rtprio(2) and POSIX(.1b) priorities, and their FreeBSD implementation: A deep dive (and sweep)

Revised – 2024/03/22

Olivier Certner
Kumacom SARL – Consulting
olce.conferences@certner.fr

***Abstract*—Although UNIX's descendants or derivatives are not hard real-time operating systems, some have support for soft real-time through allowing to assign to userland processes higher priorities normally reserved to the system, sometimes coupled with preemption of kernel tasks, making them suitable as a fundation of soft real-time system. POSIX standardized its first real-time extensions in 1993 in a document usually referred to as POSIX.1b. At that time, however, some operating systems already had support for soft real-time in the form of ad-hoc APIs, such as System V Release 4 (SVR4) with its `priocntl(2)` system call and HP/UX with its `rtprio(2)` one. FreeBSD first adopted its own `rtprio(2)` system call in 1994, largely based on HP/UX's with extensions such as idle processes. POSIX.1b extensions concerning processes were implemented later, in 1998, and some preliminary thread support added the next year. Since then, these APIs have been present in the system for applications to use.**

In this paper, we provide a thorough description of both FreeBSD's `rtprio(2)` and POSIX.1b's scheduling interfaces and embark on a journey around FreeBSD's implementation of scheduling priorities. It started with a desire to fix a few apparently simple bugs of `rtprio(2)` and to add some reasonable features and, one thing leading to another, became an almost complete rewrite of this system call and the POSIX.1b's interfaces' implementations. We will expose the many problems that the current implementation has, in terms of POSIX compliance, security and consistency and how we are fixing them. As of this writing, this project is still a work in progress, and we will report about its status during the conference presentation.

I. Introduction

Real-time systems run software that must respond to some external events in a given, short timeframe to ensure correct functioning. Traditionally, these systems are divided in two classes. *Hard real-time* systems are ones where failure to meet deadlines can have catastrophic consequences, or in other words, where a result produced too late is as bad, or worse, than a wrong result. Examples of such are digital control ones for machines in plants and brake controllers in cars. By contrast, *soft real-time* systems that miss deadlines can cause a noticeable quality of service degradation but can continue functioning, and typically previous temporary failures do not affect later operation. An example is a multimedia streaming system, where not sustaining frame rates can be noticed but is tolerable depending on the actual delays and their frequency of occurence. However, this distinction has blurred with hardware performance increases, which give much more room to meet deadlines [1], and finer analyses on the usually benign consequences of a few deadline misses. This makes UNIX-like operating systems excellent candidates as a fundation to implement a large part of real-time systems.

The IEEE POSIX P1003.4 working group was established to standardize real-time features apparently in or before 1989, as can be inferred from [2]. It nurtured three documents, initially named POSIX.4, POSIX.4a and POSIX.4b. The first is about base real-time extensions, including real-time process scheduling, our main concern in this paper, but also virtual memory locking, real-time signals, process synchronization with semaphores, shared memory, interprocess communication via simple queues, clocks and timers, and asynchronous I/O. The second is about threads, colloquially known as *pthread*, and includes thread scheduling and synchronization. The third is about additional real-time extensions, such as timeouts, execution-time clocks, a new sporadic server scheduling policy (`SCHED_SPORADIC`), interrupt control and I/O device control. It is out-of-scope for this paper, as FreeBSD does not currently implement the `SCHED_SPORADIC` scheduling policy and we have not planned to add it in the course of this work. The reader interested in an overview of these historical documents can consult [3].

Around standardization of POSIX.4 in 1993, it was decided that documents produced by the POSIX.x working groups that were amendments to the initial POSIX.1 standard would be regrouped under the same IEEE 1003.1 prefix (see for example [4]). In particular, the three documents mentioned above were respectively renamed to IEEE 1003.1b, 1003.1c and 1003.1d, and were usually referred to as POSIX.1b, POSIX.1c and POSIX.1d from that point. They were approved to standards in 1993, 1995 and 1999. Another document, prepared later, contained "advanced" real-time extensions and was finally published in 2000 as POSIX.1j.

POSIX.1 itself had been approved first in 1988, and then in 1990 as an international standard as ISO/IEC 9945-1. The ".1" in POSIX.1 standed for "System Application Program Interface" part, whereas POSIX.2, first standardized in 1992, was the "Shells and Utilities" part.

This distinction does not exist anymore, as these two parts were all incorporated and superseded by POSIX.1-2001, so POSIX.2 formally no longer exists. It is now possible to unambiguously refer to the POSIX.1 standard as just "POSIX", even if not formally correct, which we will do in most of this article.

Additionally, all amendments to POSIX.1, including the POSIX.1b, .1c, .1d and .1j documents mentioned above, were discontinued as separate documented and incorporated into POSIX.1-2001, as the options Process Scheduling (code PS), Threads (code THR), Thread Execution Scheduling (code TPS), Thread Priority Inheritance (TPI) and Thread Priority Protection (TPP).

From this point on, POSIX.1 documents have exactly coincided with the Base Specifications of the Single UNIX Specification (SUS) from the Open Group, but the timeline for new issues has differed. POSIX.1-2001 (Issue 6) is thus included in the Single UNIX Specification version 2 (SUSv2), POSIX.1-2004 (still Issue 6 but with technical corrigenda) in SUSv3 and POSIX.1-2017 (Issue 7 with technical corrigenda, published in 2018) in SUSv4. POSIX.1's Issue 7, also referred to as POSIX.1-2008, removed the Threads (THR) option, making its features mandatory. It also introduced new options covering two new mutex protocols which involve temporarily altering the priority of threads holding mutexes. These protocols are designated by the constants `PTHREAD_PRIO_INHERIT` and `PTHREAD_PRIO_PROTECT`, whereas the previous behavior of doing nothing special to prevent priority inversion is represented by `PTHREAD_PRIO_NONE`. Coupled with the introduction of robust mutexes, this led to four new options: Robust Mutex Priority Inheritance (RPI), Non-Robust Mutex Priority Inheritance (TPI), Robust Mutex Priority Protection (RPP) and Non-Robust Mutex Priority Protection (TPP). POSIX.1's Issue 8 is in preparation, with Draft 4 just completed in December 2023, and hopefully should materialize as a standard this year or in 2025. It does currently only contain very minor changes with respect to scheduling priorities. Someone interested in reading the prescribed rules on this topic thus just needs to consult a recent version of POSIX.1, such as [5] or equivalently the Base Specifications of SUSv4.

