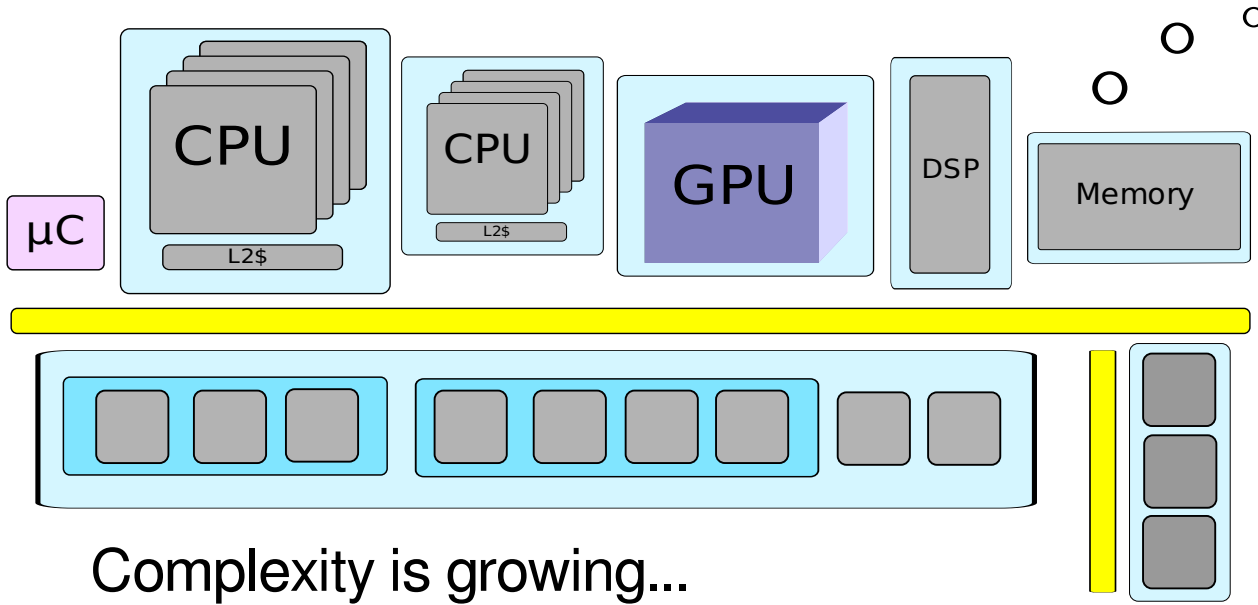


Introduction to Kernel Power Management

Kevin Hilman, Linaro

`khilman@kernel.org`

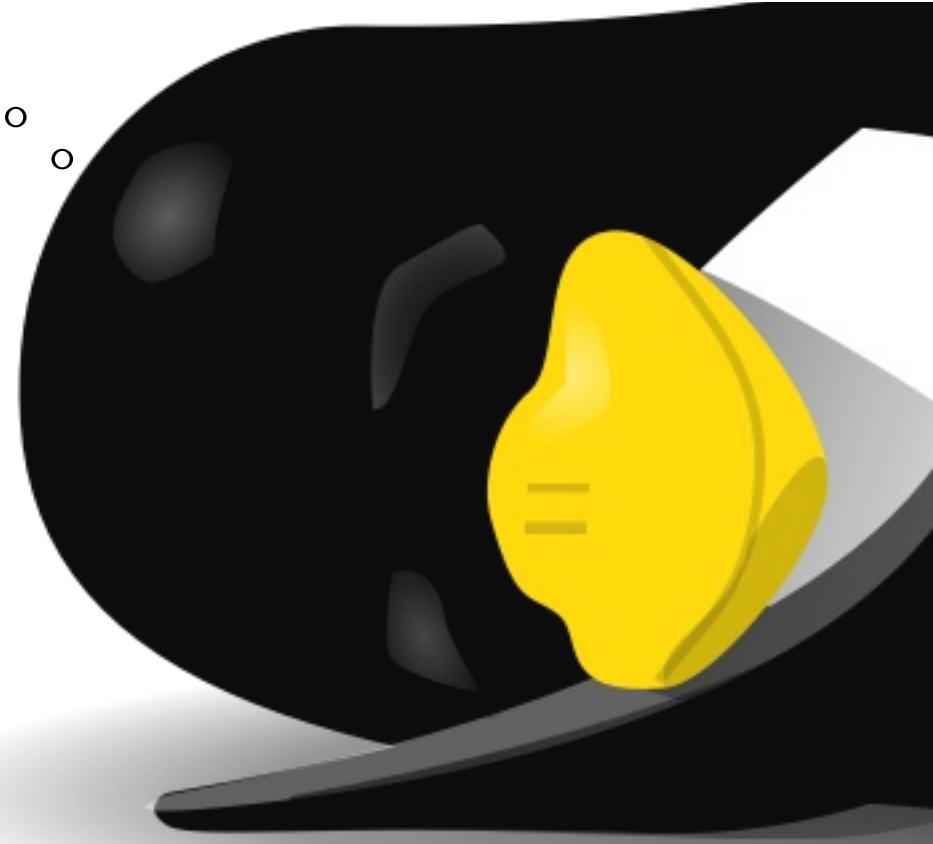
Kernel Recipes 2015, Paris



Complexity is growing...

- more CPUs
- more integrated devices
- more power domains
- micro controllers
- firmware, etc.

Kernel is evolving....



Clocks

Active PM Underlying Frameworks

Frequency scaling: clock framework

```

c1x_get_state()
c1x_set_state()

```

Voltage scaling: regulator framework

```

regulator_get_voltage()
regulator_set_voltage()
Documentation/power/regulator/connexer.txt

```

Example: driver/cpufreq/cpufreq-dt.c

CPUfreq

CPUFreqs using CPUFreq

- Scale "hot" CPU based on requirements
- Platform provides
- Platform provides for scaling "hot" CPU
- Platform provides
- Platform provides to avoid "hot" state to increase reliability - need for latency

Documentation/power/cpufreq-dt.c

Regulators

OPPs

Opening Performance Points (OPPs)

```

static of_freq_attr_t opps_attr[] = {
    { .name = "opp1", .min_uV = 1000000, .max_uV = 1200000, .flags = OPP_FLAG_NO_HOZ },
    { .name = "opp2", .min_uV = 1200000, .max_uV = 1350000, .flags = OPP_FLAG_NO_HOZ },
    { .name = "opp3", .min_uV = 1350000, .max_uV = 1500000, .flags = OPP_FLAG_NO_HOZ },
    { .name = "opp4", .min_uV = 1500000, .max_uV = 1650000, .flags = OPP_FLAG_NO_HOZ },
    { .name = "opp5", .min_uV = 1650000, .max_uV = 1800000, .flags = OPP_FLAG_NO_HOZ },
};

```

c.f. Documentation/power/opp.txt

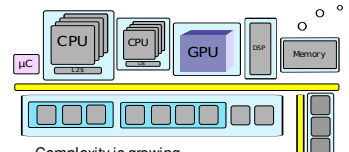
Active

Static

Static PM: System PM

- not used any software
- controlled by hardware
- system-wide, all devices
- not driver-specific
- not driver-specific

Example: arch/arm/mach-dove/pm.c



Complexity is growing...

