Linux Power Management Architecture

A review on Linux PM frameworks
Outline the architecture for advanced power management support in recent Linux kernels

*key points for effective PM and overall picture view*

Review of main subsystems

- Advanced timekeeping framework
- Clock framework
- Voltage/Power control Framework
- QoS Framework
- CPUIdle
- CPUFreq
- Linux PM

*Why we need them?*

*What they do?*

*How we can exploit them?*
Modern PM architectures
Key points for effective power management

- Exploit partial activity
disable parts of the system when not needed

- SW does part of the work, HW dependencies do the rest
  exploit existing system framework
  track dependencies (producer-consumer)
  track usage (reference counting)
  system constraints assertion
  notification-chains

  drivers support required
  aggressively get and release resources
  support efficiently OFF mode (0 volts)
  full context loss, low-latency wakeup,
  provide context saving-restore routines
  use constraints to require operational restrictions

Fine grained tracking and constraints
Modern PM architectures

Key points for effective power management

- Control power sources
  - clocks, power domains, voltage domains, memories and power IC resources
  - device drivers should exploit interfaces to control power resources

- Constraints aware drivers and subsystems

- Inactive state
  - power saving in OS idle
    - automatic choice between multiple C-States
    - OFF and retention modes
    - manual suspend/resume
    - system-wide sleep states

- Active state
  - dynamic power management
    - automatic choice between multiple P-States
Modern PM architectures

*what we should (already) have*

- CPUIdle Governor (menu governor)
- CPUFreq Governor (on-demand governor)
- CPUIdle Framework
- CPUFreq Framework
- Constraints Framework
- CPUIdle Driver
- CPUFreq Driver
- Resource Tracking Layer
- User
- Kernel
- LDM
- DSP Bridge
- Drivers
- Constraints Framework
- Clock Framework
- Voltage Framework
- PRCM API Layer 1
- PRCM API Layer 2
- SmartRefllex Driver
- PowerIC Driver

Legend:
- Arch Indep
- Arch Dependant
- OMAP34xx
- Control
- Notifications
- Constraints

Policy management Layer

Device Driver Layer

Resource Tracking Layer

Power Resource Management Layer

PRCM API Layer 1 & 2 (OMAP34xx)
Linux frameworks to support PM

Introduction

- idle-loop, timekeeping (dynamic tick) & clock tree, latency

Dynamic Power Management

- Low-power states activation for unused devices
- C-States, LDM, Suspension (RAM), Hibernation (Disk)

Dynamic Voltage and Frequency Scaling

- Adapting device performances to application needs
- P-States, OPP

- Main focus on x86 arch
  
  - custom and different PM development for SoC embedded devices
  
  - 2nd Linux PM Summit was only on 2007
  
  - increasing emphasis on embedded systems
Linux frameworks to support PM

Timeline

April 2008

2.6.9
Venkatesh Pallipadi

2.6.16
John Stultz

2.6.21
Ingo Molnar

2.6.24
Thomas Gleixner

2.6.25
Venkatesh Pallipadi

2.6.27
Mark Gross

Regulators Framework
CPUFreq OnDemand Governor
Generic Time-of-Day
Gifters base-code
Clock Source and Clock Event
Generic Dynamic Tick
Dynticks
CPUIdle framework
QoS Framework
Open Source Technology Center
IBM
linutronix
OSADL
Founding Member
IBM
Open Source Technology Center
wolfson microelectronics
VM Lab
Prior to 2.6.16

- Lake of a generic *abstraction layer*
- Tick
- Process acc.
- Profiling
- Jiffies
- Timekeeping
- Timer whell

Strongly associated with individual HW devices

**Clock source**

**Clock event device**

- Arch 1
  - TOD: Clock source
  - ISR: Clock event device
- Arch 2
  - TOD: Clock source
  - ISR: Clock event device
- Arch 3
  - TOD: Clock source
  - ISR: Clock event device

