Context Switching Accounting Mechanism

T. Castillo Girona
<toni.castillo@uoc.edu>

Miquel Angel Senar Rosell
Context Switching Basis

- Every single process \( p \) runs either in **Ring 0** or **Ring 3**.
  - **Ring 0**: low-level or hardware tasks.
  - **Ring 3**: user-space tasks.
Context Switching Basis

- Every single process $p$ runs either in **Ring 0** or **Ring 3**.
  - **Ring 0**: low-level or hardware tasks.
  - **Ring 3**: user-space tasks.
Context Switching Basis

- Every single process $p$ runs either in **Ring 0** or **Ring 3**.
  - **Ring 0**: low-level or hardware tasks.
  - **Ring 3**: user-space tasks.
Context Switching Basis

• Every single process $p$ runs either in **Ring 0** or **Ring 3**.
  - **Ring 0**: low-level or hardware tasks.
  - **Ring 3**: user-space tasks.

• $p$ runs only for a measurable time $t$. 
Context Switching Basis

- Every single process $p$ runs either in **Ring 0** or **Ring 3**.
  - **Ring 0**: low-level or hardware tasks.
  - **Ring 3**: user-space tasks.
- $p$ runs only for a measurable time $t$.
- The **GNU/Linux Scheduler** is in charge of making $p$ yield the cpu.
  - Modern GNU/Linux Kernels implements the **CFS** scheduler.
  - The Kernel executes `schedule()` to call the Scheduler.
Context Switching Basis

- Every single process $p$ runs either in **Ring 0** or **Ring 3**.
  - **Ring 0**: low-level or hardware tasks.
  - **Ring 3**: user-space tasks.
- $p$ runs only for a measurable time $t$.
- The **GNU/Linux Scheduler** is in charge of making $p$ yield the cpu.
  - Modern GNU/Linux Kernels implements the **CFS** scheduler.
  - The Kernel executes `schedule()` to call the Scheduler.
Context Switching Basis

- Every single process $p$ runs either in **Ring 0** or **Ring 3**.
  - **Ring 0**: low-level or hardware tasks.
  - **Ring 3**: user-space tasks.
- $p$ runs only for a measurable time $t$.
- The **GNU/Linux Scheduler** is in charge of making $p$ yield the cpu.
  - Modern GNU/Linux Kernels implements the CFS scheduler.
  - The Kernel executes `schedule()` to call the Scheduler.
Context Switching Basis

- Whenever \( p \) yields the processor, there is a **Context Switch**.
Context Switching Basis

- Whenever $p$ yields the processor, there is a **Context Switch**.
  - **Voluntary**:
    - $p$ has issued a **System Call**, and now the Kernel is returning from it.
    - $p$ has issued an explicit call to **schedule()**, by means of calling **sched_yield()**.
    - $p$ has ended its execution.
Context Switching Basis

- Whenever $p$ yields the processor, there is a Context Switch.
  - Voluntary:
    - $p$ has issued a System Call, and now the Kernel is returning from it.
    - $p$ has issued an explicit call to `schedule()`, by means of calling `sched_yield()`.
    - $p$ has ended its execution.
Context Switching Basis

- Whenever \( p \) yields the processor, there is a Context Switch.

  - **Voluntary**:
    - \( p \) has issued a **System Call**, and now the Kernel is returning from it.
    - \( p \) has issued an explicit call to `schedule()`, by means of calling `sched_yield()`.
    - \( p \) has ended its execution.
Context Switching Basis

- Whenever $p$ yields the processor, there is a **Context Switch**.
  - **Voluntary**:  
    - $p$ has issued a **System Call**, and now the Kernel is returning from it.  
    - $p$ has issued an explicit call to `schedule()`, by means of calling `sched_yield()`.  
    - $p$ has ended its execution.
Context Switching Basis

- Whenever $p$ yields the processor, there is a **Context Switch**.
  - **Voluntary**:
    - $p$ has issued a **System Call**, and now the Kernel is returning from it.
    - $p$ has issued an explicit call to `schedule()`, by means of calling `sched_yield()`.
    - $p$ has ended its execution.
  - **Involuntary**:
    - $p$ has been preempted by an **Interrupt Handler**.
    - $p$ has exhausted its processor’s time proportion.
    - $p$ has been preempted in favour of a recently waken up task with a higher priority.
Context Switching Basis

- Whenever \( p \) yields the processor, there is a **Context Switch**.
  - **Voluntary**:
    - \( p \) has issued a **System Call**, and now the Kernel is returning from it.
    - \( p \) has issued an explicit call to `schedule()`, by means of calling `sched_yield()`.
    - \( p \) has ended its execution.
  - **Involuntary**:
    - \( p \) has been preempted by an **Interrupt Handler**.
    - \( p \) has exhausted its processor’s time proportion.
    - \( p \) has been preempted in favour of a recently waken up task with a higher priority.
Context Switching Basis