- more CPUs
- more integrated devices
- more power domains
- micro controllers
- firmware, etc.

Kernel is evolving...

NOHZ_IDLE

kill PM: tickless idle

CONFIG_NOHZ_IDLE

- no periodic ticks while only used for "idle" or "sleep"
- Data not up to system state of tick to stop

CPUs: How deep to sleep?

CPUs: How deep to sleep?

- 1) Break down point (based on earliest time)
- 2) Latency tolerance

Performance impact:

- low "tick" value = more sleep
- low "tick" value = more sleep
- low "tick" value = more sleep

Conclusion:

- use SMP or multi-core aware

CPUidle

kill for CPUs

CPU idle state have "light"

- sleep power usage
- large wakeup latency

State Definition in DT:

- lazy state=cpu-idle

State name:

- platform-specific banks
- based on hardware string

Idle CPUs

Idle devices

Runtime PM callbacks

Use count: 1 -> 0

- device is not used
- device is not used
- device is not used

Use count: 0 -> 1

- device is used
- device is used
- device is used

Added:

- device is not used
- device is not used
- device is not used

Runtime PMAPI

Tell PM core whether device is in use

- "In-use" to use it
- "In-use" to use it
- "In-use" to use it

Example:

```

pm_runtime_get_sync()
pm_runtime_put_sync()

```

Runtime PM

kill for devices: Runtime PM

kill for devices: Runtime PM

- device is not used
- device is not used
- device is not used

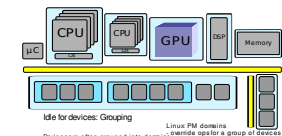
Example:

```

pm_runtime_get_sync()
pm_runtime_put_sync()

```

Idle devices



Idle for devices: Grouping

Idle for devices: Grouping

- device is not used
- device is not used
- device is not used

PM domains

Generic PM Domains (genpd)

Generic implementation of PM domains

- Based on runtime PM
- When all devices in domain are runtime suspended...
- When first device in domain is runtime resumed...

genpd in DT

Example use by device:

```

genpd {
    compatible = "generic-genpd";
    #address-cells = <1>;
    #size-cells = <0>;
    #power-domains = <0>;
    #power-domain-names = <0>;
    #power-domain-ids = <0>;
    #power-domain-ids = <0>;
};

```

PM QoS

Quality: PM QoS

Quality: PM QoS

- device is not used
- device is not used
- device is not used

PM QoS

Driver model: key concept

```

struct dev_pm_ops {
    /* ... */
};

```

```

/* ... */

```

Suspend Resume

Wakeup

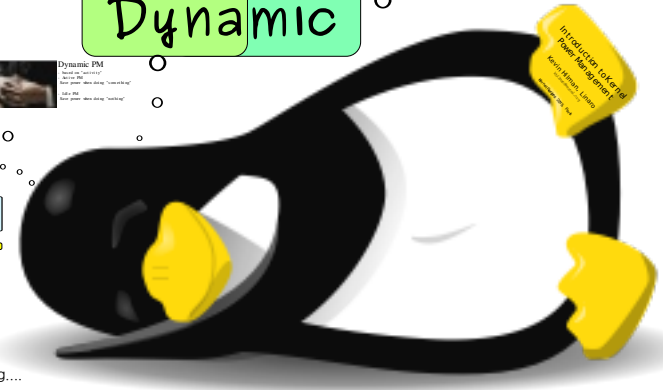
```

/* ... */

```

Dynamic

Idle



Work in Progress

Next steps

Example: power scheduling EAS

- An ongoing effort to improve energy efficiency of the scheduler
- The scheduler will be able to select the task that will be executed next
- The scheduler will be able to select the task that will be executed next
- The scheduler will be able to select the task that will be executed next

Static PM, System PM

- traditional suspend/resume

`CONFIG_PM_SLEEP=y`

- system wide, all devices
- initiated by userspace
- any device can prevent suspend



- userspace is "frozen"

(c.f. `Documentation/power/freezing-of-tasks.txt`)

MUST Read: `Documentation/power/devices.txt`

Driver model: key concept

struct dev_pm_ops

Exists in struct device_driver, struct bus_type, ...

```
struct dev_pm_ops {
    int (*prepare)(struct device *dev);
    void (*complete)(struct device *dev);
    int (*suspend)(struct device *dev);
    int (*resume)(struct device *dev);
    ...
    int (*suspend_late)(struct device *dev);
    int (*resume_early)(struct device *dev);
    ...
};
```

echo mem > /sys/power/state

Platform specific:

```
struct platform_suspend_ops
```

```
->begin()
```

```
->prepare()
```

```
->enter()
```

```
->wake()
```

```
->finish()
```

```
->end()
```

Per-device:

```
struct dev_pm_ops
```

```
->prepare()
```

```
->suspend()
```

```
->suspend_late()
```

```
->suspend_noirq()
```

```
->resume_noirq()
```

```
->resume_early()
```

```
->resume()
```

```
->complete()
```



Wakeup from Suspend

Subsystem / Driver control:

- `device_init_wakeup(dev, bool)`
- `dev_pm_set_wake_irq()`
- `dev_pm_clear_wake_irq()`

When wakeup occurs (e.g. in ISR):

- `pm_wakeup_event()`

Enable / disable from user space:

- `/sys/devices/.../power/wakeup`



Clocks

Regulators

Active PM Underlying Frameworks

```

Frequency scaling: clock framework
- clk_get_name()
- clk_get()

Voltage scaling: regulator framework
- regulator_get_voltage()
- regulator_enable()
- Documentation/power/regulator/consumer.txt

Example: driver/cpufreq/cpufreq-dt.c

```

CPUfreq

CPUFreqs using CPUFreq

```

- Scales CPU to meet requirements
- platform provider for selecting best CPU
  performance scenario
- implement algorithms to find out how to
  maintain balanced + load for heavy
  workloads
- Documentation/power/cpufreq-dt.c

