet host. Tables 5.3 and 5.4 present their results. The proportions of Context Switches after $t = 5s$ are 0.051 % and 0.419 %, respectively. By running these same programs a bit longer, we have determined that these proportions still apply. Tables 5.5 and 5.6 summarize these values after $t = 1min$ and $t = 10min$, respectively. This time, we did not run the `burnMMX` program because it has been already stated that only processes issuing a remarkable amount of system calls are likely to be voluntarily preempted.

t_i	*vc*	*non-voluntary*	*syscalls*	*rs*	*vb*
1	849	4	3812	1	848
2	1327	5	5772	2	1325
3	1806	8	7738	4	1802
4	2284	9	9700	5	2279
5	2762	10	11660	6	2756

Table 5.3: Voluntary Context Switching stats for the `wget` command; t=1s, n=5

Table 5.6 seems to show a higher number of Context Switches occurring at `ret_from_sys_call` for the **io-bound** program `io`. Its value for rs starts to be remarkable. Thus, we ran `io` a bit longer. Table 5.7 shows `io`'s stats after $t = 10h$. Despite the fact the proportion of rs in respect to the total amount of system calls performed by p is still less than 1%, as previously demonstrated, the number of Context Switches happening precisely whilst the kernel is returning from a system call is quite considerable. As discussed in section 3.3, the GNU/Linux Kernel considers any call to `schedule()` during `ret_from_sys_call` as Involuntary Context Switching. That explains the negative value for vb in table 5.7.

Therefore, it has been proved that the proportion of context switches happening during `ret_from_sys_call` is really small in regard to the total amount of Voluntary Context Switches during **process context**, as long as p is not heavily **io-bounded** or p is not issuing a large number of system calls.

t_i	*vc*	*non-voluntary*	*syscalls*	*rs*	*vb*
1	42	3	473	2	40
2	57	3	593	2	55
3	72	3	713	2	70
4	87	4	833	3	84
5	102	5	953	4	98

Table 5.4: Voluntary Context Switching stats for the `ping` command; t=1s, n=5

Program	*vc*	*non-voluntary*	*syscalls*	*rs*	*vb*	*Proportion*
`burnMMX`	2	78	43	0	0	0 %
`io`	17	80	58543968	102	5	0.0017 %
`wget`	6538	28	27196	10	6528	0.036 %
`ping`	217	2	1889	5	216	0.26 %

Table 5.5: Voluntary Context Switching stats after $t = 1min$

5.3 Involuntary Context Switching stats

Our main goal is to track every single call to `do_IRQ()` for a given process, p, so that we can determine whenever p's throughput is somehow affected by raised interrupts being fired all of a sudden. So, a particular task has been chosen: `gzip`. The experiment includes running gzip several times in order to compress a 2GB-avi file. This is called a sample, and some statistical data has been gathered from it and plotted using a simple regression test by means of using the `R` language, as previously introduced in 1.5.4. To clarify this regression test, we repeated the same experiment for a larger number of times. In all cases the process has been executed on processor $cpu = 0$, being $cpu = 0$ the processor handling all the raised interrupts, without any sort of irq balancing mechanism. Of course, some of the raised interrupts preempting our test process are not fired because of this particular test program, but others. It is remarkable then how, without a proper irq-balancing mechanism, even on a normal computer one process's throughput can possibly be reduced considerably by other non-related processes. Our experiment is shown in Listing 5.1.

Listing 5.1: Involuntary Context Switching shell-script experiment

```
#!/bin/bash
ITERATIONS=200
COMMAND="gzip -9 movie_test.avi"
CPU="0"
CPUIRQ="0"
STATSFILE=./stats.out
CPUMASK="0-3"

./irq_divert.sh $CPUIRQ >/dev/null

test -r movie_test.gz \
  && gunzip movie_test.gz

test -r $STATSFILE && rm -rf $STATSFILE
touch $STATSFILE
it=0

while [ $it -lt $ITERATIONS ]; do
    echo "ITERATION NUMBER: $it"     >> $STATSFILE
    st=`date +%s`
     /usr/bin/taskset -c "$CPU"  `echo $COMMAND` &
    ./getcsw -i -m "$CPUMASK" -p $! >> $STATSFILE
```