- Whenever $p$ yields the processor, there is a **Context Switch**.
  - **Voluntary**:
    - $p$ has issued a **System Call**, and now the Kernel is returning from it.
    - $p$ has issued an explicit call to `schedule()`, by means of calling `sched_yield()`.
    - $p$ has ended its execution.
  - **Involuntary**:
    - $p$ has been preempted by an **Interrupt Handler**.
    - $p$ has exhausted its processor’s time proportion.
    - $p$ has been preempted in favour of a recently waken up task with a higher priority.
Context Switching Basis

- Whenever $p$ yields the processor, there is a **Context Switch**.
  - **Voluntary**:
    - $p$ has issued a **System Call**, and now the Kernel is returning from it.
    - $p$ has issued an explicit call to `schedule()`, by means of calling `sched_yield()`.
    - $p$ has ended its execution.
  - **Involuntary**:
    - $p$ has been preempted by an **Interrupt Handler**.
    - $p$ has exhausted its processor’s time proportion.
    - $p$ has been preempted in favour of a recently waken up task with a higher priority.
Context Switching Basis

- Whenever $p$ yields the processor, there is a Context Switch.
  - **Voluntary**:
    - $p$ has issued a **System Call**, and now the Kernel is returning from it.
    - $p$ has issued an explicit call to `schedule()`, by means of calling `sched_yield()`.
    - $p$ has ended its execution.
  - **Involuntary**:
    - $p$ has been preempted by an **Interrupt Handler**.
    - $p$ has exhausted its processor’s time proportion.
    - $p$ has been preempted in favour of a recently waken up task with a higher priority.

- Modern GNU/Linux operating systems **does not have counters for all these particular** Context Switching sub-types.
Context Switching Basis

- Whenever $p$ yields the processor, there is a Context Switch.
  - **Voluntary**:
    - $p$ has issued a **System Call**, and now the Kernel is returning from it.
    - $p$ has issued an explicit call to `schedule()`, by means of calling `sched_yield()`.
    - $p$ has ended its execution.
  - **Involuntary**:
    - $p$ has been preempted by an **Interrupt Handler**.
    - $p$ has exhausted its processor’s time proportion.
    - $p$ has been preempted in favour of a recently waken up task with a higher priority.

- Modern GNU/Linux operating systems **does not have counters for all these particular** Context Switching sub-types.

- Our project **does add** these new counters.
Tools, APIs & ABIS

- **Tools**
  - Based on the `/proc` interface.
  - Gather **cumulative** statistics for total amount of **Voluntary** and **Involuntary** Context Switches.
  - `atsar`, reads `/proc/stat`
  - `pidstat`, reads `/proc/PID/stat`, works with **averages**.
Tools, APIs & ABIS

- **Tools**
  - Based on the `/proc` interface.
  - Gather **cumulative** statistics for total amount of **Voluntary** and **Involuntary** Context Switches.
  - `atsar`, reads `/proc/stat`
  - `pidstat`, reads `/proc/PID/stat`, works with **averages**.
Tools, APIs & ABIS

- **Tools**
  - Based on the `/proc` interface.
  - Gather **cumulative** statistics for total amount of **Voluntary** and **Involuntary** Context Switches.
  - `atsar`, reads `/proc/stat`
  - `pidstat`, reads `/proc/PID/stat`, works with **averages**.
Tools, APIs & ABIS

- **Tools**
  - Based on the `/proc` interface.
  - Gather **cumulative** statistics for total amount of **Voluntary** and **Involuntary** Context Switches.
  - `atsar`, reads `/proc/stat`
  - `pidstat`, reads `/proc/PID/stat`, works with averages.

<table>
<thead>
<tr>
<th>Time</th>
<th>ps</th>
<th>swch</th>
<th>srunq</th>
<th>nr</th>
<th>procs</th>
<th>avg1</th>
<th>lavg5</th>
<th>avg15</th>
</tr>
</thead>
<tbody>
<tr>
<td>18:14:35</td>
<td>100</td>
<td>1</td>
<td>323</td>
<td>0.29</td>
<td>0.33</td>
<td>0.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:14:36</td>
<td>1561</td>
<td>1</td>
<td>323</td>
<td>0.26</td>
<td>0.33</td>
<td>0.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:14:38</td>
<td>1872</td>
<td>1</td>
<td>323</td>
<td>0.26</td>
<td>0.33</td>
<td>0.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:14:39</td>
<td>1528</td>
<td>1</td>
<td>323</td>
<td>0.26</td>
<td>0.33</td>
<td>0.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:14:40</td>
<td>1190</td>
<td>1</td>
<td>323</td>
<td>0.26</td>
<td>0.33</td>
<td>0.32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tools, APIs & ABIS

- Tools
  - Based on the `/proc` interface.
  - Gather **cumulative** statistics for total amount of **Voluntary** and **Involuntary** Context Switches.
  - `atsar`, reads `/proc/stat`
  - `pidstat`, reads `/proc/PID/stat`, works with averages.