```

OPP

Opening Performance Points (OPPs)

opp0	opp1
100000 100000	200000 100000
300000 100000	400000 100000
600000 100000	800000 100000
1000000 100000	1500000 100000
2000000 100000	3000000 100000
4000000 100000	8000000 100000
10000000 100000	20000000 100000

c.f. Documentation/power/opp.txt

Driver model: key concept

```

struct dev_pm_ops
Exec in error device_driver struct bus_type ...

struct dev_pm_ops {
int (*prepare)(struct device *dev);
void (*suspend)(struct device *dev);
int (*suspend_noirq)(struct device *dev);
int (*freeze)(struct device *dev);
int (*thaw)(struct device *dev);
int (*resume)(struct device *dev);
int (*resume_noirq)(struct device *dev);
int (*restore)(struct device *dev);
};

```

Suspend Resume

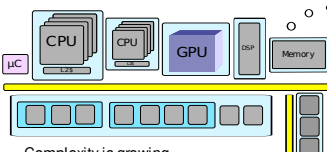
Static

Static PM: Suspend PM

```

- not used any software
- CONFIG_PM_SLEEP
- system-wide all devices
- controlled by hardware
- no driver only power sequencer
- See: Documentation/power/sleep.txt

```



Wakeup

Subsystem / Driver code

```

- dev_pm_ops.wakeup(dev, b001)
- dev_pm_ops.wake_irq(dev, b001)
- dev_pm_ops.wake_irq(dev, b001)

Wakeup from Suspend
- log in IRQ
- pm_wakeup_irq()
- pm_wakeup_irq()
- pm_wakeup_irq()

```

Complexity is growing...

- more CPUs
- more integrated devices
- more power domains
- micro controllers
- firmware, etc.

Kernel is evolving...

NOHZ_IDLE

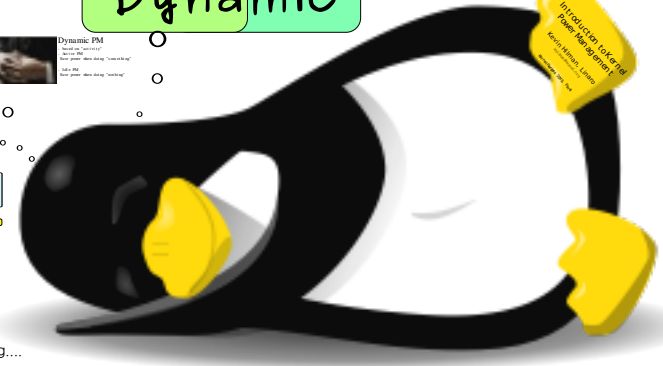
CPUIDle

Idle CPUs

Idle devices

Idle

Dynamic



kill PM: tickle idle

```

CONFIG_NOHZ_IDLE
- no periodic timer tick
- tick only for use "tick"
- or timer

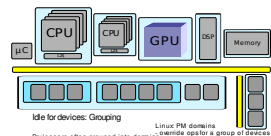
Data wakeup
- tickle idle tick tick

```

Runtime PM callbacks

Runtime PMAPI

Idle devices Runtime PM



PM domains

Runtime PM

PM QoS

Work in Progress

Next steps

Enterprise-wide scheduling EAS

- In response effort to those users efficiency of the scheduler
- EAS is a scheduler that can be used to schedule tasks in a
- task scheduler to improve the performance of the system
- EAS is a scheduler that can be used to schedule tasks in a
- task scheduler to improve the performance of the system
- EAS is a scheduler that can be used to schedule tasks in a
- task scheduler to improve the performance of the system



Dynamic PM

- based on "activity"

- Active PM

Save power when doing "something"

- Idle PM

Save power when doing "nothing"

Clocks

Regulators

Active PM: Underlying Frameworks

Frequency scaling: clock framework

- clk_get_rate()
- clk_set_rate()

Voltage scaling: regulator framework

- regulator_get_voltage()
- regulator_set_voltage()
- Documentation/power/regulator/consumer.txt

Example... drivers/cpufreq/cpufreq-dt.c

CPUFreq

CPU DVFS using CPUFreq

- Select "best" OPP based on requirements
- pluggable governors for selecting "best" OPP
- performance, powersave, ...
- ondemand: heuristics based on load, tunable
- interactive: ondemand++, tuned for latency

Documentation/cpu-freq/core.txt

... what about device DVFS? devfreq

OPPs

Operating Performance Points OPPs

- tuple of frequency, minimum voltage
 - Described in DT
- ```

cpu0: cpu0 {
 operating-points = <
/* kHz uV */
 300000 1025000
 600000 1200000
 800000 1313000
 1008000 1375000
 >;
}

```

c.f. Documentation/power/opp.txt

# Active

# Operating Performance Points OPPs

- tuple of frequency, minimum voltage
- Described in DT

```
cpu0: cpu@0 {
operating-points = <
/* kHz uV */
300000 1025000
600000 1200000
800000 1313000
1008000 1375000
>;
}
```

c.f. `Documentation/power/opp.txt`

# CPU DVFS using CPUFreq

- Select "best" OPP based on requirements
- pluggable governors for selecting "best" OPP
- performance, powersave, ...
- ondemand: heursitics based on load, tunable
- interactive: ondemand++, tuned for latency

`Documentation/cpu-freq/core.txt`

... what about device DVFS? .... devfreq

# Active PM: Underlying Frameworks

## Frequency scaling: clock framework

- `clk_get_rate()`
- `clk_set_rate()`

## Voltage scaling: regulator framework

- `regulator_get_voltage()`
- `regulator_set_voltage()`
- `Documentation/power/regulator/consumer.txt`

**Example...** `drivers/cpufreq/cpufreq-dt.c`



### Idle PM: tickless idle

```
CONFIG_NOHZ_IDLE=y
```

- stop periodic tick when idle
- only wakes for next "event" or interrupt



Don't wake up... only to press snooze and go back to sleep

# NOHZ\_IDLE

### CPUidle: How deep to sleep?

- 1) Break even point (based on enter/exit times)
  - looks at predictable events (e.g. timers)
  - compares against min residency

- 2) Latency tolerance
  - checks QoS (PM\_QOS\_CPU\_DMA\_LATENCY)
  - compares against min residency
- 3) Performance impact
  - black magic "tunables" based on load
  - favor shallower states under heavy load