Program	*vc*	*non-voluntary*	*syscalls*	*rs*	*vb*	*Proportion*
`io`	15379	56144	131231999	3505	11874	0.0026 %
`wget`	58346	191	239787	82	58264	0.034 %
`ping`	3214	47	25266	35	3179	0.1385 %

Table 5.6: Voluntary Context Switching stats after $t = 10min$

Program	*vc*	*non-voluntary*	*syscalls*	*rs*	*vb*	*Proportion*
`io`	17465	2121110	4806052284	122225	-104760	0.0025 %

Table 5.7: `io`'s Voluntary Context Switching stats after $t = 10h$

```
    et=`date +%s`
    echo "Elapsed time: `expr $et - $st`" >> $STATSFILE
    test -r movie_test.gz && gunzip movie_test.gz
    it=`expr $it + 1`
done
```

5.3.1 Statistical data without erratic hardware

time(sec)	*IRQS#*
69	3430
84	4170
81	4829
80	4324
84	4088
79	4976
80	4167
81	4104
83	4457
79	4032
...	...

Table 5.8: Involuntary Context Switching stats due to IRQS; normal hardware behaviour

Our main tool `getcsw` has been executed over a **cpu-bound** program n times; $n > 30$. We have annotated the total amount of time spent on finishing its job, that is, to compress a 2GB-avi file. Thus, it is quite obvious that our program wrote, from time to time, chunks of data to the hard disk among cpu-intensive calculations. Thus, some of the raised interrupts that affected this program's throughput were, in fact, caused by itself, as long as this task ran precisely on the same processor handling the interrupts. Table 5.8 shows the first 10 iterations of our experiment, that is, $n = 10$. We wrote a trivial bash-script in charge of running the same experiment on the same computer n times.

Apparently, table 5.8 does not seem to show us coherent data. Our main project tries to demonstrate that, as the number of raised interrupts affecting any given process increases, so does its elapsed time. Thus, we ran a first experiment 200 times, that is, $n = 200$, and we used the trivial R-script shown in Listing 5.2 to plot the data and compute a simple regression test. The regression line is shown in Figure 5.2. Two observations seem to be misplaced in figure 5.2; they occurred due to some out-of-the-blue events that positively altered the process's behaviour, in spite of the number of issued interrupts, such as: playing some video, rendering flash, etc. That is so because, apart from the evident fact of raising an interrupt, any running process can yield the processor as described in 1.2.1.

According to the statistical data gathered using **getcsw**, we can state that $69 \leq t \leq 167$ and that $3430 \leq IRQS\# \leq 9422$ for any given execution of gzip. In addition, as shown in Figure 5.2, $IRQS = 9422$ and $t = 162s$,$t = 169s$ are not common values. Therefore, it immediately follows that for any given standard computer running without hardware malfunctions, there is not much dispersion among iterations. We have computed the mean and the standard deviation excluding the misplaced values described earlier: $\bar{t} = 88,13s$, $\bar{I} = 5395.77$,$\sigma_t = 6.76s$,$\sigma_I = 880.23$. Thus, the number of raised interrupts do affect any given running process's throughput, even on a normal computer without erratic hardware involved. `irq balance` [23] can soften this problem considerably.