FreeBSD aims to implement POSIX, at least for not too obscure or cumbersome features, and currently does so for all the options we have mentioned above. POSIX has been a great success as it has influenced all UNIX-like implementations. It is however imperfect, especially on scheduling topics. For aspects it specifies either vaguely or improperly, in some cases we have surveyed the behavior of most other prominent open source implementations, including GNU/Linux, illumos, OpenBSD and NetBSD. As hinted in the abstract, FreeBSD also provides its own interface and model of scheduling priorities predating POSIX's, centered around the `rtprio(2)` system call inspired by HP/UX's one, accompanied by the command-line utilities `rtprio(1)` and `idprio(1)`.

In this paper, we describe our journey throughout FreeBSD's implementation of process and threads priorities. It starts with a description of what scheduling priorities are and how they generally influence process and thread scheduling, first by approaching the original `rtprio(2)` interface and the POSIX facilities. It then follows with some anecdotes on strange behaviors observed in the field and what we found lacking in the reports the operating system can produce, which were the motivation to deep dive in this work. We then review and rework the entire implementation of these facilities. Finally, we conclude with a status, including on-going work, and evoke possible future endeavors in this area.

II. The rtprio(2) model

A. Interface

This interface was the first implemented in FreeBSD and its internal code also serves as the basis for the later-implemented POSIX's one. It initially consisted of a single system call, `rtprio(2)`, whose signature is:

```
int rtprio
(int function, pid_t pid, struct rtprio *rtp);
```

The first argument, `function`, can take the following values:

`RTP_LOOKUP` Query the current rtprio specification.
`RTP_SET` Set the rtprio specification.

while the second, `pid`, must normally be the PID of some process, but can instead be 0 to indicate operation on the current thread only. This last feature was introduced as part of the work to allow threads of a process to be scheduled simultaneously on different processors, to be consistent with a similar change done to `sched_setscheduler()`, itself in violation of an arguably poor choice from POSIX we will talk about in subsection III-A. In restrospect, we consider this feature to have been ill-advised as it departs from the treatment of other values[1]. Today, if the current thread's priority only has to be modified and not its process', the `rtprio_thread(2)` system call described below should be used instead.

Finally, the last argument, `rtp`, serves to pass a `struct rtprio`, in or out depending on `function`. This structure represents the `rtprio(2)` view of scheduling priorities, and has the following fields:

`type` The priority *type*, or *class*.
`prio` The priority *level* in the type/class.

The priority model is that the type (or class) specifies the desired scheduling policy, whereas the level indicates the relative strength within the class. Larger values of `prio` indicate a lower priority, following the tradition established by the `nice(2)` interface and continued with `getpriority(2)`/`setpriority(2)`, and similarly to ranks in a competition to obtain processor time.

Possible values for `type` were, before this project started:

[1]This choice also shadows PID 0 used by the kernel but, given the number of threads in process 0, this could be considered a feature.

`RTP_PRIO_NORMAL`
: Stands for the normal class, i.e., traditional UNIX timesharing.

`RTP_PRIO_REALTIME`
: Stands for the real-time class.

`RTP_PRIO_FIFO`
: Stands for the FIFO class, same as the real-time class, but with the additional property that an infinite timeslice is assigned to threads/processes in this class.

`RTP_PRIO_IDLE`
: Stands for the idle class.

`RTP_PRIO_ITHD`
: Class for kernel threads running interrupt handlers.

We explain the role of these classes in subsection II-B below.

Another system call was later added, `rtprio_thread(2)`, which operates on threads instead of processes. Its signature is identical to that of `rtprio(2)` except that `pid_t pid` is replaced by `lwpid_t lwpid`. In FreeBSD, Lightweight Process (LWP) IDs, which are just thread IDs since the removal of M:N threading in 2008, share the overall same value range with Process IDs (types `pid_t` and `lwpid_t` are in fact the same, unsigned 32-bit integers). However, it is easy to determine to which kind a particular (non-negative) ID belongs to since they occupy different sub-ranges of values. Process necessarily have IDs from the range 0 to `PID_MAX` (99999, so at most 5 digits), and threads (LWPs) greater than `THREAD0_TID`, defined to be `PID_MAX+1` (100000, so at least 6 digits). `rtprio_thread(2)` accepts the special value 0 in the argument `lwpid` to mean to operate on the calling thread. By contrast with `rtprio(2)`, using 0 for this purpose does not contradict the operation for other values and does not shadow an existing thread.

Setting the priority of a process via `rtprio(2)` is equivalent to setting the priority of each of its threads via `rtprio_thread(2)`, except that the former does so atomically with respect to other such changes or the creation of new threads in the process. In reality, there is in fact no such thing as a "process priority", each thread having its own priority view. However, `rtprio(2)` is only passed a single `struct rtprio`, so calling it with `RTP_LOOKUP` has to return some synthetic result, which was chosen to be the priority view of its thread with highest scheduling priority. In our view, given this constraint, this choice is the best one since it helps administrators spot processes that have threads with high priorities. Having explained how the priority view of processes is built upon that of threads, we only consider threads in the reminder of this section.

B. Scheduling Types/Classes

Threads put in the real-time class internally have a fixed priority based on the level set with the `prio` field. As soon as they are runnable, they preempt any threads in the normal or idle class, as well as those in their class that have a lower priority (i.e., a higher level in `prio`). Threads in the FIFO class are treated mostly the same. Their priority levels are converted internally exactly as those of the real-time class. The only difference is that these threads are not subject to been descheduled after their timeslice has expired, so other threads having the exact same priority will not run until the former yield (voluntarily or because they called a system call that puts them to sleep). The acceptable values for the priority level in these classes is the range `RTP_PRIO_MIN` (0) to `RTP_PRIO_MAX` (31) (included).