Documentation/cpuidle/\*.txt

- Limitations:
- not very SMP or multi-cluster aware

# CPUidle

### Idle for CPUs

- CPU idle states have "depth"
- more power savings
  - longer wakeup latency

- State Definitions in DT
- legacy: platform-specific driver

- State entry
- platform-specific hooks
  - based on compatible string

```
enum {
 CPUIDLE_STATE_0 = 0,
 CPUIDLE_STATE_1 = 1,
 CPUIDLE_STATE_2 = 2,
 CPUIDLE_STATE_3 = 3,
 CPUIDLE_STATE_4 = 4,
 CPUIDLE_STATE_5 = 5,
 CPUIDLE_STATE_6 = 6,
 CPUIDLE_STATE_7 = 7,
 CPUIDLE_STATE_8 = 8,
 CPUIDLE_STATE_9 = 9,
 CPUIDLE_STATE_10 = 10,
 CPUIDLE_STATE_11 = 11,
 CPUIDLE_STATE_12 = 12,
 CPUIDLE_STATE_13 = 13,
 CPUIDLE_STATE_14 = 14,
 CPUIDLE_STATE_15 = 15,
 CPUIDLE_STATE_16 = 16,
 CPUIDLE_STATE_17 = 17,
 CPUIDLE_STATE_18 = 18,
 CPUIDLE_STATE_19 = 19,
 CPUIDLE_STATE_20 = 20,
 CPUIDLE_STATE_21 = 21,
 CPUIDLE_STATE_22 = 22,
 CPUIDLE_STATE_23 = 23,
 CPUIDLE_STATE_24 = 24,
 CPUIDLE_STATE_25 = 25,
 CPUIDLE_STATE_26 = 26,
 CPUIDLE_STATE_27 = 27,
 CPUIDLE_STATE_28 = 28,
 CPUIDLE_STATE_29 = 29,
 CPUIDLE_STATE_30 = 30,
 CPUIDLE_STATE_31 = 31,
};
```

# Idle CPUs

### Runtime PM: callbacks

Use count 1 --> 0

- > runtime\_pm\_suspend()
- prepare for low power state
- ensure wakeups enabled
- save context

Use count 0 --> 1

- > runtime\_pm\_wakeup()
- restore context
- etc.



Autosuspend — deferred runtime suspend

```
pm_runtime_mark_last_busy()
pm_runtime_put_autosuspend()
```

### Runtime PM API

- Tell PM core whether device is in use
- "I'm about to use it!"
  - pm\_runtime\_get()
  - increment use count, pm\_runtime\_resume()

- "I'm done... for now"
- pm\_runtime\_put()
  - Decrement use count, pm\_runtime\_idle()

- Similar to legacy clock framework usage for clock gating
- clk\_enable(), clk\_disable()

Excellent! Documentation/power/pm\_runtime.txt

# Runtime PM

### Idle for devices: Runtime PM

- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

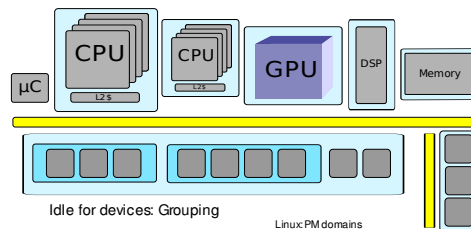
- devices are *independent*
- one device cannot prevent others from runtime suspending

- does NOT affect user space



```
struct dev_pm_ops {
 ...
 int (*runtime_suspend)(struct device *dev);
 int (*runtime_resume)(struct device *dev);
 int (*runtime_idle)(struct device *dev);
};
```

# Idle devices



### Idle for devices: Grouping

- Devices are often grouped into domains
- power gated as a group
- can be nested
- power gating has latency implications
- external regulator ramp up, etc.

- Linux PM domains
- override ops for a group of devices
  - if PM domain present, PM core uses domain callbacks instead of type/classbus

```
struct dev_pm_domain {
 struct dev_pm_ops ops;
 ...
};
```

Documentation/power/devices.txt

### Generic PM Domains (genpd)

#### Generic implementation of PM domains

- Based on runtime PM
- When all devices in domain are runtime suspended...
  - genpd->power\_off()
- When first device in domain is runtime resumed...
  - genpd->power\_on()

#### genpd governors

- allow custom decision making before power gating
- e.g. per-device QoS constraints

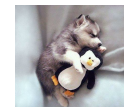
#### genpd in DT

##### Example genpd:

```
power: power-controller@12340000 {
 compatible = "foo,power-controller";
 reg = <0x12340000 0x1000>;
 #power-domain-cells = <1>;
};
```

##### Example use by device:

```
leaky=device@12350000 {
 compatible = "foo,leak-current";
 reg = <0x12350000 0x1000>;
 power-domains = <power 0>;
};
```



From: Documentation/devicetree/bindings/power/power-domain.txt

# PM domains

### Quality: PM QoS

- System-wide: e.g. PM\_QOS\_CPU\_DMA\_LATENCY
- Used by CPUidle to determine depth of idle state

#### Per-device

- attach QoS constraints with specific devices
- genpd: prevent PM domain power off
  - PM\_QOS\_PREVENT\_POWER\_OFF
- e.g. genpd: per-device wakeup latency
  - PM\_QOS\_RESUME\_LATENCY
- for use by genpd "governors"

Documentation/power/pm\_qos\_interface.txt

# PM QoS

# Idle



# Idle for CPUs

## CPU idle states have "depth"

- more power savings
- longer wakeup latency

## State Definitions in DT

- legacy: platform-specific driver

## State entry

- platform-specific hooks
- based on `compatible` string

```
idle-states {
 CPU_STBY: standby {
 compatible = "qcom,idle-state-stby",
 "arm,idle-state";
 entry-latency-us = <1>;
 exit-latency-us = <1>;
 min-residency-us = <2>;
 };
};
```

```
CPU_SPC: spc {
 compatible = "qcom,idle-state-spc",
 "arm,idle-state";
 entry-latency-us = <150>;
 exit-latency-us = <200>;
 min-residency-us = <2000>;
};
};
```

```
cpu@0 {
 [...]
 cpu-idle-states = <&CPU_STBY &CPU_SPC>;
};
```

Documentation/devicetree/bindings/arm/idle-states.txt

# CPUidle: How deep to sleep?

`drivers/cpuidle/governors/menu.c`

## 1) Break even point (based on enter/exit times)

- looks at predictable events (e.g. timers)
- compares against min residency

## 2) Latency tolerance

- checks QoS (`PM_QOS_CPU_DMA_LATENCY`)
- compares against min residency

## 3) Performance Impact:

- black magic "multiplier" based on load
- favor shallower states under heavy load

## Limitations:

- not very SMP or multi-cluster aware



`Documentation/cpuidle/*.txt`

# Idle PM: tickless idle

`CONFIG_NOHZ_IDLE=y`

- stop periodic tick when idle
- only wakes for next "event" or interrupt



Don't wake up...

only to press snooze and go back to sleep

### Idle PM: tickless idle

```
CONFIG_NOHZ_IDLE=y
```

- stop periodic tick when idle
- only wakes for next "event" or interrupt



Don't wake up... only to press snooze and go back to sleep

# NOHZ\_IDLE

### Runtime PM: callbacks

Use count 1 --> 0

- > runtime\_pm\_suspend()
- prepare for low power state
- ensure wakeups enabled
- save context

Use count 0 --> 1

- > runtime\_pm\_wakeup()
- restore context
- etc.