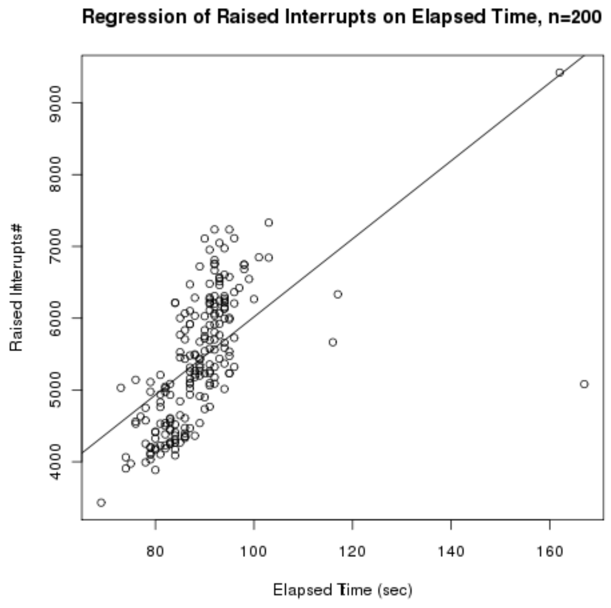


Figure 5.2: Regression of Raised Interrupts on Elapsed Time; $n = 200$

Listing 5.2: R-script Regression test to plot Figure 5.2

```r
# Read data from file:
mydata <- read.table("table.R", header=TRUE, sep=" ")
# Set time (first column) and IRQS (second column)
t <- mydata[,1]
I <- mydata[,2]
# Plot the graph on disk:
png("regression.png")
plot(t , I)
# Draw the regression line; how Interrupts affect elapsed time
#abline(lm(I~t))
title(main="Raised Interrupts on Elapsed Time, n=600 ",
  xlab="Elapsed Time (sec)", ylab="Raised Interupts#" )
dev.off()
```

In consonance with figure 5.2, as $IRQS\#$ approaches a high value, so does the elapsed time spent by the process to complete its job. Figure 5.3 really proves that the regression test does apply; we ran our experiment for a bit longer, that is, $n = 600$. We have also computed the mean and the standard deviation for this new values: $\bar{t} = 131.30s$, $\bar{I} = 12169.41$,$\sigma_t = 11.65s$,$\sigma_I = 421.40$. However, as clearly shown in Figure 5.3, as long as

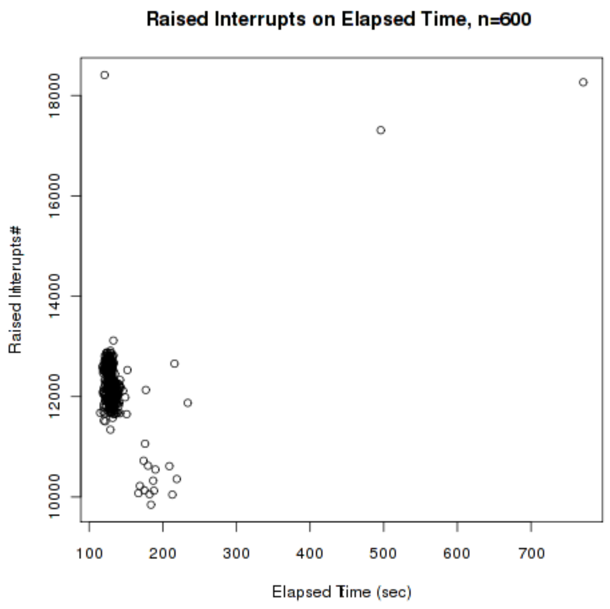


Figure 5.3: Raised Interrupts on Elapsed Time; $n = 600$

the number of iterations increases, so does the amount of misplaced data, that is, those values which time and raised interrupts do coincide with other tasks affecting them.

5.3.2 Statistical data with irq balance

`irq-balance` [23] can be installed on any flavour of GNU/Linux operating system. Its main purpose is to distribute every single interrupt among the different processors on the system, to increase its performance. Its design is heavily based on the `/proc/` interface, as described in 4.4. Running irq-balance on systems with more than just one core is recommended. Our previous experiment has been run on the same computer, this time with the irq-balance daemon taking care of the interrupt requests automatically. Figure 5.4 shows the test gathered data after running `gzip` 200 times, that is, $n = 200$.

Although the number of raised interrupts preempting our process are really small, there are a higher number of raised interrupts that seem to be misplaced on the regression chart. According to the data, irq-balance does minimize the number of interrupts affecting our process, because it does distribute them among all the available processors installed on the system, but not all the time. It appears to be a higher number of interrupts at some instant during gzip's execution that clearly come to reveal that irq-balance can provoke a great dispersion, its proportion being 10%. We have also computed the mean and the standard deviation for this new data, skipping the misplaced value of $t = 887s$ and the ones with higher IRQ# values: $\bar{t} = 127.14s$, $\bar{I} = 2274.79$, $\sigma_t = 7.01s$, $\sigma_I = 310.12$. Table 5.9 summarizes these values for our experiment with and without the presence of irq-balance, for $n = 200$.