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

18:00:49    PID  cswch/s nvcswh/s  Command
18:00:49 4152  0.02   0.01  vlc
```
Tools, APIs & ABIS

- **Tools**
  - Based on the `/proc` interface.
  - Gather **cumulative** statistics for total amount of **Voluntary** and **Involuntary** Context Switches.
    - `atsar`, reads `/proc/stat`
    - `pidstat`, reads `/proc/PID/stat`, works with **averages**.

- **Kernel ABIs**
  - `/proc/PID/sched` interface.
  - `taskstats` facility, using **Netlink** infrastructure.
Tools, APIs & ABIS

- **Tools**
  - Based on the `/proc` interface.
  - Gather *cumulative* statistics for total amount of *Voluntary* and *Involuntary* Context Switches.
  - `atsar`, reads `/proc/stat`
  - `pidstat`, reads `/proc/PID/stat`, works with averages.

- **Kernel ABIs**
  - `/proc/PID/sched` interface.
  - `taskstats` facility, using *Netlink* infrastructure.

```
vlc (4152, #threads: 7)
------------------------------------------
se.exec_start : 3196505.837931
se.run_time  : 370749.896090
se.sum_exec_run_time : 72.197356
se.avg_overlap : 0.080572
se.avg_wakeup  : 1.135731
se.avg_running : 0.030190
nr_switches   : 126
nr_voluntary_switches : 102
nr_involuntary_switches : 24
se.load.weight : 1024
policy        : 0
prio          : 120
clock.delta   : 276
```
Tools, APIs & ABIS

- **Tools**
  - Based on the `/proc` interface.
  - Gather **cumulative** statistics for total amount of **Voluntary** and **Involuntary** Context Switches.
  - `atsar`, reads `/proc/stat`
  - `pidstat`, reads `/proc/PID/stat`, works with **averages**.

- **Kernel ABIs**
  - `/proc/PID/sched` interface.
  - `taskstats` facility, using **Netlink** infrastructure.
Tools, APIs & ABIS

- Advantages of using the `taskstats` interface.
  - Easy to communicate with the GNU/Linux Kernel.
  - There is no need to develop a Linux Kernel Module.
  - A client tool written in C running in user-space, `getcw.c`. 
Tools, APIs & ABIS

- Advantages of using the `taskstats` interface.
  - Easy to communicate with the GNU/Linux Kernel.
  - There is no need to develop a Linux Kernel Module.
  - A client tool written in C running in user-space, `getcw.c`.

Context Switching Accounting Mechanism 2012 Toni Castillo Girona - <toni.castillo@uoc.edu> - p. 5
Tools, APIs & ABIS

- Advantages of using the `taskstats` interface.
  - Easy to communicate with the GNU/Linux Kernel.
  - There is no need to develop a **Linux Kernel Module**.
  - A client tool written in C running in user-space, `getcw.c`. 
Tools, APIs & ABIS

- **Advantages of using the `taskstats` interface.**
  - Easy to communicate with the GNU/Linux Kernel.
  - There is no need to develop a **Linux Kernel Module**.
  - A client tool written in C running in user-space, `getcw.c`.
Tools, APIs & ABIS

- **Advantages of using the `taskstats` interface.**
  - Easy to communicate with the GNU/Linux Kernel.
  - There is no need to develop a **Linux Kernel Module**.
  - A client tool written in C running in user-space, `getcw.c`.

- **Extending the `taskstats` interface...**

```c
struct taskstats {
  ...
  /* New version: 8*/
  u64  dummy;
}

/* Let's fill "dummy": */
stats->dummy = 6 6 6;

#define MIN VERSION 8

/* Get dummy if we don't have the right version */
  (t->version >= MIN VERSION) ? t->dummy: 0;
...
```
Tools, APIs & ABIS

- Advantages of using the `taskstats` interface.
  - Easy to communicate with the GNU/Linux Kernel.
  - There is no need to develop a Linux Kernel Module.
  - A client tool written in C running in user-space, `getcw.c`.

- Extending the `taskstats` interface...
- ... and reading it back from user-space.