Autosuspend - deferred runtime suspend

```

pruntime_pm_runtime_suspend(struct device *dev)
{
 pruntime_pm_mark_last_busy(dev);
 pruntime_pm_get_autosuspend();
}

```

### Runtime PM API

Tell PM core whether device is in use

- "I'm about to use it"
- > runtime\_pm\_get(), \_sync()
- increment use count, pm\_runtime\_resume()

"I'm done... for now"

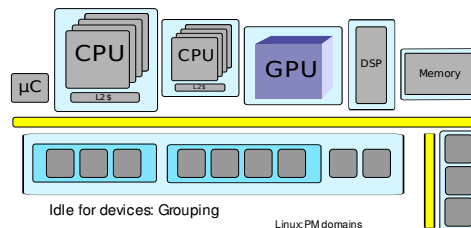
- > runtime\_pm\_put(), \_sync()
- decrement use count, pm\_runtime\_idle()

Similar to legacy clock framework usage for clock gating

- clk\_enable(), clk\_disable()

Excellent! Documentation/power/pm\_runtime.txt

# Runtime PM



### Idle for devices: Grouping

- Devices are often grouped into domains
- power gated as a group
- can be nested
- power gating has latency implications
- external regulator ramp up, etc.

### Linux PM domains

- override ops for a group of devices
- if PM domain present, PM core uses domain callbacks instead of type/classbus

```

struct dev_pm_domain {
 struct dev_pm_ops ops;
 ...
};

```

Documentation/power/devices.txt

### Generic PM Domains (genpd)

Generic implementation of PM domains

- Based on runtime PM
- When all devices in domain are runtime suspended... genpd->power\_off()
- When first device in domain is runtime resumed... genpd->power\_on()

genpd governors

- allow custom decision making before power gating
- e.g. per-device QoS constraints

genpd in DT

Example genpd:

```

power: power-controller@12340000 {
 compatible = "foo,power-controller";
 reg = <0x12340000 0x1000>;
 #power-domain-cells = <1>;
};

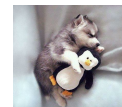
```

Example use by device:

```

leaky-device@12350000 {
 compatible = "foo,leaky-current";
 reg = <0x12350000 0x1000>;
 power-domains = <power 0>;
};

```



From: Documentation/devicetree/bindings/power/power-domain.txt

### CPUidle: How deep to sleep?

drivers/cpuidle/governors/menu.c

1) Break even point (based on enter/exit times)

- looks at predictable events (e.g. timers)
- compares against min residency

2) Latency tolerance

- checks QoS (PM\_QOS\_CPU\_DMA\_LATENCY)
- compares against min residency

3) Performance impact

- black magic "trickles" based on load
- favor shallower states under heavy load



Documentation/cpuidle/\*.txt

Limitations:

- not very SMP or multi-cluster aware

# CPUidle

### Idle for CPUs

CPU idle states have "depth"

- more power savings
- longer wakeup latency

State Definitions in DT

- legacy: platform-specific driver

State entry

- platform-specific hooks
- based on compatible string

```

idle-states {
 #idle-states = 1;
 compatible = "arm,cortex-a9";
 state-no-wakeup {
 compatible = "arm,cortex-a9";
 state-no-wakeup {
 compatible = "arm,cortex-a9";
 state-no-wakeup {
 compatible = "arm,cortex-a9";
 }
 }
 }
};

```

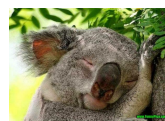
# Idle CPUs

### Idle for devices: Runtime PM

- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

- devices are *independent*
- one device cannot prevent others from runtime suspending

- does NOT affect user space



```

struct dev_pm_ops {
 ...
 int (*runtime_suspend)(struct device *dev);
 int (*runtime_resume)(struct device *dev);
 int (*runtime_idle)(struct device *dev);
};

```

# Idle devices

# PM domains

### Quality: PM QoS

System-wide: e.g. PM\_QOS\_CPU\_DMA\_LATENCY

- Used by CPUidle to determine depth of idle state

Per-device

- attach QoS constraints with specific devices
- genpd: prevent PM domain power off
- > PM\_QOS\_CPU\_DMA\_LATENCY
- e.g. genpd: per-device wakeup latency
- > PM\_QOS\_RESUME\_LATENCY
- for use by genpd "governors"

Documentation/power/pm\_qos\_interface.txt

# PM QoS

# Idle

# Idle for devices: Runtime PM

- per-device idle
- single device at a time
- idleness controlled by driver, based on activity
  
- devices are *independent*
- one device cannot prevent others from runtime suspending
  
- does **NOT** affect user space



```
struct dev_pm_ops {
 ...
 int (*runtime_suspend)(struct device *dev);
 int (*runtime_resume)(struct device *dev);
 int (*runtime_idle)(struct device *dev);
};
```

# Runtime PM API

Tell PM core whether device is in use

"I'm about to use it"

- `pm_runtime_get()`, `_sync()`
- Increment use count, `pm_runtime_resume()`

"I'm done... for now"

- `pm_runtime_put()`, `_sync()`
- Decrement use count, `pm_runtime_idle()`

Similar to legacy clock framework usage for clock gating

- `clk_enable()`, `clk_disable()`

**Excellent:** `Documentation/power/pm_runtime.txt`



# Runtime PM: callbacks

## Use count: 1 --> 0

- `->runtime_suspend()`
- prepare for low-power state
- ensure wakeups enabled
- save context

## Use count: 0 --> 1

- `->runtime_resume()`
- restore context
- etc.