Threads in the idle class also have a fixed priority internally, based on the level set with the `prio` field. By contrast, they only can preempt other threads with lower priority in this class, and are themselves immediately preempted by runnable threads in the other classes. The acceptable values for the priority level in the idle class is the same as for the real-time class.

Threads in the interrupt kernel threads class historically have had a fixed priority internally assigned, but now can be changed by the kernel on particular circumstances (in case of timeslice exhaustion). Attempts to put a thread in this class by any user, including `root`, fails with `EINVAL`.

Threads in the normal class are supposedly scheduled by the system in a timesharing manner. To this end, the scheduler internally assigns them a dynamic priority that depends on their runtime behavior and the nice value of their process. The `rtprio(2)` interface is ill-defined in this regard. It allows to retrieve the current, dynamic internal priority level, which is mostly useless, and gives the appearance that one can set it, although in practice doing so has almost no practical effect since the scheduler recomputes it after a thread has been woken up or when it has exhausted its timeslice. The range of accepted values is 0 (no symbolic constant is provided for it is provided) to `PRI_MAX_TIMESHARE-PRI_MIN_TIMESHARE` (included). The symbolic constants of the higher bound are internal priority levels and are thus subject to change in new releases of FreeBSD. Although this does not happen frequently, the range of internal timesharing priorities was widened during the recent FreeBSD 13 to 14 transition, silently changing the higher bound for `RTP_PRIO_NORMAL` from 103 to 135, which is strictly speaking an unannounced API breakage.

C. Administration

To launch or move a process into the `RTP_PRIO_FIFO` or `RTP_PRIO_REALTIME` class, the calling process must have privilege `PRIV_SCHED_RTPRIO`. This administrative restriction prevents non-authorized users from privatizing the machine, or more likely, lock it up completely since real-time threads have higher priorities than regular kernel threads. In a vanilla FreeBSD, granting this privilege can be achieved by either running the process as `root` or, if the administrator enabled the `mac_priority(4)` module without altering its default configuration, as a user that has been added to the `realtime` user group.

The same kind of restriction exists for the `RTP_PRIO_IDLE` class, for which the corresponding privilege is `PRIV_SCHED_IDPRIO` and the user group used by the `mac_pri-`

ority(4) module is idletime. However, it is possible to disable it entirely by setting the sysctl(8) knob security.bsd.unprivileged_idprio is set to 1. This is a security measure to avoid potential deadlocks given that, at introduction of this feature, the kernel did not internally perform priority propagation. Today, there are still kernel primitives not performing it, and we have not reviewed the extent to which they are used. However, the kernel temporarily raises the internal priority of threads executing within it as they are about to be blocked for resources, so they will eventually run even if nominally in the idle class. Thus, setting the knob to 1 should not in practice present a security risk. At introduction of the mac_priority(4) module, this knob was marked as deprecated, but for the reason we have just developed, we will likely undeprecate it.

III. The POSIX model

A. Overview

The POSIX model can be seen as essentially similar to the rtprio(2) one, and differs on a few mostly small features. What was called scheduling types (or classes) in the latter are called scheduling policies by POSIX. There is no standardized idle scheduling policy, but instead a sporadic server one specified by options Process Sporadic Server (code SS) or Thread Sporadic Server (code TSP), which no open source UNIX operating system implements[2]. Priority levels are ordered in the opposite direction: Greater priority levels mean higher priority. The SCHED_OTHER policy is completely implementation-defined, and is assumed to be used by threads or processes that do not require a real-time scheduling policy, which all UNIX operating systems we know about dedicate to timesharing. There are, however, two notable conceptual additions.

First, POSIX, in [5]'s XSH book, imposes more stringent rules on priority levels and how to schedule threads from them. It states that priority levels are global and that the system must conceptually organize runnable threads into ordered lists, with exactly one list per priority level. When the system has an available processor, it must select the thread at head of the highest priority non-empty thread list, irrespective of its scheduling policy. Additionally, depending on the reason why threads are added to some priority list, they must be so either at head or tail. We will elaborate on this point in subsection III-C.

Second, independent scheduling attributes are attached both to processes and threads. Whether the scheduling of a given thread is affected by its process' attributes depends on its *scheduling contention scope*. Threads that have *system contention scope* are scheduled solely based on their own scheduling attributes, irrespective of their process'. Threads with *process contention scope* are scheduled according to both. The process' settings affect system-wide scheduling of the *kernel-scheduled entities* (KSE) that in the end run the process' threads, and threads scheduled by a KSE compete within it based on their respective settings[3]. This is intended to model a two-level scheme where on one hand the kernel schedules KSEs and on the other hand a userland threading library schedules threads as seen by the applications onto the KSEs.

This distinction between system and process contention scope seems to be mostly obsolete these days in practice. We do not know of any current open source implementation actually supporting process contention scope. FreeBSD's former M:N threading implementation [6], which POSIX refers to as "the hybrid model" of thread implementation, was called KSE and had more elaborate rules, but was finally discarded for its complexity. Today, FreeBSD will accept requests to change a thread's contention scope attribute to PTHREAD_SCOPE_PROCESS without them having any practical effect, instead of returning an error with code ENOTSUP as, for example, glibc does.

Additionally, if we always assume system contention scope, processes' scheduling attributes should never have any effect, and consequently the functions that operate on them should be essentially useless. However, this is inconsistent with the mandated behavior of functions setpriority() and nice(), which operate on processes but must also affect all their system-scoped threads. Moreover, if we consider that, from a scheduling point of view, a non-multi-threaded process can be equated with its single thread, the functions operating on processes should also affect that single thread. This view was prevalent when threading was added in various operating systems, and explains why none of the implementations we have surveyed follow the implication we have just mentioned. We will detail our choices in this respect, and compare them with those of other implementations, in subsubsection IV-C5.