```
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
    1        279        22init
dummy
```
Tools, APIs & ABIS

- **Advantages of using the** `taskstats` **interface.**
  - Easy to communicate with the GNU/Linux Kernel.
  - There is no need to develop a **Linux Kernel Module**.
  - A client tool written in C running in user-space, `getcw.c`.

- Extending the `taskstats` interface...
- ... and reading it back from user-space.
- The `task_struct` data structure has to be altered accordingly.

```c
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
1 279 22 init
dummy 666
```
Design & Implementation

- We have **added new counters** to the GNU/Linux Kernel.

<table>
<thead>
<tr>
<th>Counter</th>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsyscalls</td>
<td>-</td>
<td>Total amount of issued syscalls</td>
</tr>
<tr>
<td>nvcsw_ext[]</td>
<td>0</td>
<td>Calls to <code>schedule()</code> at <code>ret_from_syscall</code>. Voluntary task’s exit. 1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Calls to the <code>sched_yield()</code> system call. Number of calls to <code>schedule()</code> per syscall.</td>
</tr>
<tr>
<td>nsyscalls.schedule[]</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>nvcsw_ext[]</td>
<td>0</td>
<td>Calls to <code>schedule()</code> during <code>try_to_wake_up()</code>.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Calls to <code>schedule()</code> during <code>do_irq()</code>.</td>
</tr>
<tr>
<td>nvcsw_ext.exit</td>
<td>-</td>
<td>Involuntary task’s exit. 1</td>
</tr>
</tbody>
</table>
Design & Implementation

- We have **added new counters** to the GNU/Linux Kernel.
- We have extended the `task_struct` and `taskstats` data structures to implement them.