Time tracked using *periodic timer ticks*
The Linux Time System
Towards a new implementation

- **Required abstractions**
  - **Clock source (CS) management**
    - provide read access to monotonically increasing time values
    - use human-oriented values (nanoseconds)
  - **Clock synchronization**
    - system time value and clock source drifts correction
    - use reference time sources (e.g. NTP or GPS/GSM)
  - **Time-of-day representation**
    - applying drifts corrections to clock source
  - **Clock event device (CED) management**
    - schedule next event interrupt using clock event device
    - minimize arch dependent code, allow easy run-time addition of new CED
  - **Removing tick dependencies**
    - better support high resolution timers and precise timer scheduling
    - by replace the CTW (Cascading Timer Wheel) mechanism
Why a new Timer Subsystem?

CTW (1997) has unacceptable worst case performances

Insertion is $O(1)$, but cascading require interrupt disabling...

CTW is the better solution for long-term protocol-timeout related timers:
- expiration time within the first category
- removed before they expire or have to be cascaded

is based on periodic interrupt (jiffies)
support only low-resolution time values

Propose solution: use two category for timers

Timeouts – low-resolution, almost always removed
using the existing CTW

Timers – high-resolution, usually expire
using the new hrtimers framework, based on human time units [ns], kept in per-CPU time sorted list implemented using red-black tree ($O(\log N)$)
Since 2.6.16

- Using **human-based time units** [ns]
- Better data structures to support other timer code rework (e.g. high-resolution timers and dynamic-tick)
- Timer queues run from the **normal timer softirq**
- Low-resolution jiffies based timers

**hrtimers**

**Timekeeping**

**Tick**

**Process acc.**

**Profiling**

**Jiffies**

**Timer whell**
Why a Generic Time of Day Implementation?
- allow sharing of clock source code across archs
  - move large portion of code out of arch-specific areas
  - limit arch-dependent code to HW interface and setup
- avoid interpolation based computation of human-time values
  - compensate clock drifts as clock source offsets to get TOD
  - previous implementation track compensated TOD directly and derive increasing human-time values from it (error addition problem)

Proposed solution: a new generic TOD framework
- presented by John Stultz at OLS 2005, merged in 2.6.16
  - cleanup and simplify by arch-independent common code
  - use nanoseconds as fundamental time unit
  - modular design supporting run-time add/remove of time source
  - break tick-based dependency to avoid interpolation issues
The Linux Time System

*The Generic Time-of-Day (GTOD)*

- **Tick**
- **Process acc.**
- **Profiling**
- **Jiffies**
- **Timer wheel**
- **Timekeeping**
- **Shared HW**
- **Clock Source**
- **Clock synch.**
- **hrtimers**

- **Arch-independent and dynamic clock source management**
- **Human-based time units time of day representation**
- **Arch-dependent code limited to direct HW interface and setup procedure**

- **Arch 2**
  - **TOD**
  - **Clock source**
  - **HW**
  - **ISR**
  - **Clock event device**
  - **HW**

- **Arch 3**
  - **TOD**
  - **Clock source**
  - **HW**
  - **ISR**
  - **Clock event device**
  - **HW**
Why a Clock Event Device Abstraction?
provide an abstraction level for timekeeping and time-related activities

- substantial reduction in arch-dependent code
- support either periodic or individual programmable events
- build the base for a generic dynamic tick implementation

Proposed solution: by Thomas Gleixner, merged in 2.6.21

- Registration interface: run-time configuration of clock event device
  property based definition (e.g. capabilities, max/min delta, mult/shift)
  support generic interrupt setup code (i.e. call-back based handlers)
  support for power-management

- Event distribution interface: bind clock events related services to clock event sources
  classical time services (e.g. process accounting, profiling, periodic tick)
  next event interrupt (e.g. high-resolution timers, dynamic tick)
Why high-resolution timers?