## Autosuspend --- deferred runtime suspend

- `pm_runtime_set_autosuspend_delay()`
- `pm_runtime_mark_last_busy()`
- `pm_runtime_put_autosuspend()`



### Idle PM: tickless idle

```
CONFIG_NOHZ_IDLE=y
```

- stop periodic tick when idle
- only wakes for next "event" or interrupt



Don't wake up... only to press snooze and go back to sleep

# NOHZ\_IDLE

### CPUidle: How deep to sleep?

- 1) Break even point (based on enter/exit times)
  - looks at predictable events (e.g. timers)
  - compares against min residency

- 2) Latency tolerance
  - checks QoS (PM\_QOS\_CPU\_DMA\_LATENCY)
  - compares against min residency
- 3) Performance impact
  - black magic "tunables" based on load
  - favor shallower states under heavy load



Documentation/cpuidle/\*.txt

- Limitations:
- not very SMP or multi-cluster aware

# CPUidle

### Idle for CPUs

- CPU idle states have "depth"
- more power savings
  - longer wakeup latency

- State Definitions in DT
- legacy: platform-specific driver

- State entry
- platform-specific hooks
  - based on compatible string

```
enum {
 CPU_IDLE_STATES = 1,
 CPU_IDLE_STATE_0 = "idle",
 CPU_IDLE_STATE_1 = "idle-noirq",
 CPU_IDLE_STATE_2 = "idle-deep",
 CPU_IDLE_STATE_3 = "idle-deepest",
};

enum {
 CPU_IDLE_STATE_0 = "idle",
 CPU_IDLE_STATE_1 = "idle-noirq",
 CPU_IDLE_STATE_2 = "idle-deep",
 CPU_IDLE_STATE_3 = "idle-deepest",
};

enum {
 CPU_IDLE_STATE_0 = "idle",
 CPU_IDLE_STATE_1 = "idle-noirq",
 CPU_IDLE_STATE_2 = "idle-deep",
 CPU_IDLE_STATE_3 = "idle-deepest",
};
```

# Idle CPUs

### Runtime PM: callbacks

Use count 1 --> 0

- > runtime\_pm\_suspend()
- prepare for low power state
- ensure wakeups enabled
- save context

Use count 0 --> 1

- > runtime\_pm\_wakeup()
- restore context
- etc.



Autosuspend — deferred runtime suspend

```
pm_runtime_mark_last_busy()
pm_runtime_put_autosuspend()
```

### Runtime PM API

- Tell PM core whether device is in use
- "I'm about to use it!"
  - pm\_runtime\_get()
  - increment use count, pm\_runtime\_resume()

- "I'm done... for now"
- pm\_runtime\_put()
  - Decrement use count, pm\_runtime\_idle()

- Similar to legacy clock framework usage for clock gating
- clk\_enable(), clk\_disable()

Excellent! Documentation/power/pm\_runtime.txt

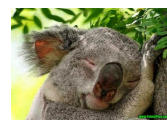
# Runtime PM

### Idle for devices: Runtime PM

- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

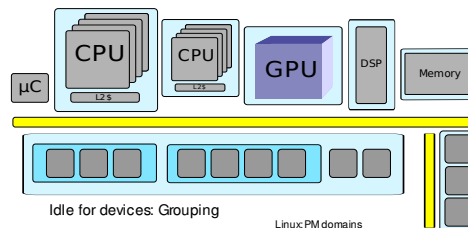
- devices are *independent*
- one device cannot prevent others from runtime suspending

- does NOT affect user space



```
struct dev_pm_ops {
 ...
 int (*runtime_suspend)(struct device *dev);
 int (*runtime_resume)(struct device *dev);
 int (*runtime_idle)(struct device *dev);
};
```

# Idle devices



### Idle for devices: Grouping

- Devices are often grouped into domains
- power gated as a group
- can be nested
- power gating has latency implications
- external regulator ramp up, etc.

### Linux PM domains

- override ops for a group of devices
- if PM domain present, PM core uses domain callbacks instead of type/classbus

```
struct dev_pm_domain {
 struct dev_pm_ops ops;
 ...
};
```

Documentation/power/devices.txt

### Generic PM Domains (genpd)

#### Generic implementation of PM domains

- Based on runtime PM
- When all devices in domain are runtime suspended...
  - genpd->power\_off()
- When first device in domain is runtime resumed...
  - genpd->power\_on()

#### genpd governors

- allow custom decision making before power gating
- e.g. per-device QoS constraints

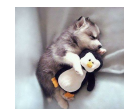
#### genpd in DT

##### Example genpd:

```
power: power-controller@12340000 {
 compatible = "foo,power-controller";
 reg = <0x12340000 0x1000>;
 #power-domain-cells = <1>;
};
```

##### Example use by device:

```
leaky-device@12350000 {
 compatible = "foo,leaky-current";
 reg = <0x12350000 0x1000>;
 power-domains = <&power_0>;
};
```



From: Documentation/devicetree/bindings/power/power-domain.txt

# PM domains

### Quality: PM QoS

- System-wide: e.g. PM\_QOS\_CPU\_DMA\_LATENCY
- Used by CPUidle to determine depth of idle state

