Linux Scheduler Documentation

The kernel development community

Jan 15, 2023

CONTENTS

CHAPTER

ONE

COMPLETIONS - “WAIT FOR COMPLETION” BARRIER APIS

1.1 Introduction:

If you have one or more threads that must wait for some kernel activity to have reached a point or a specific state, completions can provide a race-free solution to this problem. Semantically they are somewhat like a pthread_barrier() and have similar use-cases.

Completions are a code synchronization mechanism which is preferable to any misuse of locks/semaphores and busy-loops. Any time you think of using yield() or some quirky msleep(1) loop to allow something else to proceed, you probably want to look into using one of the wait_for_completion*() calls and complete() instead.

The advantage of using completions is that they have a well defined, focused purpose which makes it very easy to see the intent of the code, but they also result in more efficient code as all threads can continue execution until the result is actually needed, and both the waiting and the signalling is highly efficient using low level scheduler sleep/wakeup facilities.

Completions are built on top of the waitqueue and wakeup infrastructure of the Linux scheduler. The event the threads on the waitqueue are waiting for is reduced to a simple flag in ‘struct completion’, appropriately called “done”.

As completions are scheduling related, the code can be found in kernel/sched/completion.c.

1.2 Usage:

There are three main parts to using completions:

- the initialization of the ‘struct completion’ synchronization object
- the waiting part through a call to one of the variants of wait_for_completion(),
- the signaling side through a call to complete() or complete_all().

There are also some helper functions for checking the state of completions. Note that while initialization must happen first, the waiting and signaling part can happen in any order. I.e. it’s entirely normal for a thread to have marked a completion as ‘done’ before another thread checks whether it has to wait for it.

To use completions you need to #include <linux/completion.h> and create a static or dynamic variable of type ‘struct completion’, which has only two fields:

```
struct completion {
        unsigned int done;
        wait_queue_head_t wait;
};
```

This provides the ->wait waitqueue to place tasks on for waiting (if any), and the ->done completion flag for indicating whether it's completed or not.

Completions should be named to refer to the event that is being synchronized on. A good example is:

```
wait_for_completion(&early_console_added);

complete(&early_console_added);
```

Good, intuitive naming (as always) helps code readability. Naming a completion 'complete' is not helpful unless the purpose is super obvious...

1.3 Initializing completions:

Dynamically allocated completion objects should preferably be embedded in data structures that are assured to be alive for the life-time of the function/driver, to prevent races with asynchronous complete() calls from occurring.

Particular care should be taken when using the _timeout() or _killable()/_interruptible() variants of wait_for_completion(), as it must be assured that memory de-allocation does not happen until all related activities (complete() or reinit_completion()) have taken place, even if these wait functions return prematurely due to a timeout or a signal triggering.

Initializing of dynamically allocated completion objects is done via a call to init_completion():

```
init_completion(&dynamic_object->done);
```

In this call we initialize the waitqueue and set ->done to 0, i.e. "not completed" or "not done".

The re-initialization function, reinit_completion(), simply resets the ->done field to 0 ("not done"), without touching the waitqueue. Callers of this function must make sure that there are no racy wait_for_completion() calls going on in parallel.

Calling init_completion() on the same completion object twice is most likely a bug as it re-initializes the queue to an empty queue and enqueued tasks could get "lost" - use reinit_completion() in that case, but be aware of other races.

For static declaration and initialization, macros are available.

For static (or global) declarations in file scope you can use DECLARE_COMPLETION():

```
static DECLARE_COMPLETION(setup_done);
DECLARE_COMPLETION(setup_done);
```

Note that in this case the completion is boot time (or module load time) initialized to 'not done' and doesn't require an init_completion() call.

When a completion is declared as a local variable within a function, then the initialization should always use DECLARE_COMPLETION_ONSTACK() explicitly, not just to make lockdep happy, but also to make it clear that limited scope had been considered and is intentional:

```
DECLARE_COMPLETION_ONSTACK(setup_done)
```

Note that when using completion objects as local variables you must be acutely aware of the short life time of the function stack: the function must not return to a calling context until all activities (such as waiting threads) have ceased and the completion object is completely unused.

To emphasise this again: in particular when using some of the waiting API variants with more complex outcomes, such as the timeout or signalling (_timeout(), _killable() and _interruptible()) variants, the wait might complete prematurely while the object might still be in use by another thread - and a return from the wait_on_completion*() caller function will deallocate the function stack and cause subtle data corruption if a complete() is done in some other thread. Simple testing might not trigger these kinds of races.

If unsure, use dynamically allocated completion objects, preferably embedded in some other long lived object that has a boringly long life time which exceeds the life time of any helper threads using the completion object, or has a lock or other synchronization mechanism to make sure complete() is not called on a freed object.

A naive DECLARE_COMPLETION() on the stack triggers a lockdep warning.

1.4 Waiting for completions:

For a thread to wait for some concurrent activity to finish, it calls wait_for_completion() on the initialized completion structure:

```
void wait_for_completion(struct completion *done)
```

A typical usage scenario is:

```
CPU#1                                   CPU#2

struct completion setup_done;

init_completion(&setup_done);
initialize_work(...,&setup_done,...);

/* run non-dependent code */            /* do setup */

wait_for_completion(&setup_done);       complete(setup_done);
```

This is not implying any particular order between wait_for_completion() and the call to complete() - if the call to complete() happened before the call to wait_for_completion() then the waiting side simply will continue immediately as all dependencies are satisfied; if not, it will block until completion is signaled by complete().

Note that wait_for_completion() is calling spin_lock_irq()/spin_unlock_irq(), so it can only be called safely when you know that interrupts are enabled. Calling it from IRQs-off atomic contexts will result in hard-to-detect spurious enabling of interrupts.

The default behavior is to wait without a timeout and to mark the task as uninterruptible. wait_for_completion() and its variants are only safe in process context (as they can sleep) but not in atomic context, interrupt context, with disabled IRQs, or preemption is disabled - see also try_wait_for_completion() below for handling completion in atomic/interrupt context.

As all variants of wait_for_completion() can (obviously) block for a long time depending on the nature of the activity they are waiting for, so in most cases you probably don't want to call this with held mutexes.

1.5 wait_for_completion*() variants available:

The below variants all return status and this status should be checked in most(/all) cases - in cases where the status is deliberately not checked you probably want to make a note explaining this (e.g. see arch/arm/kernel/smp.c:__cpu_up()).

A common problem that occurs is to have unclean assignment of return types, so take care to assign return-values to variables of the proper type.

Checking for the specific meaning of return values also has been found to be quite inaccurate, e.g. constructs like:

```
if (!wait_for_completion_interruptible_timeout(...))
```

... would execute the same code path for successful completion and for the interrupted case - which is probably not what you want:

```
int wait_for_completion_interruptible(struct completion *done)
```

This function marks the task TASK_INTERRUPTIBLE while it is waiting. If a signal was received while waiting it will return -ERESTARTSYS; 0 otherwise:

```
unsigned long wait_for_completion_timeout(struct completion *done, unsigned␣
↪long timeout)
```

The task is marked as TASK_UNINTERRUPTIBLE and will wait at most 'timeout' jiffies. If a timeout occurs it returns 0, else the remaining time in jiffies (but at least 1).

Timeouts are preferably calculated with msecs_to_jiffies() or usecs_to_jiffies(), to make the code largely HZ-invariant.

If the returned timeout value is deliberately ignored a comment should probably explain why (e.g. see drivers/mfd/wm8350-core.c wm8350_read_auxadc()):

```
long wait_for_completion_interruptible_timeout(struct completion *done,␣
↪unsigned long timeout)
```

This function passes a timeout in jiffies and marks the task as TASK_INTERRUPTIBLE. If a signal was received it will return -ERESTARTSYS; otherwise it returns 0 if the completion timed out, or the remaining time in jiffies if completion occurred.

Further variants include _killable which uses TASK_KILLABLE as the designated tasks state and will return -ERESTARTSYS if it is interrupted, or 0 if completion was achieved. There is a _timeout variant as well:

```
long wait_for_completion_killable(struct completion *done)
long wait_for_completion_killable_timeout(struct completion *done, unsigned␣
↪long timeout)
```

The _io variants wait_for_completion_io() behave the same as the non-_io variants, except for accounting waiting time as 'waiting on IO', which has an impact on how the task is accounted in scheduling/IO stats:

```
void wait_for_completion_io(struct completion *done)
unsigned long wait_for_completion_io_timeout(struct completion *done, unsigned␣
↪long timeout)
```

1.6 Signaling completions:

A thread that wants to signal that the conditions for continuation have been achieved calls complete() to signal exactly one of the waiters that it can continue:

```
void complete(struct completion *done)
```

... or calls complete_all() to signal all current and future waiters:

```
void complete_all(struct completion *done)
```

The signaling will work as expected even if completions are signaled before a thread starts waiting. This is achieved by the waiter "consuming" (decrementing) the done field of 'struct completion'. Waiting threads wakeup order is the same in which they were enqueued (FIFO order).

If complete() is called multiple times then this will allow for that number of waiters to continue - each call to complete() will simply increment the done field. Calling complete_all() multiple times is a bug though. Both complete() and complete_all() can be called in IRQ/atomic context safely.

There can only be one thread calling complete() or complete_all() on a particular 'struct completion' at any time - serialized through the wait queue spinlock. Any such concurrent calls to complete() or complete_all() probably are a design bug.

Signaling completion from IRQ context is fine as it will appropriately lock with spin_lock_irqsave()/spin_unlock_irqrestore() and it will never sleep.

1.7 try_wait_for_completion()/completion_done():

The try_wait_for_completion() function will not put the thread on the wait queue but rather returns false if it would need to enqueue (block) the thread, else it consumes one posted completion and returns true:

```
bool try_wait_for_completion(struct completion *done)
```

Finally, to check the state of a completion without changing it in any way, call completion_done(), which returns false if there are no posted completions that were not yet consumed by waiters (implying that there are waiters) and true otherwise:

```
bool completion_done(struct completion *done)
```

Both try_wait_for_completion() and completion_done() are safe to be called in IRQ or atomic context.

CHAPTER

TWO

CPU SCHEDULER IMPLEMENTATION HINTS FOR ARCHITECTURE SPECIFIC CODE

Nick Piggin, 2005

2.1 Context switch

1. Runqueue locking By default, the switch_to arch function is called with the runqueue locked. This is usually not a problem unless switch_to may need to take the runqueue lock. This is usually due to a wake up operation in the context switch. See arch/ia64/include/asm/switch_to.h for an example.

To request the scheduler call switch_to with the runqueue unlocked, you must *#define __ARCH_WANT_UNLOCKED_CTXSW* in a header file (typically the one where switch_to is defined).

Unlocked context switches introduce only a very minor performance penalty to the core scheduler implementation in the CONFIG_SMP case.

2.2 CPU idle

Your cpu_idle routines need to obey the following rules:

1. Preempt should now disabled over idle routines. Should only be enabled to call schedule() then disabled again.
2. need_resched/TIF_NEED_RESCHED is only ever set, and will never be cleared until the running task has called schedule(). Idle threads need only ever query need_resched, and may never set or clear it.
3. When cpu_idle finds (need_resched() == 'true'), it should call schedule(). It should not call schedule() otherwise.
4. The only time interrupts need to be disabled when checking need_resched is if we are about to sleep the processor until the next interrupt (this doesn't provide any protection of need_resched, it prevents losing an interrupt):

    4a. Common problem with this type of sleep appears to be:

```
local_irq_disable();
if (!need_resched()) {
        local_irq_enable();
```

```
        *** resched interrupt arrives here ***
        __asm__("sleep until next interrupt");
}
```

5. TIF_POLLING_NRFLAG can be set by idle routines that do not need an interrupt to wake them up when need_resched goes high. In other words, they must be periodically polling need_resched, although it may be reasonable to do some background work or enter a low CPU priority.

    - 5a. If TIF_POLLING_NRFLAG is set, and we do decide to enter an interrupt sleep, it needs to be cleared then a memory barrier issued (followed by a test of need_resched with interrupts disabled, as explained in 3).

arch/x86/kernel/process.c has examples of both polling and sleeping idle functions.

2.3 Possible arch/ problems

Possible arch problems I found (and either tried to fix or didn't):

ia64 - is safe_halt call racy vs interrupts? (does it sleep?) (See #4a)

sh64 - Is sleeping racy vs interrupts? (See #4a)

sparc - IRQs on at this point(?), change local_irq_save to _disable.

- TODO: needs secondary CPUs to disable preempt (See #1)

CHAPTER

THREE

CFS BANDWIDTH CONTROL

Note: This document only discusses CPU bandwidth control for SCHED_NORMAL. The SCHED_RT case is covered in *Real-Time group scheduling*

CFS bandwidth control is a CONFIG_FAIR_GROUP_SCHED extension which allows the specification of the maximum CPU bandwidth available to a group or hierarchy.

The bandwidth allowed for a group is specified using a quota and period. Within each given "period" (microseconds), a task group is allocated up to "quota" microseconds of CPU time. That quota is assigned to per-cpu run queues in slices as threads in the cgroup become runnable. Once all quota has been assigned any additional requests for quota will result in those threads being throttled. Throttled threads will not be able to run again until the next period when the quota is replenished.

A group's unassigned quota is globally tracked, being refreshed back to cfs_quota units at each period boundary. As threads consume this bandwidth it is transferred to cpu-local "silos" on a demand basis. The amount transferred within each of these updates is tunable and described as the "slice".

3.1 Burst feature

This feature borrows time now against our future underrun, at the cost of increased interference against the other system users. All nicely bounded.

Traditional (UP-EDF) bandwidth control is something like:

(U = Sum u_i) <= 1

This guaranteeds both that every deadline is met and that the system is stable. After all, if U were > 1, then for every second of walltime, we'd have to run more than a second of program time, and obviously miss our deadline, but the next deadline will be further out still, there is never time to catch up, unbounded fail.

The burst feature observes that a workload doesn't always executes the full quota; this enables one to describe u_i as a statistical distribution.

For example, have u_i = {x,e}_i, where x is the p(95) and x+e p(100) (the traditional WCET). This effectively allows u to be smaller, increasing the efficiency (we can pack more tasks in the system), but at the cost of missing deadlines when all the odds line up. However, it does maintain stability, since every overrun must be paired with an underrun as long as our x is above the average.

That is, suppose we have 2 tasks, both specify a p(95) value, then we have a p(95)*p(95) = 90.25% chance both tasks are within their quota and everything is good. At the same time we have a p(5)p(5) = 0.25% chance both tasks will exceed their quota at the same time (guaranteed deadline fail). Somewhere in between there's a threshold where one exceeds and the other doesn't underrun enough to compensate; this depends on the specific CDFs.

At the same time, we can say that the worst case deadline miss, will be Sum e_i; that is, there is a bounded tardiness (under the assumption that x+e is indeed WCET).

The interferenece when using burst is valued by the possibilities for missing the deadline and the average WCET. Test results showed that when there many cgroups or CPU is under utilized, the interference is limited. More details are shown in: https://lore.kernel.org/lkml/5371BD36-55AE-4F71-B9D7-B86DC32E3D2B@linux.alibaba.com/

3.2 Management

Quota, period and burst are managed within the cpu subsystem via cgroupfs.

Note: The cgroupfs files described in this section are only applicable to cgroup v1. For cgroup v2, see Documentation/admin-guide/cgroup-v2.rst.

- cpu.cfs_quota_us: run-time replenished within a period (in microseconds)
- cpu.cfs_period_us: the length of a period (in microseconds)
- cpu.stat: exports throttling statistics [explained further below]
- cpu.cfs_burst_us: the maximum accumulated run-time (in microseconds)

The default values are:

```
cpu.cfs_period_us=100ms
cpu.cfs_quota_us=-1
cpu.cfs_burst_us=0
```

A value of -1 for cpu.cfs_quota_us indicates that the group does not have any bandwidth restriction in place, such a group is described as an unconstrained bandwidth group. This represents the traditional work-conserving behavior for CFS.

Writing any (valid) positive value(s) no smaller than cpu.cfs_burst_us will enact the specified bandwidth limit. The minimum quota allowed for the quota or period is 1ms. There is also an upper bound on the period length of 1s. Additional restrictions exist when bandwidth limits are used in a hierarchical fashion, these are explained in more detail below.

Writing any negative value to cpu.cfs_quota_us will remove the bandwidth limit and return the group to an unconstrained state once more.

A value of 0 for cpu.cfs_burst_us indicates that the group can not accumulate any unused bandwidth. It makes the traditional bandwidth control behavior for CFS unchanged. Writing any (valid) positive value(s) no larger than cpu.cfs_quota_us into cpu.cfs_burst_us will enact the cap on unused bandwidth accumulation.

Any updates to a group's bandwidth specification will result in it becoming unthrottled if it is in a constrained state.

3.3 System wide settings

For efficiency run-time is transferred between the global pool and CPU local “silos” in a batch fashion. This greatly reduces global accounting pressure on large systems. The amount transferred each time such an update is required is described as the “slice”.

This is tunable via procfs:

```
/proc/sys/kernel/sched_cfs_bandwidth_slice_us (default=5ms)
```

Larger slice values will reduce transfer overheads, while smaller values allow for more fine-grained consumption.

3.4 Statistics

A group’s bandwidth statistics are exported via 5 fields in cpu.stat.

cpu.stat:

- nr_periods: Number of enforcement intervals that have elapsed.
- nr_throttled: Number of times the group has been throttled/limited.
- throttled_time: The total time duration (in nanoseconds) for which entities of the group have been throttled.
- nr_bursts: Number of periods burst occurs.
- burst_time: Cumulative wall-time (in nanoseconds) that any CPUs has used above quota in respective periods.