B. Scheduling Policies

The standardized scheduling policies are, expressed in terms of threads since we assume system contention scope for threads as we have just explained:

SCHED_FIFO
: Threads at the same priority level are scheduled in FIFO order. Each one runs to completion or up to some blocking system call. Threads of higher priorities preempt threads of lower priority.

SCHED_RR
: Similar to SCHED_FIFO, except that threads cannot run more than some time quantum in a row, in which case they are put back to the tail of their priority's level list.

SCHED_OTHER
: Was described at beginning of subsection III-A.

SCHED_SPORADIC
: Was described at beginning of subsection III-A.

[2]Linux has a specific deadline policy, SCHED_DEADLINE, meant to address a similar high-level problem, but differing substantially.

[3]The POSIX description in [5]'s XSH book is much more vague than that and consists only of a single example. It is our interpretation that this example strongly hints at this rule.

Each scheduling policies has its own parameters. Together, the specification of a policy and its parameters form a particular value of *scheduling attributes*. POSIX specifies that the policies `SCHED_FIFO` and `SCHED_RR` only have a single parameter, a priority level (`sched_priority`, see subsection III-C below). This priority level is also used by policy `SCHED_SPORADIC` together with other parameters. Implementations must provide at least 32 distinct priority levels in each of these three policies. It is implementation-defined which parameters are used by policy `SCHED_OTHER`, but in practice, all the operating systems we looked at make use of the priority level, and only it, as the associated parameters, but they implement different ranges of admissible values, sometimes reduced to a single value.

In FreeBSD, policy `SCHED_FIFO` is completely equivalent to `rtprio(2)`'s class `RTP_PRIO_FIFO`, `SCHED_RR` to `RTP_PRIO_REALTIME` and `SCHED_OTHER` to `RTP_PRIO_NORMAL`, since these policies are in fact internally implemented by these classes. Thus, policy `SCHED_OTHER` has as its parameters only the priority level, like `SCHED_FIFO` and `SCHED_RR`. However, since POSIX mandates that greater levels must indicate higher priorities, `rtprio(2)`'s levels cannot be used directly and have to be translated. The current transformation operation is simple and applies to all policies, thanks to all priority level ranges having a minimum value of 0, be they on the `rtprio(2)` or POSIX side. It is an involution, so starting from any side: Levels are mirrored around 0 and then translated so that the smallest value (the initial maximum level) coincides with 0 again. As an example, for policies `SCHED_FIFO` and `SCHED_RR`, value 0 is transformed into value 31 of the `rtprio(2)` range, 1 into 30, 2 into 29 and so on.

FreeBSD currently does not implement any non-standard policy but will soon do so. As they are relevant to FreeBSD's Linuxulator, here is the list of non-standard policies Linux currently implements:

`SCHED_IDLE`
: An idle policy. It differs with FreeBSD's `RTP_PRIO_IDLE` in that it admits a single priority level (0) and that threads with other policies do not necessarily preempt them immediately.

`SCHED_BATCH`
: A timesharing policy, similar to `SCHED_OTHER`, indicating that the process should never be considered interactive.

`SCHED_DEADLINE`
: A deadline policy, suitable for real-time treatment of aperiodic events, like `SCHED_SPORADIC`, but substantially differing from it in its operation.

In Linux, the admissible priority levels for `SCHED_FIFO` and `SCHED_RR` are the range 1 to 99. For all other policies, only 0 can be used. This made Linux fully POSIX-compliant with respect to the constraints on priority levels until the introduction of `SCHED_DEADLINE`, where threads so scheduled always preempt threads with different policies although they have a priority level of 0, indicating lowest priority.

POSIX specifies that, by default, just-created processes inherit the scheduling attributes of their parent if the latter follows the `SCHED_FIFO` or `SCHED_RR` policy, but says nothing about other classes. All systems we have reviewed always abide by this rule regardless of the parent process' scheduling policy.

C. Scheduling Interface

1) Live Processes: We start by giving an exhaustive list of all functions that can modify scheduling attributes of live processes:

`sched_setscheduler()`
: Set a process' scheduling policy and its parameters, together forming the scheduling attributes.

`sched_setparam()`
: Set the parameters associated with a process' scheduling policy.

Each of these functions change settings that can be retrieved by another function whose name can be obtained by substituting `set` with `get`.

Parameters for any policy are communicated through a `sched_param` objects, which at a minimum has a `sched_priority` field to contain the priority level in the policy, if applicable. Additional ones are required when the implementation supports the sporadic server policy. Acceptable level values for a given policy are not standardized, nor the `SCHED_RR` policy's time quantum. They can be determined at runtime thanks to the following functions:

`sched_get_priority_min()`
: Return the minimal priority level value for a given scheduling policy. quantum for the `SCHED_RR` policy.

`sched_get_priority_max()`
: Return the maximal one.

`sched_rr_get_interval()`
: Return the current time quantum for the `SCHED_RR` policy.

Another attribute, part of earlier POSIX.1 versions, can influence scheduling: The nice value. According to the standard, it is a per-process value between 0 and $2 \cdot \{\texttt{NZERO}\} - 1$, where the symbolic constant `{NZERO}` must be at least 20. In practice, in all the surveyed systems, `{NZERO}` is defined to be 20[4]. All BSD systems also accept $2 \cdot \{\texttt{NZERO}\}$ as a valid, distinct value, as they always have historically. Linux's implementation of threads causes it to have a nice value per thread. POSIX leaves the exact effect of the nice value mostly at the implementation's discretion. Its only requirement is that it must not influence scheduling of threads or processes with policies `SCHED_FIFO` or `SCHED_RR` in any way. In all the surveyed implementations, the nice value only has an effect coupled with the `SCHED_OTHER` policy or other ones that can be assimilated to timesharing,

[4]In FreeBSD, it has always been 0, as inherited from 4.4BSD, and is not exposed as a symbolic constant to users although it should. This is going to be fixed in this project's course.

i.e., SCHED_BATCH in Linux, and SCHED_IA (interactive) and SCHED_FSS (fair-share) in illumos, with for the latter the addition of the fixed-priority one (SCHED_FX).

