Energy Efficiency in Operating Systems

Björn Brömstrup

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

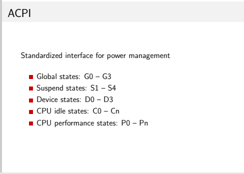
December 3, 2014

Outline

ACPI

Standardized interface for power management

- Global states: G0 – G3
- Suspend states: S1 – S4
- Device states: D0 – D3
- CPU idle states: C0 – Cn
- CPU performance states: P0 – Pn



ACPI

Standardized interface for power management

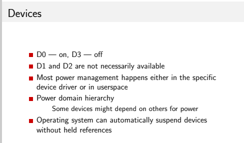
- Global states: G0 – G3
- Suspend states: S1 – S4
- Device states: D0 – D3
- CPU idle states: C0 – Cn
- CPU performance states: P0 – Pn

- As we have seen in the first presentation (Core Energy Efficiency) ACPI has defined multiple states for power management.
- Global states refer to the power state of the entire machine, i.e on, standby, off.
- Suspend states are different standby states, for example suspend-to-ram and suspend-to-disk.
- These are usually invoked from userspace and are therefore not very interesting from a kernel standpoint.
- We are going to talk shortly about the device states and then focus on the CPU.

Devices

- D0 — on, D3 — off
- D1 and D2 are not necessarily available
- Most power management happens either in the specific device driver or in userspace
- Power domain hierarchy

  Some devices might depend on others for power
- Operating system can automatically suspend devices without held references



Devices

- D0 — on, D3 — off
- D1 and D2 are not necessarily available
- Most power management happens either in the specific device driver or in userspace
- Power domain hierarchy
  - Some devices might depend on others for power
- Operating system can automatically suspend devices without held references

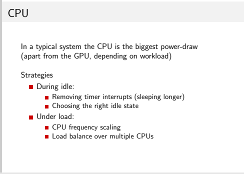
- Device states can't account for the variety in needs of different devices.
- Ultimately, the OS can't do a lot without knowing the specifics of the devices.
- What it can do is keeping track of power dependencies and which devices are actually in use.

CPU

In a typical system the CPU is the biggest power-draw (apart from the GPU, depending on workload)

Strategies

- During idle:
  - Removing timer interrupts (sleeping longer)
  - Choosing the right idle state
- Under load:
  - CPU frequency scaling
  - Load balance over multiple CPUs



CPU

In a typical system the CPU is the biggest power-draw (apart from the GPU, depending on workload)

Strategies

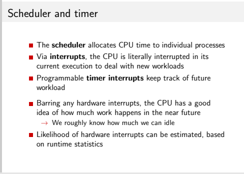
- During idle:
  - Removing timer interrupts (sleeping longer)
  - Choosing the right idle state
- Under load:
  - CPU frequency scaling
  - Load balance over multiple CPUs

- Under load, some CPUs might use more than 50% the power of the entire machine.
- Therefore, we can achieve big gains in power efficiency by managing the CPU correctly.

Scheduler and timer

- The **scheduler** allocates CPU time to individual processes
- Via **interrupts**, the CPU is literally interrupted in its current execution to deal with new workloads
- Programmable **timer interrupts** keep track of future workload

- Barring any hardware interrupts, the CPU has a good idea of how much work happens in the near future
  $\rightarrow$ We roughly know how much we can idle
- Likelihood of hardware interrupts can be estimated, based on runtime statistics



Scheduler and timer

- The **scheduler** allocates CPU time to individual processes
- Via **interrupts**, the CPU is literally interrupted in its current execution to deal with new workloads
- Programmable **timer interrupts** keep track of future workload
- Barring any hardware interrupts, the CPU has a good idea of how much work happens in the near future
  → We roughly know how much we can idle
- Likelihood of hardware interrupts can be estimated, based on runtime statistics

- Before we can understand how CPU power management works, we have to review a few concepts of operating systems.
- The kernel manages computation time on CPUs in slices.
- When multiple processes run on the same CPU, the scheduler will let them run each for a short period of time in sequence.
- The scheduler sets a timer interrupt to regain control after letting a process run.