This interface is read-only.

3.5 Hierarchical considerations

The interface enforces that an individual entity’s bandwidth is always attainable, that is: max(c_i) <= C. However, over-subscription in the aggregate case is explicitly allowed to enable work-conserving semantics within a hierarchy:

e.g. Sum (c_i) may exceed C

[Where C is the parent’s bandwidth, and c_i its children]

There are two ways in which a group may become throttled:

a. it fully consumes its own quota within a period

b. a parent’s quota is fully consumed within its period

In case b) above, even though the child may have runtime remaining it will not be allowed to until the parent’s runtime is refreshed.

3.6 CFS Bandwidth Quota Caveats

Once a slice is assigned to a cpu it does not expire. However all but 1ms of the slice may be returned to the global pool if all threads on that cpu become unrunnable. This is configured at compile time by the min_cfs_rq_runtime variable. This is a performance tweak that helps prevent added contention on the global lock.

The fact that cpu-local slices do not expire results in some interesting corner cases that should be understood.

For cgroup cpu constrained applications that are cpu limited this is a relatively moot point because they will naturally consume the entirety of their quota as well as the entirety of each cpu-local slice in each period. As a result it is expected that nr_periods roughly equal nr_throttled, and that cpuacct.usage will increase roughly equal to cfs_quota_us in each period.

For highly-threaded, non-cpu bound applications this non-expiration nuance allows applications to briefly burst past their quota limits by the amount of unused slice on each cpu that the task group is running on (typically at most 1ms per cpu or as defined by min_cfs_rq_runtime). This slight burst only applies if quota had been assigned to a cpu and then not fully used or returned in previous periods. This burst amount will not be transferred between cores. As a result, this mechanism still strictly limits the task group to quota average usage, albeit over a longer time window than a single period. This also limits the burst ability to no more than 1ms per cpu. This provides better more predictable user experience for highly threaded applications with small quota limits on high core count machines. It also eliminates the propensity to throttle these applications while simultanously using less than quota amounts of cpu. Another way to say this, is that by allowing the unused portion of a slice to remain valid across periods we have decreased the possibility of wastefully expiring quota on cpu-local silos that don't need a full slice's amount of cpu time.

The interaction between cpu-bound and non-cpu-bound-interactive applications should also be considered, especially when single core usage hits 100%. If you gave each of these applications half of a cpu-core and they both got scheduled on the same CPU it is theoretically possible that the non-cpu bound application will use up to 1ms additional quota in some periods, thereby preventing the cpu-bound application from fully using its quota by that same amount. In these instances it will be up to the CFS algorithm (see *CFS Scheduler*) to decide which application is chosen to run, as they will both be runnable and have remaining quota. This runtime discrepancy will be made up in the following periods when the interactive application idles.

3.7 Examples

1. Limit a group to 1 CPU worth of runtime:

   ```
   If period is 250ms and quota is also 250ms, the group will get
   1 CPU worth of runtime every 250ms.

   # echo 250000 > cpu.cfs_quota_us /* quota = 250ms */
   # echo 250000 > cpu.cfs_period_us /* period = 250ms */
   ```

2. Limit a group to 2 CPUs worth of runtime on a multi-CPU machine

   With 500ms period and 1000ms quota, the group can get 2 CPUs worth of runtime every 500ms:

```
# echo 1000000 > cpu.cfs_quota_us /* quota = 1000ms */
# echo 500000 > cpu.cfs_period_us /* period = 500ms */

The larger period here allows for increased burst capacity.
```

3. Limit a group to 20% of 1 CPU.

   With 50ms period, 10ms quota will be equivalent to 20% of 1 CPU:

```
# echo 10000 > cpu.cfs_quota_us /* quota = 10ms */
# echo 50000 > cpu.cfs_period_us /* period = 50ms */
```

   By using a small period here we are ensuring a consistent latency response at the expense of burst capacity.

4. Limit a group to 40% of 1 CPU, and allow accumulate up to 20% of 1 CPU additionally, in case accumulation has been done.

   With 50ms period, 20ms quota will be equivalent to 40% of 1 CPU. And 10ms burst will be equivalent to 20% of 1 CPU:

```
# echo 20000 > cpu.cfs_quota_us /* quota = 20ms */
# echo 50000 > cpu.cfs_period_us /* period = 50ms */
# echo 10000 > cpu.cfs_burst_us /* burst = 10ms */
```

   Larger buffer setting (no larger than quota) allows greater burst capacity.

CHAPTER

FOUR

DEADLINE TASK SCHEDULING

4.1 0. WARNING

Fiddling with these settings can result in an unpredictable or even unstable system behavior. As for -rt (group) scheduling, it is assumed that root users know what they're doing.

4.2 1. Overview

The SCHED_DEADLINE policy contained inside the sched_dl scheduling class is basically an implementation of the Earliest Deadline First (EDF) scheduling algorithm, augmented with a mechanism (called Constant Bandwidth Server, CBS) that makes it possible to isolate the behavior of tasks between each other.

4.3 2. Scheduling algorithm

4.3.1 2.1 Main algorithm

SCHED_DEADLINE [18] uses three parameters, named "runtime", "period", and "deadline", to schedule tasks. A SCHED_DEADLINE task should receive "runtime" microseconds of execution time every "period" microseconds, and these "runtime" microseconds are available within "deadline" microseconds from the beginning of the period. In order to implement this behavior, every time the task wakes up, the scheduler computes a "scheduling deadline" consistent with the guarantee (using the CBS[2,3] algorithm). Tasks are then scheduled using EDF[1] on these scheduling deadlines (the task with the earliest scheduling deadline is selected for execution). Notice that the task actually receives "runtime" time units within "deadline" if a proper "admission control" strategy (see Section "4. Bandwidth management") is used (clearly, if the system is overloaded this guarantee cannot be respected).

Summing up, the CBS[2,3] algorithm assigns scheduling deadlines to tasks so that each task runs for at most its runtime every period, avoiding any interference between different tasks (bandwidth isolation), while the EDF[1] algorithm selects the task with the earliest scheduling deadline as the one to be executed next. Thanks to this feature, tasks that do not strictly comply with the "traditional" real-time task model (see Section 3) can effectively use the new policy.

In more details, the CBS algorithm assigns scheduling deadlines to tasks in the following way:

- Each SCHED_DEADLINE task is characterized by the “runtime”, “deadline”, and “period” parameters;
- The state of the task is described by a “scheduling deadline”, and a “remaining runtime”. These two parameters are initially set to 0;
- When a SCHED_DEADLINE task wakes up (becomes ready for execution), the scheduler checks if:

```
         remaining runtime                  runtime
---------------------------------    >    ---------
scheduling deadline - current time           period
```

  then, if the scheduling deadline is smaller than the current time, or this condition is verified, the scheduling deadline and the remaining runtime are re-initialized as

  scheduling deadline = current time + deadline remaining runtime = runtime

  otherwise, the scheduling deadline and the remaining runtime are left unchanged;

- When a SCHED_DEADLINE task executes for an amount of time t, its remaining runtime is decreased as:

```
remaining runtime = remaining runtime - t
```

  (technically, the runtime is decreased at every tick, or when the task is descheduled / preempted);

- When the remaining runtime becomes less or equal than 0, the task is said to be “throttled” (also known as “depleted” in real-time literature) and cannot be scheduled until its scheduling deadline. The “replenishment time” for this task (see next item) is set to be equal to the current value of the scheduling deadline;
- When the current time is equal to the replenishment time of a throttled task, the scheduling deadline and the remaining runtime are updated as:

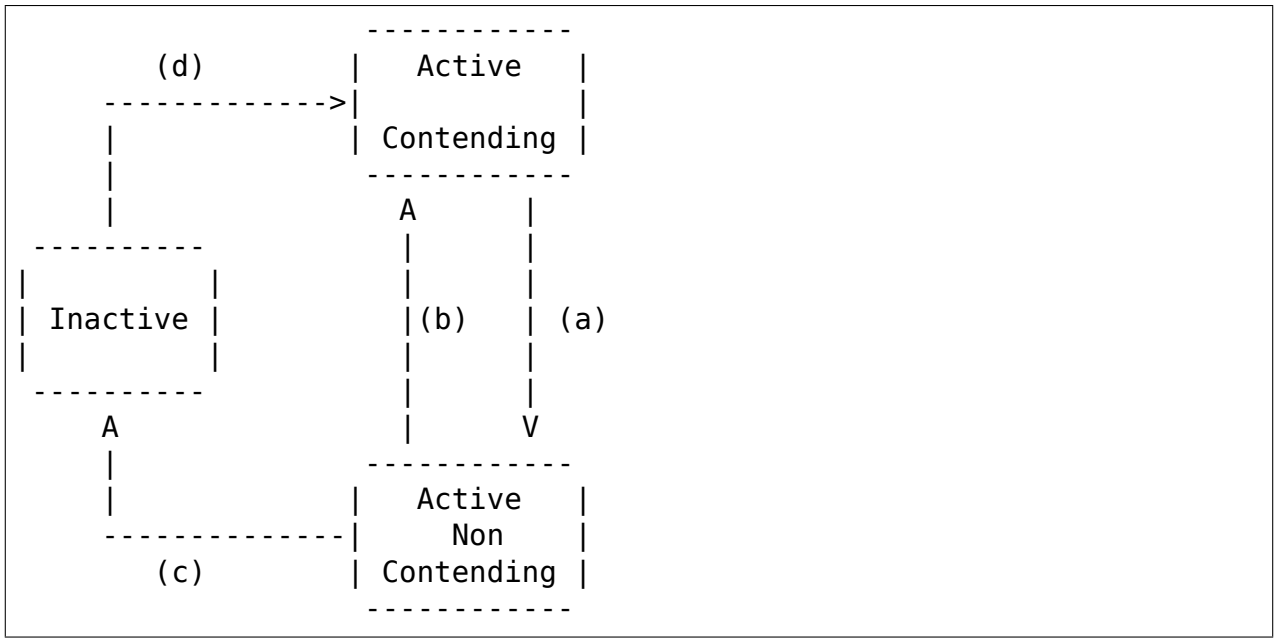
```
scheduling deadline = scheduling deadline + period
remaining runtime = remaining runtime + runtime
```

The SCHED_FLAG_DL_OVERRUN flag in sched_attr’s sched_flags field allows a task to get informed about runtime overruns through the delivery of SIGXCPU signals.

4.3.2 2.2 Bandwidth reclaiming

Bandwidth reclaiming for deadline tasks is based on the GRUB (Greedy Reclamation of Unused Bandwidth) algorithm [15, 16, 17] and it is enabled when flag SCHED_FLAG_RECLAIM is set.

The following diagram illustrates the state names for tasks handled by GRUB:



A task can be in one of the following states:

- ActiveContending: if it is ready for execution (or executing);
- ActiveNonContending: if it just blocked and has not yet surpassed the 0-lag time;
- Inactive: if it is blocked and has surpassed the 0-lag time.

State transitions:

(a) When a task blocks, it does not become immediately inactive since its bandwidth cannot be immediately reclaimed without breaking the real-time guarantees. It therefore enters a transitional state called ActiveNonContending. The scheduler arms the “inactive timer” to fire at the 0-lag time, when the task’s bandwidth can be reclaimed without breaking the real-time guarantees.

The 0-lag time for a task entering the ActiveNonContending state is computed as:

$$\text{deadline} - \frac{(\text{runtime} * \text{dl\_period})}{\text{dl\_runtime}}$$

where runtime is the remaining runtime, while dl_runtime and dl_period are the reservation parameters.

(b) If the task wakes up before the inactive timer fires, the task re-enters the ActiveContending state and the “inactive timer” is canceled. In addition, if the task wakes up on a different runqueue, then the task’s utilization must be removed from the previous runqueue’s active utilization and must be added to the new

runqueue's active utilization. In order to avoid races between a task waking up on a runqueue while the "inactive timer" is running on a different CPU, the "dl_non_contending" flag is used to indicate that a task is not on a runqueue but is active (so, the flag is set when the task blocks and is cleared when the "inactive timer" fires or when the task wakes up).

(c) When the "inactive timer" fires, the task enters the Inactive state and its utilization is removed from the runqueue's active utilization.

(d) When an inactive task wakes up, it enters the ActiveContending state and its utilization is added to the active utilization of the runqueue where it has been enqueued.

For each runqueue, the algorithm GRUB keeps track of two different bandwidths:

- Active bandwidth (running_bw): this is the sum of the bandwidths of all tasks in active state (i.e., ActiveContending or ActiveNonContending);
- Total bandwidth (this_bw): this is the sum of all tasks "belonging" to the runqueue, including the tasks in Inactive state.

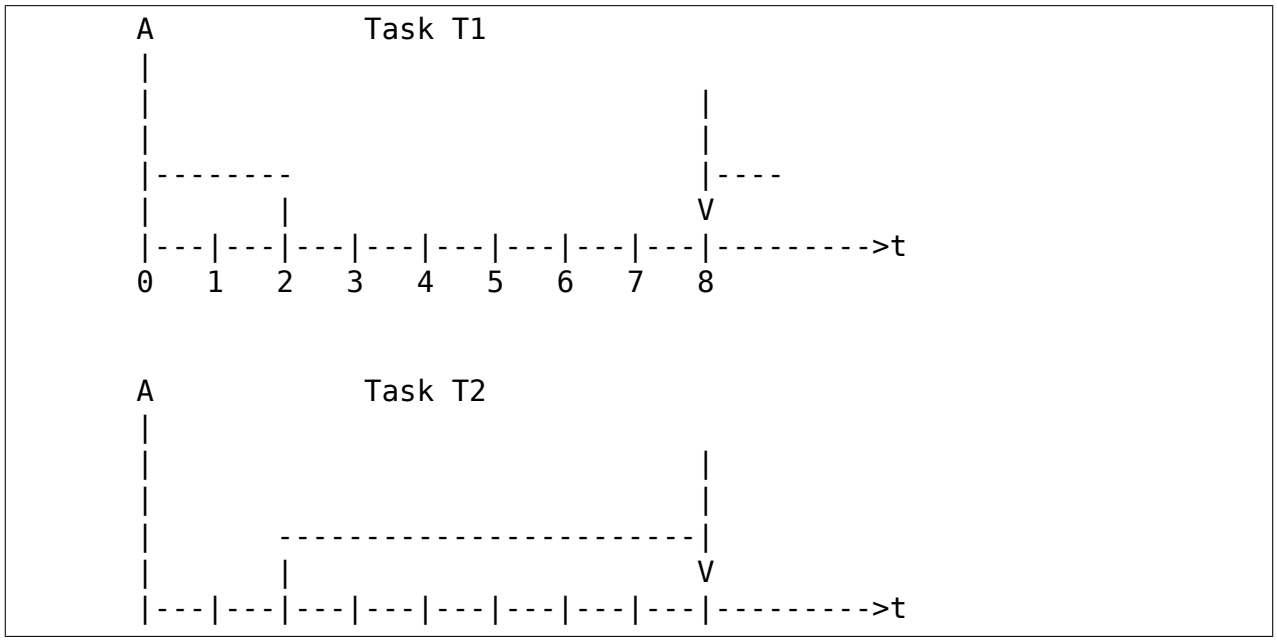
The algorithm reclaims the bandwidth of the tasks in Inactive state. It does so by decrementing the runtime of the executing task Ti at a pace equal to

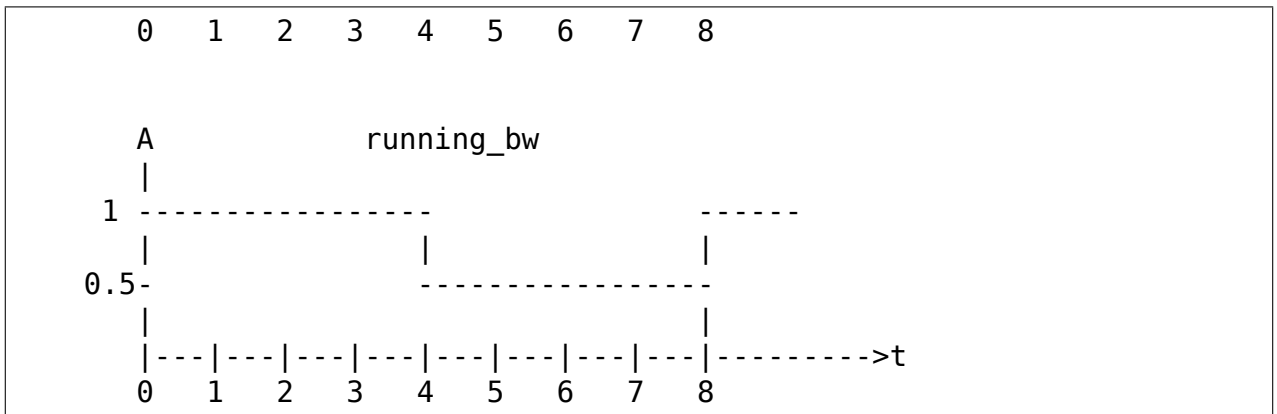
dq = -max{ Ui / Umax, (1 - Uinact - Uextra) } dt

where:

- Ui is the bandwidth of task Ti;
- Umax is the maximum reclaimable utilization (subjected to RT throttling limits);
- Uinact is the (per runqueue) inactive utilization, computed as (this_bq - running_bw);
- Uextra is the (per runqueue) extra reclaimable utilization (subjected to RT throttling limits).