Modification of the nice value can be done through the following functions:

`nice()` Increment (or decrement) the nice value of the calling process.

`setpriority()` Directly set the nice value of some process, or all processes in a process group or belonging to some user.

These functions take and/or return *offset nice values*, that is, the nice value minus {NZERO}, whose range is thus typically -20 to 19 or 20. The counterpart of `setpriority()` for reading the (offset) nice value is `getpriority()`. Function `nice()` not only modifies the value but also returns the new (offset) nice value after incrementation.

2) Live Threads: This is the list of functions that can modify scheduling attributes on live threads:

`pthread_setschedparam()`
Set a thread's scheduling policy and parameters.

`pthread_setschedprio()`
Change a thread's `sched_priority` attribute.

Function `pthread_setschedparam()` has a counterpart to read the scheduling attributes, called `pthread_getschedparam()`. Function `pthread_setschedprio()` has no such counterpart.

Users must watch out for discrepancies between the process and thread interfaces. In particular, names are confusing: `sched_setscheduler()` maps to `pthread_setschedparam()`, whereas `sched_setparam()` is similar to but subtly different from `pthread_setschedprio()`. As some thread's priority is changed using the above functions, if the thread is runnable, it must be conceptually moved from its old priority's list to its new priority's one. In this case, it must be inserted at tail except if the change of priority was performed through `pthread_setschedprio()` and, as a result, either its priority decreased, in which case insertion must happen at head, or stayed the same, in which case no removal and insertion should take place and the thread should keep its position in the (unchanged) list. Thus, `pthread_setschedprio()` performs a different thread re-scheduling than `pthread_setschedparam()`, or the `sched_setparam()` or `sched_setscheduler()` process functions when only changing the priority level. The other difference between `sched_setparam()` and `pthread_setschedprio()` is straightforward: The former allows to change all parameters, whereas the latter can only change `sched_priority`.

Fortunately, in practice, these differences are mostly unimportant. The latter does not matter for most policies as they only have a single parameter, `sched_priority`. The former can be important to implement userland mechanism to bound priority inversion when using semaphores or if the operating system does not provide the mutex protocols `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT`. But besides being a relatively rare case, the implementations we have surveyed do not implement the re-scheduling distinctions above and have all these functions behave as the standard-prescribed `pthread_setschedprio()`.

3) Thread Attributes: Concerning threads, it is also possible to change the attributes related to scheduling in some `pthread_attr_t` object to influence a future thread creation with the following functions:

`pthread_attr_setinheritsched()`
Set the `inheritsched` attribute indicating whether the created thread will inherit the scheduling policy, its parameters and the thread contention scope, or if those set in the attribute objects will be used instead. Possible values are `PTHREAD_INHERIT_SCHED` and `PTHREAD_EXPLICIT_SCHED`.

`pthread_attr_setschedpolicy()`
Set the scheduling policy to use on thread creation, if at that time the `inheritsched` attribute's value is `PTHREAD_EXPLICIT_SCHED`.

`pthread_attr_setschedparam()`
Set the scheduling parameters to use on thread creation, if at that time the `inheritsched` attribute's value is `PTHREAD_EXPLICIT_SCHED`.

`pthread_attr_setscope()`
Set the scheduling scope to use on thread creation, if at that time the `inheritsched` attribute's value is `PTHREAD_EXPLICIT_SCHED`.

Each of these functions has a counterpart to retrieve the information it sets, whose name can be obtained by substituting `set` with `get`. We can draw a parallel between some of these functions and those that operate on live threads. The `pthread_attr_setschedpolicy()` and `pthread_attr_setschedparam()` functions combined provide the same functionality as `pthread_setschedparam()`, whereas `pthread_attr_setschedparam()` has roughly the same as `pthread_setschedprio()`.

4) Spawn Attributes: Process spawning via `posix_spawn` can be configured by passing a `posix_spawnattr_t` attributes object, whose scheduling-related attributes can be set by the following functions:

`posix_spawnattr_setflags()`
Allows to set a number of flags controlling which process attributes must be changed as prescribed by the `posix_spawnattr_t` attributes object. The scheduling-related flags are:

`POSIX_SPAWN_SETSCHEDPARAM`
If set, apply the parameters recorded by `posix_spawnattr_setschedparam()`.

`POSIX_SPAWN_SETSCHEDULER`
If set, apply the parameters recorded by both `posix_spawnattr_setschedpolicy()` and `posix_spawnattr_setschedparam()`.

`posix_spawnattr_setschedpolicy()`
Set the scheduling policy.

`posix_spawnattr_setschedparam()`
Set the scheduling parameters.

Again, each of these functions has a counterpart to retrieve the information it sets, whose name can be obtained by substituting `set` with `get`.

5) Mutexes: We hereby list the functions operating on mutexes influencing how the processes holding them are scheduled:

`pthread_mutexattr_setprotocol()`
Sets the priority inversion avoidance protocol that the mutexes created with the passed attributes object should follow. Possible values are:

`PTHREAD_PRIO_NONE`
Do nothing. This is the default value.

`PTHREAD_PRIO_INHERIT`
Have holders of such a mutex run at least at the highest priority among all threads blocking on this lock.

`PTHREAD_PRIO_PROTECT`
Have holders of such a mutex run at least at the priority ceiling for this lock.

`pthread_mutex_setprioceiling()`
Sets the priority ceiling used for mutexes that use the `PTHREAD_PRIO_PROTECT` priority inversion avoidance protocol.

Again, and as always for functions modifying an attributes object, each of these functions has a counterpart to retrieve the information it sets, whose name can be obtained by substituting `set` with `get`.

IV. Rationalizing FreeBSD's Priorities

A. The Starting Point

We wanted to make sure that certain batch jobs, including long running compilations, would not disturb other interactive and batch executions. Of course, it is possible to give a high nice value to the former, but on FreeBSD this currently does not have much practical effect for CPU-bound jobs. So we turned instead to using the idle class instead, so that the special batch jobs can always be immediately preempted by any other normal process.