Ticks and timers

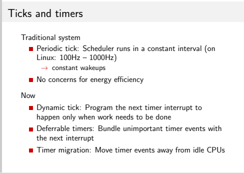
Traditional system

- Periodic tick: Scheduler runs in a constant interval (on Linux: 100Hz – 1000Hz)

  $\rightarrow$ constant wakeups
- No concerns for energy efficiency

Now

- Dynamic tick: Program the next timer interrupt to happen only when work needs to be done
- Deferrable timers: Bundle unimportant timer events with the next interrupt
- Timer migration: Move timer events away from idle CPUs



Ticks and timers

Traditional system

- Periodic tick: Scheduler runs in a constant interval (on Linux: 100Hz – 1000Hz)
  $\rightarrow$ constant wakeups
- No concerns for energy efficiency

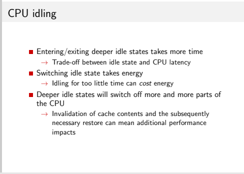
Now

- Dynamic tick: Program the next timer interrupt to happen only when work needs to be done
- Deferrable timers: Bundle unimportant timer events with the next interrupt
- Timer migration: Move timer events away from idle CPUs

- In the past, no concerns were given to energy efficiency and the scheduler was simply run in a constant interval.
- This means that the CPU will be woken up every few milliseconds, even if there is no work to do.
- There are various ways to reduce this overhead.

CPU idling

- Entering/exiting deeper idle states takes more time
  - → Trade-off between idle state and CPU latency
- Switching idle state takes energy
  - → Idling for too little time can *cost* energy
- Deeper idle states will switch off more and more parts of the CPU
  - → Invalidation of cache contents and the subsequently necessary restore can mean additional performance impacts



CPU idling

- Entering/exiting deeper idle states takes more time
  - → Trade-off between idle state and CPU latency
- Switching idle state takes energy
  - → Idling for too little time can *cost* energy
- Deeper idle states will switch off more and more parts of the CPU
  - → Invalidation of cache contents and the subsequently necessary restore can mean additional performance impacts

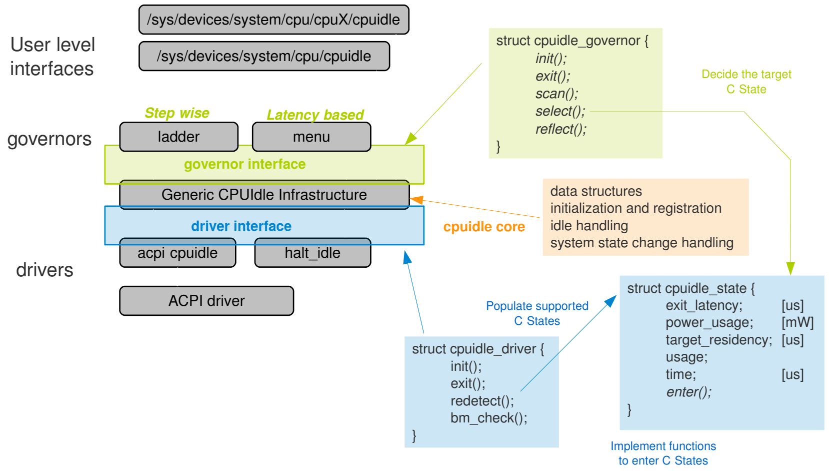
- CPU idling refers to shutting down the power to the CPU when it isn't used, much like any device. It wakes up automatically when an interrupt occurs.
- Idle states are the C-States in ACPI.
- Idling is not as easy as "go to idle when there is nothing to do."
- There are various trade-offs to keep track of.
- On top of that, deciding on an idle state has to be efficient as well.