- exploit brand-new time subsystems reworks (GTOD, hrtimers)
- *require just arch-independent code by modifying hrtimers framework*
- accurate delivery of high-resolution timers

Proposed solution:

- presented by Thomas Gleixner at OLS 2006
- *switch to high-resolution mode late in the boot process*
- exploiting clock source and clock event device dynamic registration
- ensure kernel compatibility on non high-resolution platforms
- *support next event scheduling and next event interrupt handler*
- softirq based execution of hrtimer queues, independent from tick bound callbacks from next-event handler (with some limitations, e.g. for nanosleep)
- notification to clock event distribution code about periodic interval expiration
Why a Tick-less Kernel?

- processors are quite good about saving power when idle
  - HW has support to reduce leakage on idle states
- avoid processor wakeups when nothing is happening
  - by turning off the period timer tick when in idle state
  - looking at the timer queue to see when the next timer expires
    - CPUIdle can exploit this information
    - in the absence of other events (e.g. hardware interrupts), the system will
      sleep until the nearest timer is due
      - enable the periodic tick once the processor goes out of the idle state

- Proposed solution: merged in 2.6.21
  - room for further development (i.e. full tickless systems)
    - time slice is controlled by the scheduler, variable frequency profiling, and
      a complete removal of jiffies in the future
Nomadik status

support for disabling the timer tick during idle was first merged in 2.6.6 (2004, arch-s390)
we are using an old interface, not the new one provided by clock source devices, but...


[ARM] Remove obsolete and unused ARM dyntick support

`dyntick` is superseded by the `clocksource/clockevent infrastructure`, using the `NO_HZ` configuration option. No one implements `dyntick` on ARM anymore, so it's pointless keeping it around. Remove `dyntick` support.
Why deferrable timers?

better exploit dynamic-tick, by allowing to sleep longer

not all timers has to run as soon as the requested period has expired

non-critical timeouts can run some fraction of a second later

i.e. when the processor wakes up for other reasons

Proposed solution: new function added to the internal kernel API

`init_timer_deferrable(struct timer_list *timer)`

timers defined as deferrable are recognized by the kernel

will not be considered when the kernel makes its "when should the next timer interrupt be?" decision

`timer_list, workqueue,`
Linux Timekeeping Subsystem

Complete support for RT and PM

- Shared HW
- Clock Source
  - Clock synch.
  - TOD
- hrtimers
- Next event
  - Timekeeping
  - Shared HW
- Dynamic tick
  - Clock events
  - ISR
  - Event distribution
  - Process acc.
  - Profiling
  - Jiffies
  - Timer whell

Architectures:
- Arch 1
  - HW
  - HW
  - HW
  - HW
- Arch 2
  - HW
  - HW
  - HW
- Arch 3
  - ISR
  - HW
Define an (abstract) API for all clock related functionalities

dependencies (clock tree), rates (get/set), status (enable/disable)

Since 2.6.16

standard open source solution

Initially added for ARM, now multiple architectures use it

In 2.6.25: ARM (OMAP, PXA, SA1100, AT91, ...), Avr32, PowerPC, ...

recent proposals on LKML for a generic clock API implementation

Used by device drivers

reference counting

re-entrant code

May support layered structure

platform generic layer and board specific low-level
The Clock Framework
centralized control for all clock related functionalities

- Nomadik status (2.6.24)
  Dummy support just for CLCD and UART clocks
  No tracking for Main and System clock, USB, ...
  empty get/set methods, seem that work is in progress...
  Custom Clock Framework for set-top-boxes
Power/Voltage Framework

- track device's power dependencies

- provide a framework analogous to the clock framework but for power/voltage regulators control
  - voltage regulators dependency tracking
  - satisfy client devices needs depending on their state
    - optimize generators usage and efficiency

- no support in mainline, but different patch-set available
  - ARM patch by Nokia (no too much general)
  - Platform independent framework by Wolfson Microelectronics

- Nomadik status (2.6.24)
  - n*k15: limited platform power domains
  - n*k30: much more opportunities
- Announced on LKML in Feb 2008, presented on ELC 2008
  