Launching new processes in the idle class is as easy as typing, e.g.:

```
$ idprio 16 ./myjob
```

at a command shell, thanks to the provided `idprio(1)` command-line utility. This currently requires privilege, as detailed in subsection II-C.

If these jobs are to be launched always by some particular user, an alternative is to use login classes to automatically set the priority of this user's new login shells, such as those spawned by `su -`. To this end, the administrator can change the `login.conf(5)` configuration file by using the `priority` capability for the target user. This capability is documented to accept (offset) nice values, but also accepts higher or lower values that are interpreted as requests to put the calling process into the idle or real-time class respectively[5]. The intervention of an administrator will soon not be required anymore, as we have recently committed changes that make the login class machinery take into account a `priority` capability in each user's `~/.login_conf` file.

One can also use the command `idprio(1)` (or `rtprio(1)` indifferently) to see in which class some process is. A job launched by the above command and executing such a command will report:

```
idprio: idle priority 16
```

Strangely, we noticed that launching `idprio(1)` on normal processes returned something a bit different:

```
idprio: normal priority
```

where no priority level is reported. This piqued our interest, and thanks to the simplicity of the `rtprio(2)` interface, in minutes we had our own program to query the priority status of a process and that would always report the priority even if the target process was in the normal class. Launching it from a shell and with an argument of 0 for the PID, we got:

```
$ ./prio
Current priority: 0.
RT prio: Type: RTP_PRIO_NORMAL, prio: 0.
```

where the first line reports the result of calling `getpriority(2)` and the second that of calling `rtprio(2)`. Curious about the reported priority level, we soon built a second program to change it, since there is no utility similar to `idprio(1)` or `rtprio(1)` for the `RTP_PRIO_NORMAL` class. With it, we tried to set the priority level to 10:

```
$ ./set_rtprio 0 NORMAL 10
Current priority: 0.
RT prio: Type: RTP_PRIO_NORMAL, prio: 10.
```

which seemed to work, as the reported priority immediately after having called `rtprio(2)` with `RTP_SET` was the requested one. However, reissuing almost immediately the first program above (`./prio`) gave the same first result, as if we had never changed the level in the meantime.

While we were thinking about what could be going on, we nonchalantly launched `./prio` again several times in a row, which at some point and to our astonishment, reported this:

```
$ ./prio
Current priority: 0.
RT prio: Type: RTP_PRIO_NORMAL, prio: 1.
```

i.e., the priority level had increased. Continuing, we could make that level slowly but steadily increase. But as we would stop launching it for a while, the first subsequent launch would report a lower priority than the previous launch, usually 0 after enough wait.

Something was definitely going on. `rtprio(2)` was probably reporting dynamic priorities for processes in the `RTP_PRIO_NORMAL` class, which a later analysis would confirm. These are most often useless to users or administrators as

[5]This has been the case since 1998. We have recently changed the manual page to reflect the real range of admissible values and their meaning.

they are kernel-controlled. Worse, setting a level seemed to have no effect, but we were worried it could affect at least temporarily process scheduling. A later analysis confirmed it did not, although the level was temporarily stored in a `struct thread` field always recomputed before the scheduler actually makes a decision based on it.

We were also curious about the `RTP_PRIO_ITHD` class. Having listed processes with `ps -o rtprio -axl` and sorted them by their `PRI` value (the kernel's internal value), we took the first two processes having highest priorities, and were granted by this (most columns elided for clarity):

```
RTPRIO              PID  PRI  NI  COMMAND
kernel:4294967248   14   -68  0   [usb]
intr:48             12   -52  0   [intr]
```

which is somewhat surprising given that:

- The `prio` field of `struct rtprio` is an unsigned byte.
- There is no kernel class, and no `RTP_PRIO_KERNEL` symbolic constant.

We leave as an exercize to the curious reader to list all threads pertaining to the above processes and look at their `RTPRIO` and `PRI` values, e.g., by typing (with the appropriate process IDs on its system):

```
$ ps -o tid,rtprio,pid,pri,ni,command \
-p 12 -p 14 -H | tail +2 | sort -k4 -n
```

and guess what is their relationship to the same values listed for their process. As a bonus question, can you figure out a relationship between `PRI` and `RTPRIO` for the same thread/process?

Running our small `./prio` program on these two processes, we got:

```
$ ./prio 12
Current priority: 0.
RT prio: Type: RTP_PRIO_ITHD, prio: 0.
$ ./prio 14
Current priority: 0.
RT prio: Type: RTP_PRIO_NORMAL, prio: 0.
```

which is again astonishing given it appears in contradiction with the `PRI` values reported by `ps` above, which seemed to indicate that process 14 (`[usb]`) had a higher priority than process 12 (`[intr]`).

Finally, we noticed that, in repeated launches, `./prio 12` most of the time reports 0 for the `prio` field, but very rarely reports an odd, sometimes big value.

We had seen enough. There were probably a lot of strange behaviors to analyze in the area, bugs to fix, documentation to write. Having been interested in scheduling matters for a long time [7], we thought it would also be a good entry point to FreeBSD's schedulers.

B. Goals

We first expose a list of unsurprising high-level goals stemming from generally good design, and then more precise ones prompted by our peculiar experience of using the scheduling priorities interfaces, as related notably in the previous subsection, and analyzing and reviewing their implementations, which led to the series of changes reported in the next subsection.

1) High-Level: First, FreeBSD needs to provide fully documented scheduling priority interfaces with stable and predictable behavior. Moreover, considering that the users of these interfaces are or will mostly be developers of real-time applications such as multimedia ones, users wanting to launch processes that should not disturb the rest of the system or on the contrary should imperatively always run, or administrators wanting to efficiently monitor and understand system behavior, these interfaces must depend on documented behavior and not on their chosen implementations, which can be subject to change, as recently happened (see subsection II-B).