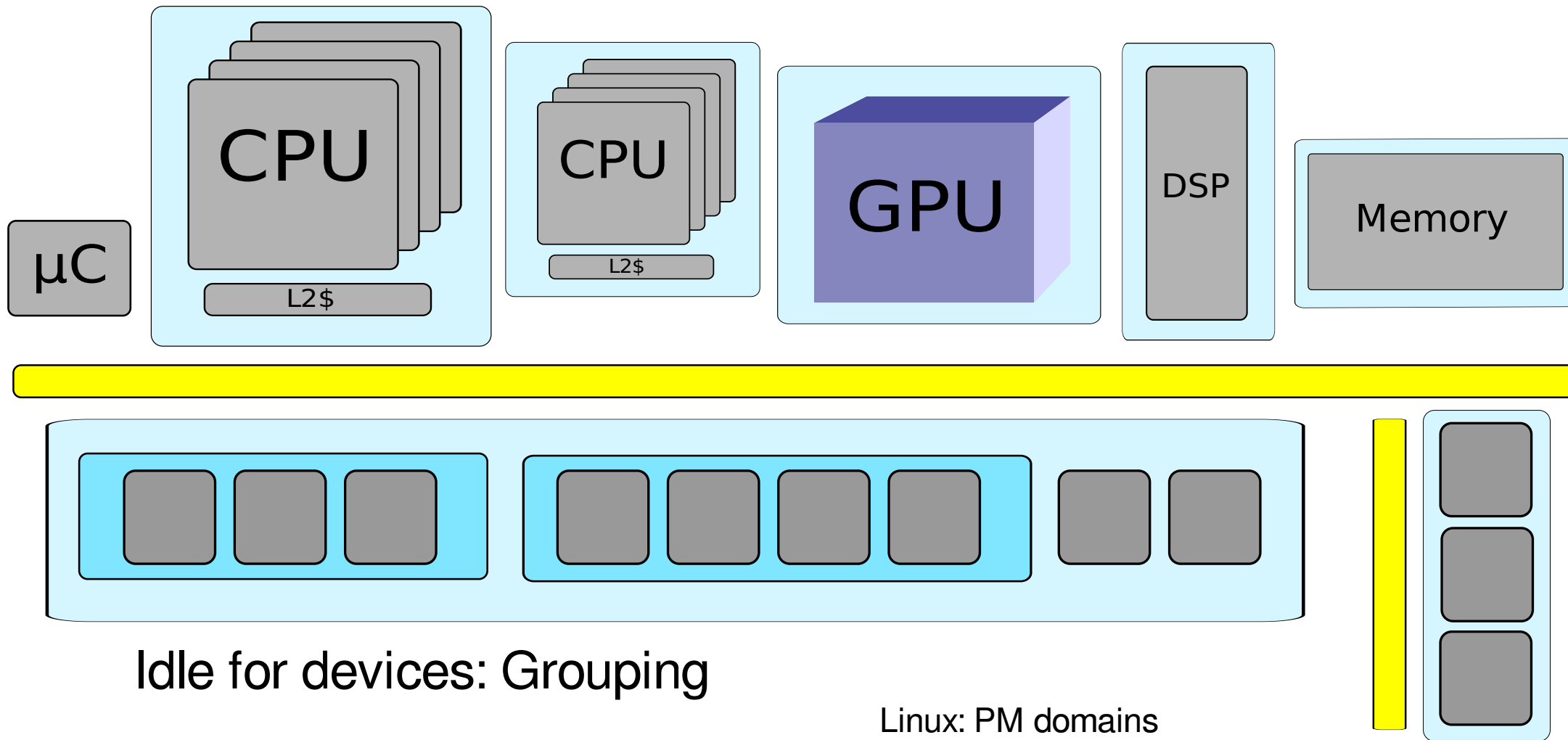
#### Per-device

- attach QoS constraints with specific devices
- genpd: prevent PM domain power off
  - PM\_QOS\_PREVENT\_POWER\_OFF
- e.g. genpd: per-device wakeup latency
  - PM\_QOS\_PREWAKE\_LATENCY
- for use by genpd "governors"

Documentation/power/pm\_qos\_interface.txt

# PM QoS

# Idle



## Idle for devices: Grouping

Devices are often grouped into domains

- power gated as a group
- can be nested
- power gating has latency implications
- external regulator ramp up, etc.

Documentation/power/devices.txt

## Linux: PM domains

- override ops for a group of devices
- if PM domain present, PM core uses domain callbacks instead of type/class/bus

```
struct dev_pm_domain {
 struct dev_pm_ops ops;
 ...
};
```

# Generic PM Domains (genpd)

## Generic implementation of PM domains

- Based on runtime PM
- When all devices in domain are runtime suspended...

```
genpd->power_off()
```

- When first device in domain is runtime resumed...

```
genpd->power_on()
```

## genpd governors

- allow custom decision making before power gating
- e.g. per-device QoS constraints

# genpd in DT

## Example genpd:

```
power: power-controller@12340000 {
 compatible = "foo,power-controller";
 reg = <0x12340000 0x1000>;
 #power-domain-cells = <1>;
};
```

## Example use by device:

```
leaky-device@12350000 {
 compatible = "foo,i-leak-current";
 reg = <0x12350000 0x1000>;
 power-domains = <&power 0>;
};
```



**From:** `Documentation/devicetree/bindings/power/power-domain.txt`

# Quality: PM QoS

**System-wide: e.g** `PM_QOS_CPU_DMA_LATENCY`

- Used by CPUidle to determine depth of idle state

## Per-device

- attach QoS constraints with specific devices
- genpd: prevent PM domain power off
  - `PM_QOS_FLAG_NO_POWER_OFF`
- e.g. genpd: per-device wakeup latency
  - `DEV_PM_QOS_RESUME_LATENCY`
  - for use by genpd "governors"

`Documentation/power/pm_qos_interface.txt`

**Clocks**

**Regulators**

Active PM Underlying Frameworks

Frequency scaling: clock framework

```

 <+> clk_get_val()
 <+> clk_get_val()

 Voltage scaling: regulator framework
 <+> regulator_get_voltage()
 <+> regulator_set_voltage()
 <+> Documentation/power/regulator/consumer.txt

```

Example: driver/opfreq/opufreq-dt.c

**CPUfreq**

CPUFreqs using CPUFreq

- Allow user to tune frequency
- Provide governor for selecting best OPP
- Performance governor
- Implement governors to suit on hand, suitable to various situations, used for heavy

```

 ... while (dev_drvst) {
 ...
 }

```

**Opening Performance Points**

```

OPP#s

 type of frequency, minimum voltage
 opp0: opp0s [
 oppscaling-opp0s = 1
 /t_opp0: 1
 500000 1020000
 600000 1200000
 650000 1300000
 700000 1375000
]
 .. c.f. Documentation/power/opp.txt

```

**Active**

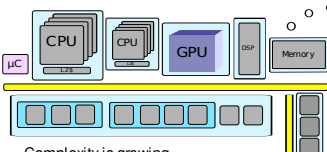
**Static**

Static PM: Suspend PM

- not used any software
- controlled by hardware
- system-wide, all devices
- set driver only power support



**Dynamic**



Complexity is growing...  
 - more CPUs  
 - more integrated devices  
 - more power domains  
 - micro controllers  
 - firmware, etc.

Kernel is evolving...

**NOHZ\_IDLE**

kill PM: tickless idle

```

CONFIG_NOHZ_IDLE
- no periodic timer ticks
- only ticks for use "tick" or "idle"
- tickless idle
- tickless idle
- tickless idle

```

**CPUIDle**

CPUIDle: How deep to sleep?

- 1) Break even point (based on estimated time)
- 2) Latency tolerance

Performance impact:

- low latency
- high energy efficiency

**CPUIidle**

kill for CPUs

- CPU idle time have "light"
- low power usage
- large wakeup latency

State Definition in DT:

- low: states=cpu-idle

**Idle CPUs**

**Idle devices**

**Runtime PM callbacks**

```

Use count: 1 -> 0
- use count: 1 -> 0
- use count: 0 -> 1
- use count: 0 -> 1

```

**Runtime PMAPI**

Tell PM core whether device is in use

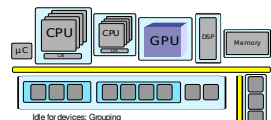
- "On/Off to use"
- "I'm done - firmware"
- "I'm done - hardware"
- "I'm done - firmware"

**Runtime PM**

**Idle devices Runtime PM**

idle devices Runtime PM

- idle devices Runtime PM
- idle devices Runtime PM
- idle devices Runtime PM



Generic PM Domains (genpd)

- Based on runtime PM
- When all device in domain are runtime suspended...
- When first device in domain is runtime resumed...