```c
struct task_struct {
    ...
    unsigned long nsyscalls;
    unsigned long nvcswn_ext[3];
    atomic64_t nivcswn_ext[2];
    unsigned long nivcswn_ext_exit;
    unsigned long nsycalls_schedule[___NR_syscall_max+1];
    ...
}
```
Design & Implementation

- We have **added new counters** to the GNU/Linux Kernel.
- We have extended the `task_struct` and `taskstats` data structures to implement them.
- In order to read their data, we have implemented a client C program communicating with the Kernel via `NetLink`, called `getcsw.c`. 
Design & Implementation

- We have **added new counters** to the GNU/Linux Kernel.
- We have extended the `task_struct` and `taskstats` data structures to implement them.
- In order to read their data, we have implemented a client C program communicating with the Kernel via **NetLink**, called `getcsw.c`.
- Our counters can ...
  - account the total amount of syscalls per task.
  - account the total amount of calls to the `sched_yield()` function.
  - account the total feasible amount of **context switches** whilst returning from a system call.
  - account preemption due to **Interrupts**.
  - account preemption due to `try_to_wake_up()`.
  - determine whether \( p \) has ended its execution **on its own accord** or not.
Design & Implementation

- We have **added new counters** to the GNU/Linux Kernel.
- We have extended the `task_struct` and `taskstats` data structures to implement them.
- In order to read their data, we have implemented a client C program communicating with the Kernel via `NetLink`, called `getcsw.c`.
- Our counters can ...  
  - account the total amount of syscalls per task.
  - account the total amount of calls to the `sched_yield()` function.
  - account the total feasible amount of **context switches** whilst returning from a system call.
  - account preemption due to **Interrupts**.
  - account preemption due to `try_to_wake_up()`.
  - determine whether \( p \) has ended its execution on its own accord or not.
Design & Implementation

- We have **added new counters** to the GNU/Linux Kernel.
- We have extended the `task_struct` and `taskstats` data structures to implement them.
- In order to read their data, we have implemented a client C program communicating with the Kernel via `NetLink`, called `getcsw.c`.
- Our counters can ...  
  - account the total amount of syscalls per task.
  - account the total amount of calls to the `sched_yield()` function.
  - account the total feasible amount of context switches whilst returning from a system call.
  - account preemption due to `Interrupts`.
  - account preemption due to `try_to_wake_up()`.
  - determine whether \( p \) has ended its execution **on its own accord** or not.
Design & Implementation

- We have **added new counters** to the GNU/Linux Kernel.
- We have extended the `task_struct` and `taskstats` data structures to implement them.
- In order to read their data, we have implemented a client C program communicating with the Kernel via `NetLink`, called `getcsw.c`.
- Our counters can ...
  - account the total amount of syscalls per task.
  - account the total amount of calls to the `sched_yield()` function.
  - account the total feasible amount of **context switches** whilst returning from a system call.
  - account preemption due to **Interrupts**.
  - account preemption due to `try_to_wake_up()`.
  - determine whether \( p \) has ended its execution **on its own accord** or not.
Design & Implementation

- We have **added new counters** to the GNU/Linux Kernel.
- We have extended the `task_struct` and `taskstats` data structures to implement them.
- In order to read their data, we have implemented a client C program communicating with the Kernel via **NetLink**, called `getcsw.c`.
- Our counters can ...
  - account the total amount of syscalls per task.
  - account the total amount of calls to the `sched_yield()` function.
  - account the total feasible amount of context switches whilst returning from a system call.
  - account preemption due to **Interrupts**.
  - account preemption due to `try_to_wake_up()`.
  - determine whether \( p \) has ended its execution on its own accord or not.
Design & Implementation

- We have **added new counters** to the GNU/Linux Kernel.
- We have extended the `task_struct` and `taskstats` data structures to implement them.
- In order to read their data, we have implemented a client C program communicating with the Kernel via `NetLink`, called `getcsw.c`.
- Our counters can ...
  - account the total amount of syscalls per task.
  - account the total amount of calls to the `sched_yield()` function.
  - account the total feasible amount of **context switches** whilst returning from a system call.
  - account preemption due to **Interrupts**.
  - account preemption due to `try_to_wake_up()`.
  - determine whether \( p \) has ended its execution **on its own accord** or not.
Design & Implementation

• We have **added new counters** to the GNU/Linux Kernel.
• We have extended the `task_struct` and `taskstats` data structures to implement them.
• In order to read their data, we have implemented a client C program communicating with the Kernel via `NetLink`, called `getcsw.c`.
• Our counters can ...
  ◦ **account the total amount of syscalls per task.**
  ◦ **account the total amount of calls to the `sched_yield()` function.**
  ◦ **account the total feasible amount of context switches whilst returning from a system call.**
  ◦ **account preemption due to `Interrupts`**.
  ◦ **account preemption due to `try_to_wake_up()`**.
  ◦ **determine whether \( p \) has ended its execution on its own accord or not.**
Design & Implementation

- We have **added new counters** to the GNU/Linux Kernel.
- We have extended the `task_struct` and `taskstats` data structures to implement them.
- In order to read their data, we have implemented a client C program communicating with the Kernel via **NetLink**, called `getcsw.c`.
- Our counters can ...
  - account the total amount of syscalls per task.
  - account the total amount of calls to the `sched_yield()` function.
  - account the total feasible amount of **context switches** whilst returning from a system call.
  - account preemption due to **Interrupts**.
  - account preemption due to `try_to_wake_up()`.
  - determine whether `p` has ended its execution **on its own accord** or not.
- To enable these counters, a **patch** has to be applied to the GNU/Linux Kernel.
Execution Examples

- Our main interface to the Kernel is `getcsw.c`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-p PID</code></td>
<td>Mandatory. Gets process <code>PID</code>'s stats.</td>
<td><code>./getcsw -p 2345</code></td>
</tr>
<tr>
<td><code>-i</code></td>
<td>Shows Involuntary Context Switches extended counters.</td>
<td><code>./getcsw -p 2345 -i</code></td>
</tr>
<tr>
<td><code>-l</code></td>
<td>Gets process <code>PID</code>'s stats at infinite intervals of 1s.</td>
<td><code>./getcsw -p 1 -l</code></td>
</tr>
<tr>
<td><code>-d t</code></td>
<td>Alters the default <code>-l</code>'s delay of 1s to <code>t</code> seconds.</td>
<td><code>./getcsw -p 45 -l -d 10</code></td>
</tr>
<tr>
<td><code>-m mask</code></td>
<td>Sets which cpu(s) we are listening to finishing tasks.</td>
<td><code>./getcsw -p 1 -m &quot;0,3&quot;</code></td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Enables verbosity.</td>
<td><code>./getcsw -p 2345 -l -v</code></td>
</tr>
</tbody>
</table>
Execution Examples

- Our main interface to the Kernel is `getcsw.c`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p PID</td>
<td>Mandatory. Gets process PID’s stats.</td>
<td>./getcsw -p 2345</td>
</tr>
<tr>
<td>-i</td>
<td>Shows Involuntary Context Switches extended counters.</td>
<td>./getcsw -p 2345 -i</td>
</tr>
<tr>
<td>-l</td>
<td>Gets process PID’s stats at infinite intervals of 1s.</td>
<td>./getcsw -p 1 -l</td>
</tr>
<tr>
<td>-d t</td>
<td>Alters the default -l’s delay of 1s to t seconds.</td>
<td>./getcsw -p 45 -1 -d 10</td>
</tr>
<tr>
<td>-m mask</td>
<td>Sets which cpu(s) we are listening to finishing tasks.</td>
<td>./getcsw -p 1 -m &quot;0,3&quot;</td>
</tr>
<tr>
<td>-v</td>
<td>Enables verbosity.</td>
<td>./getcsw -p 2345 -1 -v</td>
</tr>
</tbody>
</table>

Having a peak at the counters

```
./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 –
```
Execution Examples

• Our main interface to the Kernel is `getcsw.c`.