  Standard and generic kernel interfaces to dynamically control voltage regulators and current sinks

- Better exploit regulators dynamics

  e.g. consumer with 10mA load:
  70% @ Normal =~ 13mA
  90% @ Sleep =~ 11mA

  Saving ~2mA
Some real world examples:

**CPUFreq**: voltage control matching operating frequency

**CPUIdle**: voltage control matching idle state

**LCD back lighting**: control via white LED current reduction

**Audio**: analog supply control, components control

FM-Tuner, Speaker Amplifier when using Headphone

**NAND & NOR**: idle power control

e.g. R/W=35mA, Erase=40mA, Erase+R/W=55mA, Idle=1mA
- Four separate interfaces
The Latency Framework
Explicit system-wide latency-expectation infrastructure

- **Latency is the elapsing time between service request and service delivery**
- **Could influence application behaviors**
  - e.g. audiocodec DMA buffer refill
- **Power saving actions could influence overall system latency**
  - we should consider their impact on overall system latency
- **Need for a system framework that trace instantaneous latency allowed**
The Latency Framework
Explicit system-wide latency-expectation infrastructure

- include/linux/latency.h

The system can tune its operations to the minimum latency requirements in effect at the moment

multiple drivers can announce their maximum accepted latency

drivers know device operating modes at any given moment and the corresponding expected system response time

collect and summarize these expectations globally

cumulated result can be used by power management and similar users
to make decisions that have trade-offs
The Latency Framework
Explicit system-wide latency-expectation infrastructure

- Since 2.6.19, an interface where drivers can:
  - announce the maximum latency [us] that they can deal with
  - modify this latency
  - give up their constraint
  - a function where the code that decides on power saving strategy can query the current global desired maximum
  - a notifier chain allow interested subsystems know when the maximum acceptable latency has changed

Currently used by the generic cpuidle driver on x86 arch

- Nomadik status (2.6.24)
  not exploited: because cpuidle is not correctly used?! could be interesting for some streaming devices
    Audio, Camera, WiFi, Bluetooth, CLCD, ...
Example:
- **user**: idle loop code
  
  *higher C-state saves more power, but has a higher exit latency*
  
  *idle loop code use these informations to make a good decision which C-state to use*
  
- **announcer**: audio driver
  
  *knowns it will get an interrupt when the hardware has 200 usec of samples left in the DMA buffer*
  
  *set a latency constraint of, say, 150 usec*
LatencyTOP
measuring and fixing Linux latency

- tool for software developers (both kernel and userspace), identifying where in the system latency is happening
  what kind of operation/action is causing the latency to happen
  both at system level or on a per process level
- focuses on the cases
  the applications want to run and execute useful code, but there's some resource that's not currently available (and the kernel then blocks the process)
- Kernel patch needed
  keep track of what high level operation it is doing
  limited number of annotations
  output on procfs or using a ncurses based GUI
Here's some output of LatencyTOP, collected for a make -j4 of a kernel on a quad core system

<table>
<thead>
<tr>
<th>Cause</th>
<th>Maximum [msec]</th>
<th>Average [msec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>process fork</td>
<td>1097.7</td>
<td>2.5</td>
</tr>
<tr>
<td>Reading from file</td>
<td>1097.0</td>
<td>0.1</td>
</tr>
<tr>
<td>updating atime</td>
<td>850.4</td>
<td>60.1</td>
</tr>
<tr>
<td>Locking buffer head</td>
<td>433.1</td>
<td>94.3</td>
</tr>
<tr>
<td>Writing to file</td>
<td>381.8</td>
<td>0.6</td>
</tr>
<tr>
<td>Synchronous bufferhead read</td>
<td>318.5</td>
<td>16.3</td>
</tr>
<tr>
<td>Waiting for buffer IO</td>
<td>298.8</td>
<td>7.8</td>
</tr>
</tbody>
</table>

With a small change to the EXT3 journaling layer to fix a priority inversion problem
- `linux/pm_qos_params.h`

Kernel infrastructure to implement a coordination mechanism to facilitate communication among devices (drivers), users (applications) and system (power manager).