cpuidle

cpuidle is a Linux kernel subsystem to manage CPU idling

- The decision of which idle state to choose is delegated to one of two governors
  - ladder
  - menu
- Governors rate themselves on how effective they are on a given system and the one with the higher rating is chosen
- Constraints, like latency requirements, are tracked with a Quality of Service (QoS) subsystem

cpuidle



User level interfaces
/sys/devices/system/cpu/cpuX/cpuidle
/sys/devices/system/cpu/cpuidle
governors
Step wise
Latency based
ladder
menu
governor interface
Generic CPUIdle Infrastructure
driver interface
acpi cpuidle
halt_idle
drivers
ACPI driver
cpuidle core
Decide the target C State
Populate supported C States
Implement functions to enter C States

```
struct cpuidle_governor {
        init();
        exit();
        scan();
        select();
        reflect();
}
```

data structures
initialization and registration
idle handling
system state change handling

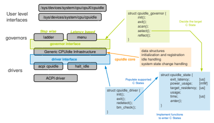
```
struct cpuidle_driver {
        init();
        exit();
        redetect();
        bm_check();
}
```

```
struct cpuidle_state {
        exit_latency;       [us]
        power_usage;        [mW]
        target_residency;   [us]
        usage;
        time;               [us]
        enter();
}
```

Figure: cpuidle in the Linux kernel

source: Patrick Bellasi, Linux Power Management Architecture, http://ilinuxkernel.com/Backup/Data/Linux.Power.Management.Architecture.-.A.review.on.Linux.PM.frameworks.December.2010.pdf

cpuidle


User level interfaces
/sys/devices/system/cpu/cpuX/cpuidle
/sys/devices/system/cpu/cpuidle
governors
Step wise
Latency based
ladder
menu
governor interface
Generic CPUIdle Infrastructure
driver interface
cpuidle core
drivers
acpi cpuidle
halt_idle
ACPI driver
data structures
initialization and registration
idle handling
system state change handling
Decide the target C State
Populate supported C States
Implement functions to enter C States

```
struct cpuidle_governor {
        init();
        exit();
        scan();
        select();
        reflect();
}
```

```
struct cpuidle_driver {
        init();
        exit();
        redetect();
        bm_check();
}
```

```
struct cpuidle_state {
        exit_latency;       [us]
        power_usage;        [mW]
        target_residency;   [us]
        usage;
        time;               [us]
        enter();
}
```

Figure: cpuidle in the Linux kernel

source: Patrick Bellasi, Linux Power Management Architecture, http://ilinuxkernel.com/Backup/Data/Linux.Power.Management.Architecture.-.A.review.on.Linux.PM.frameworks.December.2010.pdf

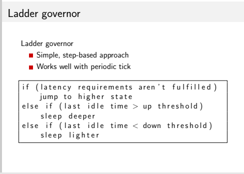
- This is the structure of cpuidle. I thought it would be quite interesting to see how something like this is actually implemented.
- On the right, you can see the data structures that represents a governor and idle states and are used to communicate between the different parts of the kernel.
- When the scheduler decides to idle, it will first call cpuidle (the generic infrastructure in the middle) to decide on the next idle state. cpuidle then passes this request on to the correct governor which will eventually return an idle state to cpuidle and the scheduler.
- Once the scheduler has finished all preparations it will call cpuidle with the idle state, which will pass the request on to the cpu driver, which then finally changes the cpu state.