Let's now see a trivial example of two deadline tasks with runtime equal to 4 and period equal to 8 (i.e., bandwidth equal to 0.5):





- Time t = 0:

  Both tasks are ready for execution and therefore in ActiveContending state.
  Suppose Task T1 is the first task to start execution.
  Since there are no inactive tasks, its runtime is decreased as dq = -1 dt.

- Time t = 2:

  Suppose that task T1 blocks
  Task T1 therefore enters the ActiveNonContending state. Since its remaining
  runtime is equal to 2, its 0-lag time is equal to t = 4.
  Task T2 start execution, with runtime still decreased as dq = -1 dt since
  there are no inactive tasks.

- Time t = 4:

  This is the 0-lag time for Task T1. Since it didn't woken up in the
  meantime, it enters the Inactive state. Its bandwidth is removed from
  running_bw.
  Task T2 continues its execution. However, its runtime is now decreased as
  dq = - 0.5 dt because Uinact = 0.5.
  Task T2 therefore reclaims the bandwidth unused by Task T1.

- Time t = 8:

  Task T1 wakes up. It enters the ActiveContending state again, and the
  running_bw is incremented.

4.3.3 2.3 Energy-aware scheduling

When cpufreq's schedutil governor is selected, SCHED_DEADLINE implements the GRUB-PA [19] algorithm, reducing the CPU operating frequency to the minimum value that still allows to meet the deadlines. This behavior is currently implemented only for ARM architectures.

A particular care must be taken in case the time needed for changing frequency is of the same order of magnitude of the reservation period. In such cases, setting a fixed CPU frequency results in a lower amount of deadline misses.

4.4 3. Scheduling Real-Time Tasks

> **Warning:** This section contains a (not-thorough) summary on classical deadline scheduling theory, and how it applies to SCHED_DEADLINE. The reader can "safely" skip to Section 4 if only interested in seeing how the scheduling policy can be used. Anyway, we strongly recommend to come back here and continue reading (once the urge for testing is satisfied :P) to be sure of fully understanding all technical details.

There are no limitations on what kind of task can exploit this new scheduling discipline, even if it must be said that it is particularly suited for periodic or sporadic real-time tasks that need guarantees on their timing behavior, e.g., multimedia, streaming, control applications, etc.

4.4.1 3.1 Definitions

A typical real-time task is composed of a repetition of computation phases (task instances, or jobs) which are activated on a periodic or sporadic fashion. Each job J_j (where J_j is the j^th job of the task) is characterized by an arrival time r_j (the time when the job starts), an amount of computation time c_j needed to finish the job, and a job absolute deadline d_j, which is the time within which the job should be finished. The maximum execution time max{c_j} is called "Worst Case Execution Time" (WCET) for the task. A real-time task can be periodic with period P if r_{j+1} = r_j + P, or sporadic with minimum inter-arrival time P is r_{j+1} >= r_j + P. Finally, d_j = r_j + D, where D is the task's relative deadline. Summing up, a real-time task can be described as

Task = (WCET, D, P)

The utilization of a real-time task is defined as the ratio between its WCET and its period (or minimum inter-arrival time), and represents the fraction of CPU time needed to execute the task.

If the total utilization U=sum(WCET_i/P_i) is larger than M (with M equal to the number of CPUs), then the scheduler is unable to respect all the deadlines. Note that total utilization is defined as the sum of the utilizations WCET_i/P_i over all the real-time tasks in the system. When considering multiple real-time tasks, the parameters of the i-th task are indicated with the "_i" suffix. Moreover, if the total utilization is larger than M, then we risk starving non- real-time tasks by real-time tasks. If, instead, the

total utilization is smaller than M, then non real-time tasks will not be starved and the system might be able to respect all the deadlines. As a matter of fact, in this case it is possible to provide an upper bound for tardiness (defined as the maximum between 0 and the difference between the finishing time of a job and its absolute deadline). More precisely, it can be proven that using a global EDF scheduler the maximum tardiness of each task is smaller or equal than

((M − 1) · WCET_max − WCET_min)/(M − (M − 2) · U_max) + WCET_max

where WCET_max = max{WCET_i} is the maximum WCET, WCET_min=min{WCET_i} is the minimum WCET, and U_max = max{WCET_i/P_i} is the maximum utilization[12].

4.4.2 3.2 Schedulability Analysis for Uniprocessor Systems

If M=1 (uniprocessor system), or in case of partitioned scheduling (each real-time task is statically assigned to one and only one CPU), it is possible to formally check if all the deadlines are respected. If D_i = P_i for all tasks, then EDF is able to respect all the deadlines of all the tasks executing on a CPU if and only if the total utilization of the tasks running on such a CPU is smaller or equal than 1. If D_i != P_i for some task, then it is possible to define the density of a task as WCET_i/min{D_i,P_i}, and EDF is able to respect all the deadlines of all the tasks running on a CPU if the sum of the densities of the tasks running on such a CPU is smaller or equal than 1:

sum(WCET_i / min{D_i, P_i}) <= 1

It is important to notice that this condition is only sufficient, and not necessary: there are task sets that are schedulable, but do not respect the condition. For example, consider the task set {Task_1,Task_2} composed by Task_1=(50ms,50ms,100ms) and Task_2=(10ms,100ms,100ms). EDF is clearly able to schedule the two tasks without missing any deadline (Task_1 is scheduled as soon as it is released, and finishes just in time to respect its deadline; Task_2 is scheduled immediately after Task_1, hence its response time cannot be larger than 50ms + 10ms = 60ms) even if

50 / min{50,100} + 10 / min{100, 100} = 50 / 50 + 10 / 100 = 1.1

Of course it is possible to test the exact schedulability of tasks with D_i != P_i (checking a condition that is both sufficient and necessary), but this cannot be done by comparing the total utilization or density with a constant. Instead, the so called “processor demand” approach can be used, computing the total amount of CPU time h(t) needed by all the tasks to respect all of their deadlines in a time interval of size t, and comparing such a time with the interval size t. If h(t) is smaller than t (that is, the amount of time needed by the tasks in a time interval of size t is smaller than the size of the interval) for all the possible values of t, then EDF is able to schedule the tasks respecting all of their deadlines. Since performing this check for all possible values of t is impossible, it has been proven[4,5,6] that it is sufficient to perform the test for values of t between 0 and a maximum value L. The cited papers contain all of the mathematical details and explain how to compute h(t) and L. In any case, this kind of analysis is too complex as well as too time-consuming to be performed on-line. Hence, as explained in Section 4 Linux uses an admission test based on the tasks’ utilizations.

4.4.3 3.3 Schedulability Analysis for Multiprocessor Systems

On multiprocessor systems with global EDF scheduling (non partitioned systems), a sufficient test for schedulability can not be based on the utilizations or densities: it can be shown that even if D_i = P_i task sets with utilizations slightly larger than 1 can miss deadlines regardless of the number of CPUs.

Consider a set {Task_1,...Task_{M+1}} of M+1 tasks on a system with M CPUs, with the first task Task_1=(P,P,P) having period, relative deadline and WCET equal to P. The remaining M tasks Task_i=(e,P-1,P-1) have an arbitrarily small worst case execution time (indicated as “e” here) and a period smaller than the one of the first task. Hence, if all the tasks activate at the same time t, global EDF schedules these M tasks first (because their absolute deadlines are equal to t + P - 1, hence they are smaller than the absolute deadline of Task_1, which is t + P). As a result, Task_1 can be scheduled only at time t + e, and will finish at time t + e + P, after its absolute deadline. The total utilization of the task set is U = M · e / (P - 1) + P / P = M · e / (P - 1) + 1, and for small values of e this can become very close to 1. This is known as “Dhall’s effect”[7]. Note: the example in the original paper by Dhall has been slightly simplified here (for example, Dhall more correctly computed lim_{e->0}U).

More complex schedulability tests for global EDF have been developed in real-time literature[8,9], but they are not based on a simple comparison between total utilization (or density) and a fixed constant. If all tasks have D_i = P_i, a sufficient schedulability condition can be expressed in a simple way:

sum(WCET_i / P_i) <= M - (M - 1) · U_max

where U_max = max{WCET_i / P_i}[10]. Notice that for U_max = 1, M - (M - 1) · U_max becomes M - M + 1 = 1 and this schedulability condition just confirms the Dhall’s effect. A more complete survey of the literature about schedulability tests for multi-processor real-time scheduling can be found in [11].

As seen, enforcing that the total utilization is smaller than M does not guarantee that global EDF schedules the tasks without missing any deadline (in other words, global EDF is not an optimal scheduling algorithm). However, a total utilization smaller than M is enough to guarantee that non real-time tasks are not starved and that the tardiness of real-time tasks has an upper bound[12] (as previously noted). Different bounds on the maximum tardiness experienced by real-time tasks have been developed in various papers[13,14], but the theoretical result that is important for SCHED_DEADLINE is that if the total utilization is smaller or equal than M then the response times of the tasks are limited.

4.4.4 3.4 Relationship with SCHED_DEADLINE Parameters

Finally, it is important to understand the relationship between the SCHED_DEADLINE scheduling parameters described in Section 2 (runtime, deadline and period) and the real-time task parameters (WCET, D, P) described in this section. Note that the tasks’ temporal constraints are represented by its absolute deadlines d_j = r_j + D described above, while SCHED_DEADLINE schedules the tasks according to scheduling deadlines (see Section 2). If an admission test is used to guarantee that the scheduling deadlines are respected, then SCHED_DEADLINE can be used to schedule real-time tasks guaranteeing that all the jobs’ deadlines of a task are respected. In order to do this, a task must be scheduled by setting:

- runtime >= WCET
- deadline = D
- period <= P

IOW, if runtime >= WCET and if period is <= P, then the scheduling deadlines and the absolute deadlines (d_j) coincide, so a proper admission control allows to respect the jobs' absolute deadlines for this task (this is what is called "hard schedulability property" and is an extension of Lemma 1 of [2]). Notice that if runtime > deadline the admission control will surely reject this task, as it is not possible to respect its temporal constraints.

References:

1 - C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the Association for Computing Machinery, 20(1), 1973.

2 - L. Abeni , G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. Proceedings of the 19th IEEE Real-time Systems Symposium, 1998. http://retis.sssup.it/~giorgio/paps/1998/rtss98-cbs.pdf

3 - L. Abeni. Server Mechanisms for Multimedia Applications. ReTiS Lab Technical Report. http://disi.unitn.it/~abeni/tr-98-01.pdf

4 - J. Y. Leung and M.L. Merril. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. Information Processing Letters, vol. 11, no. 3, pp. 115-118, 1980.

5 - S. K. Baruah, A. K. Mok and L. E. Rosier. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. Proceedings of the 11th IEEE Real-time Systems Symposium, 1990.

6 - S. K. Baruah, L. E. Rosier and R. R. Howell. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time tasks on One Processor. Real-Time Systems Journal, vol. 4, no. 2, pp 301-324, 1990.

7 - S. J. Dhall and C. L. Liu. On a real-time scheduling problem. Operations research, vol. 26, no. 1, pp 127-140, 1978.

8 - T. Baker. Multiprocessor EDF and Deadline Monotonic Schedulability Analysis. Proceedings of the 24th IEEE Real-Time Systems Symposium, 2003.

9 - T. Baker. An Analysis of EDF Schedulability on a Multiprocessor. IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 8, pp 760-768, 2005.

10 - J. Goossens, S. Funk and S. Baruah, Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. Real-Time Systems Journal, vol. 25, no. 2–3, pp. 187–205, 2003.

11 - R. Davis and A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. ACM Computing Surveys, vol. 43, no. 4, 2011. http://www-users.cs.york.ac.uk/~robdavis/papers/MPSurveyv5.0.pdf

12 - U. C. Devi and J. H. Anderson. Tardiness Bounds under Global EDF Scheduling on a Multiprocessor. Real-Time Systems Journal, vol. 32, no. 2, pp 133-189, 2008.

13 - P. Valente and G. Lipari. An Upper Bound to the Lateness of Soft Real-Time Tasks Scheduled by EDF on Multiprocessors. Proceedings of the 26th IEEE Real-Time Systems Symposium, 2005.

14 - J. Erickson, U. Devi and S. Baruah. Improved tardiness bounds for Global EDF. Proceedings of the 22nd Euromicro Conference on Real-Time Systems, 2010.

15 - G. Lipari, S. Baruah, Greedy reclamation of unused bandwidth in constant-bandwidth servers, 12th IEEE Euromicro Conference on Real-Time Systems, 2000.

16 - L. Abeni, J. Lelli, C. Scordino, L. Palopoli, Greedy CPU reclaiming for SCHED DEADLINE. In Proceedings of the Real-Time Linux Workshop (RTLWS), Dusseldorf, Germany, 2014.

17 - L. Abeni, G. Lipari, A. Parri, Y. Sun, Multicore CPU reclaiming: parallel or sequential?. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2016.

18 - J. Lelli, C. Scordino, L. Abeni, D. Faggioli, Deadline scheduling in the Linux kernel, Software: Practice and Experience, 46(6): 821-839, June 2016.

19 - C. Scordino, L. Abeni, J. Lelli, Energy-Aware Real-Time Scheduling in the Linux Kernel, 33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018), Pau, France, April 2018.

4.5 4. Bandwidth management

As previously mentioned, in order for -deadline scheduling to be effective and useful (that is, to be able to provide “runtime” time units within “deadline”), it is important to have some method to keep the allocation of the available fractions of CPU time to the various tasks under control. This is usually called “admission control” and if it is not performed, then no guarantee can be given on the actual scheduling of the -deadline tasks.

As already stated in Section 3, a necessary condition to be respected to correctly schedule a set of real-time tasks is that the total utilization is smaller than M. When talking about -deadline tasks, this requires that the sum of the ratio between runtime and period for all tasks is smaller than M. Notice that the ratio runtime/period is equivalent to the utilization of a “traditional” real-time task, and is also often referred to as “bandwidth”. The interface used to control the CPU bandwidth that can be allocated to -deadline tasks is similar to the one already used for -rt tasks with real-time group scheduling (a.k.a. RT-throttling - see *Real-Time group scheduling*), and is based on readable/ writable control files located in procfs (for system wide settings). Notice that per-group settings (controlled through cgroupfs) are still not defined for -deadline tasks, because more discussion is needed in order to figure out how we want to manage SCHED_DEADLINE bandwidth at the task group level.

A main difference between deadline bandwidth management and RT-throttling is that -deadline tasks have bandwidth on their own (while -rt ones don't!), and thus we don't need a higher level throttling mechanism to enforce the desired bandwidth. In other words, this means that interface parameters are only used at admission control time (i.e., when the user calls sched_setattr()). Scheduling is then performed considering actual tasks' parameters, so that CPU bandwidth is allocated to SCHED_DEADLINE tasks respecting their needs in terms of granularity. Therefore, using this simple interface we can put a cap on total utilization of -deadline tasks (i.e., Sum (runtime_i / period_i) < global_dl_utilization_cap).

4.5.1 4.1 System wide settings

The system wide settings are configured under the /proc virtual file system.

For now the -rt knobs are used for -deadline admission control and the -deadline runtime is accounted against the -rt runtime. We realize that this isn't entirely desirable; however, it is better to have a small interface for now, and be able to change it easily later. The ideal situation (see 5.) is to run -rt tasks from a -deadline server; in which case the -rt bandwidth is a direct subset of dl_bw.

This means that, for a root_domain comprising M CPUs, -deadline tasks can be created while the sum of their bandwidths stays below:

M * (sched_rt_runtime_us / sched_rt_period_us)

It is also possible to disable this bandwidth management logic, and be thus free of oversubscribing the system up to any arbitrary level. This is done by writing -1 in /proc/sys/kernel/sched_rt_runtime_us.

4.5.2 4.2 Task interface

Specifying a periodic/sporadic task that executes for a given amount of runtime at each instance, and that is scheduled according to the urgency of its own timing constraints needs, in general, a way of declaring:

- a (maximum/typical) instance execution time,
- a minimum interval between consecutive instances,
- a time constraint by which each instance must be completed.

Therefore:

- a new struct sched_attr, containing all the necessary fields is provided;
- the new scheduling related syscalls that manipulate it, i.e., sched_setattr() and sched_getattr() are implemented.

For debugging purposes, the leftover runtime and absolute deadline of a SCHED_DEADLINE task can be retrieved through /proc/<pid>/sched (entries dl.runtime and dl.deadline, both values in ns). A programmatic way to retrieve these values from production code is under discussion.

4.5.3 4.3 Default behavior

The default value for SCHED_DEADLINE bandwidth is to have rt_runtime equal to 950000. With rt_period equal to 1000000, by default, it means that -deadline tasks can use at most 95%, multiplied by the number of CPUs that compose the root_domain, for each root_domain. This means that non -deadline tasks will receive at least 5% of the CPU time, and that -deadline tasks will receive their runtime with a guaranteed worst-case delay respect to the “deadline” parameter. If “deadline” = “period” and the cpuset mechanism is used to implement partitioned scheduling (see Section 5), then this simple setting of the bandwidth management is able to deterministically guarantee that -deadline tasks will receive their runtime in a period.

Finally, notice that in order not to jeopardize the admission control a -deadline task cannot fork.

4.5.4 4.4 Behavior of sched_yield()

When a SCHED_DEADLINE task calls sched_yield(), it gives up its remaining runtime and is immediately throttled, until the next period, when its runtime will be replenished (a special flag dl_yielded is set and used to handle correctly throttling and runtime replenishment after a call to sched_yield()).

