FINAL PROJECT
COMPONENT-BASED SYSTEMS DESIGN

SANT'ANNA SCHOOL OF ADVANCED STUDIES

MSC IN EMBEDDED COMPUTING SYSTEMS

Unprivileged Sched Deadline

Authors:
Leonardo Lai

Email:
leonardo.lai@santannapisa.it

Date: April 15, 2020

Contents

1 Introduction **3**

2 Background **3**
2.1 SCHED_DEADLINE 3
2.2 Linux PAM 3
2.3 Unprivileged DEADLINE patch 4

3 Access control policy extension **5**
3.1 Cumulative bandwidth 5
3.1.1 Bandwidth radix tree 6
3.2 Group cumulative bandwidth 7
3.2.1 Semantics 8
3.2.2 Implementation 10

4 Source code modifications **10**

5 KConfig **11**

6 Notes **12**

7 Tests **12**

8 Credits **13**

1 Introduction

The Linux Kernel features three real-time scheduling policies, SCHED_RR, SCHED_FIFO and SCHED_DEADLINE, the latter being dynamic-priority. Normally, their access is restricted only to privileged users (CAP_SYS_NICE); since kernel 2.6.12, however, a resource limit called RLIMIT_RTPRIO allows unprivileged processes to change policy and priorities, under the constraint of an upper bound on the priority. The policy SCHED_DEADLINE is excluded from this mechanism, because the deadline tasks run at the highest priority in the system; it is essentially a security feature, preventing unprivileged users to maliciously jeopardize the overall system responsiveness by spawning an arbitrarily large number of such tasks. On the other hand, a practical limitation is a complete impossibility for normal users to exploit deadline scheduling functionalities, not even sparingly. A patch has already been proposed to address this issue: the idea is to define a flexible policy to rule the access from unprivileged users to SCHED_DEADLINE, in a way that prevents them from take more resources than allowed. Yet, that patch is still incomplete, as it does not enforce any limit on the cumulative bandwidth held by a domain; all the admittance tests are instead performed on a per-process basis. This project aims to fill that vacancy, implementing a mechanism to monitor the current overall processor usage for each user/group, and specify a versatile set of rules to limit them.

2 Background

2.1 SCHED_DEADLINE

SCHED_DEADLINE is a CPU scheduler that is available in the mainline Linux Kernel since version 3.14. It implements the *GEDF (Global Earliest Deadline First)* protocol and *CBS (Constant Bandwidth Server)*, hence it is often the preferred scheduling policy for Linux-based real time systems.
A periodic real time task is typically modelled with three parameters: the computation time C, the period T and the deadline D. To enable the SCHED_DEADLINE policy for a task, these values must be specified (in nanoseconds) in a `sched_attr` structure, which is eventually passed as argument to the `sched_setattr` syscall.
A requirement is that:

$$sched\_runtime \leq sched\_deadline \leq sched\_period$$

The kernel checks for this and other conditions to be satisfied when modifying the policy. A thread must be privileged to use SCHED_DEADLINE, since it will execute at the highest priority.

2.2 Linux PAM

PAM (Pluggable Authentication Module) is a flexible module for user authentication in Linux systems. The rationale behind Linux PAM is to decouple the application

logic from the authentication part, making the former easier to develop and the latter more configurable. Linux PAM supports different authentication schemes (from passwords to smart cards) and offers many session handling features. It is typically deployed as a set of shared libraries, and configured via `/etc/pam.conf` or files in the `/etc/pam.d` directory.
One of the modules, `pam_limits`, sets limits on the system resources that can be obtained in a user session. These are specified in `/etc/security/limits.conf` and consist in a set of rules with the following syntax pattern:

$$< domain >< type >< item >< value >$$

where:

- $< domain >$ is the subject to whom the rule is applied. It can be either a user, a group ('@' prefix), a range of users/groups or everyone ('*').
- $< type >$ is either *hard* or *soft*. The soft limits are the ones actually applied, whereas the hard limits represent upper bounds that soft ones can not exceed.
- $< item >$ is the name of the resource controlled by the limit. There are about two dozens of different items that can be configured.
- $< value >$ is the actual value applied to the limit.