Ladder governor

Ladder governor

- Simple, step-based approach
- Works well with periodic tick

```
if (latency requirements aren't fulfilled)
    jump to higher state
else if (last idle time > up threshold)
    sleep deeper
else if (last idle time < down threshold)
    sleep lighter
```


Ladder governor

Ladder governor

- Simple, step-based approach
- Works well with periodic tick

```
if (latency requirements aren't fulfilled)
    jump to higher state
else if (last idle time > up threshold)
    sleep deeper
else if (last idle time < down threshold)
    sleep lighter
```

- The ladder government simply steps through the idle states, depending on the last idle time.
- This means, if the load changes rapidly, then the state it chooses will be very often sub-optimal.
- For example, when the last state was the deepest state but the next few sleep times are very short, it would make sense to immediately jump into a light sleep state, but instead the ladder governor will only move slowly towards lighter sleep.
- However, it has the benefit of being very easy to calculate.

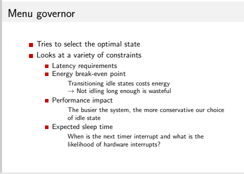
Menu governor

- Tries to select the optimal state
- Looks at a variety of constraints
  - Latency requirements
  - Energy break-even point

    Transitioning idle states costs energy
    $\rightarrow$ Not idling long enough is wasteful
  - Performance impact

    The busier the system, the more conservative our choice of idle state
  - Expected sleep time

    When is the next timer interrupt and what is the likelihood of hardware interrupts?



Menu governor

- Tries to select the optimal state
- Looks at a variety of constraints
  - Latency requirements
  - Energy break-even point

    Transitioning idle states costs energy
    $\rightarrow$ Not idling long enough is wasteful
  - Performance impact

    The busier the system, the more conservative our choice of idle state
  - Expected sleep time

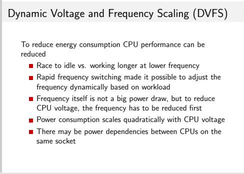
    When is the next timer interrupt and what is the likelihood of hardware interrupts?

- Pretty much all modern systems use the menu governor.

Dynamic Voltage and Frequency Scaling (DVFS)

To reduce energy consumption CPU performance can be reduced

- Race to idle vs. working longer at lower frequency
- Rapid frequency switching made it possible to adjust the frequency dynamically based on workload
- Frequency itself is not a big power draw, but to reduce CPU voltage, the frequency has to be reduced first
- Power consumption scales quadratically with CPU voltage
- There may be power dependencies between CPUs on the same socket



Dynamic Voltage and Frequency Scaling (DVFS)

To reduce energy consumption CPU performance can be reduced

- Race to idle vs. working longer at lower frequency
- Rapid frequency switching made it possible to adjust the frequency dynamically based on workload
- Frequency itself is not a big power draw, but to reduce CPU voltage, the frequency has to be reduced first
- Power consumption scales quadratically with CPU voltage
- There may be power dependencies between CPUs on the same socket

- Frequency scaling refers to reducing the CPU frequency when the CPU is operational, i.e. not idle.
- These are the P-states in ACPI.
- There is an inherent trade-off between P- and C-states. You can either use a lot of power over a short time period to get work done quickly and then idle longer, or use less power over a longer time period and don't idle as long.

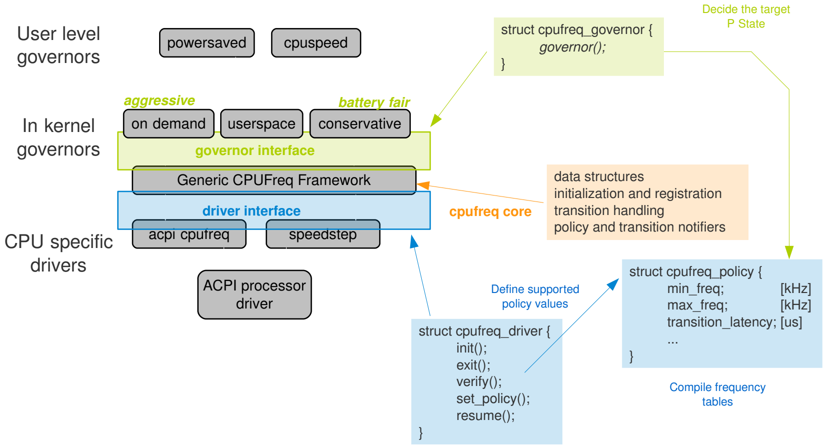
cpufreq

cpufreq is a Linux kernel subsystem to manage CPU frequency states and changes

- A policy is a frequency range in which the CPU needs to stay

  The policy is determined through hardware constraints and explicit setting in userspace
- Governors decide which P-state within the current policy to choose
- The active governor decides by itself when to switch frequency. It is not called by the scheduler