This behavior of sched_yield() allows the task to wake-up exactly at the beginning of the next period. Also, this may be useful in the future with bandwidth reclaiming mechanisms, where sched_yield() will make the leftoever runtime available for reclamation by other SCHED_DEADLINE tasks.

4.6 5. Tasks CPU affinity

-deadline tasks cannot have an affinity mask smaller that the entire root_domain they are created on. However, affinities can be specified through the cpuset facility (Documentation/admin-guide/cgroup-v1/cpusets.rst).

4.6.1 5.1 SCHED_DEADLINE and cpusets HOWTO

An example of a simple configuration (pin a -deadline task to CPU0) follows (rt-app is used to create a -deadline task):

```
mkdir /dev/cpuset
mount -t cgroup -o cpuset cpuset /dev/cpuset
cd /dev/cpuset
mkdir cpu0
echo 0 > cpu0/cpuset.cpus
echo 0 > cpu0/cpuset.mems
echo 1 > cpuset.cpu_exclusive
echo 0 > cpuset.sched_load_balance
echo 1 > cpu0/cpuset.cpu_exclusive
echo 1 > cpu0/cpuset.mem_exclusive
echo $$ > cpu0/tasks
```

```
rt-app -t 100000:10000:d:0 -D5 # it is now actually superfluous to␣
↪specify
                                # task affinity
```

4.7 6. Future plans

Still missing:

- programmatic way to retrieve current runtime and absolute deadline
- refinements to deadline inheritance, especially regarding the possibility of retaining bandwidth isolation among non-interacting tasks. This is being studied from both theoretical and practical points of view, and hopefully we should be able to produce some demonstrative code soon;
- (c)group based bandwidth management, and maybe scheduling;
- access control for non-root users (and related security concerns to address), which is the best way to allow unprivileged use of the mechanisms and how to prevent non-root users "cheat" the system?

As already discussed, we are planning also to merge this work with the EDF throttling patches [https://lore.kernel.org/r/cover.1266931410.git.fabio@helm.retis] but we still are in the preliminary phases of the merge and we really seek feedback that would help us decide on the direction it should take.

4.8 Appendix A. Test suite

The SCHED_DEADLINE policy can be easily tested using two applications that are part of a wider Linux Scheduler validation suite. The suite is available as a GitHub repository: https://github.com/scheduler-tools.

The first testing application is called rt-app and can be used to start multiple threads with specific parameters. rt-app supports SCHED_{OTHER,FIFO,RR,DEADLINE} scheduling policies and their related parameters (e.g., niceness, priority, runtime/deadline/period). rt-app is a valuable tool, as it can be used to synthetically recreate certain workloads (maybe mimicking real use-cases) and evaluate how the scheduler behaves under such workloads. In this way, results are easily reproducible. rt-app is available at: https://github.com/scheduler-tools/rt-app.

Thread parameters can be specified from the command line, with something like this:

```
# rt-app -t 100000:10000:d -t 150000:20000:f:10 -D5
```

The above creates 2 threads. The first one, scheduled by SCHED_DEADLINE, executes for 10ms every 100ms. The second one, scheduled at SCHED_FIFO priority 10, executes for 20ms every 150ms. The test will run for a total of 5 seconds.

More interestingly, configurations can be described with a json file that can be passed as input to rt-app with something like this:

```
# rt-app my_config.json
```

The parameters that can be specified with the second method are a superset of the command line options. Please refer to rt-app documentation for more details (*<rt-app-sources>/doc/*.json*).

The second testing application is a modification of schedtool, called schedtool-dl, which can be used to setup SCHED_DEADLINE parameters for a certain pid/application. schedtool-dl is available at: https://github.com/scheduler-tools/schedtool-dl.git.

The usage is straightforward:

```
# schedtool -E -t 10000000:100000000 -e ./my_cpuhog_app
```

With this, my_cpuhog_app is put to run inside a SCHED_DEADLINE reservation of 10ms every 100ms (note that parameters are expressed in microseconds). You can also use schedtool to create a reservation for an already running application, given that you know its pid:

```
# schedtool -E -t 10000000:100000000 my_app_pid
```

4.9 Appendix B. Minimal main()

We provide in what follows a simple (ugly) self-contained code snippet showing how SCHED_DEADLINE reservations can be created by a real-time application developer:

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <linux/unistd.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <sys/syscall.h>
#include <pthread.h>

#define gettid() syscall(__NR_gettid)

#define SCHED_DEADLINE       6

/* XXX use the proper syscall numbers */
#ifdef __x86_64__
#define __NR_sched_setattr           314
#define __NR_sched_getattr           315
#endif

#ifdef __i386__
```

```
#define __NR_sched_setattr           351
#define __NR_sched_getattr           352
#endif

#ifdef __arm__
#define __NR_sched_setattr           380
#define __NR_sched_getattr           381
#endif

static volatile int done;

struct sched_attr {
     __u32 size;

     __u32 sched_policy;
     __u64 sched_flags;

     /* SCHED_NORMAL, SCHED_BATCH */
     __s32 sched_nice;

     /* SCHED_FIFO, SCHED_RR */
     __u32 sched_priority;

     /* SCHED_DEADLINE (nsec) */
     __u64 sched_runtime;
     __u64 sched_deadline;
     __u64 sched_period;
};

int sched_setattr(pid_t pid,
               const struct sched_attr *attr,
               unsigned int flags)
{
     return syscall(__NR_sched_setattr, pid, attr, flags);
}

int sched_getattr(pid_t pid,
               struct sched_attr *attr,
               unsigned int size,
               unsigned int flags)
{
     return syscall(__NR_sched_getattr, pid, attr, size, flags);
}

void *run_deadline(void *data)
{
     struct sched_attr attr;
     int x = 0;
     int ret;
     unsigned int flags = 0;
```

```
     printf("deadline thread started [%ld]\n", gettid());

     attr.size = sizeof(attr);
     attr.sched_flags = 0;
     attr.sched_nice = 0;
     attr.sched_priority = 0;

     /* This creates a 10ms/30ms reservation */
     attr.sched_policy = SCHED_DEADLINE;
     attr.sched_runtime = 10 * 1000 * 1000;
     attr.sched_period = attr.sched_deadline = 30 * 1000 * 1000;

     ret = sched_setattr(0, &attr, flags);
     if (ret < 0) {
             done = 0;
             perror("sched_setattr");
             exit(-1);
     }

     while (!done) {
             x++;
     }

     printf("deadline thread dies [%ld]\n", gettid());
     return NULL;
}

int main (int argc, char **argv)
{
     pthread_t thread;

     printf("main thread [%ld]\n", gettid());

     pthread_create(&thread, NULL, run_deadline, NULL);

     sleep(10);

     done = 1;
     pthread_join(thread, NULL);

     printf("main dies [%ld]\n", gettid());
     return 0;
}
```

CHAPTER

FIVE

CFS SCHEDULER

5.1 1. OVERVIEW

CFS stands for “Completely Fair Scheduler,” and is the new “desktop” process scheduler implemented by Ingo Molnar and merged in Linux 2.6.23. It is the replacement for the previous vanilla scheduler’s SCHED_OTHER interactivity code.

80% of CFS’s design can be summed up in a single sentence: CFS basically models an “ideal, precise multi-tasking CPU” on real hardware.

“Ideal multi-tasking CPU” is a (non-existent :-)) CPU that has 100% physical power and which can run each task at precise equal speed, in parallel, each at 1/nr_running speed. For example: if there are 2 tasks running, then it runs each at 50% physical power — i.e., actually in parallel.

On real hardware, we can run only a single task at once, so we have to introduce the concept of “virtual runtime.” The virtual runtime of a task specifies when its next timeslice would start execution on the ideal multi-tasking CPU described above. In practice, the virtual runtime of a task is its actual runtime normalized to the total number of running tasks.

5.2 2. FEW IMPLEMENTATION DETAILS

In CFS the virtual runtime is expressed and tracked via the per-task p->se.vruntime (nanosec-unit) value. This way, it’s possible to accurately timestamp and measure the “expected CPU time” a task should have gotten.

> Small detail: on “ideal” hardware, at any time all tasks would have the same p->se.vruntime value — i.e., tasks would execute simultaneously and no task would ever get “out of balance” from the “ideal” share of CPU time.

CFS’s task picking logic is based on this p->se.vruntime value and it is thus very simple: it always tries to run the task with the smallest p->se.vruntime value (i.e., the task which executed least so far). CFS always tries to split up CPU time between runnable tasks as close to “ideal multitasking hardware” as possible.

Most of the rest of CFS’s design just falls out of this really simple concept, with a few add-on embellishments like nice levels, multiprocessing and various algorithm variants to recognize sleepers.

5.3 3. THE RBTREE

CFS's design is quite radical: it does not use the old data structures for the runqueues, but it uses a time-ordered rbtree to build a "timeline" of future task execution, and thus has no "array switch" artifacts (by which both the previous vanilla scheduler and RSDL/SD are affected).

CFS also maintains the rq->cfs.min_vruntime value, which is a monotonic increasing value tracking the smallest vruntime among all tasks in the runqueue. The total amount of work done by the system is tracked using min_vruntime; that value is used to place newly activated entities on the left side of the tree as much as possible.

The total number of running tasks in the runqueue is accounted through the rq->cfs.load value, which is the sum of the weights of the tasks queued on the runqueue.

CFS maintains a time-ordered rbtree, where all runnable tasks are sorted by the p->se.vruntime key. CFS picks the "leftmost" task from this tree and sticks to it. As the system progresses forwards, the executed tasks are put into the tree more and more to the right — slowly but surely giving a chance for every task to become the "leftmost task" and thus get on the CPU within a deterministic amount of time.

Summing up, CFS works like this: it runs a task a bit, and when the task schedules (or a scheduler tick happens) the task's CPU usage is "accounted for": the (small) time it just spent using the physical CPU is added to p->se.vruntime. Once p->se.vruntime gets high enough so that another task becomes the "leftmost task" of the time-ordered rbtree it maintains (plus a small amount of "granularity" distance relative to the leftmost task so that we do not over-schedule tasks and trash the cache), then the new leftmost task is picked and the current task is preempted.

5.4 4. SOME FEATURES OF CFS

CFS uses nanosecond granularity accounting and does not rely on any jiffies or other HZ detail. Thus the CFS scheduler has no notion of "timeslices" in the way the previous scheduler had, and has no heuristics whatsoever. There is only one central tunable (you have to switch on CONFIG_SCHED_DEBUG):

    /proc/sys/kernel/sched_min_granularity_ns

which can be used to tune the scheduler from "desktop" (i.e., low latencies) to "server" (i.e., good batching) workloads. It defaults to a setting suitable for desktop workloads. SCHED_BATCH is handled by the CFS scheduler module too.

Due to its design, the CFS scheduler is not prone to any of the "attacks" that exist today against the heuristics of the stock scheduler: fiftyp.c, thud.c, chew.c, ring-test.c, massive_intr.c all work fine and do not impact interactivity and produce the expected behavior.

The CFS scheduler has a much stronger handling of nice levels and SCHED_BATCH than the previous vanilla scheduler: both types of workloads are isolated much more aggressively.

SMP load-balancing has been reworked/sanitized: the runqueue-walking assumptions are gone from the load-balancing code now, and iterators of the scheduling modules are used. The balancing code got quite a bit simpler as a result.

5.5 5. Scheduling policies

CFS implements three scheduling policies:

- SCHED_NORMAL (traditionally called SCHED_OTHER): The scheduling policy that is used for regular tasks.
- SCHED_BATCH: Does not preempt nearly as often as regular tasks would, thereby allowing tasks to run longer and make better use of caches but at the cost of interactivity. This is well suited for batch jobs.
- SCHED_IDLE: This is even weaker than nice 19, but its not a true idle timer scheduler in order to avoid to get into priority inversion problems which would deadlock the machine.

SCHED_FIFO/_RR are implemented in sched/rt.c and are as specified by POSIX.

The command chrt from util-linux-ng 2.13.1.1 can set all of these except SCHED_IDLE.

5.6 6. SCHEDULING CLASSES

The new CFS scheduler has been designed in such a way to introduce “Scheduling Classes,” an extensible hierarchy of scheduler modules. These modules encapsulate scheduling policy details and are handled by the scheduler core without the core code assuming too much about them.

sched/fair.c implements the CFS scheduler described above.

sched/rt.c implements SCHED_FIFO and SCHED_RR semantics, in a simpler way than the previous vanilla scheduler did. It uses 100 runqueues (for all 100 RT priority levels, instead of 140 in the previous scheduler) and it needs no expired array.

Scheduling classes are implemented through the sched_class structure, which contains hooks to functions that must be called whenever an interesting event occurs.

This is the (partial) list of the hooks:

- enqueue_task(...)

  Called when a task enters a runnable state. It puts the scheduling entity (task) into the red-black tree and increments the nr_running variable.

- dequeue_task(...)

  When a task is no longer runnable, this function is called to keep the corresponding scheduling entity out of the red-black tree. It decrements the nr_running variable.

- yield_task(...)

  This function is basically just a dequeue followed by an enqueue, unless the compat_yield sysctl is turned on; in that case, it places the scheduling entity at the right-most end of the red-black tree.

- check_preempt_curr(...)

  This function checks if a task that entered the runnable state should preempt the currently running task.

- pick_next_task(...)

  This function chooses the most appropriate task eligible to run next.

- set_curr_task(...)

  This function is called when a task changes its scheduling class or changes its task group.

- task_tick(...)

  This function is mostly called from time tick functions; it might lead to process switch. This drives the running preemption.

5.7 7. GROUP SCHEDULER EXTENSIONS TO CFS

Normally, the scheduler operates on individual tasks and strives to provide fair CPU time to each task. Sometimes, it may be desirable to group tasks and provide fair CPU time to each such task group. For example, it may be desirable to first provide fair CPU time to each user on the system and then to each task belonging to a user.

CONFIG_CGROUP_SCHED strives to achieve exactly that. It lets tasks to be grouped and divides CPU time fairly among such groups.

CONFIG_RT_GROUP_SCHED permits to group real-time (i.e., SCHED_FIFO and SCHED_RR) tasks.

CONFIG_FAIR_GROUP_SCHED permits to group CFS (i.e., SCHED_NORMAL and SCHED_BATCH) tasks.

> These options need CONFIG_CGROUPS to be defined, and let the administrator create arbitrary groups of tasks, using the "cgroup" pseudo filesystem. See Documentation/admin-guide/cgroup-v1/cgroups.rst for more information about this filesystem.

When CONFIG_FAIR_GROUP_SCHED is defined, a "cpu.shares" file is created for each group created using the pseudo filesystem. See example steps below to create task groups and modify their CPU share using the "cgroups" pseudo filesystem:

```
# mount -t tmpfs cgroup_root /sys/fs/cgroup
# mkdir /sys/fs/cgroup/cpu
# mount -t cgroup -ocpu none /sys/fs/cgroup/cpu
# cd /sys/fs/cgroup/cpu