```
<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p</td>
<td>Mandatory. Gets process PID’s stats.</td>
<td>./getcsw -p 2345</td>
</tr>
<tr>
<td>-i</td>
<td>Shows Involuntary Context Switches extended counters.</td>
<td>./getcsw -p 2345 -i</td>
</tr>
<tr>
<td>-l</td>
<td>Gets process PID’s stats at infinite intervals of 1s.</td>
<td>./getcsw -p 1 -l</td>
</tr>
<tr>
<td>-d t</td>
<td>Alters the default -l’s delay of 1s to t seconds.</td>
<td>./getcsw -p 45 -l -d 10</td>
</tr>
<tr>
<td>-m mask</td>
<td>Sets which cpu(s) we are listening to finishing tasks.</td>
<td>./getcsw -p 1 -m &quot;0,3&quot;</td>
</tr>
<tr>
<td>-v</td>
<td>Enables verbosity.</td>
<td>./getcsw -p 2345 -l -v</td>
</tr>
</tbody>
</table>
```

Reading the Involuntary Context Switches extended counters

```
./getcsw -p 1 -i

<table>
<thead>
<tr>
<th>Task</th>
<th>voluntary</th>
<th>nonvoluntary</th>
<th>wake_up</th>
<th>IRQS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>71586</td>
<td>142</td>
<td>139</td>
<td>64</td>
</tr>
</tbody>
</table>
```
### Execution Examples

- Our main interface to the Kernel is `getcsw.c`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-p PID</code></td>
<td>Mandatory. Gets process PID’s stats.</td>
<td><code>./getcsw -p 2345</code></td>
</tr>
<tr>
<td><code>-i</code></td>
<td>Shows Involuntary Context Switches extended counters.</td>
<td><code>./getcsw -p 2345 -i</code></td>
</tr>
<tr>
<td><code>-l</code></td>
<td>Gets process PID’s stats at infinite intervals of 1s.</td>
<td><code>./getcsw -p 1 -l</code></td>
</tr>
<tr>
<td><code>-d t</code></td>
<td>Alters the default -l’s delay of 1s to t seconds.</td>
<td><code>./getcsw -p 45 -l -d 10</code></td>
</tr>
<tr>
<td><code>-m mask</code></td>
<td>Sets which cpu(s) we are listening to finishing tasks.</td>
<td><code>./getcsw -p 1 -m &quot;0,3&quot;</code></td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Enables verbosity.</td>
<td><code>./getcsw -p 2345 -l -v</code></td>
</tr>
</tbody>
</table>

Reading the counters at infinite intervals of time $t = 10$ seconds.

```
./getcsw -p 'ps -C ping|tail -l|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              | –                |
| ...                     |                        |                  |                  |                  |                  |                  |
```
Execution Examples

• Our main interface to the Kernel is `getcsw.c`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p PID</td>
<td>Mandatory. Gets process PID’s stats.</td>
<td>./getcsw -p 2345</td>
</tr>
<tr>
<td>-i</td>
<td>Shows Involuntary Context Switches extended counters.</td>
<td>./getcsw -p 2345 -i</td>
</tr>
<tr>
<td>-l</td>
<td>Gets process PID’s stats at infinite intervals of 1s.</td>
<td>./getcsw -p 1 -l</td>
</tr>
<tr>
<td>-d t</td>
<td>Alters the default -l’s delay of 1s to t seconds.</td>
<td>./getcsw -p 45 -l -d 10</td>
</tr>
<tr>
<td>-m mask</td>
<td>Sets which cpu(s) we are listening to finishing tasks.</td>
<td>./getcsw -p 1 -m &quot;0,3&quot;</td>
</tr>
<tr>
<td>-v</td>
<td>Enables verbosity.</td>
<td>./getcsw -p 2345 -l -v</td>
</tr>
</tbody>
</table>

Waiting for a task 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
```
Execution Examples