cpufreq



```
struct cpufreq_governor {
        governor();
}
```

data structures
initialization and registration
transition handling
policy and transition notifiers

```
struct cpufreq_policy {
        min_freq;           [kHz]
        max_freq;           [kHz]
        transition_latency; [us]
        ...
}
```

```
struct cpufreq_driver {
        init();
        exit();
        verify();
        set_policy();
        resume();
}
```

Figure: cpufreq in the Linux kernel

source: Patrick Bellasi, Linux Power Management Architecture, http://ilinuxkernel.com/Backup/Data/Linux.Power.Management.Architecture.-.A.review.on.Linux.PM.frameworks.December.2010.pdf

cpufreq

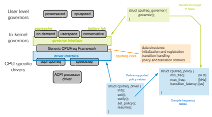

Decide the target P State
User level governors
powersaved
cpuspeed
struct cpufreq_governor { governor(); }
aggressive
battery fair
In kernel governors
on demand
userspace
conservative
governor interface
Generic CPUFreq Framework
cpufreq core
data structures
initialization and registration
transition handling
policy and transition notifiers
driver interface
CPU specific drivers
acpi cpufreq
speedstep
ACPI processor driver
Define supported policy values
struct cpufreq_policy { min_freq; [kHz] max_freq; [kHz] transition_latency; [us] ... }
struct cpufreq_driver { init(); exit(); verify(); set_policy(); resume(); }
Compile frequency tables

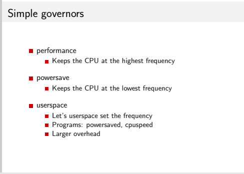
Figure: cpufreq in the Linux kernel

source: Patrick Bellasi, Linux Power Management Architecture, http://ilinuxkernel.com/Backup/Data/Linux.Power.Management.Architecture.-.A.review.on.Linux.PM.frameworks.December.2010.pdf

- This is the structure of cpufreq. It is deliberately designed similar to cpuidle.
- The main difference to cpuidle is that the scheduler doesn't call cpufreq at all. Instead the governors decide themselves when and how to change the frequency.
- When a governor wants to change the frequency, it first requests the current policy and available P-states through the cpufreq framework. When it has made its decision it will tell cpufreq, which will then handle the transition and call the driver. Should the policy change, for example because it was set manually, cpufreq can also request the governor to choose a frequency at any time.
- The user level governors here are programs that run in userspace.

Simple governors

- performance
  - Keeps the CPU at the highest frequency
- powersave
  - Keeps the CPU at the lowest frequency
- userspace
  - Let's userspace set the frequency
  - Programs: powersaved, cpuspeed
  - Larger overhead



Simple governors

- performance
  - Keeps the CPU at the highest frequency
- powersave
  - Keeps the CPU at the lowest frequency
- userspace
  - Let's userspace set the frequency
  - Programs: powersaved, cpuspeed
  - Larger overhead

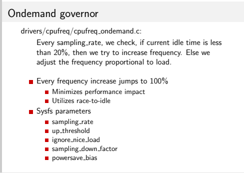
- The simple governors have been around for a long time.
- They typically aren't used anymore.
- Userspace governors were used when frequency switching wasn't as advanced and the larger overhead wasn't a problem.
- Certain intel CPUs use a custom intel driver in place of cpufreq, which exports its governors as "performance" and "powersave" as well. But those governors differ from these.