The special values *-1*, *unlimited* and *infinity* indicate no limit. The limits become effective when the user performs login.
In the Linux kernel, these limits are represented by the `rlimit` structure. Each process (`task_struct`) contains an array of these elements, one for every resource type. The system `prlimit` (with its flavors `setrlimit` and `getrlimit`) is used to retrieve and modify such limits.

2.3 Unprivileged DEADLINE patch

A few years ago F. Aromolo proposed a set of patches to the Linux kernel and PAM, exploiting the resource limits control mechanism to implement a tunable access control policy for SCHED_DEADLINE features.
The patches introduce six new resource limit types:

- `runtime`: maximum runtime
- `periodmin`: minimum period
- `periodmax`: maximum period
- `deadlinemin`: minimum deadline
- `deadlinemax`: maximum deadline
- `bandwidth`: maximum bandwidth

These limits are used to perform some admittance checks on the deadline parameters when a process modifies the SCHED_DEADLINE parameters, according to the following policy:

1. Privileged processes can arbitrarily change any scheduling policy parameter.
2. When an unprivileged process tries to set the SCHED_DEADLINE policy for another task that is *currently using a different policy*, all the deadline scheduling parameters must lie within the ranges defined by the aforementioned resource limits.
3. When an unprivileged process tries to modify the SCHED_DEADLINE attributes for a task that is *already using SCHED_DEADLINE*, all the new scheduling parameters are checked against *effective* limits, computed as the minimum/maximum between the current value and the corresponding lower/upper rlimits.

These modifications allow unprivileged processes to use the SCHED_DEADLINE scheduling policy under some restrictions, making it potentially easier for a lot of real time applications to exploit the best scheduling features provided by Linux.
Nevertheless, security issues still exist: what happens if an unprivileged task spawns two tasks, each with the maximum allowed bandwidth for its domain? The admittance test succeeds both times because it checks the scheduling parameters of each individual task, which are indeed within the valid range. However, the overall bandwidth used by these tasks is twice the corresponding resource limits, which is clearly undesirable in any system where unprivileged users should not be able to claim an arbitrarily large amount of CPU bandwidth.

3 Access control policy extension

3.1 Cumulative bandwidth

As said in the previous section, a domain-wise mechanism to control the overall CPU usage for real time tasks is needed. The core idea is to define a new resource limit type, named `rlimit_cum_bandwidth`, to set an upper bound on the cumulative bandwidth used by a user or group. Although adding a new resource type is relatively straightforward in the kernel, the semantics of `rlimit_cum_bandwidth` is inherently very different from the other limits. Indeed, while the other limits are tested for each task individually, and are thus *stateless*, checking if the cumulative bandwidth is within the allowed range requires global information about the other tasks too, hence its implementation must adopt a *stateful* approach. In other words, it is necessary to introduce some kind of data structure in the kernel to store the bandwidth used by each domain, and make this information available every time an admittance test is performed. Basically, the test must verify that, for a domain d, its current bandwidth plus the bandwidth variation does not exceed its cumulative bandwidth resource limit[1].

[1] Actually, the implemented policy is more sophisticated than this; the rationale is the same, though

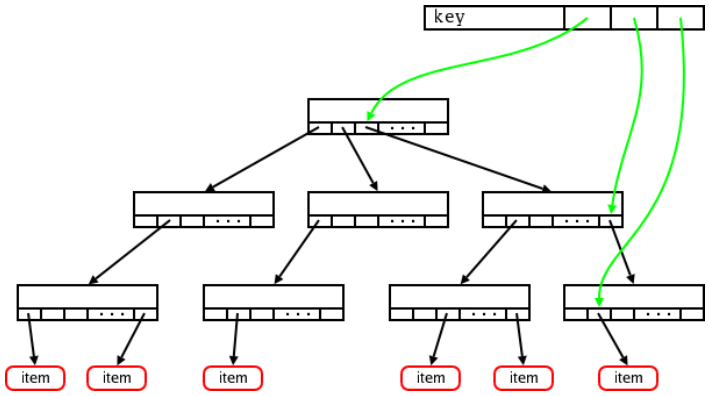

Figure 1: Radix tree: a chunk of the key is used to index within the first-level node, the next chunk to index within the second-level one, and so on...