- Our main interface to the Kernel is `getcsw.c`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-p PID</code></td>
<td>Mandatory. Gets process PID’s stats.</td>
<td><code>./getcsw -p 2345</code></td>
</tr>
<tr>
<td><code>-i</code></td>
<td>Shows Involuntary Context Switches extended counters.</td>
<td><code>./getcsw -p 2345 -i</code></td>
</tr>
<tr>
<td><code>-l</code></td>
<td>Gets process PID’s stats at infinite intervals of 1s.</td>
<td><code>./getcsw -p 1 -l</code></td>
</tr>
<tr>
<td><code>-d t</code></td>
<td>Alters the default -1’s delay of 1s to t seconds.</td>
<td><code>./getcsw -p 45 -l -d 10</code></td>
</tr>
<tr>
<td><code>-m mask</code></td>
<td>Sets which cpu(s) we are listening to finishing tasks.</td>
<td><code>./getcsw -p 1 -m &quot;0,3&quot;</code></td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Enables verbosity.</td>
<td><code>./getcsw -p 2345 -l -v</code></td>
</tr>
</tbody>
</table>

- We have written some trivial **scripts** to add more functionality:
Execution Examples

- Our main interface to the Kernel is `getcsw.c`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p PID</td>
<td>Mandatory. Gets process PID’s stats.</td>
<td><code>.getcsw -p 2345</code></td>
</tr>
<tr>
<td>-i</td>
<td>Shows Involuntary Context Switches extended counters.</td>
<td><code>.getcsw -p 2345 -i</code></td>
</tr>
<tr>
<td>-l</td>
<td>Gets process PID’s stats at infinite intervals of 1s.</td>
<td><code>.getcsw -p 1 -l</code></td>
</tr>
<tr>
<td>-d t</td>
<td>Alters the default -l’s delay of 1s to t seconds.</td>
<td><code>.getcsw -p 45 -l -d 10</code></td>
</tr>
<tr>
<td>-m mask</td>
<td>Sets which cpu(s) we are listening to finishing tasks.</td>
<td><code>.getcsw -p 1 -m &quot;0,3&quot;</code></td>
</tr>
<tr>
<td>-v</td>
<td>Enables verbosity.</td>
<td><code>.getcsw -p 2345 -l -v</code></td>
</tr>
</tbody>
</table>

- We have written some trivial **scripts** to add more functionality:
  - `fschedyield.sh`, in charge of looking for running tasks that are calling the `sched_yield()` function.

```
<table>
<thead>
<tr>
<th>Task</th>
<th>Calls</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>5700</td>
<td>1</td>
<td><code>/usr/sbin/kernelloops</code></td>
</tr>
<tr>
<td>6432</td>
<td>3</td>
<td><code>.io</code></td>
</tr>
</tbody>
</table>
```
Execution Examples

• Our main interface to the Kernel is `getcsw.c`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p PID</td>
<td>Mandatory. Gets process PID’s stats.</td>
<td>./getcsw -p 2345</td>
</tr>
<tr>
<td>-i</td>
<td>Shows Involuntary Context Switches extended counters.</td>
<td>./getcsw -p 2345 -i</td>
</tr>
<tr>
<td>-l</td>
<td>Gets process PID’s stats at infinite intervals of 1s.</td>
<td>./getcsw -p 1 -l</td>
</tr>
<tr>
<td>-d t</td>
<td>Alters the default -l’s delay of 1s to t seconds.</td>
<td>./getcsw -p 45 -l -d 10</td>
</tr>
<tr>
<td>-m mask</td>
<td>Sets which cpu(s) we are listening to finishing tasks.</td>
<td>./getcsw -p 1 -m &quot;0,3&quot;</td>
</tr>
<tr>
<td>-v</td>
<td>Enables verbosity.</td>
<td>./getcsw -p 2345 -l -v</td>
</tr>
</tbody>
</table>

• We have written some trivial **scripts** to add more functionality:
  
  ◦ `fschedyield.sh`, **in charge of looking for running tasks that are calling the sched_yield() function.**
  
  ◦ `fcalltable.sh`, **in charge of building a table of calls to the scheduler per each system call during their return.**
Generating Hardware Interrupts at will

- An **Interrupt** always preempts \( p \).
Generating Hardware Interrupts at will

- An **Interrupt** always preempts \( p \).
- Whenever an Interrupt is raised, the **GNU/Linux Kernel** handles it by calling `do_IRQ()`.

![Diagram of interrupt handling process]
Generating Hardware Interrupts at will

- An Interrupt always preempts \( p \).
- Whenever an Interrupt is raised, the GNU/Linux Kernel handles it by calling `do_IRQ()`.
- Most Interrupts are maskable: they can be ignored.
Generating Hardware Interrupts at will

- An **Interrupt** always preempts $p$.
- Whenever an Interrupt is raised, the **GNU/Linux Kernel** handles it by calling `do_IRQ()`.
- Most Interrupts are **maskable**: they can be ignored.
- **NMIs** cannot be ignored; they are ideal to test our project.
Generating Hardware Interrupts at will

- An **Interrupt** always preempts \( p \).
- Whenever an Interrupt is raised, the **GNU/Linux Kernel** handles it by calling `do_IRQ()`.
- Most Interrupts are **maskable**: they can be ignored.
- **NMIs** cannot be ignored; they are ideal to test our project.
- To generate NMIs at will, we need to alter the **Kernel IDT**, so that `int $0x2` can be executed with \( DPL = 3 \).