- devices have specific **power management capabilities**
  - *devices talk in terms of* **latencies, time outs, throughput**
  - *drivers could better address power management**
  - *expose power management mechanisms*

- applications have specific **performances needs**

- **Since 2.6.25**
  - work-on-progress, used for iwl4965 WiFi driver on x86
- Provide infrastructure for the registration of:
  Dependents: register their QoS needs
e.g.
  Watchers: keep track of the current QoS needs of the system
e.g. cpuidle, wifi drivers, ...

- **3 basic classes of QoS parameter**
  latency [us], timeout [us], throughput [kbs]
  platform customizable constraints set
  Interrupt latency, power domain latency, frequency/OPP
  maintain lists of pm_qos requests and aggregated requirements
  kernel notification tree for each parameter

- **User mode interface for request QoS**
  using simple char device node
The CPUIdle Framework [1]

Do nothing, efficiently...

- Generic processor *idle power management* framework
  
  *support for multiple idle states with different characteristics*
  
  power consumptions, state preservation, state constraints, entry/exit latency

- Support trade-off efficiently between application expectations and idle state power saving

- Clean interface
  
  *abstract between idle-driver and idle-governor*
  
  separate arch-specific idle driver (mechanisms) from arch-independent power management governors (policies)

  *provide a convenient user-space interface*

- Since 2.6.24

  *X86 - ACPI, for OMAP mapping to ACPI states*
The CPUIdle Framework [1]

Do nothing, efficiently...

User-level interfaces

```
/sys/devices/system/cpu/cpuX/cpuidle
```

Governors

**Step-wise**
- ladder

**Latency-based**
- menu

Generic CPUIdle Infrastructure

**Driver interface**

- acpi-cpuidle
- halt_idle

**ACPI driver**

Governor interface

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

Data structures

- initialization and registration
- idle handling
- system state change handling

Cpuidle core

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

Implement functions to enter C-States

Decide the target C-State

Populate supported C-States
### The CPUIdle Framework [1]

**The OMAP43xx implementation**

- The following C states are supported in Cpuidle driver:

<table>
<thead>
<tr>
<th>State Description</th>
<th>Latency (μs)</th>
<th>Residency (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0 – System executing code</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C1 – MPU WFI + Core active</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>C2 – MPU CSWR + Core active</td>
<td>100</td>
<td>300</td>
</tr>
<tr>
<td>C3 – MPU OFF + Core active</td>
<td>3300</td>
<td>4000</td>
</tr>
<tr>
<td>C4 – MPU CSWR + Core CSWR</td>
<td>10000</td>
<td>12000</td>
</tr>
<tr>
<td>C5 – MPU OFF + Core CSWR</td>
<td>11500</td>
<td>15000</td>
</tr>
<tr>
<td>C6 – MPU OFF + Core OFF</td>
<td>20000</td>
<td>300000</td>
</tr>
</tbody>
</table>

- **Menu governor takes the following into account to decide the target sleep state:**
  - Next timer expiry in the system
  - Comparing target residency with available sleep time
  - Comparing exit latency with system wide latency constraints
  - Checking for activity in the core domain

- **Dynamic tick based on support in kernel**
identifying what is causing system wake-ups

*what kind of operation/action prevent long-time permanence in low-power consumption states?*

*support software developers, both kernel-space and user-space*

**main features**

*show how well your system is using various HW power-saving features*

  *both C- and P-States statistics are collected*

*show you the culprit software components that are preventing optimal usage of your HW power savings*

  *userspace should prefer deferrable timers*

*provide you with tuning suggestions to achieve low power consumption*

**support for CPUIdle since v1.10**

*previsou version used the ACPI interface*

*a patch make it possible to collect statistics on both C- and P-States*
PowerTOP
example session running on OMAP3430
The CPUFreq Framework [9]