Ondemand governor

drivers/cpufreq/cpufreq_ondemand.c:

> Every sampling_rate, we check, if current idle time is less than 20%, then we try to increase frequency. Else we adjust the frequency proportional to load.

- Every frequency increase jumps to 100%
  - Minimizes performance impact
  - Utilizes race-to-idle
- Sysfs parameters
  - sampling_rate
  - up_threshold
  - ignore_nice_load
  - sampling_down_factor
  - powersave_bias



Ondemand governor

drivers/cpufreq/cpufreq_ondemand.c:

Every sampling_rate, we check, if current idle time is less than 20%, then we try to increase frequency. Else we adjust the frequency proportional to load.

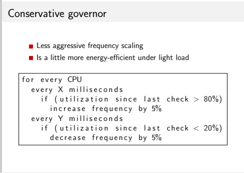
- Every frequency increase jumps to 100%
  - Minimizes performance impact
  - Utilizes race-to-idle
- Sysfs parameters
  - sampling_rate
  - up_threshold
  - ignore_nice_load
  - sampling_down_factor
  - powersave_bias

- The ondemand governor is the default governor for most systems today.
- Finding a good heuristic for frequency scaling is quite difficult. The algorithm in the ondemand governor has been tuned over the years since its implementation.
- That's also why it is changed quite frequently and exports a lot of parameters.

Conservative governor

- Less aggressive frequency scaling
- Is a little more energy-efficient under light load

```
for every CPU
  every X milliseconds
    if (utilization since last check > 80%)
      increase frequency by 5%
  every Y milliseconds
    if (utilization since last check < 20%)
      decrease frequency by 5%
```



Conservative governor

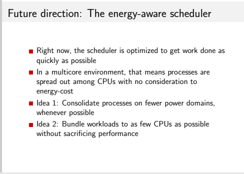
- Less aggressive frequency scaling
- Is a little more energy-efficient under light load

```
for every CPU
  every X milliseconds
    if (utilization since last check > 80%)
      increase frequency by 5%
  every Y milliseconds
    if (utilization since last check < 20%)
      decrease frequency by 5%
```

- The conservative governor scales the frequency step-by-step similar to what the ladder governor does for idle states.
- It's useful for scenarios in which there is a constant, but light, load on the CPUs.
- Smartphones and tablets running Android don't use this governor. They have other governor implementations.

Future direction: The energy-aware scheduler

- Right now, the scheduler is optimized to get work done as quickly as possible
- In a multicore environment, that means processes are spread out among CPUs with no consideration to energy-cost
- Idea 1: Consolidate processes on fewer power domains, whenever possible
- Idea 2: Bundle workloads to as few CPUs as possible without sacrificing performance



Future direction: The energy-aware scheduler

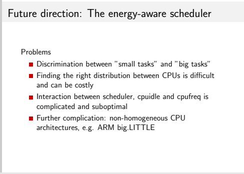
- Right now, the scheduler is optimized to get work done as quickly as possible
- In a multicore environment, that means processes are spread out among CPUs with no consideration to energy-cost
- Idea 1: Consolidate processes on fewer power domains, whenever possible
- Idea 2: Bundle workloads to as few CPUs as possible without sacrificing performance

- If, for example, you have a few light tasks, each using only a few percent of the CPU, then you could move all of them onto a single CPU without a performance impact.

Future direction: The energy-aware scheduler

Problems

- Discrimination between "small tasks" and "big tasks"
- Finding the right distribution between CPUs is difficult and can be costly
- Interaction between scheduler, cpuidle and cpufreq is complicated and suboptimal
- Further complication: non-homogeneous CPU architectures, e.g. ARM big.LITTLE