# mkdir multimedia      # create "multimedia" group of tasks
# mkdir browser         # create "browser" group of tasks

# #Configure the multimedia group to receive twice the CPU bandwidth
# #that of browser group

# echo 2048 > multimedia/cpu.shares
# echo 1024 > browser/cpu.shares

# firefox &     # Launch firefox and move it to "browser" group
# echo <firefox_pid> > browser/tasks
```

```
# #Launch gmplayer (or your favourite movie player)
# echo <movie_player_pid> > multimedia/tasks
```

CHAPTER

SIX

SCHEDULER DOMAINS

Each CPU has a “base” scheduling domain (struct sched_domain). The domain hierarchy is built from these base domains via the ->parent pointer. ->parent MUST be NULL terminated, and domain structures should be per-CPU as they are locklessly updated.

Each scheduling domain spans a number of CPUs (stored in the ->span field). A domain’s span MUST be a superset of it child’s span (this restriction could be relaxed if the need arises), and a base domain for CPU i MUST span at least i. The top domain for each CPU will generally span all CPUs in the system although strictly it doesn’t have to, but this could lead to a case where some CPUs will never be given tasks to run unless the CPUs allowed mask is explicitly set. A sched domain’s span means “balance process load among these CPUs”.

Each scheduling domain must have one or more CPU groups (struct sched_group) which are organised as a circular one way linked list from the ->groups pointer. The union of cpumasks of these groups MUST be the same as the domain’s span. The group pointed to by the ->groups pointer MUST contain the CPU to which the domain belongs. Groups may be shared among CPUs as they contain read only data after they have been set up. The intersection of cpumasks from any two of these groups may be non empty. If this is the case the SD_OVERLAP flag is set on the corresponding scheduling domain and its groups may not be shared between CPUs.

Balancing within a sched domain occurs between groups. That is, each group is treated as one entity. The load of a group is defined as the sum of the load of each of its member CPUs, and only when the load of a group becomes out of balance are tasks moved between groups.

In kernel/sched/core.c, trigger_load_balance() is run periodically on each CPU through scheduler_tick(). It raises a softirq after the next regularly scheduled rebalancing event for the current runqueue has arrived. The actual load balancing workhorse, run_rebalance_domains()->rebalance_domains(), is then run in softirq context (SCHED_SOFTIRQ).

The latter function takes two arguments: the runqueue of current CPU and whether the CPU was idle at the time the scheduler_tick() happened and iterates over all sched domains our CPU is on, starting from its base domain and going up the ->parent chain. While doing that, it checks to see if the current domain has exhausted its rebalance interval. If so, it runs load_balance() on that domain. It then checks the parent sched_domain (if it exists), and the parent of the parent and so forth.

Initially, load_balance() finds the busiest group in the current sched domain. If it succeeds, it looks for the busiest runqueue of all the CPUs’ runqueues in that group. If it manages to find such a runqueue, it locks both our initial CPU’s runqueue and the newly found busiest one and starts moving tasks from it to our runqueue. The exact number of tasks amounts to an imbalance previously computed while iterating over this sched domain’s groups.

6.1 Implementing sched domains

The “base” domain will “span” the first level of the hierarchy. In the case of SMT, you’ll span all siblings of the physical CPU, with each group being a single virtual CPU.

In SMP, the parent of the base domain will span all physical CPUs in the node. Each group being a single physical CPU. Then with NUMA, the parent of the SMP domain will span the entire machine, with each group having the cpumask of a node. Or, you could do multi-level NUMA or Opteron, for example, might have just one domain covering its one NUMA level.

The implementor should read comments in include/linux/sched/sd_flags.h: SD_* to get an idea of the specifics and what to tune for the SD flags of a sched_domain.

Architectures may override the generic domain builder and the default SD flags for a given topology level by creating a sched_domain_topology_level array and calling set_sched_topology() with this array as the parameter.

The sched-domains debugging infrastructure can be enabled by enabling CONFIG_SCHED_DEBUG and adding ‘sched_verbose’ to your cmdline. If you forgot to tweak your cmdline, you can also flip the /sys/kernel/debug/sched/verbose knob. This enables an error checking parse of the sched domains which should catch most possible errors (described above). It also prints out the domain structure in a visual format.

CHAPTER

SEVEN

CAPACITY AWARE SCHEDULING

7.1 1. CPU Capacity

7.1.1 1.1 Introduction

Conventional, homogeneous SMP platforms are composed of purely identical CPUs. Heterogeneous platforms on the other hand are composed of CPUs with different performance characteristics - on such platforms, not all CPUs can be considered equal.

CPU capacity is a measure of the performance a CPU can reach, normalized against the most performant CPU in the system. Heterogeneous systems are also called asymmetric CPU capacity systems, as they contain CPUs of different capacities.

Disparity in maximum attainable performance (IOW in maximum CPU capacity) stems from two factors:

- not all CPUs may have the same microarchitecture (µarch).
- with Dynamic Voltage and Frequency Scaling (DVFS), not all CPUs may be physically able to attain the higher Operating Performance Points (OPP).

Arm big.LITTLE systems are an example of both. The big CPUs are more performance-oriented than the LITTLE ones (more pipeline stages, bigger caches, smarter predictors, etc), and can usually reach higher OPPs than the LITTLE ones can.

CPU performance is usually expressed in Millions of Instructions Per Second (MIPS), which can also be expressed as a given amount of instructions attainable per Hz, leading to:

```
capacity(cpu) = work_per_hz(cpu) * max_freq(cpu)
```

7.1.2 1.2 Scheduler terms

Two different capacity values are used within the scheduler. A CPU's `capacity_orig` is its maximum attainable capacity, i.e. its maximum attainable performance level. A CPU's `capacity` is its `capacity_orig` to which some loss of available performance (e.g. time spent handling IRQs) is subtracted.

Note that a CPU's `capacity` is solely intended to be used by the CFS class, while `capacity_orig` is class-agnostic. The rest of this document will use the term `capacity` interchangeably with `capacity_orig` for the sake of brevity.

7.1.3 1.3 Platform examples

1.3.1 Identical OPPs

Consider an hypothetical dual-core asymmetric CPU capacity system where

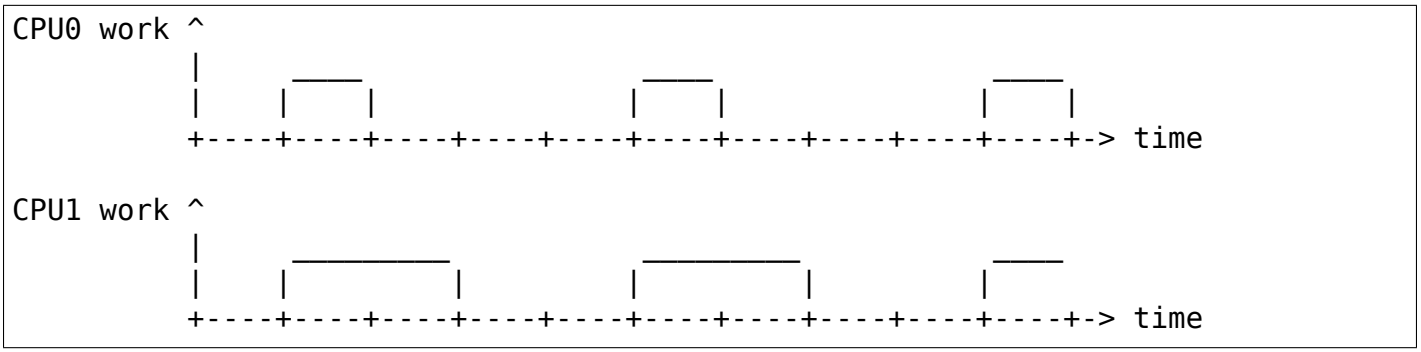
- work_per_hz(CPU0) = W
- work_per_hz(CPU1) = W/2
- all CPUs are running at the same fixed frequency

By the above definition of capacity:

- capacity(CPU0) = C
- capacity(CPU1) = C/2

To draw the parallel with Arm big.LITTLE, CPU0 would be a big while CPU1 would be a LITTLE.

With a workload that periodically does a fixed amount of work, you will get an execution trace like so:



CPU0 has the highest capacity in the system (C), and completes a fixed amount of work W in T units of time. On the other hand, CPU1 has half the capacity of CPU0, and thus only completes W/2 in T.

1.3.2 Different max OPPs

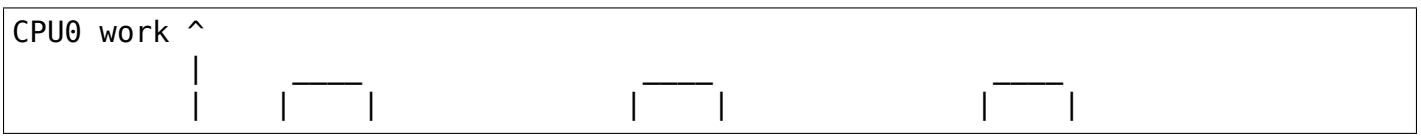
Usually, CPUs of different capacity values also have different maximum OPPs. Consider the same CPUs as above (i.e. same work_per_hz()) with:

- max_freq(CPU0) = F
- max_freq(CPU1) = 2/3 * F

This yields:

- capacity(CPU0) = C
- capacity(CPU1) = C/3

Executing the same workload as described in 1.3.1, which each CPU running at its maximum frequency results in:



```
          +----+----+----+----+----+----+----+----+----+----+-> time

                              workload on CPU1
CPU1 work ^
          |     _______________      _______________      ____
          |    |               |    |               |    |
          +----+----+----+----+----+----+----+----+----+----+-> time
```

7.1.4 1.4 Representation caveat

It should be noted that having a *single* value to represent differences in CPU performance is somewhat of a contentious point. The relative performance difference between two different µarchs could be X% on integer operations, Y% on floating point operations, Z% on branches, and so on. Still, results using this simple approach have been satisfactory for now.

7.2 2. Task utilization

7.2.1 2.1 Introduction

Capacity aware scheduling requires an expression of a task's requirements with regards to CPU capacity. Each scheduler class can express this differently, and while task utilization is specific to CFS, it is convenient to describe it here in order to introduce more generic concepts.

Task utilization is a percentage meant to represent the throughput requirements of a task. A simple approximation of it is the task's duty cycle, i.e.:

```
task_util(p) = duty_cycle(p)
```

On an SMP system with fixed frequencies, 100% utilization suggests the task is a busy loop. Conversely, 10% utilization hints it is a small periodic task that spends more time sleeping than executing. Variable CPU frequencies and asymmetric CPU capacities complexify this somewhat; the following sections will expand on these.

7.2.2 2.2 Frequency invariance

One issue that needs to be taken into account is that a workload's duty cycle is directly impacted by the current OPP the CPU is running at. Consider running a periodic workload at a given frequency F:

```
CPU work ^
         |     ____                ____                ____
         |    |    |              |    |              |    |
         +----+----+----+----+----+----+----+----+----+----+-> time
```

This yields duty_cycle(p) == 25%.

Now, consider running the *same* workload at frequency F/2:

```
CPU work ^
         |     _________           _________           ____
         |    |         |         |         |         |
         +----+----+----+----+----+----+----+----+----+----+-> time
```

This yields duty_cycle(p) == 50%, despite the task having the exact same behaviour (i.e. executing the same amount of work) in both executions.

The task utilization signal can be made frequency invariant using the following formula:

```
task_util_freq_inv(p) = duty_cycle(p) * (curr_frequency(cpu) / max_
↪frequency(cpu))
```

Applying this formula to the two examples above yields a frequency invariant task utilization of 25%.

7.2.3 2.3 CPU invariance

CPU capacity has a similar effect on task utilization in that running an identical workload on CPUs of different capacity values will yield different duty cycles.

Consider the system described in 1.3.2., i.e.:

```
- capacity(CPU0) = C
- capacity(CPU1) = C/3
```

Executing a given periodic workload on each CPU at their maximum frequency would result in:

```
CPU0 work ^
          |     ____              ____              ____
          |    |    |            |    |            |    |
          +----+----+----+----+----+----+----+----+----+----+-> time

CPU1 work ^
          |     _______________   _______________   ____
          |    |               |  |               |  |
          +----+----+----+----+----+----+----+----+----+----+-> time
```

IOW,

- duty_cycle(p) == 25% if p runs on CPU0 at its maximum frequency
- duty_cycle(p) == 75% if p runs on CPU1 at its maximum frequency

The task utilization signal can be made CPU invariant using the following formula:

```
task_util_cpu_inv(p) = duty_cycle(p) * (capacity(cpu) / max_capacity)
```

with `max_capacity` being the highest CPU capacity value in the system. Applying this formula to the above example above yields a CPU invariant task utilization of 25%.

7.2.4 2.4 Invariant task utilization

Both frequency and CPU invariance need to be applied to task utilization in order to obtain a truly invariant signal. The pseudo-formula for a task utilization that is both CPU and frequency invariant is thus, for a given task p:

```
                                    curr_frequency(cpu)   capacity(cpu)
task_util_inv(p) = duty_cycle(p) * ------------------- * -------------
                                    max_frequency(cpu)    max_capacity
```

In other words, invariant task utilization describes the behaviour of a task as if it were running on the highest-capacity CPU in the system, running at its maximum frequency.

Any mention of task utilization in the following sections will imply its invariant form.

7.2.5 2.5 Utilization estimation

Without a crystal ball, task behaviour (and thus task utilization) cannot accurately be predicted the moment a task first becomes runnable. The CFS class maintains a handful of CPU and task signals based on the Per-Entity Load Tracking (PELT) mechanism, one of those yielding an *average* utilization (as opposed to instantaneous).

This means that while the capacity aware scheduling criteria will be written considering a "true" task utilization (using a crystal ball), the implementation will only ever be able to use an estimator thereof.

7.3 3. Capacity aware scheduling requirements

7.3.1 3.1 CPU capacity

Linux cannot currently figure out CPU capacity on its own, this information thus needs to be handed to it. Architectures must define arch_scale_cpu_capacity() for that purpose.

The arm and arm64 architectures directly map this to the arch_topology driver CPU scaling data, which is derived from the capacity-dmips-mhz CPU binding; see Documentation/devicetree/bindings/arm/cpu-capacity.txt.

7.3.2 3.2 Frequency invariance

As stated in 2.2, capacity-aware scheduling requires a frequency-invariant task utilization. Architectures must define arch_scale_freq_capacity(cpu) for that purpose.

Implementing this function requires figuring out at which frequency each CPU have been running at. One way to implement this is to leverage hardware counters whose increment rate scale with a CPU's current frequency (APERF/MPERF on x86, AMU on arm64). Another is to directly hook into cpufreq frequency transitions, when the kernel is aware of the switched-to frequency (also employed by arm/arm64).

7.4 4. Scheduler topology

During the construction of the sched domains, the scheduler will figure out whether the system exhibits asymmetric CPU capacities. Should that be the case:

- The sched_asym_cpucapacity static key will be enabled.
- The SD_ASYM_CPUCAPACITY_FULL flag will be set at the lowest sched_domain level that spans all unique CPU capacity values.
- The SD_ASYM_CPUCAPACITY flag will be set for any sched_domain that spans CPUs with any range of asymmetry.

The sched_asym_cpucapacity static key is intended to guard sections of code that cater to asymmetric CPU capacity systems. Do note however that said key is *system-wide*. Imagine the following setup using cpusets:

```
capacity    C/2          C

          ________   ________
         /        \ /        \
CPUs     0  1  2  3  4  5  6  7
         \__/  \_____________/
cpusets   cs0         cs1
```

Which could be created via:

```
mkdir /sys/fs/cgroup/cpuset/cs0
echo 0-1 > /sys/fs/cgroup/cpuset/cs0/cpuset.cpus
echo 0 > /sys/fs/cgroup/cpuset/cs0/cpuset.mems

mkdir /sys/fs/cgroup/cpuset/cs1
echo 2-7 > /sys/fs/cgroup/cpuset/cs1/cpuset.cpus
echo 0 > /sys/fs/cgroup/cpuset/cs1/cpuset.mems

echo 0 > /sys/fs/cgroup/cpuset/cpuset.sched_load_balance
```

Since there *is* CPU capacity asymmetry in the system, the sched_asym_cpucapacity static key will be enabled. However, the sched_domain hierarchy of CPUs 0-1 spans a single capacity value: SD_ASYM_CPUCAPACITY isn't set in that hierarchy, it describes an SMP island and should be treated as such.

Therefore, the 'canonical' pattern for protecting codepaths that cater to asymmetric CPU capacities is to:

- Check the sched_asym_cpucapacity static key
- If it is enabled, then also check for the presence of SD_ASYM_CPUCAPACITY in the sched_domain hierarchy (if relevant, i.e. the codepath targets a specific CPU or group thereof)

7.5 5. Capacity aware scheduling implementation

7.5.1 5.1 CFS

5.1.1 Capacity fitness

The main capacity scheduling criterion of CFS is:

```
task_util(p) < capacity(task_cpu(p))
```

This is commonly called the capacity fitness criterion, i.e. CFS must ensure a task "fits" on its CPU. If it is violated, the task will need to achieve more work than what its CPU can provide: it will be CPU-bound.

Furthermore, uclamp lets userspace specify a minimum and a maximum utilization value for a task, either via sched_setattr() or via the cgroup interface (see Documentation/admin-guide/cgroup-v2.rst). As its name imply, this can be used to clamp task_util() in the previous criterion.

5.1.2 Wakeup CPU selection

CFS task wakeup CPU selection follows the capacity fitness criterion described above. On top of that, uclamp is used to clamp the task utilization values, which lets userspace have more leverage over the CPU selection of CFS tasks. IOW, CFS wakeup CPU selection searches for a CPU that satisfies:

```
clamp(task_util(p), task_uclamp_min(p), task_uclamp_max(p)) < capacity(cpu)
```

By using uclamp, userspace can e.g. allow a busy loop (100% utilization) to run on any CPU by giving it a low uclamp.max value. Conversely, it can force a small periodic task (e.g. 10% utilization) to run on the highest-performance CPUs by giving it a high uclamp.min value.

Note: Wakeup CPU selection in CFS can be eclipsed by Energy Aware Scheduling (EAS), which is described in *Energy Aware Scheduling*.

5.1.3 Load balancing

A pathological case in the wakeup CPU selection occurs when a task rarely sleeps, if at all - it thus rarely wakes up, if at all. Consider:

```
w == wakeup event

capacity(CPU0) = C
capacity(CPU1) = C / 3

                          workload on CPU0
CPU work ^
         |     _________           _________           ____
         |    |         |         |         |         |
```

```
          +----+----+----+----+----+----+----+----+----+----+-> time
               w                   w                   w

                            workload on CPU1
CPU work ^
          |      ____________________________________________
          |     |
          +----+----+----+----+----+----+----+----+----+----+->
               w
```

This workload should run on CPU0, but if the task either:

- was improperly scheduled from the start (inaccurate initial utilization estimation)
- was properly scheduled from the start, but suddenly needs more processing power

then it might become CPU-bound, IOW `task_util(p) > capacity(task_cpu(p))`; the CPU capacity scheduling criterion is violated, and there may not be any more wakeup event to fix this up via wakeup CPU selection.

Tasks that are in this situation are dubbed “misfit” tasks, and the mechanism put in place to handle this shares the same name. Misfit task migration leverages the CFS load balancer, more specifically the active load balance part (which caters to migrating currently running tasks). When load balance happens, a misfit active load balance will be triggered if a misfit task can be migrated to a CPU with more capacity than its current one.

7.5.2 5.2 RT

5.2.1 Wakeup CPU selection

RT task wakeup CPU selection searches for a CPU that satisfies:

```
task_uclamp_min(p) <= capacity(task_cpu(cpu))
```

while still following the usual priority constraints. If none of the candidate CPUs can satisfy this capacity criterion, then strict priority based scheduling is followed and CPU capacities are ignored.

7.5.3 5.3 DL

5.3.1 Wakeup CPU selection

DL task wakeup CPU selection searches for a CPU that satisfies:

```
task_bandwidth(p) < capacity(task_cpu(p))
```

while still respecting the usual bandwidth and deadline constraints. If none of the candidate CPUs can satisfy this capacity criterion, then the task will remain on its current CPU.

CHAPTER

EIGHT

ENERGY AWARE SCHEDULING

8.1 1. Introduction

Energy Aware Scheduling (or EAS) gives the scheduler the ability to predict the impact of its decisions on the energy consumed by CPUs. EAS relies on an Energy Model (EM) of the CPUs to select an energy efficient CPU for each task, with a minimal impact on throughput. This document aims at providing an introduction on how EAS works, what are the main design decisions behind it, and details what is needed to get it to run.

Before going any further, please note that at the time of writing:

```
/!\ EAS does not support platforms with symmetric CPU topologies /!\
```

EAS operates only on heterogeneous CPU topologies (such as Arm big.LITTLE) because this is where the potential for saving energy through scheduling is the highest.

The actual EM used by EAS is _not_ maintained by the scheduler, but by a dedicated framework. For details about this framework and what it provides, please refer to its documentation (see Documentation/power/energy-model.rst).

8.2 2. Background and Terminology

To make it clear from the start:

- energy = [joule] (resource like a battery on powered devices)
- power = energy/time = [joule/second] = [watt]

The goal of EAS is to minimize energy, while still getting the job done. That is, we want to maximize:

```
performance [inst/s]
--------------------
    power [W]
```

which is equivalent to minimizing:

```
energy [J]
-----------
instruction
```

while still getting ‘good’ performance. It is essentially an alternative optimization objective to the current performance-only objective for the scheduler. This alternative considers two objectives: energy-efficiency and performance.

The idea behind introducing an EM is to allow the scheduler to evaluate the implications of its decisions rather than blindly applying energy-saving techniques that may have positive effects only on some platforms. At the same time, the EM must be as simple as possible to minimize the scheduler latency impact.

In short, EAS changes the way CFS tasks are assigned to CPUs. When it is time for the scheduler to decide where a task should run (during wake-up), the EM is used to break the tie between several good CPU candidates and pick the one that is predicted to yield the best energy consumption without harming the system’s throughput. The predictions made by EAS rely on specific elements of knowledge about the platform’s topology, which include the ‘capacity’ of CPUs, and their respective energy costs.

8.3 3. Topology information

EAS (as well as the rest of the scheduler) uses the notion of ‘capacity’ to differentiate CPUs with different computing throughput. The ‘capacity’ of a CPU represents the amount of work it can absorb when running at its highest frequency compared to the most capable CPU of the system. Capacity values are normalized in a 1024 range, and are comparable with the utilization signals of tasks and CPUs computed by the Per-Entity Load Tracking (PELT) mechanism. Thanks to capacity and utilization values, EAS is able to estimate how big/busy a task/CPU is, and to take this into consideration when evaluating performance vs energy trade-offs. The capacity of CPUs is provided via arch-specific code through the arch_scale_cpu_capacity() callback.

The rest of platform knowledge used by EAS is directly read from the Energy Model (EM) framework. The EM of a platform is composed of a power cost table per ‘performance domain’ in the system (see Documentation/power/energy-model.rst for futher details about performance domains).

The scheduler manages references to the EM objects in the topology code when the scheduling domains are built, or re-built. For each root domain (rd), the scheduler maintains a singly linked list of all performance domains intersecting the current rd->span. Each node in the list contains a pointer to a struct em_perf_domain as provided by the EM framework.

The lists are attached to the root domains in order to cope with exclusive cpuset configurations. Since the boundaries of exclusive cpusets do not necessarily match those of performance domains, the lists of different root domains can contain duplicate elements.

Example 1. Let us consider a platform with 12 CPUs, split in 3 performance domains (pd0, pd4 and pd8), organized as follows:

```
CPUs:   0 1 2 3 4 5 6 7 8 9 10 11
PDs:   |--pd0--|--pd4--|---pd8---|
RDs:   |----rd1----|-----rd2-----|
```

Now, consider that userspace decided to split the system with two exclusive cpusets, hence creating two independent root domains, each containing 6 CPUs. The two root domains are denoted rd1 and rd2 in the above figure. Since pd4 intersects with both rd1 and rd2, it will be present in the linked list ‘->pd’ attached to each of them:

- rd1->pd: pd0 -> pd4

- rd2->pd: pd4 -> pd8

Please note that the scheduler will create two duplicate list nodes for pd4 (one for each list). However, both just hold a pointer to the same shared data structure of the EM framework.

Since the access to these lists can happen concurrently with hotplug and other things, they are protected by RCU, like the rest of topology structures manipulated by the scheduler.

EAS also maintains a static key (sched_energy_present) which is enabled when at least one root domain meets all conditions for EAS to start. Those conditions are summarized in Section 6.

8.4 4. Energy-Aware task placement

EAS overrides the CFS task wake-up balancing code. It uses the EM of the platform and the PELT signals to choose an energy-efficient target CPU during wake-up balance. When EAS is enabled, select_task_rq_fair() calls find_energy_efficient_cpu() to do the placement decision. This function looks for the CPU with the highest spare capacity (CPU capacity - CPU utilization) in each performance domain since it is the one which will allow us to keep the frequency the lowest. Then, the function checks if placing the task there could save energy compared to leaving it on prev_cpu, i.e. the CPU where the task ran in its previous activation.

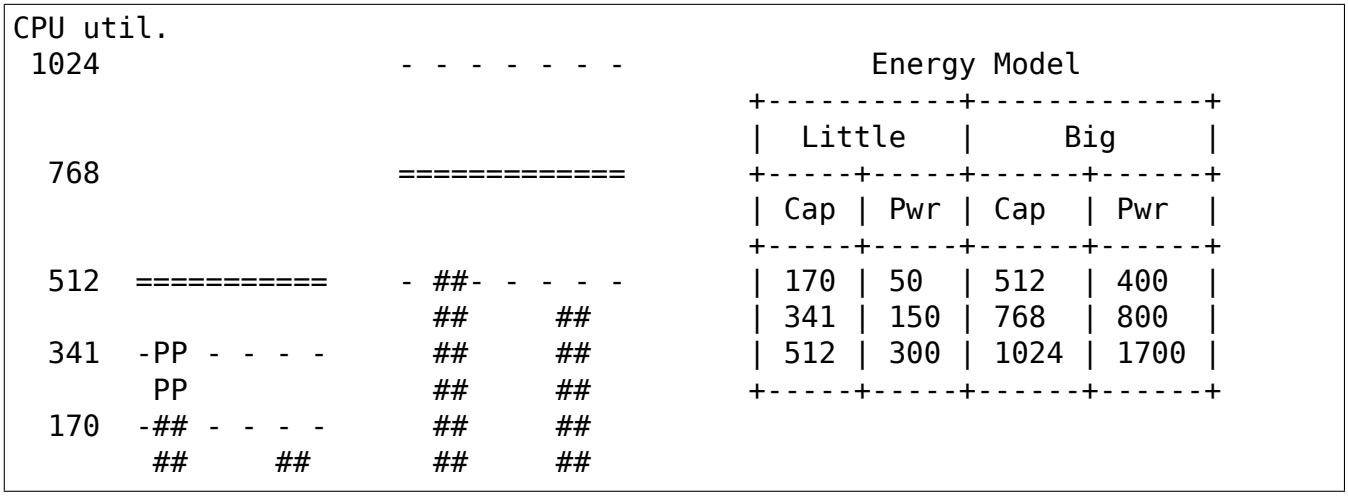
find_energy_efficient_cpu() uses compute_energy() to estimate what will be the energy consumed by the system if the waking task was migrated. compute_energy() looks at the current utilization landscape of the CPUs and adjusts it to 'simulate' the task migration. The EM framework provides the em_pd_energy() API which computes the expected energy consumption of each performance domain for the given utilization landscape.

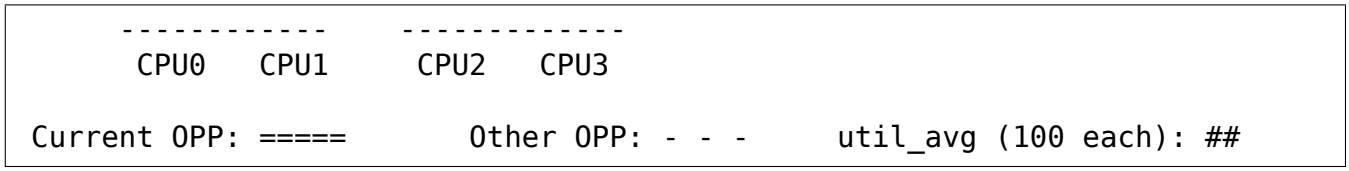
An example of energy-optimized task placement decision is detailed below.

Example 2. Let us consider a (fake) platform with 2 independent performance domains composed of two CPUs each. CPU0 and CPU1 are little CPUs; CPU2 and CPU3 are big.

The scheduler must decide where to place a task P whose util_avg = 200 and prev_cpu = 0.

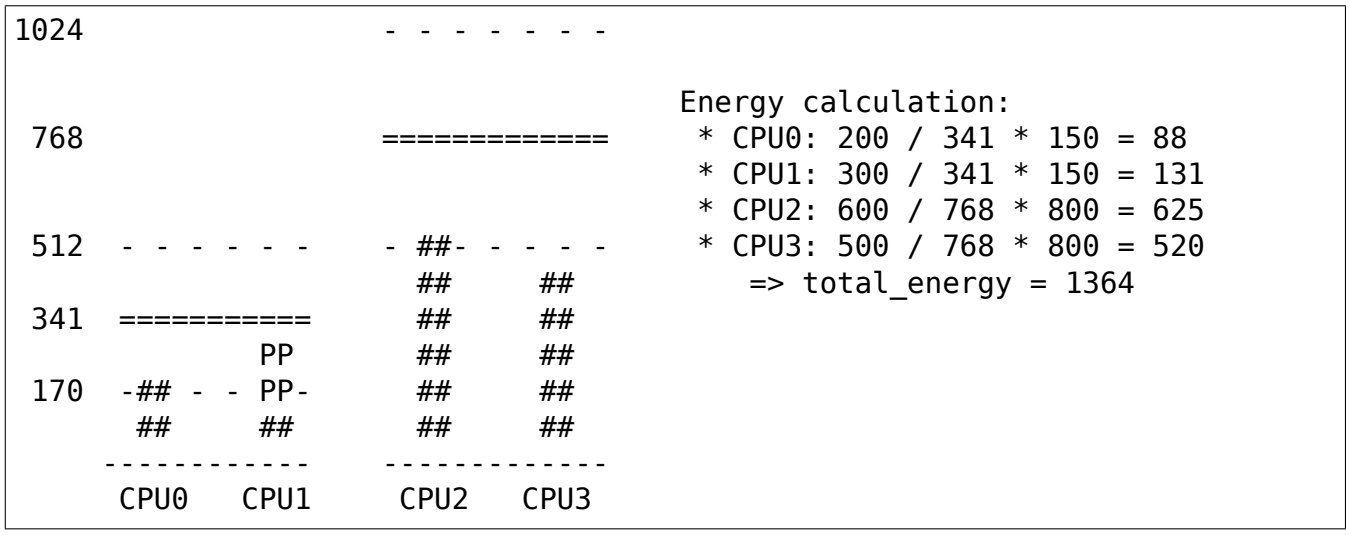
The current utilization landscape of the CPUs is depicted on the graph below. CPUs 0-3 have a util_avg of 400, 100, 600 and 500 respectively Each performance domain has three Operating Performance Points (OPPs). The CPU capacity and power cost associated with each OPP is listed in the Energy Model table. The util_avg of P is shown on the figures below as 'PP':



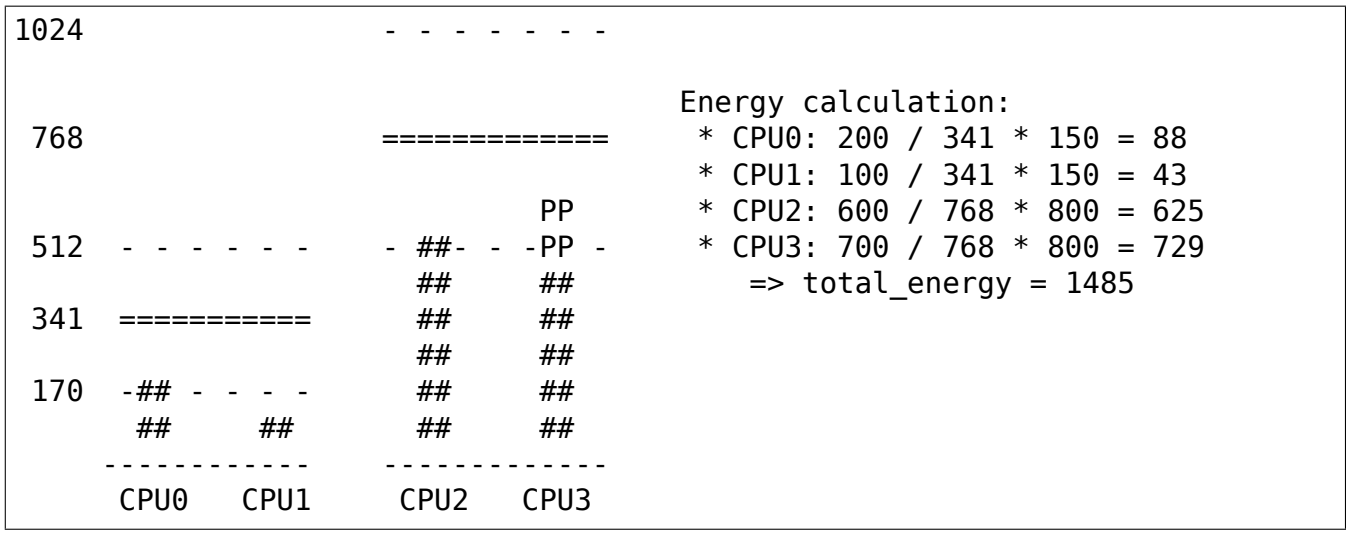


find_energy_efficient_cpu() will first look for the CPUs with the maximum spare capacity in the two performance domains. In this example, CPU1 and CPU3. Then it will estimate the energy of the system if P was placed on either of them, and check if that would save some energy compared to leaving P on CPU0. EAS assumes that OPPs follow utilization (which is coherent with the behaviour of the schedutil CPUFreq governor, see Section 6. for more details on this topic).

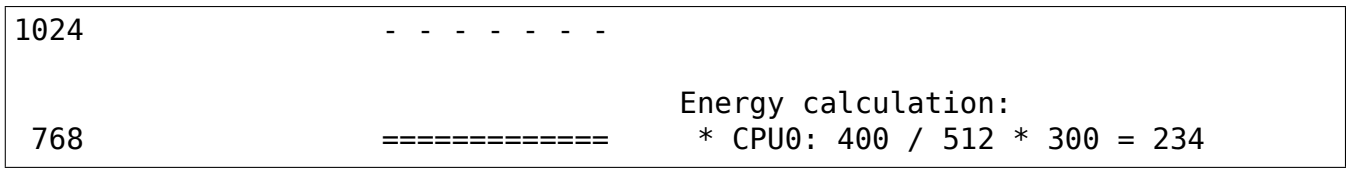
Case 1. P is migrated to CPU1:

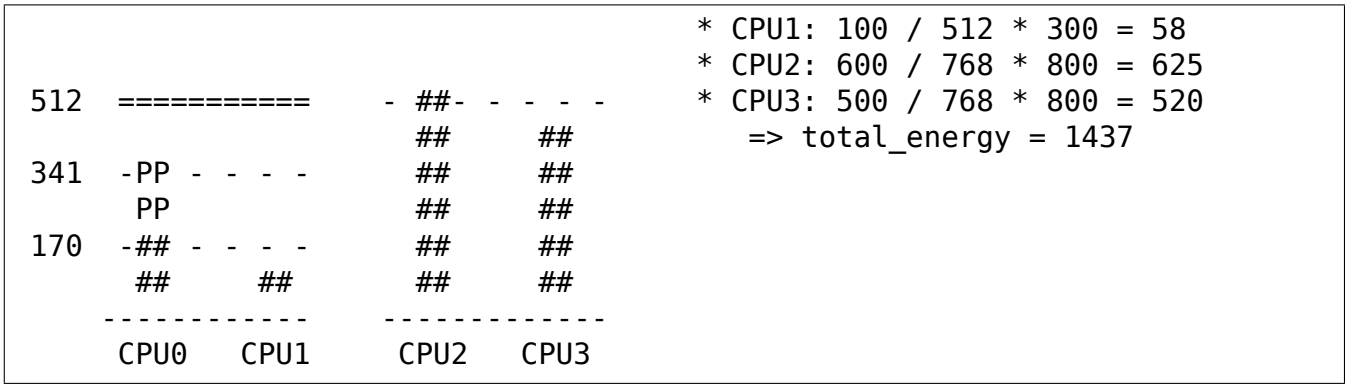


Case 2. P is migrated to CPU3:



Case 3. P stays on prev_cpu / CPU 0:





From these calculations, the Case 1 has the lowest total energy. So CPU 1 is be the best candidate from an energy-efficiency standpoint.

Big CPUs are generally more power hungry than the little ones and are thus used mainly when a task doesn't fit the littles. However, little CPUs aren't always necessarily more energy-efficient than big CPUs. For some systems, the high OPPs of the little CPUs can be less energy-efficient than the lowest OPPs of the bigs, for example. So, if the little CPUs happen to have enough utilization at a specific point in time, a small task waking up at that moment could be better of executing on the big side in order to save energy, even though it would fit on the little side.

And even in the case where all OPPs of the big CPUs are less energy-efficient than those of the little, using the big CPUs for a small task might still, under specific conditions, save energy. Indeed, placing a task on a little CPU can result in raising the OPP of the entire performance domain, and that will increase the cost of the tasks already running there. If the waking task is placed on a big CPU, its own execution cost might be higher than if it was running on a little, but it won't impact the other tasks of the little CPUs which will keep running at a lower OPP. So, when considering the total energy consumed by CPUs, the extra cost of running that one task on a big core can be smaller than the cost of raising the OPP on the little CPUs for all the other tasks.

The examples above would be nearly impossible to get right in a generic way, and for all platforms, without knowing the cost of running at different OPPs on all CPUs of the system. Thanks to its EM-based design, EAS should cope with them correctly without too many troubles. However, in order to ensure a minimal impact on throughput for high-utilization scenarios, EAS also implements another mechanism called 'over-utilization'.

8.5 5. Over-utilization

From a general standpoint, the use-cases where EAS can help the most are those involving a light/medium CPU utilization. Whenever long CPU-bound tasks are being run, they will require all of the available CPU capacity, and there isn't much that can be done by the scheduler to save energy without severly harming throughput. In order to avoid hurting performance with EAS, CPUs are flagged as 'over-utilized' as soon as they are used at more than 80% of their compute capacity. As long as no CPUs are over-utilized in a root domain, load balancing is disabled and EAS overridess the wake-up balancing code. EAS is likely to load the most energy efficient CPUs of the system more than the others if that can be done without harming throughput. So, the load-balancer is disabled to prevent it from breaking the energy-efficient task placement found by EAS. It is safe to do so when the system isn't overutilized since being below the 80% tipping point implies that:

a. there is some idle time on all CPUs, so the utilization signals used by EAS are likely to accurately represent the 'size' of the various tasks in the system;

b. all tasks should already be provided with enough CPU capacity, regardless of their nice values;

c. since there is spare capacity all tasks must be blocking/sleeping regularly and balancing at wake-up is sufficient.

As soon as one CPU goes above the 80% tipping point, at least one of the three assumptions above becomes incorrect. In this scenario, the 'overutilized' flag is raised for the entire root domain, EAS is disabled, and the load-balancer is re-enabled. By doing so, the scheduler falls back onto load-based algorithms for wake-up and load balance under CPU-bound conditions. This provides a better respect of the nice values of tasks.

Since the notion of overutilization largely relies on detecting whether or not there is some idle time in the system, the CPU capacity 'stolen' by higher (than CFS) scheduling classes (as well as IRQ) must be taken into account. As such, the detection of overutilization accounts for the capacity used not only by CFS tasks, but also by the other scheduling classes and IRQ.

8.6 6. Dependencies and requirements for EAS

Energy Aware Scheduling depends on the CPUs of the system having specific hardware properties and on other features of the kernel being enabled. This section lists these dependencies and provides hints as to how they can be met.

8.6.1 6.1 - Asymmetric CPU topology

As mentioned in the introduction, EAS is only supported on platforms with asymmetric CPU topologies for now. This requirement is checked at run-time by looking for the presence of the SD_ASYM_CPUCAPACITY_FULL flag when the scheduling domains are built.

See *Capacity Aware Scheduling* for requirements to be met for this flag to be set in the sched_domain hierarchy.

Please note that EAS is not fundamentally incompatible with SMP, but no significant savings on SMP platforms have been observed yet. This restriction could be amended in the future if proven otherwise.

8.6.2 6.2 - Energy Model presence

EAS uses the EM of a platform to estimate the impact of scheduling decisions on energy. So, your platform must provide power cost tables to the EM framework in order to make EAS start. To do so, please refer to documentation of the independent EM framework in Documentation/power/energy-model.rst.

Please also note that the scheduling domains need to be re-built after the EM has been registered in order to start EAS.

EAS uses the EM to make a forecasting decision on energy usage and thus it is more focused on the difference when checking possible options for task placement. For EAS it doesn't matter whether the EM power values are expressed in milli-Watts or in an 'abstract scale'.

8.6.3 6.3 - Energy Model complexity

The task wake-up path is very latency-sensitive. When the EM of a platform is too complex (too many CPUs, too many performance domains, too many performance states, ...), the cost of using it in the wake-up path can become prohibitive. The energy-aware wake-up algorithm has a complexity of:

C = Nd * (Nc + Ns)

with: Nd the number of performance domains; Nc the number of CPUs; and Ns the total number of OPPs (ex: for two perf. domains with 4 OPPs each, Ns = 8).

A complexity check is performed at the root domain level, when scheduling domains are built. EAS will not start on a root domain if its C happens to be higher than the completely arbitrary EM_MAX_COMPLEXITY threshold (2048 at the time of writing).

If you really want to use EAS but the complexity of your platform's Energy Model is too high to be used with a single root domain, you're left with only two possible options:

1. split your system into separate, smaller, root domains using exclusive cpusets and enable EAS locally on each of them. This option has the benefit to work out of the box but the drawback of preventing load balance between root domains, which can result in an unbalanced system overall;
2. submit patches to reduce the complexity of the EAS wake-up algorithm, hence enabling it to cope with larger EMs in reasonable time.

8.6.4 6.4 - Schedutil governor

EAS tries to predict at which OPP will the CPUs be running in the close future in order to estimate their energy consumption. To do so, it is assumed that OPPs of CPUs follow their utilization.

Although it is very difficult to provide hard guarantees regarding the accuracy of this assumption in practice (because the hardware might not do what it is told to do, for example), schedutil as opposed to other CPUFreq governors at least _requests_ frequencies calculated using the utilization signals. Consequently, the only sane governor to use together with EAS is schedutil, because it is the only one providing some degree of consistency between frequency requests and energy predictions.

Using EAS with any other governor than schedutil is not supported.

8.6.5 6.5 Scale-invariant utilization signals

In order to make accurate prediction across CPUs and for all performance states, EAS needs frequency-invariant and CPU-invariant PELT signals. These can be obtained using the architecture-defined arch_scale{cpu,freq}_capacity() callbacks.

Using EAS on a platform that doesn't implement these two callbacks is not supported.

8.6.6 6.6 Multithreading (SMT)

EAS in its current form is SMT unaware and is not able to leverage multithreaded hardware to save energy. EAS considers threads as independent CPUs, which can actually be counter-productive for both performance and energy.

EAS on SMT is not supported.

CHAPTER

NINE

SCHEDUTIL

Note: All this assumes a linear relation between frequency and work capacity, we know this is flawed, but it is the best workable approximation.

9.1 PELT (Per Entity Load Tracking)

With PELT we track some metrics across the various scheduler entities, from individual tasks to task-group slices to CPU runqueues. As the basis for this we use an Exponentially Weighted Moving Average (EWMA), each period (1024us) is decayed such that y^32 = 0.5. That is, the most recent 32ms contribute half, while the rest of history contribute the other half.

Specifically:

ewma_sum(u) := u_0 + u_1*y + u_2*y^2 + ...

ewma(u) = ewma_sum(u) / ewma_sum(1)

Since this is essentially a progression of an infinite geometric series, the results are composable, that is ewma(A) + ewma(B) = ewma(A+B). This property is key, since it gives the ability to recompose the averages when tasks move around.

Note that blocked tasks still contribute to the aggregates (task-group slices and CPU runqueues), which reflects their expected contribution when they resume running.

Using this we track 2 key metrics: 'running' and 'runnable'. 'Running' reflects the time an entity spends on the CPU, while 'runnable' reflects the time an entity spends on the runqueue. When there is only a single task these two metrics are the same, but once there is contention for the CPU 'running' will decrease to reflect the fraction of time each task spends on the CPU while 'runnable' will increase to reflect the amount of contention.

For more detail see: kernel/sched/pelt.c

9.2 Frequency / CPU Invariance

Because consuming the CPU for 50% at 1GHz is not the same as consuming the CPU for 50% at 2GHz, nor is running 50% on a LITTLE CPU the same as running 50% on a big CPU, we allow architectures to scale the time delta with two ratios, one Dynamic Voltage and Frequency Scaling (DVFS) ratio and one microarch ratio.

For simple DVFS architectures (where software is in full control) we trivially compute the ratio as:

```
          f_cur
r_dvfs := -----
          f_max
```

For more dynamic systems where the hardware is in control of DVFS we use hardware counters (Intel APERF/MPERF, ARMv8.4-AMU) to provide us this ratio. For Intel specifically, we use:

```
         APERF
f_cur := ----- * P0
         MPERF

           4C-turbo;  if available and turbo enabled
f_max := { 1C-turbo;  if turbo enabled
           P0;        otherwise

                  f_cur
r_dvfs := min( 1, ----- )
                  f_max
```

We pick 4C turbo over 1C turbo to make it slightly more sustainable.

r_cpu is determined as the ratio of highest performance level of the current CPU vs the highest performance level of any other CPU in the system.

    r_tot = r_dvfs * r_cpu

The result is that the above ‘running’ and ‘runnable’ metrics become invariant of DVFS and CPU type. IOW. we can transfer and compare them between CPUs.

For more detail see:

- kernel/sched/pelt.h:update_rq_clock_pelt()
- arch/x86/kernel/smpboot.c:”APERF/MPERF frequency ratio computation.”
- *Capacity Aware Scheduling*:”1. CPU Capacity + 2. Task utilization”

9.3 UTIL_EST / UTIL_EST_FASTUP

Because periodic tasks have their averages decayed while they sleep, even though when running their expected utilization will be the same, they suffer a (DVFS) ramp-up after they are running again.

To alleviate this (a default enabled option) UTIL_EST drives an Infinite Impulse Response (IIR) EWMA with the 'running' value on dequeue – when it is highest. A further default enabled option UTIL_EST_FASTUP modifies the IIR filter to instantly increase and only decay on decrease.

A further runqueue wide sum (of runnable tasks) is maintained of:

util_est := Sum_t max(t_running, t_util_est_ewma)

For more detail see: kernel/sched/fair.c:util_est_dequeue()

9.4 UCLAMP

It is possible to set effective u_min and u_max clamps on each CFS or RT task; the runqueue keeps an max aggregate of these clamps for all running tasks.

For more detail see: include/uapi/linux/sched/types.h

9.5 Schedutil / DVFS

Every time the scheduler load tracking is updated (task wakeup, task migration, time progression) we call out to schedutil to update the hardware DVFS state.

The basis is the CPU runqueue's 'running' metric, which per the above it is the frequency invariant utilization estimate of the CPU. From this we compute a desired frequency like:

```
           max( running, util_est );  if UTIL_EST
u_cfs := { running;                   otherwise

             clamp( u_cfs + u_rt , u_min, u_max );    if UCLAMP_TASK
u_clamp := { u_cfs + u_rt;                            otherwise

u := u_clamp + u_irq + u_dl;          [approx. see source for more detail]