Second, we intend to continue supporting both the `rtprio(2)` and POSIX interface, the first as FreeBSD's native one and with the second's implementation on top of the first. Among the reasons for this choice, there are backwards compatibility, an economy of changes since this preserves the existing design, but also some leeway to experiment with changes which are not clearly, or definitely not POSIX-compliant. At the same time, we intend that both interfaces mostly provide the same functionality in the same fashion, to the extent permitted by POSIX.

Third, reporting tools available to the user and administrator must be documented and, to avoid cognitive burden, exhibit information in exactly the same format, or at least the spirit, used to change the policies through the programming interfaces.

Four, there must ideally be not foot-shooting possible in the default configuration. This includes an administrator or user inadvertently locking the machines or messing with the priority of kernel processes/threads.

2) Peculiar: From the high-level goals above, given the particular nature of the problem, we have drawn some additional ones.

First, we intend to change the semantics of levels in class `RTP_PRIO_NORMAL` to coincide with nice values. Users and administrators are uninterested in having some dynamic priority returned in a range that is not well defined and can change between releases. Using nice values instead is familiar and stable, both for reporting but also to give real meaning to change requests through the `rtprio(2)` API. Logically, this change should be propagated to scheduling policy `SCHED_OTHER`, as it is the counterpart of class `RTP_PRIO_NORMAL`. However, we are not completely sure doing so can be considered POSIX-compliant, since according to it the system must choose a thread from the highest priority level list to run on an available processor, and even niced processes are at some point executed even if there are other higher priority timesharing processes. That said, Linux and illumos strictly speaking also violate this rule.

Second, we want to block tampering with kernel processes/threads scheduling attributes by default, even by the administrator. But, in the course of experimentations,

it may be useful to allow them to be changed, which could be enabled through some `sysctl(8)` knob.

C. Fixes and Improvements

In this subsection, we list all changes we have already coded and tested, in a format similar to release notes. At time of this writing, they have not been publicly disclosed yet.

1) Proper Separation of Concerns: Do not use internal scheduler classes as the value returned in field `type` of `struct rtprio`.

Some kernel thread was using an internal function of the `rtprio(2)` implementation to change its own priority, blocking other changes. Use the scheduler API instead.

2) Input Validation: The scheduling policy passed to the POSIX API for threads is now checked, and rejected if unknown instead of being treated as `SCHED_OTHER`.

If scheduling attributes passed at thread creation cannot be honored, e.g., because they are incorrect or the creating thread does not have enough privileges, thread creation now fails as mandated by POSIX.

3) Thread Attributes: Stop resetting the scheduling parameters (i.e., currently, the priority level) when setting the scheduling policy via `pthread_attr_setschedpolicy()`.

Enforce invalid defaults on missing scheduling attributes if they have to be reset at thread creation. More concretely, callers that set the `inheritsched` attribute to `PTHREAD_EXPLICIT_SCHED` will be forced to call both `pthread_attr_setschedpolicy()` and `pthread_attr_setschedparam()` at least once, else thread creation with these thread attributes will fail.

4) Translation of Interfaces and Convergence: Factorization of in-kernel duplicated formulas, between the `rtprio(2)` space and that of POSIX for common classes/policies (FIFO, realtime).

Suppress all translation in userland, more precisely in the `libthr` threading library, and have them performed by the kernel only (see also IV-C5).

Implement a new `SCHED_IDLE` scheduling policy, with 32 priority levels, as a straightforward translation of the `RTP_PRIO_IDLE` scheduling class.

Return `EOVERFLOW` in case translation fails (no corresponding classes). We intend that, as this project ends, translations are always possible between internal priorities, `rtprio(2)` and POSIX ones, so `EOVERFLOW` should never be returned in practice. Still, applications will have to be prepared to handle this case as any error case, or specifically, since we may not be able to maintain this property in the future.

The way has been paved so that it is possible to easily change the range bounds of admissible priorities for each (POSIX) scheduling policy, which we may do as a consequence of a strict interpretation of the standard regarding priority levels.

Assert that there is exactly one internal priority level reserved for one level of the `RTP_PRIO_FIFO`, `RTP_PRIO_REALTIME` and `RTP_PRIO_IDLE` classes, since the implementation assumes this property in a few places (now very few, following code factorization).

In Linuxulator's `ioprio_get()`'s implementation, add mapping of processes in the `RTP_PRIO_FIFO` class to the appropriate ioprio priority exactly as those in class `RTP_PRIO_REALTIME`. Generally improve Linuxulator's ioprio conversion to `rtprio(2)`.

Fix the Linuxulator's priority level conversions between Linux's POSIX levels and our native ones, which was mathematically slightly incorrect and violated layers (assumed a particular translation between POSIX and `rtprio(2)` levels). This also entailed suppressing duplicated code.

The Linuxulator now supports the (Linux's) scheduling policies `SCHED_IDLE` and `SCHED_BATCH`. The first one has only a single admissible priority level (0), which is currently translated into a value near the median of FreeBSD's admissible range for `SCHED_IDLE`. Policy `SCHED_BATCH` is for now simply translated to FreeBSD's `SCHED_OTHER`, since the ULE scheduler is normally able to automatically determine whether some thread is to be considered interactive. We are nonetheless planning to introduce a native `SCHED_BATCH` by allowing to pass a hint to the scheduler.

5) System Calls: Provide two new scheduling system calls operating on threads, `thr_sched_set()` and `thr_sched_get()`, that accept POSIX scheduling attributes natively. Modify the `thr_new()` system call to accept them as well. This allows to suppress translation code duplication (see also IV-C4). Make `thr_sched_set()` accept a special policy number, `SCHED_CURRENT`, to indicate that only the scheduling parameters are to be changed, in support of `pthread_setschedprio()`.

The computation of some process' synthetic scheduling attributes based on the maximum priority of its threads was wrong, because the ordering of `rtprio(2)` classes is not congruent with that of their numerical values. Moreover, its code was duplicated in the Linuxulator. It has been factorized and fixed, and is now based on the scheduler's internal priorities in order to cirmcumvent translation problems for non-exported scheduler priority ranges.