3.1.1 Bandwidth radix tree

The Linux kernel source tree contains implementations in C of many data structures, and leverages them to efficiently carry out its routines. *Linked list* is probably the most known example, but there are also *queues*, *bit arrays*, *RB trees*, *B+ trees*, *radix trees*, and many others. Each has pros and cons in terms of access time (lookup/insert/removal) and memory footprint, therefore choosing the most suitable data structure strictly depends on the application and usage assumptions.
Let's focus on the case of cumulative bandwidth: what is needed is a structure to associate the identifier of a domain (UID/GID) to a value corresponding to its overall bandwidth usage. We can make the following considerations:

1. Although Linux supports up to 65536 users (UID), any reasonable system many fewer users and groups, hardly beyond a hundred.
2. A user may belong to multiple groups, rarely more than few dozens though.
3. The identifiers of users and groups are often assigned consecutively, so they are bounded within a limited range.
4. Real time jobs execute very often, however their tasks are spawned once and modifications to their deadline scheduling attributes are sporadic.

These remarks make the *Radix Tree* a very good candidate to store the cumulative bandwidth information.
A radix tree is a data structure similar to a trie, where leaf nodes are merged to their parents. Each node contains multiple pointers to child nodes, and the item values are stored in leaves. The key (integer/string) is split in chunks (bits/letters), which are used as indices to navigate through the tree one level after the other. In the Linux kernel, each node contains pointers up to 64 children, that is 6 bits are used to index at each level; empty slots contain null pointers. An example of a radix tree is shown in Fig. 1.
The lookup and insertion operations have a $\mathcal{O}(\log n)$ time complexity (like most trees, indeed), and the typical low-sparsity of the UID/GID values make it efficient on the memory side too.

The cumulative bandwidth kernel patch defines two radix trees, one for users and the other groups, mapping the identifier to the respective bandwidth usage, which is represented in millionths by a `u64` integer attribute.
Consider now the user bandwidth radix tree; there are three distinct events when it is accessed:

- **Login**: all the resource limits are parsed, including the cumulative bandwidth one for the user that just logged in. If `limits.conf` specifies a stricter limit in a group of which the user is member, that value is applied instead.

- **Task policy is set to SCHED_DEADLINE**: If an entry in the radix tree did not already exist for this specific user, a new one is dynamically created. The used bandwidth value is set equal to the previous value plus the new increment, unless that results in a cumulative bandwidth violation. In the latter case, the modification does not take effect and the operation fails as a whole.
  Note: moving from SCHED_IDLE to any other policy, including SCHED_DEADLINE, is always forbidden, hence no bandwidth is reserved in this particular case.

- **Task policy changes from SCHED_DEADLINE to another:** An amount of bandwidth equal to that currently used by the task is subtracted from the corresponding entry in the radix tree.

- **Task is killed**: Within the `do_exit` routine, an amount of bandwidth equal to the one claimed by the dying is subtracted from the corresponding entry in the radix tree.

The whole bandwidth radix tree structure is globally protected by a mutex, necessary to prevent modifications by concurrent tasks that would potentially create data inconsistencies. The lock is expected to have a minimal impact on the system performance, since the event of changing the deadline attributes of a task is considered relatively rare.

3.2 Group cumulative bandwidth

The resource limit interface between the kernel and PAM was developed a long time ago, much earlier than the relatively recent release of SCHED_DEADLINE. Furthermore, it was not designed to take into account semantics different from plain minimum/maximum to be checked once right before granting permissions, without an explicit need for a stateful representation of the system. Even though PAM limits has the notion of groups, its meaning is to apply the limit individually to each member of the group, selecting the strictest one whenever multiple rules apply. In other words, the limit is not bound to the group itself (that would be ideal to realize group bandwidth control), but broadcast to its members.
If one wants to implement more advanced access control features on top of the PAM skeleton, yet without drastically changing the interface for compatibility reasons, some ugly tricks and workarounds become necessary.