f_des := min( f_max, 1.25 u * f_max )
```

XXX IO-wait: when the update is due to a task wakeup from IO-completion we boost 'u' above.

This frequency is then used to select a P-state/OPP or directly munged into a CPPC style request to the hardware.

XXX: deadline tasks (Sporadic Task Model) allows us to calculate a hard f_min required to satisfy the workload.

Because these callbacks are directly from the scheduler, the DVFS hardware interaction should be 'fast' and non-blocking. Schedutil supports rate-limiting DVFS requests for when hardware interaction is slow and expensive, this reduces effectiveness.

For more information see: kernel/sched/cpufreq_schedutil.c

9.6 NOTES

- On low-load scenarios, where DVFS is most relevant, the ‘running’ numbers will closely reflect utilization.
- In saturated scenarios task movement will cause some transient dips, suppose we have a CPU saturated with 4 tasks, then when we migrate a task to an idle CPU, the old CPU will have a ‘running’ value of 0.75 while the new CPU will gain 0.25. This is inevitable and time progression will correct this. XXX do we still guarantee f_max due to no idle-time?
- Much of the above is about avoiding DVFS dips, and independent DVFS domains having to re-learn / ramp-up when load shifts.

CHAPTER

TEN

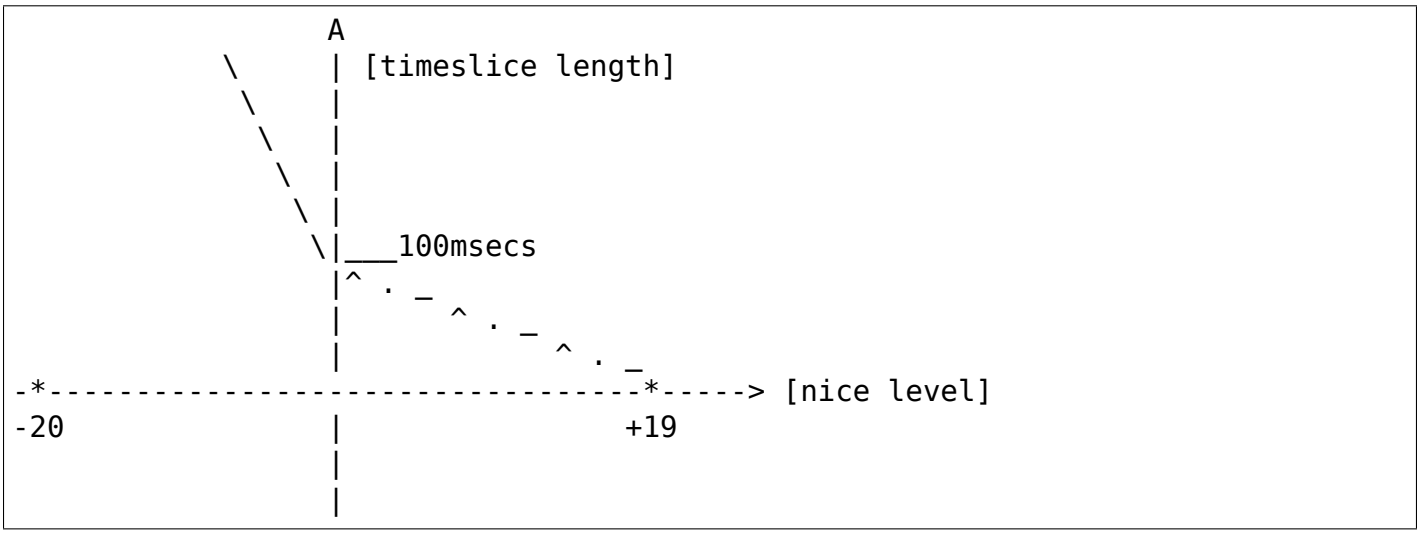
SCHEDULER NICE DESIGN

This document explains the thinking about the revamped and streamlined nice-levels implementation in the new Linux scheduler.

Nice levels were always pretty weak under Linux and people continuously pestered us to make nice +19 tasks use up much less CPU time.

Unfortunately that was not that easy to implement under the old scheduler, (otherwise we'd have done it long ago) because nice level support was historically coupled to timeslice length, and timeslice units were driven by the HZ tick, so the smallest timeslice was 1/HZ.

In the O(1) scheduler (in 2003) we changed negative nice levels to be much stronger than they were before in 2.4 (and people were happy about that change), and we also intentionally calibrated the linear timeslice rule so that nice +19 level would be _exactly_ 1 jiffy. To better understand it, the timeslice graph went like this (cheesy ASCII art alert!):



So that if someone wanted to really renice tasks, +19 would give a much bigger hit than the normal linear rule would do. (The solution of changing the ABI to extend priorities was discarded early on.)

This approach worked to some degree for some time, but later on with HZ=1000 it caused 1 jiffy to be 1 msec, which meant 0.1% CPU usage which we felt to be a bit excessive. Excessive _not_ because it's too small of a CPU utilization, but because it causes too frequent (once per millisec) rescheduling. (and would thus trash the cache, etc. Remember, this was long ago when hardware was weaker and caches were smaller, and people were running number crunching apps at nice +19.)

So for HZ=1000 we changed nice +19 to 5msecs, because that felt like the right minimal granularity - and this translates to 5% CPU utilization. But the fundamental HZ-sensitive property for nice+19 still remained, and we never got a single complaint about nice +19 being too _weak_ in terms of CPU utilization, we only got complaints about it (still) being too _strong_ :-)

To sum it up: we always wanted to make nice levels more consistent, but within the constraints of HZ and jiffies and their nasty design level coupling to timeslices and granularity it was not really viable.

The second (less frequent but still periodically occurring) complaint about Linux's nice level support was its asymmetry around the origin (which you can see demonstrated in the picture above), or more accurately: the fact that nice level behavior depended on the _absolute_ nice level as well, while the nice API itself is fundamentally "relative":

    int nice(int inc);

    asmlinkage long sys_nice(int increment)

(the first one is the glibc API, the second one is the syscall API.) Note that the 'inc' is relative to the current nice level. Tools like bash's "nice" command mirror this relative API.

With the old scheduler, if you for example started a niced task with +1 and another task with +2, the CPU split between the two tasks would depend on the nice level of the parent shell - if it was at nice -10 the CPU split was different than if it was at +5 or +10.

A third complaint against Linux's nice level support was that negative nice levels were not 'punchy enough', so lots of people had to resort to run audio (and other multimedia) apps under RT priorities such as SCHED_FIFO. But this caused other problems: SCHED_FIFO is not starvation proof, and a buggy SCHED_FIFO app can also lock up the system for good.

The new scheduler in v2.6.23 addresses all three types of complaints:

To address the first complaint (of nice levels being not "punchy" enough), the scheduler was decoupled from 'time slice' and HZ concepts (and granularity was made a separate concept from nice levels) and thus it was possible to implement better and more consistent nice +19 support: with the new scheduler nice +19 tasks get a HZ-independent 1.5%, instead of the variable 3%-5%-9% range they got in the old scheduler.

To address the second complaint (of nice levels not being consistent), the new scheduler makes nice(1) have the same CPU utilization effect on tasks, regardless of their absolute nice levels. So on the new scheduler, running a nice +10 and a nice 11 task has the same CPU utilization "split" between them as running a nice -5 and a nice -4 task. (one will get 55% of the CPU, the other 45%.) That is why nice levels were changed to be "multiplicative" (or exponential) - that way it does not matter which nice level you start out from, the 'relative result' will always be the same.

The third complaint (of negative nice levels not being "punchy" enough and forcing audio apps to run under the more dangerous SCHED_FIFO scheduling policy) is addressed by the new scheduler almost automatically: stronger negative nice levels are an automatic side-effect of the recalibrated dynamic range of nice levels.

CHAPTER

ELEVEN

REAL-TIME GROUP SCHEDULING

11.1 0. WARNING

> Fiddling with these settings can result in an unstable system, the knobs are root only and assumes root knows what he is doing.

Most notable:

- very small values in sched_rt_period_us can result in an unstable system when the period is smaller than either the available hrtimer resolution, or the time it takes to handle the budget refresh itself.
- very small values in sched_rt_runtime_us can result in an unstable system when the runtime is so small the system has difficulty making forward progress (NOTE: the migration thread and kstopmachine both are real-time processes).

11.2 1. Overview

11.2.1 1.1 The problem

Realtime scheduling is all about determinism, a group has to be able to rely on the amount of bandwidth (eg. CPU time) being constant. In order to schedule multiple groups of realtime tasks, each group must be assigned a fixed portion of the CPU time available. Without a minimum guarantee a realtime group can obviously fall short. A fuzzy upper limit is of no use since it cannot be relied upon. Which leaves us with just the single fixed portion.

11.2.2 1.2 The solution

CPU time is divided by means of specifying how much time can be spent running in a given period. We allocate this “run time” for each realtime group which the other realtime groups will not be permitted to use.

Any time not allocated to a realtime group will be used to run normal priority tasks (SCHED_OTHER). Any allocated run time not used will also be picked up by SCHED_OTHER.

Let’s consider an example: a frame fixed realtime renderer must deliver 25 frames a second, which yields a period of 0.04s per frame. Now say it will also have to play some music and respond to input, leaving it with around 80% CPU time dedicated for the graphics. We can then give this group a run time of 0.8 * 0.04s = 0.032s.

This way the graphics group will have a 0.04s period with a 0.032s run time limit. Now if the audio thread needs to refill the DMA buffer every 0.005s, but needs only about 3% CPU time to do so, it can do with a 0.03 * 0.005s = 0.00015s. So this group can be scheduled with a period of 0.005s and a run time of 0.00015s.

The remaining CPU time will be used for user input and other tasks. Because realtime tasks have explicitly allocated the CPU time they need to perform their tasks, buffer underruns in the graphics or audio can be eliminated.

NOTE: the above example is not fully implemented yet. We still lack an EDF scheduler to make non-uniform periods usable.

11.3 2. The Interface

11.3.1 2.1 System wide settings

The system wide settings are configured under the /proc virtual file system:

/proc/sys/kernel/sched_rt_period_us: The scheduling period that is equivalent to 100% CPU bandwidth

/proc/sys/kernel/sched_rt_runtime_us: A global limit on how much time realtime scheduling may use. Even without CONFIG_RT_GROUP_SCHED enabled, this will limit time reserved to realtime processes. With CONFIG_RT_GROUP_SCHED it signifies the total bandwidth available to all realtime groups.

- Time is specified in us because the interface is s32. This gives an operating range from 1us to about 35 minutes.
- sched_rt_period_us takes values from 1 to INT_MAX.
- sched_rt_runtime_us takes values from -1 to (INT_MAX - 1).
- A run time of -1 specifies runtime == period, ie. no limit.

11.3.2 2.2 Default behaviour

The default values for sched_rt_period_us (1000000 or 1s) and sched_rt_runtime_us (950000 or 0.95s). This gives 0.05s to be used by SCHED_OTHER (non-RT tasks). These defaults were chosen so that a run-away realtime tasks will not lock up the machine but leave a little time to recover it. By setting runtime to -1 you'd get the old behaviour back.

By default all bandwidth is assigned to the root group and new groups get the period from /proc/sys/kernel/sched_rt_period_us and a run time of 0. If you want to assign bandwidth to another group, reduce the root group's bandwidth and assign some or all of the difference to another group.

Realtime group scheduling means you have to assign a portion of total CPU bandwidth to the group before it will accept realtime tasks. Therefore you will not be able to run realtime tasks as any user other than root until you have done that, even if the user has the rights to run processes with realtime priority!

11.3.3 2.3 Basis for grouping tasks

Enabling CONFIG_RT_GROUP_SCHED lets you explicitly allocate real CPU bandwidth to task groups.

This uses the cgroup virtual file system and "<cgroup>/cpu.rt_runtime_us" to control the CPU time reserved for each control group.

For more information on working with control groups, you should read Documentation/admin-guide/cgroup-v1/cgroups.rst as well.

Group settings are checked against the following limits in order to keep the configuration schedulable:

Sum_{i} runtime_{i} / global_period <= global_runtime / global_period

For now, this can be simplified to just the following (but see Future plans):

Sum_{i} runtime_{i} <= global_runtime

11.4 3. Future plans

There is work in progress to make the scheduling period for each group ("<cgroup>/cpu.rt_period_us") configurable as well.

The constraint on the period is that a subgroup must have a smaller or equal period to its parent. But realistically its not very useful _yet_ as its prone to starvation without deadline scheduling.

Consider two sibling groups A and B; both have 50% bandwidth, but A's period is twice the length of B's.

- group A: period=100000us, runtime=50000us
  - this runs for 0.05s once every 0.1s
- group B: period= 50000us, runtime=25000us
  - this runs for 0.025s twice every 0.1s (or once every 0.05 sec).

This means that currently a while (1) loop in A will run for the full period of B and can starve B's tasks (assuming they are of lower priority) for a whole period.

The next project will be SCHED_EDF (Earliest Deadline First scheduling) to bring full deadline scheduling to the linux kernel. Deadline scheduling the above groups and treating end of the period as a deadline will ensure that they both get their allocated time.

Implementing SCHED_EDF might take a while to complete. Priority Inheritance is the biggest challenge as the current linux PI infrastructure is geared towards the limited static priority levels 0-99. With deadline scheduling you need to do deadline inheritance (since priority is inversely proportional to the deadline delta (deadline - now)).

This means the whole PI machinery will have to be reworked - and that is one of the most complex pieces of code we have.

CHAPTER

TWELVE

SCHEDULER STATISTICS

Version 15 of schedstats dropped counters for some sched_yield: yld_exp_empty, yld_act_empty and yld_both_empty. Otherwise, it is identical to version 14.

Version 14 of schedstats includes support for sched_domains, which hit the mainline kernel in 2.6.20 although it is identical to the stats from version 12 which was in the kernel from 2.6.13-2.6.19 (version 13 never saw a kernel release). Some counters make more sense to be per-runqueue; other to be per-domain. Note that domains (and their associated information) will only be pertinent and available on machines utilizing CONFIG_SMP.

In version 14 of schedstat, there is at least one level of domain statistics for each cpu listed, and there may well be more than one domain. Domains have no particular names in this implementation, but the highest numbered one typically arbitrates balancing across all the cpus on the machine, while domain0 is the most tightly focused domain, sometimes balancing only between pairs of cpus. At this time, there are no architectures which need more than three domain levels. The first field in the domain stats is a bit map indicating which cpus are affected by that domain.

These fields are counters, and only increment. Programs which make use of these will need to start with a baseline observation and then calculate the change in the counters at each subsequent observation. A perl script which does this for many of the fields is available at

http://eaglet.pdxhosts.com/rick/linux/schedstat/

Note that any such script will necessarily be version-specific, as the main reason to change versions is changes in the output format. For those wishing to write their own scripts, the fields are described here.

12.1 CPU statistics

cpu<N> 1 2 3 4 5 6 7 8 9

First field is a sched_yield() statistic:

1) # of times sched_yield() was called

Next three are schedule() statistics:

2) This field is a legacy array expiration count field used in the O(1) scheduler. We kept it for ABI compatibility, but it is always set to zero.

3) # of times schedule() was called

4) # of times schedule() left the processor idle

Next two are try_to_wake_up() statistics:

5) # of times try_to_wake_up() was called
6) # of times try_to_wake_up() was called to wake up the local cpu

Next three are statistics describing scheduling latency:

7) sum of all time spent running by tasks on this processor (in nanoseconds)
8) sum of all time spent waiting to run by tasks on this processor (in nanoseconds)
9) # of timeslices run on this cpu

12.2 Domain statistics

One of these is produced per domain for each cpu described. (Note that if CONFIG_SMP is not defined, *no* domains are utilized and these lines will not appear in the output.)

domain<N> <cpumask> 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

The first field is a bit mask indicating what cpus this domain operates over.

The next 24 are a variety of load_balance() statistics in grouped into types of idleness (idle, busy, and newly idle):

1) # of times in this domain load_balance() was called when the cpu was idle
2) # of times in this domain load_balance() checked but found the load did not require balancing when the cpu was idle
3) # of times in this domain load_balance() tried to move one or more tasks and failed, when the cpu was idle
4) sum of imbalances discovered (if any) with each call to load_balance() in this domain when the cpu was idle
5) # of times in this domain pull_task() was called when the cpu was idle
6) # of times in this domain pull_task() was called even though the target task was cache-hot when idle
7) # of times in this domain load_balance() was called but did not find a busier queue while the cpu was idle
8) # of times in this domain a busier queue was found while the cpu was idle but no busier group was found
9) # of times in this domain load_balance() was called when the cpu was busy
10) # of times in this domain load_balance() checked but found the load did not require balancing when busy
11) # of times in this domain load_balance() tried to move one or more tasks and failed, when the cpu was busy
12) sum of imbalances discovered (if any) with each call to load_balance() in this domain when the cpu was busy
13) # of times in this domain pull_task() was called when busy

14) # of times in this domain pull_task() was called even though the target task was cache-hot when busy

15) # of times in this domain load_balance() was called but did not find a busier queue while the cpu was busy

16) # of times in this domain a busier queue was found while the cpu was busy but no busier group was found

17) # of times in this domain load_balance() was called when the cpu was just becoming idle

18) # of times in this domain load_balance() checked but found the load did not require balancing when the cpu was just becoming idle

19) # of times in this domain load_balance() tried to move one or more tasks and failed, when the cpu was just becoming idle

20) sum of imbalances discovered (if any) with each call to load_balance() in this domain when the cpu was just becoming idle

21) # of times in this domain pull_task() was called when newly idle

22) # of times in this domain pull_task() was called even though the target task was cache-hot when just becoming idle

23) # of times in this domain load_balance() was called but did not find a busier queue while the cpu was just becoming idle

24) # of times in this domain a busier queue was found while the cpu was just becoming idle but no busier group was found

Next three are active_load_balance() statistics:

25) # of times active_load_balance() was called

26) # of times active_load_balance() tried to move a task and failed

27) # of times active_load_balance() successfully moved a task

Next three are sched_balance_exec() statistics:

28) sbe_cnt is not used

29) sbe_balanced is not used

30) sbe_pushed is not used

Next three are sched_balance_fork() statistics:

31) sbf_cnt is not used

32) sbf_balanced is not used

33) sbf_pushed is not used

Next three are try_to_wake_up() statistics:

34) # of times in this domain try_to_wake_up() awoke a task that last ran on a different cpu in this domain

35) # of times in this domain try_to_wake_up() moved a task to the waking cpu because it was cache-cold on its own cpu anyway

36) # of times in this domain try_to_wake_up() started passive balancing

12.3 /proc/<pid>/schedstat

schedstats also adds a new /proc/<pid>/schedstat file to include some of the same information on a per-process level. There are three fields in this file correlating for that process to:

1) time spent on the cpu (in nanoseconds)
2) time spent waiting on a runqueue (in nanoseconds)
3) # of timeslices run on this cpu

A program could be easily written to make use of these extra fields to report on how well a particular process or set of processes is faring under the scheduler's policies. A simple version of such a program is available at

http://eaglet.pdxhosts.com/rick/linux/schedstat/v12/latency.c

CHAPTER

THIRTEEN

SCHEDULER DEBUGFS

Booting a kernel with CONFIG_SCHED_DEBUG=y will give access to scheduler specific debug files under /sys/kernel/debug/sched. Some of those files are described below.

13.1 numa_balancing

numa_balancing directory is used to hold files to control NUMA balancing feature. If the system overhead from the feature is too high then the rate the kernel samples for NUMA hinting faults may be controlled by the *scan_period_min_ms, scan_delay_ms, scan_period_max_ms, scan_size_mb* files.

13.1.1 scan_period_min_ms, scan_delay_ms, scan_period_max_ms, scan_size_mb

Automatic NUMA balancing scans tasks address space and unmaps pages to detect if pages are properly placed or if the data should be migrated to a memory node local to where the task is running. Every "scan delay" the task scans the next "scan size" number of pages in its address space. When the end of the address space is reached the scanner restarts from the beginning.

In combination, the "scan delay" and "scan size" determine the scan rate. When "scan delay" decreases, the scan rate increases. The scan delay and hence the scan rate of every task is adaptive and depends on historical behaviour. If pages are properly placed then the scan delay increases, otherwise the scan delay decreases. The "scan size" is not adaptive but the higher the "scan size", the higher the scan rate.

Higher scan rates incur higher system overhead as page faults must be trapped and potentially data must be migrated. However, the higher the scan rate, the more quickly a tasks memory is migrated to a local node if the workload pattern changes and minimises performance impact due to remote memory accesses. These files control the thresholds for scan delays and the number of pages scanned.

`scan_period_min_ms` is the minimum time in milliseconds to scan a tasks virtual memory. It effectively controls the maximum scanning rate for each task.

`scan_delay_ms` is the starting "scan delay" used for a task when it initially forks.

`scan_period_max_ms` is the maximum time in milliseconds to scan a tasks virtual memory. It effectively controls the minimum scanning rate for each task.

`scan_size_mb` is how many megabytes worth of pages are scanned for a given scan.

CHAPTER

FOURTEEN

SCHEDULER PELT C PROGRAM

```c
/*
 * The following program is used to generate the constants for
 * computing sched averages.
 *
 * ============================================================
 *                C program (compile with -lm)
 * ============================================================
 */

#include <math.h>
#include <stdio.h>

#define HALFLIFE 32
#define SHIFT 32

double y;

void calc_runnable_avg_yN_inv(void)
{
        int i;
        unsigned int x;

        /* To silence -Wunused-but-set-variable warnings. */
        printf("static const u32 runnable_avg_yN_inv[] __maybe_unused = {");
        for (i = 0; i < HALFLIFE; i++) {
                x = ((1UL<<32)-1)*pow(y, i);

                if (i % 6 == 0) printf("\n\t");
                printf("0x%8x, ", x);
        }
        printf("\n};\n\n");
}

int sum = 1024;

void calc_runnable_avg_yN_sum(void)
{
        int i;
```

```
        printf("static const u32 runnable_avg_yN_sum[] = {\n\t    0,");
        for (i = 1; i <= HALFLIFE; i++) {
                if (i == 1)
                        sum *= y;
                else
                        sum = sum*y + 1024*y;

                if (i % 11 == 0)
                        printf("\n\t");

                printf("%5d,", sum);
        }
        printf("\n};\n\n");
}

int n = -1;
/* first period */
long max = 1024;

void calc_converged_max(void)
{
        long last = 0, y_inv = ((1UL<<32)-1)*y;

        for (; ; n++) {
                if (n > -1)
                        max = ((max*y_inv)>>SHIFT) + 1024;
                        /*
                         * This is the same as:
                         * max = max*y + 1024;
                         */

                if (last == max)
                        break;

                last = max;
        }
        n--;
        printf("#define LOAD_AVG_PERIOD %d\n", HALFLIFE);
        printf("#define LOAD_AVG_MAX %ld\n", max);
//        printf("#define LOAD_AVG_MAX_N %d\n\n", n);
}

void calc_accumulated_sum_32(void)
{
        int i, x = sum;

        printf("static const u32 __accumulated_sum_N32[] = {\n\t     0,");
        for (i = 1; i <= n/HALFLIFE+1; i++) {
                if (i > 1)
                        x = x/2 + sum;
```

```
                if (i % 6 == 0)
                        printf("\n\t");

                printf("%6d,", x);
        }
        printf("\n};\n\n");
}

void main(void)
{
        printf("/* Generated by Documentation/scheduler/sched-pelt; do not␣
↪modify. */\n\n");

        y = pow(0.5, 1/(double)HALFLIFE);

        calc_runnable_avg_yN_inv();
//        calc_runnable_avg_yN_sum();
        calc_converged_max();
//        calc_accumulated_sum_32();
}
```