	-	`irq-balance`
σ_t (s)	6.71	7.01
σ_I	880.23	310.12

Table 5.9: Mean and standard deviation comparison with and without irq-balance; $n = 200$

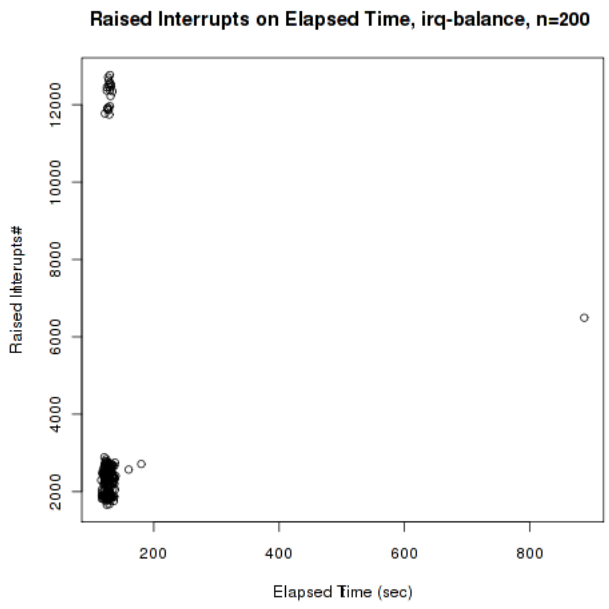


Figure 5.4: Raised Interrupts on Elapsed Time with irq-balance; $n = 200$

In consonance with table 5.9, the number of raised interrupts provoking gzip to relinquish the processor due to a `do_IRQ()` call are almost half the ones without running irq-balance. However, and despite that, the standard deviation associated with the time spent to finish its job is not quite as good as that. The problem relies on the ground that the system can be occupied in doing some other tasks at the time we ran our experiment. That means the number of interrupts do affect any running process, but there are other variables to consider.

5.3.3 Statistical data with erratic hardware without `irq-balance`

Whenever a running task behaves erratically, we have to determine whether this is due to the fact there is something wrong directly related to that particular task or, on the contrary, whatever is affecting its normal execution has nothing to do with it. Let's consider a computer where our task is running on that is behaving fine, and that there are no other cpu-bound or even io-bound tasks running. In such scenario, it would be fairly feasible to believe that our task is going to run optimally. After an unknown amount of time, this particular computer starts working erratically because of an unknown hardware malfunction. In spite of the kernel log messages, the user sending jobs is not concerned at all about such matters; the only abnormal thing the user is actually aware of is basically the time elapsed for that particular task to complete its job.

Our Involuntary Context Switching counters can detect such anomalies. Whereas the GNU/Linux Kernel only includes generic interrupt counters for all the running tasks on the system, we can track every single interrupt request that is preempting our traced process, as previously and profusely detailed.

Listing 5.3: Shell-script to generate $N \times MAX$ NMIs

```
P=./int_nmi
N=30
MAX=10
i=0
```

```
WAITSECS=30s
while [ true ]; do
    $P -n $N
    i=`expr $i + 1`
    if [ $i -eq $MAX ]; then
        i=0
        sleep $WAITSECS
    fi
done

exit 0
```

In order to demonstrate this, we have executed the same experiment, but this time provoking hundreds of NMIs in a loop, as described in 4.3.1. We ran the script shown in Listing 5.3 whilst gzip was running, with $n = 200$. We have computed the mean and the standard deviation for this new data: $\bar{t} = 387.27s$, $\bar{I} = 8769.72$, $\sigma_t = 107.27s$, $\sigma_I = 4355.14$. Figure 5.6 shows this new gathered data.