Therefore, two additional resource limits are introduced to cope with the necessity of a group-wise bandwidth control: RLIM_GROUP_CUM_BANDWIDTH, to specify the bandwidth limit assigned to a specific group, and RLIM_GROUP_CUM_BW_GID, which bypasses the PAM rlimits shortcomings by declaring the target GID in the limit < *value* > field. Note how the latter information is redundant, as a unique mapping from a group-like < *domain* > to GID already exists; however, the kernel is not aware at any point of the < *domain* > field value that caused a certain resource limit to be set.
Both RLIM_GROUP_CUM_BW_GID and RLIM_GROUP_CUM_BANDWIDTH can be specified only within a limits.conf entry of type group (i.e. starting with @), else it has no effect. RLIM_GROUP_CUM_BW_GID informs the kernel about the GID relative to the next RLIM_GROUP_CUM_BANDWIDTH it will deal with. Specifically, when the kernel receives a RLIM_GROUP_CUM_BW_GID in do_prlimit, it saves that value in a variable, and retrieves it the next type it receives a RLIM_GROUP_CUM_BANDWIDTH instead. For each domain, Linux PAM parses the resource limits in order of type, where each type is a unique integer: since RLIM_GROUP_CUM_BW_GID has a lower identifier than RLIM_GROUP_CUM_BANDWIDTH, the former will always be parsed before (as it should be), regardless of the order in limits.conf.

3.2.1 Semantics

A major issue arises around the meaning of group bandwidth limits. When dealing with user domains, it is easy because a user is always denied access to SCHED_DEADLINE unless a matching entry exists in limits.conf. Following the same rationale, one may say that a group (i.e. its users) whose RLIM_GROUP_CUM_BANDWIDTH is missing in the configuration is excluded from SCHED_DEADLINE. However, this would imply that any user cannot use SCHED_DEADLINE unless *all* of its groups are explicitly allowed to do so, which is very unpractical because a user may belong to dozens of groups. Conversely, a semantic where undefined groups are allowed by default to use deadline features would still force the system administrator to write hundreds of deny rules to achieve a reasonably secure configuration.
A solution which is meaningful, flexible and manageable is possible, but it requires to consider both users and groups at the same time, as shown in Table 1.
The user cumulative bandwidth can be:

- **Undefined**: a limit of type RLIM_CUM_BANDWIDTH does not exist for that user
- **Sufficient**: a limit exist, and the current request to allocate bandwidth would not violate that limit
- **Insufficient**: a limit exist, and the current request to allocate bandwidth would violate that limit

The group cumulative bandwidth can be:

- **All undefined**: No limit of type RLIM_GROUP_CUM_BANDWIDTH exists for any group of which the user is member.

- **1+ sufficient, no over**: The user belongs to at least one group with an explicit `RLIM_GROUP_CUM_BANDWIDTH` rule and sufficient bandwidth to serve the current request, and none of its other groups (if any) are short of bandwidth, i.e. would violate their threshold.
- **1+ over**: The user belongs to at least one group with an explicit `RLIM_GROUP_CUM_BANDWIDTH` rule but an insufficient bandwidth to serve the current request.

In all the cases where the user cumulative bandwidth is beyond limit, the only reasonable policy is to deny access at the admittance test. Similarly, requests shall be declined when at least one group is exceeding its maximum cumulative bandwidth, otherwise the rule would break. In the other cases, where no rule is clearly violated, the idea is to let a fairly permissive policy, where the only necessary condition to pass the admittance test is to satisfy at least one explicit rule (i.e. be a user with sufficient bandwidth, or belong to a group that does). If no rule applies, the task is still denied access, just like the old-fashioned rlimits.
Let's consider a few practical scenarios from the point of view of a sysadmin who wants to customize `limits.conf` to enable a specific deadline access policy:

- **Allow only a certain user**: add an entry of type RLIM_CUM_BANDWIDTH, specifying the user name as domain.
- **Allow only a specific group**: add a RLIM_GROUP_CUM_BANDWIDTH entry, specifying the group name as domain. A RLIM_GROUP_CUM_BW_GID entry specifying the GID is needed too.
- **Allow a specific group to have cumulative bandwidth B, but each member cannot exceed b (with b < B)**: add a RLIM_GROUP_CUM_BANDWIDTH entry with value B to the group, and an entry of type RLIM_CUM_BANDWIDTH with value b to the same group.
- **Allow a specific group to have cumulative bandwidth B, but a determined subset of members cannot exceed b (with b < B)**: create a new group comprising the subset members, and set a stricter RLIM_GROUP_CUM_BANDWIDTH for the subgroup.
- **Allow a specific user U to have more bandwidth than its group G would be allowed to**: although this case is not particularly sound, a viable solution

Groups BW / User BW	All undefined	1+ sufficient, no over	1+ over
Undefined	DENY	ALLOW	DENY
Sufficient	ALLOW	ALLOW	DENY
Insufficient	DENY	DENY	DENY

Table 1: Group cumulative bandwidth: admittance test based on user and groups state

still exists. Create a clone G' of the G group, but without the user U; enforce the group limits to G' instead of G; create a dedicate shallower rule for user U.

As just shown, this policy is very expressive, allowing a lot of configuration flexibility.

3.2.2 Implementation

To implement group cumulative bandwidth control, another radix tree as been added along with the one for user cumulative bandwidth. Since the kernel does not have any dedicated data structure to store rlimits that affect an entire group (they are only stored in the processes' `task_struct`), the cumulative bandwidth upper bound has been embedded in the radix tree nodes, in a `dl_domain_cum_bw` structure:

```
struct dl_domain_cum_bw {
        u64 used;       // currently in-use
        u64 reserved;   // max
};
```

The attribute `reserved` contains the maximum total bandwidth that the corresponding domain (group) can ask, while `used` stores the currently used amount.
Both trees are initially empty. When a user logs in, `limits.conf` is parsed, then entries with `used` equal to 0 and `reserved` equal to the rlimit in the group radix tree, for each group with an explicit rule; the user radix tree is left untouched. When a request for bandwidth is made, a node for the corresponding user is added to the user radix tree; if the admittance test succeeds (i.e. all user and group constraints are satisfied), then the attribute `reserved` is modified in the user tree node, and also in every node of the group tree such that the user is a member and an explicit rule exists.
Similarly to the user bandwidth radix tree, a mutex protects the group one too. The reason behind locking the entire data structure rather than individual nodes is due to the necessity of checking multiple elements one after the other to decide whether to accept or not the bandwidth modification: if the node values change in the meanwhile, inconsistencies may arise later.

4 Source code modifications

In order to implement the functionalities described in the previous section, changes were made both to the Linux Kernel and Linux PAM sources. The reference kernel version is 5.5. The reference PAM version is 1.3.1.

Linux kernel:

- `fs/proc/base.c`: definition of names and description of the new rlimits
- `include/asm-generic/resource.h`: definition of rlimits default values

- `include/linux/sched/deadline.h`: definition of bandwidth data structures and functions
- `include/uapi/asm-generic/resource.h`: definition of rlimits IDs
- `kernel/exit.c`: implementation of bandwidth reclaim when a process dies
- `kernel/sched/core.c`: implementation of bandwidth allocation/reclaim upon change of scheduling policy or attributes
- `kernel/sched/deadline.c`: implementation of utilities to manipulate the bandwidth structures
- `kernel/sys.c`: implementation of cumulative bandwidth limits setting for groups
- `security/Kconfig`: new Kconfig option to select whether to include the unprivileged deadline features when compiling the kernel

Linux PAM:

- `modules/pam_limits/limits.conf`: default configuration (description)
- `modules/pam_limits/pam_limits.c`: definition of rlimits IDs, configuration parsing and setup

The modifications are shared as a set of patches, which can be directly applied on the respective source trees using `patch` or `git-apply`:

```
patch -p1 < <patch_name>
```

After applying the changes, the kernel must be recompiled and installed. Similarly, Linux PAM must be recompiled and reinstalled as well, along with shadows. *Note: make sure to backup everything before reinstalling PAM, since any configuration mistake can potentially lock you out of the system.*
Instructions to reinstall PAM can be found in this link:
`http://www.linuxfromscratch.org/blfs/view/stable/postlfs/linux-pam.html`