Future direction: The energy-aware scheduler

Problems

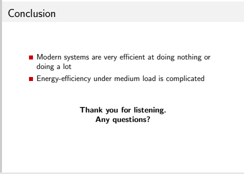
- Discrimination between "small tasks" and "big tasks"
- Finding the right distribution between CPUs is difficult and can be costly
- Interaction between scheduler, cpuidle and cpufreq is complicated and suboptimal
- Further complication: non-homogeneous CPU architectures, e.g. ARM big.LITTLE

- Judging how much a task utilizes a CPU is often hard.
- The task may switch from using a lot of processing power to using only a little, or vice versa, very quickly. Short lived tasks might be impossible to judge.
- Moving tasks from one CPU to another is also expensive.
- Kernel devs are very wary of possible performance impacts and further complication of the scheduler. The implementation of an energy-aware scheduler should be accompanied by a complete redesign of the power management structure. However, big changes to the kernel are difficult and take time.

Conclusion

- Modern systems are very efficient at doing nothing or doing a lot
- Energy-efficiency under medium load is complicated

Thank you for listening.
Any questions?



Conclusion

- Modern systems are very efficient at doing nothing or doing a lot
- Energy-efficiency under medium load is complicated

Thank you for listening.
Any questions?

- Constant loads bring many trade-offs.
- The energy-aware scheduler would be a big improvement.
- There might never be one solution that works well for all situations.

Kernel sources and documentation

- Documentation/cpuidle/*
- Documentation/cpu-freq/*
- Documentation/scheduler/*
- Documentation/timers/*
- drivers/cpufreq/cpufreq*
- drivers/cpuidle/cpuidle*
- include/linux/cpufreq.h
- include/linux/cpuidle.h
- kernel/sched/idle.c

References

- Patrick Bellasi, Linux Power Management Architecture, http://ilinuxkernel.com/Backup/Data/Linux.Power.Management.Architecture.-.A.review.on.Linux.PM.frameworks.December.2010.pdf
- Vaidyanathan Srinivasan, Gautham R Shenoy, Srivatsa Vaddagiri, Dipankar Sarma, Energy-aware task and interrupt management in Linux, https://www.kernel.org/doc/ols/2008/ols2008v2-pages-187-198.pdf
- Venkatesh Pallipadi, Shaohua Li, Adam Belay, cpuidle — Do nothing, efficiently..., https://www.kernel.org/doc/ols/2007/ols2007v2-pages-119-126.pdf
- Venkatesh Pallipadi, Alexey Starikovskiy, The Ondemand Governor, https://www.kernel.org/doc/ols/2006/ols2006v2-pages-223-238.pdf
- Len Brown, Anil Keshavamurthy, David Shaohua Li, Robert Moore, Venkatesh Pallipadi, Luming Yu, ACPI in Linux, https://www.kernel.org/doc/ols/2005/ols2005v1-pages-59-76.pdf
- Rafael J. Wysocki, Runtime Power Management Framework, https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_wysocki2.pdf
- Rafael J. Wysocki, Power Management In The Linux* Kernel, http://events.linuxfoundation.org/sites/events/files/slides/kernel_PM_plain.pdf
- Tate Hornbeck, Peter Hokanson ,Power Management in the Linux Kernel ,www.ruf.rice.edu/ mobile/elec518/lectures/2011-tatepeter.pdf
- corbet, Deferrable timers, http://lwn.net/Articles/228143/
- corbet, Clockevents and dyntick, http://lwn.net/Articles/223185/
- Jonathan Corbet, Per-entity load tracking, http://lwn.net/Articles/531853/
- Jonathan Corbet, Power-aware scheduling meets a line in the sand, http://lwn.net/Articles/552885/
- Libby Clark, Boosting Linux Power Efficiency with Kernel Scheduler Updates, https://www.linux.com/news/featured-blogs/200-libby-clark/715486-boosting-linux-power-efficiency-with-kernel-scheduler-updates/
- Preeti U. Murthy, Overview of the Current Approaches to Enhance the Linux Scheduler, https://events.linuxfoundation.org/images/stories/slides/lfcs2013_murthy.pdf