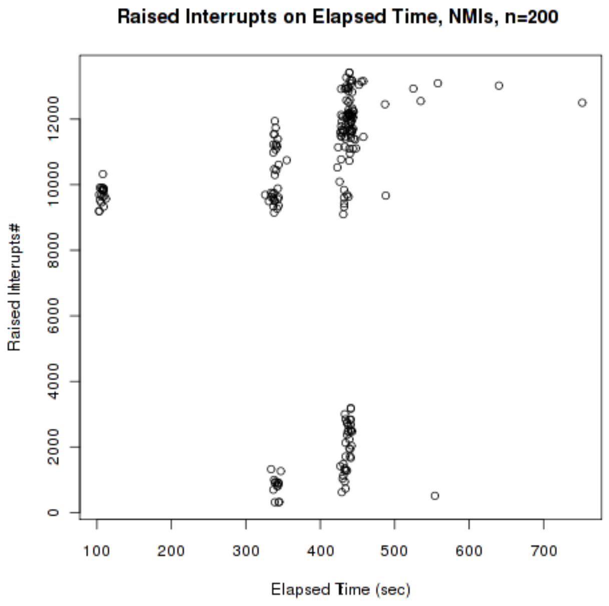


Figure 5.5: Raised Interrupts on Elapsed Time; hardware malfunction; $n = 200$

We ran the same experiment 600 times, that is, $n = 600$. As it is clearly shown in Figure 5.6, the number of interrupts preempting the process do provoke this very same task to increase its execution time considerably. There is some dispersion among the data, also.

	Normal	`irq-balance`	NMIs
t(s)	17779	26214	75132
IRQs#	1082867	659144	1701326

Table 5.10: Total time spent and number of raised interrupts summary; $n = 200$

Table 5.10 summarizes the total amount of time spent and raised interrupts for $n = 200$ in each one of those previous cases. These figures do include all those apparently misplaced values, because they represent the real time spent after executing the experiment 200 times. Although the total amount of time spent in finishing gzip's job should be smaller whilst distributing the interrupts among all the cores available

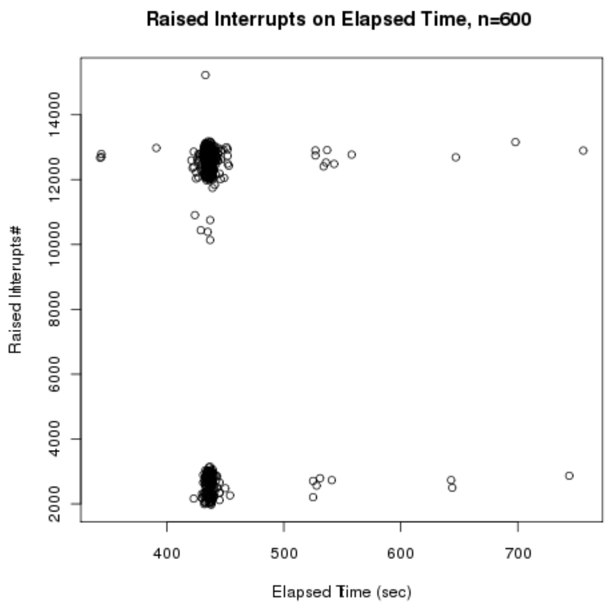


Figure 5.6: Raised Interrupts on Elapsed Time; hardware malfunction; $n = 600$

on the system by means of running `irq-balance`, that is not precisely accurate in real environments. That is so, as previously stated, because we have executed every single experiment during normal workload times, that is, whilst displaying a video on you tube or writing this final report. All tests have been conducted this way. Thus, in spite of having less interrupts preempting gzip, it is still feasible to increase its execution time. However, figures related to the unusual amount of NMIs do emphasize that there is an abnormal or even erratic behaviour being involved in the system, if one could compare these times with the ones in the first column or even the second one in table 5.10.

Chapter 6

Conclusions

Context Switching has a real impact on any running process' throughput. The ways a context switch come to happen are either voluntary or involuntary. Although modern operating systems - in fact, not only GNU/Linux - do include some sort of measuring tools to account them, there were no actual mechanisms to differentiate among their own subtype. Thus, our project has implemented a sort of counters to gather and then plot dot graphs that allow us to determine whether the time spent in doing some particular task is being affected by an unknown number of external events to the process or, on the contrary, we have to perfect its implementation.