5 KConfig

The Linux kernel supports hundreds of different configuration options to choose what features will be compiled in the kernel, or excluded, or built as separate modules. This is possible thanks to *KConfig*, a powerful config tool. It makes the configuration modular, based on a set of files (Kconfig files), and the dependencies easier to manage. `make menuconfig` is a Linux kernel tool leveraging KConfig that offers a user-friendly interface to configure the kernel build options.
For this project, a new configuration option has been added in `make menuconfig`, under the section *security*. This option allows one to enable/disable support for unprivileged SCHED_DEADLINE, including all the previously described features. If disabled, all the code provided by this patch is not even compiled.

6 Notes

- A user should never change its group memberships while running deadline tasks, otherwise bandwidth reclamation may behave incorrectly. Moreover, a logout/login is strongly advised after such changes.
- `pam_limits` advanced <domain> syntax is not explicitly supported.
- RLIM_GROUP_CUM_BW_GID and RLIM_GROUP_CUM_BANDWIDTH must be considered as an atomic pair. Declaring one without the other is always a configuration mistake. While their relative order does not matter at all, the system admin shall never interleave two or more of these pairs, belonging to different groups (e.g. specify GID for X, then GID for Y, then BANDWIDTH for X, then BANDWIDTH for Y), otherwise the behaviour is undefined.
- RLIM_GROUP_CUM_BW_GID and RLIM_GROUP_CUM_BANDWIDTH only support *soft* limits. Specifying *hard* limits is allowed, but has no effect at all.

7 Tests

The correctness of the code has been tested by means of a suite of shell scripts.
Each test is executed individually, and the environment is reset after every experiment. Most of the tests involve spawning one or more processes (typically background 'sleep') and changing their scheduling policy with `chrt`:

```
chrt –d –T <runtime> –P <period> –D <deadline> –p 0 <pid>
```

Some dummy users and groups were created specifically for testing purposes. Here follows the `limits.conf` file used in the trial it. Note that the numerical values are arbitrary, by no means they should be considered as a reference for any real system.

```
*          soft   runtime              1000000000
*          hard   runtime              1500000000
*          soft   periodmin            1500000
*          hard   periodmin            100000
*          soft   periodmax            4000000000
*          hard   periodmax            5000000000
*          soft   deadlinemin          1000000
*          hard   deadlinemin          100000
*          soft   deadlinemax          3000000000
*          hard   deadlinemax          4000000000
*          soft   bandwidth            400000
*          hard   bandwidth            600000
*          soft   cum_bandwidth        600000
*          hard   cum_bandwidth        800000
@wheel     soft   group_cum_bw_gid     1001
@wheel     soft   group_cum_bandwidth  700000
```

The batch contains the following tests:

Description	Result	Pass
Task attempts to use sched deadline with runtime too large	The policy change fails	Y
Task attempts to use sched deadline with period too large	The policy change fails	Y
Task attempts to use sched deadline with deadline too large	The policy change fails	Y
Task attempts to use sched deadline with bandwidth too large	The policy change fails	Y
Two tasks attempt to use sched deadline with cumulative bandwidth too large	The policy change fails for the second task	Y
Two tasks attempt to use sched deadline with cumulative bandwidth within limits	The policy change succeeds	Y
Two tasks (different user, same group) attempt to use sched deadline with group cumulative bandwidth within limits	The policy change succeeds	Y
Two tasks (different user, same group) attempt to use sched deadline with group cumulative bandwidth beyond limits	The policy change fails	Y
A deadline task changes policy to batch, then changes back to deadline	The policy change succeeds and the bandwidth is correctly reclaimed/reassigned in between	Y

Table 2: Functional tests

The above tests are clearly not exhaustive, but cover most of the typical use cases.

8 Credits

The project has been developed by Leonardo Lai (MSc student in Embedded Computing Systems) under the supervision of prof. Tommaso Cucinotta. A preliminary version of the SCHED_DEADLINE unprivileged access, without support for user/-group bandwidth allocation, was proposed by Federico Aromolo (PhD candidate).