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

- By accounting preemption due to do_IRQ(), we can determine whenever there is an external problem affecting $p$’s throughput.
Preliminary Results

- By accounting preemption due to `do_IRQ()` , we can determine whenever there is an external problem affecting p’s throughput.

- As $IRQS \rightarrow \infty$, the time spent by $p$ increases.
Preliminary Results

- By accounting preemption due to `do_IRQ()` , we can determine whenever there is an external problem affecting $p$'s throughput.
- As $IRS \rightarrow \infty$, the time spent by $p$ increases.
- IRQ-balance can help to diminish this time by reducing the number of interrupts preempting $p$. 

![Graph showing relationship between elapsed time and number of interrupts]
Preliminary Results

- By accounting preemption due to `do_IRQ()` , we can determine whenever there is an external problem affecting `p`'s throughput.
- As `IRQS → ∞`, the time spent by `p` increases.
- `IRQ-balance` can help to diminish this time by reducing the number of interrupts preempting `p`. 

![Regression of Raised Interrupts on Elapsed Time, n=200](image1)

![Raised Interrupts on Elapsed Time, irq-balance, n=200](image2)
Preliminary Results

- Whenever there is a hardware malfunction, the data starts being chaotic.
Preliminary Results

- Whenever there is a **hardware malfunction**, the data starts being chaotic.

- It’s proved that, whenever \( p \) **yields the processor due to a raised interrupt**, at some measurable intervals of time \( t_i \), its time spent in doing its job increases.

<table>
<thead>
<tr>
<th>( t(s) )</th>
<th>Normal</th>
<th>irq-balance</th>
<th>NMIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>17779</td>
<td>26214</td>
<td>75132</td>
<td></td>
</tr>
<tr>
<td>IRQs#</td>
<td>1082867</td>
<td>659144</td>
<td>1701326</td>
</tr>
</tbody>
</table>

Regression of Raised Interrupts on Elapsed Time, \( n=200 \)
Raised Interrupts on Elapsed Time, irq-balance, \( n=200 \)
Raised Interrupts on Elapsed Time, NMIs, \( n=200 \)
Future Work

- Architecture-independent,
- Re-ran the experiments adding some entropy for larger values of \( n \),
- Extend `getcsw` to integrate it with plot facilities,
- Analyse the real impact of our counters inside the Scheduler,
- ...

Future Work

- Architecture-independent,
- Re-ran the experiments adding some entropy for larger values of \( n \),
- Extend `getcsw` to integrate it with plot facilities,
- Analyse the real impact of our counters inside the Scheduler,
- ...

Future Work
Future Work

- Architecture-independent,
- Re-ran the experiments adding some entropy for larger values of \( n \),
- Extend `getcsw` to integrate it with plot facilities,
- Analyse the real impact of our counters inside the Scheduler,
- ...

...
Future Work

- Architecture-independent,
- Re-ran the experiments adding some entropy for larger values of $n$,
- **Extend `getcsw` to integrate it with plot facilities,**
- Analyse the real impact of our counters inside the Scheduler,
- ...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
...

Architectures-independent,
Re-ran the experiments adding some entropy for larger values of $n$,
**Extend `getcsw` to integrate it with plot facilities,**
Analyse the real impact of our counters inside the Scheduler,
Future Work

- Architecture-independent,
- Re-ran the experiments adding some entropy for larger values of \( n \),
- Extend `getcsw` to integrate it with plot facilities,
- Analyse the real impact of our counters inside the Scheduler,
- ...

Context Switching Basis
Tools, APIs & ABIS
Design & Implementation
Execution Examples
Generating Hardware Interrupts at will
Preliminary Results
Future Work
Thanks for coming!
Future Work

- Architecture-independent,
- Re-ran the experiments adding some entropy for larger values of $n$,
- Extend `getcsw` to integrate it with plot facilities,
- Analyse the real impact of our counters inside the Scheduler,
- ...

Future Work

- Architecture-independent,
- Re-ran the experiments adding some entropy for larger values of $n$,
- Extend `getcsw` to integrate it with plot facilities,
- Analyse the real impact of our counters inside the Scheduler,
- ...

...
Thanks for coming!