Our Context Switching Mechanism does alter the entire system's behaviour. It does have a cost in time and resources, such as memory, because it directly deals with internal data-structures. Therefore, this approach is merely experimental and it has to be put to test in real environments. All the work has been conducted on a Virtual Machine, thus the figures can possibly vary on actual computers running the operating system natively.

6.1 Future Work

The way these counters are implemented are architecture-dependant. Thus, it has to be mandatory to develop a newer solution capable of implementing a good-layer of abstraction. Instead of altering internal data structures and hooking some low-level kernel code, another approach would consist on deploying **Loadable Kernel Modules** and make use of the GNU/Linux Kprobes facility. Besides, our `getcsw` gathering tool has to be improved so that it can not only get some statistical data, but plot it as well in real-time. This way it would be feasible to trace a particular task constantly from the very beginning by means of graphical data. Our experiments must be re-considered by adding some entropy, and then being re-run for larger values of n.

Bibliography

[1] The Completely Fair Schedule Algorithm. `http://en.wikipedia.org/wiki/Completely_Fair_Scheduler`.

[2] R. Love. Linux Kernel Development. *Addison-Wesley Professional,* Chapter IV: Process Scheduling, June. 2010.

[3] J. Corbet, A. Rubini & G. Kroah-Hartman. Linux Device Drivers, 3rd edition. *O'Reilly,* Chapter 2: Building and Running Modules; Chapter 3: Char Drivers, February. 2005.

[4] Linux Kernel TaskStats. Linux Kernel Documentation, *Documentation/accounting/-taskstats.txt.*

[5] J. Keniston, P. S Panchamukhi. Linux Kernel Probes. Linux Kernel Documentation, *Documentation/kprobes.txt.*

[6] N. Matloff. The Art Of R Programming. *William Pollock,* 2011.

[7] Oracle VirtualBox.
`https://www.virtualbox.org/`

[8] Performance API.
`http://icl.cs.utk.edu/papi/index.html`

[9] Atsar, `http://freecode.com/projects/atsar`.

[10] System Statistics (`sysstat`). `http://sebastien.godard.pagesperso-orange.fr/`.

[11] The GNU/Linux Kernel TaskStats. Linux Kernel Documentation, *Documentation/accounting/taskstats.txt.*

[12] `getdelays.c`, a C program sample of `Kernel TaskStats` and **Netlink** interface.

[13] Taskstats: add context-switch counters.
`http://www.mail-archive.com/git-commits-head@vger.kernel.org/msg16941.html`.

[14] T. Castillo Girona. Context Switching Accounting, PAC1: Working Plan, *Section: Context Switching Mechanism*, pages 2-3.

[15] P. N. Ayuso, R. M. Gasca, L. Lefèvre. Communicating between the kernel and user-space in Linux using Netlink sockets. Software: Practice and Experience, 40(9):797-810, August 2010.

[16] The GNU/Linux Kernel TaskStats: The struct taskstats, GNU/Linux Kernel Documentation. *Documentation/accounting/taskstats-struct.txt.*

[17] List of available processors' counter registers, sorted by cpu-name. `http://oprofile.sourceforge.net/docs/`

[18] T. Castillo Girona. Context Switching Accounting, PAC2: Tools, AP Is & ABIs analysis. *Section: Kernel ABIs*, pages 4-10.

[19] T. Castillo Girona. Context Switching Accounting, PAC2: Tools, APIs & ABIs analysis. *Section: Kernel ABIs , subsection 4.1: task_struct data structure*, page 5.

[20] T. Castillo Girona. Context Switching Accounting, PAC2: Tools, APIs & ABIs analysis. *Section 4.2.3: Extending `taskstats`*, page 8.

[21] T. Castillo Girona. Context Switching Accounting, PAC2: Tools, APIs & ABIs analysis. *Section 5: Setup Framework*, page 11.

[22] T. Castillo Girona. Context Switching Accounting, PAC2: Tools, APIs & ABIs analysis. *Section 4.2.4: `task_struct` and Kernel Taskstats relationship*, page 10.

[23] IRQBALANCE - handholding your interrupts for power and performance `http://irqbalance.org/documentation.html`

[24] SMP IRQ affinity *Documentation/IRQ-affinity.txt.*