Change the behavior of the POSIX process API to always operate on all threads of a process, instead of just the main thread. With this change, FreeBSD will behave like illumos and NetBSD. Linux is in a league of its own, as it operates on a process' main thread except if the passed PID is 0, in which case it does so on the calling thread instead. Fix the Linuxulator to behave exactly this way.

Allow callers of the POSIX process API that mean to modify settings to be informed of a concurrent change of priority affecting some of the process' threads in a way that can prevent the requested change to be completely performed (because, e.g., some thread had its priority lowered and the caller has insufficient privileges to raise it), in which case they may decide to reissue the call later on to obtain some "definitive" result (success or failure).

6) Privilege Checks: Factorize privilege checks to assign a process to the `RTP_PRIO_FIFO`, `RTP_PRIO_REALTIME` or `RTP_PRIO_IDLE` classes. This fixes insufficient checks in Linuxulator's implementation of `ioprio_set()`.

Remove incoherent checking for privilege `PRIV_SCHED_SETPOLICY` at thread creation when the scheduling attributes specify to assign the process to `RTP_PRIO_FIFO` or `RTP_PRIO_REALTIME`, instead of checking for `PRIV_SCHED_RTPRIO`.

Remove unused `PRIV_SCHED_SETPARAM` privilege.

Always require privilege to be able to raise priorities, whether by changing the class or the level. For consistency with `setpriority()`, we have reused the `PRIV_SCHED_SETPRIORITY` privilege already checked to decrease the nice value. Make module `mac_priority(4)` grant this privilege to users that are members of group `realtime`, because without they would not be able to actually put a process in the realtime class. A consequence of this change is that processes put in the idle class cannot rise their level within that class even if they have privilege `PRIV_SCHED_IDPRIO`.

Remove the additional, inconsistent check for the `PRIV_SCHED_SET` privilege in the implementation of `sched_setscheduler()`, and then remove this privilege entirely as it is no more of use. Generally fix insufficient privilege checks for tampering with processes or just obtaining their current attributes, in particular raising the priority level or changing the scheduling class.

7) Miscellaneous: Internally represent POSIX scheduling attributes with a new structure, `struct sched_attr`, containing the scheduling policy and its parameters.

Make sure that the necessary infrastructure is present on the kernel side to support versioning of the `struct sched_attr` structure and scenarios where an old `libthr` is used on a newer kernel (e.g., compatibility jails), and even converse ones where `libthr` is newer (this part is not implemented, but now could be if desired). To this end, we ensured that the threads' scheduling-related system calls properly support versioning of the parameter holding a `struct sched_attr` structure. Decouple the definitions of this structure as seen by userland (in `libthr`) and by the kernel, since the latter will have to support more than one version as the current one is bumped.

Make `libthr` cache results of the `sched_get_priority_min()` and `sched_get_priority_max()` system calls, and remove its hardcoded priority bounds. Use this cache also to outright reject unknown policies that some caller wants to set on a thread attributes object (via `pthread_attr_setschedpolicy()`).

Add checks that priority levels are in compliance with POSIX for `SCHED_FIFO` and `SCHED_RR`: non-negative, at least 32 levels.

Add an internal explicitly invalid priority value, to be used as a sentinel in several cases.

Cleanup header pollution by removing the inclusion of `sys/sys/rtprio.h` from `sys/sys/proc.h`. Also remove all mentions of `struct rtprio` in the latter.

Add the proper event audit tags to system calls related to scheduling. These events have existed for a long time (since 2006 for most), but have been used only in the Linuxulator.

Enforce the separation of concerns between the `sys_*()` and `kern_*()` functions that implement the `thr_new()` system call. Functions in the first family are system calls' entry points, whereas those in the second are internal kernel functions that operate on structures already allocated from kernel memory.

V. Status and Future Work

As of this writing, the changes we have done so far (see subsection IV-C) have been lightly tested. The next steps are to introduce systematic tests for them in FreeBSD's test suite and have them reviewed. Implementations and the scheduler were so intertwined that is was hard to come with independent, incremental changes that can make sense on their own. Moreover, some of them are significantly modified by later ones in the series, so we felt that submitting the first ones separately for review would mostly waste reviewers' and the author's time. Given the amount of changes, we hope that the necessary code reviews will not take too much time. Some of the design choices we made are debatable, such as whether policy `SCHED_OTHER` should only have a single priority level, and we may change our views on them as long as the overall consistency is maintained and they do not violate the high-level goals stated in subsection IV-B1.

A new round of changes concerning the schedulers' view of priorities is in progress. Besides cleaning sometimes ambiguous internals, they will enable accurate, readily comprehensible and stable reporting of thread's scheduling attributes. To this end, they will be complemented by a new `rtprio(2)` class for kernel threads.

Acknowledgments

We would like to thank the FreeBSD Foundation for sponsoring this work, and in particular Ed Maste for allowing us to work on this project.

References

[1] G. Lipari and L. Palopoli, "Real-time scheduling: from hard to soft real-time systems," *ArXiv*, vol. abs/1512.01978, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:12204476

[2] B. O. Gallmeister and C. Lanier, "Early experience with POSIX 1003.4 and POSIX 1003.4 A," *[1991] Proceedings Twelfth Real-Time Systems Symposium*, pp. 190–198, 1991. [Online]. Available: https://api.semanticscholar.org/CorpusID:206524607

[3] M. G. Harbour, "REAL-TIME POSIX: AN OVERVIEW," 1993. [Online]. Available: https://api.semanticscholar.org/CorpusID:1719497

[4] B. O. Gallmeister, *POSIX.4: programming for the real world*. USA: O'Reilly & Associates, Inc., 1995.

[5] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, 2018.

[6] J. Evans, "Kernel-Scheduled Entities for FreeBSD," 2000. [Online]. Available: https://api.semanticscholar.org/CorpusID:264752129

[7] O. Certner, "Programming Environment, Run-Time System and Simulator for Many-Core Machines," Theses, Université Paris Sud - Paris XI, Dec. 2010. [Online]. Available: https://theses.hal.science/tel-00826616