**CPU active power consumption optimization**

- **Generic processor** active power management framework
  - support for multiple performance states with different characteristics
  - power consumptions

- **Clean interface**
  - abstract between driver and governor
  - separate arch-specific cpu driver (mechanisms) from arch-independent power management governors (policies)
  - provide a convenient user-space interface

- **Since 2.6.9**
  - on-demand governor
  - scaling based on load
  - exploit race-to-idle
  - more recently added support for conservative governor
  - implementing a better battery-fair policy

\[
0.5s \times 24W + 0.5s \times 1W = 12.5J \\
0.25s \times 34W + 0.75s \times 1W = 9.25J
\]
CPUFreq
optimized CPU power consumptions

User-level governors
- powersaved
- cpuspeed

In-kernel governors
- on-demand
- userspace
- conservative
- aggressive
- battery-fair

CPU-specific drivers
- ACPI processor driver
- acpi-cpufreq
- speedstep

Generic CPUFreq Framework

Driver interface

Governor interface

CPUFreq core

Data structures
- initialization and registration
- transition handling
- policy and transition notifiers

Governor interface

- on-demand
- userspace
- conservative
- aggressive
- battery-fair

ACPI processor driver

Define supported policy values

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

Compile frequency tables

User-level governors

struct cpufreq_governor {
  governor();
}

In-kernel governors

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

CPU-specific drivers

Patrick Bellasi bellasi@elet.polimi.it
Every driver should implement suspend/resume calls
register to the LDM (Linux Device Model)
release clocks and save context in suspend call and restore
these when resume is called
drivers which have already released their clocks and have
saved their context need not do anything in their suspend call

Drivers should support OFF modes
all registers in the power domain are reset when the power
domain goes to OFF
OFF mode could introduce considerable latency for wakeup
the system can enter chip off through two paths:
- *Idle loop*
- *Suspend/Resume*
Device drivers responsibilities

- aggressively manage request/release of clocks
  - through clock framework => control their clocks on a request basis
  - not transaction based => use inactivity timer to cut off clocks after a period of inactivity

- need to register with the LDM
  - implement suspend() and resume() calls

- specify constraints when required for its functionalities
  - e.g. min. frequency, max latency, ...

- need to implement context save/restore
  - be aware of power OFF mode

- configure optimal power settings
  - according to standing system constraints
Examples of changes being made

- Transaction based drivers will control clocks based on activity
e.g. i2c driver enables clocks when there are pending requests and
disables them when there are no pending requests

- Camera – Clocks will be enabled as long as the driver is required

- Display – The fbdev inactivity timer (which is tied to user activity) can be used
to turn off display clock

- MMC – Clocks are controlled on a per command basis

- GPTimer – Clocks are controlled as per requirement
  i.e. when a timer is in use, they will be enabled and will be disabled when
  they are not in use

- UART – Console clocks are cut in the idle loop (before putting core domain to
  retention) and other UART clocks could be controlled on a need basis

- USB – Clocks can be controlled as per requirement (only when transfers are
  going on)
When all drivers in a power domain release their clocks, the power domain can go to RET or OFF state. The shared resource framework programs the power domain to target state depending on the latency constraints in the system.

Drivers can follow any of the following methods to save and restore their context:

- **Always save/ always restore**
  
  Drivers which do not have lots of registers can always save context and restore context because it will not cause a lot of overhead.

- **Early save/ restore on demand**
  
  Drivers save context every time they release their clocks but restore it only if the power domain has actually gone to off after saving. This makes sense for drivers which have a large restore time with save time being minimal.
In Nomadik?.... quite everything :-/
References

2. Walfson Microelectronics, Linux Voltage and Current Regulator Framework
3. An API for specifying latency constraints, LWN Article, August 2006.
10. R. Woodruff, Linux system power management on OMAP3430, Embedded Linux Conference, April 2008. Silicon Valley, US.
11. PowerTOP,