**PM domains**

genpd in DT

```

Example genpd:
- allow customization making below power gating
- e.g. per-device QoS constraints

```

**PM QoS**

**Quality: PM QoS**

Quality: PM QoS

- Quality: PM QoS
- Quality: PM QoS

**Driver model: key concept**

```

struct dev_pm_ops
{
 // ...
}

```

```

echo mem > /sys/power/state

Platform specific:
struct platform_suspend_ops
{
 <+> suspend()
 <+> resume()
 }

```

```

Wakeups from Suspend
Subsystem / Driver used:
- dev_pm_ops.wakeup
- dev_pm_ops.wakeup

```

**Suspend Resume**

**Wakeups**

**Work in Progress**

**Next steps**

Enterprise-wide scheduling EAS

- No response after 100ms to reduce efficiency of the scheduler
- No response after 100ms to reduce efficiency of the scheduler
- No response after 100ms to reduce efficiency of the scheduler

## Runtime PM, genpd ideas

### Unify idle for CPUs and devices

- use runtime PM for CPUs
- use runtime PM for CPU-connected "extras" (e.g. GIC, PMUs, VFP, CoreSight, etc.)
- combine into a "CPU genpd"

### Extend to CPU Clusters

- model clusters as genpd made up of "CPU genpd"s plus shared L2\$
- when CPUs in cluster are idle (runtime suspended) cluster genpd can hit low-power state (off)

Work in  
Progress

Next Steps

## Energy-aware scheduling: EAS

An on-going effort to improve energy efficiency of the scheduler.

- Teach the scheduler new heuristics for task placement to take advantage of energy-saving hardware
- Integrate CPUidle and CPUfreq with the scheduler
  - scheduler tracks load statistics for its own decision making (e.g. task placement, load balancing, etc.)
- CPUidle / CPUfreq governor decisions are based on their own load-based calculations, heuristics (and some black magic).

These are to be replaced by scheduler-driven data



# Runtime PM, genpd ideas

## Unify idle for CPUs and devices

- use runtime PM for CPUs
- use runtime PM for CPU-connected "extras" (e.g. GIC, PMUs, VFP, CoreSight, etc.)
- combine into a "CPU genpd"

## Extend to CPU Clusters

- model clusters as genpd made up of "CPU genpd"s plus shared L2\$
- when CPUs in cluster are idle (runtime suspended) cluster genpd can hit low-power state (off)

# Energy-aware scheduling: EAS

An on-going effort to improve energy efficiency of the scheduler.

- Teach the scheduler new heuristics for task placement to take advantage of energy-saving hardware
- Integrate CPUidle and CPUfreq with the scheduler
  - scheduler tracks load statistics for its own decision making (e.g. task placement, load balancing, etc.)
  - CPUidle / CPUfreq governor decisions are based on their own load-based calculations, heuristics (and some black magic).

These are to be replaced by scheduler-driven data

# Clocks

# Regulators

Active PM Underlying Frameworks

```
Frequency scaling: clock framework
+ clock_get_rate()
+ clock_set_rate()

Voltage scaling: regulator framework
+ regulator_get_voltage()
+ regulator_set_voltage()
+ Documentation/power/regulator/consumer.txt
```

```
Example: driver/cpufreq/cpufreq-dt.c
```

# CPUfreq

CPUfreqs using CPUfreq

- Start that CPU based on requirements
- Check for governor for selecting best CPU profile
- implement algorithm to find out load, switch to nearest related + load for heavy
- Documentation/proc/sys/kernel/cpu/governors
- ... what does CPUfreq do? ... datag

# OPPs

Opening Performance Points OPPs

```
typedef struct opp {
 struct device_node *np;
 struct opp_params opp;
 struct opp_table *table;
};

/* ...
 * 000000 1000000
 * 000000 1000000
 * 000000 1375000
 * ...
 */
```

c.f. Documentation/power/opp.txt

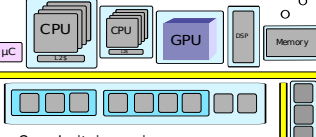
**Driver model: key concept**

```
struct dev_pm_ops {
 /* ...
 suspend_late()
 suspend()
 suspend_late()
 suspend()
 resume_early()
 resume()
 resume_late()
 resume()
 */
};
```

# Static

Static PM: Suspend PM

- Not used any software
- Configured in hardware
- system-wide, all devices
- controlled by hardware
- no driver code to manage suspend/resume



Complexity is growing...

- more CPUs
- more integrated devices
- more power domains
- micro controllers
- firmware, etc.

Kernel is evolving...

# NOHZ\_IDLE

kill PM: tickless idle

CPUIdle: How deep to sleep?

- 1) Break down point (based on earliest timing)
- 2) Latency tolerance
- Performance Impact:
  - less time in deep sleep
  - more time in shallow sleep
  - more time in deep sleep

# CPUIDle

kill for CPUs

- CPU idle state have "light"
- sleep power usage
- large wakeup latency
- State Definition in DT
- latency of state-to-idle driver
- State name
- platform-specific hooks
- handle the hardware timing

# Idle CPUs

# Idle devices

# Runtime PM

kill for devices: Runtime PM

- per-device state
- single driver is in control of state
- allow the driver to control the device
- don't use for hardware
- no of device state program when driver runtime suspend/resume

# Idle devices

Runtime PM APIs

- Tell PM core whether device is in use
- "I'm awake to use it"
- "I'm done - firmware"
- "I'm done - hardware"
- "I'm done - driver"
- "I'm done - user"

Runtime PM QoS

- system-wide
- per-device
- per-driver
- per-app
- per-user

# PM domains

Generic PM Domains (genpd)

Generic implementation of PM domains

- Based on runtime PM
- When all device in domain are runtime suspended... genpd is in NOHZ\_IDLE
- When first device in domain is runtime resumed... genpd is in DT

Example use by device:

```
struct genpd_ops {
 /* ...
 suspend()
 resume()
 */
};
```

# Suspend Resume

# Wakeups

**Subsystem: Driver model**

```
dev_pm_ops {
 /* ...
 suspend_late()
 suspend()
 suspend_late()
 suspend()
 resume_early()
 resume()
 resume_late()
 resume()
 */
};
```

# Dynamic

Dynamic PM: Suspend PM

- Not used any software
- Configured in hardware
- system-wide, all devices
- controlled by hardware
- no driver code to manage suspend/resume

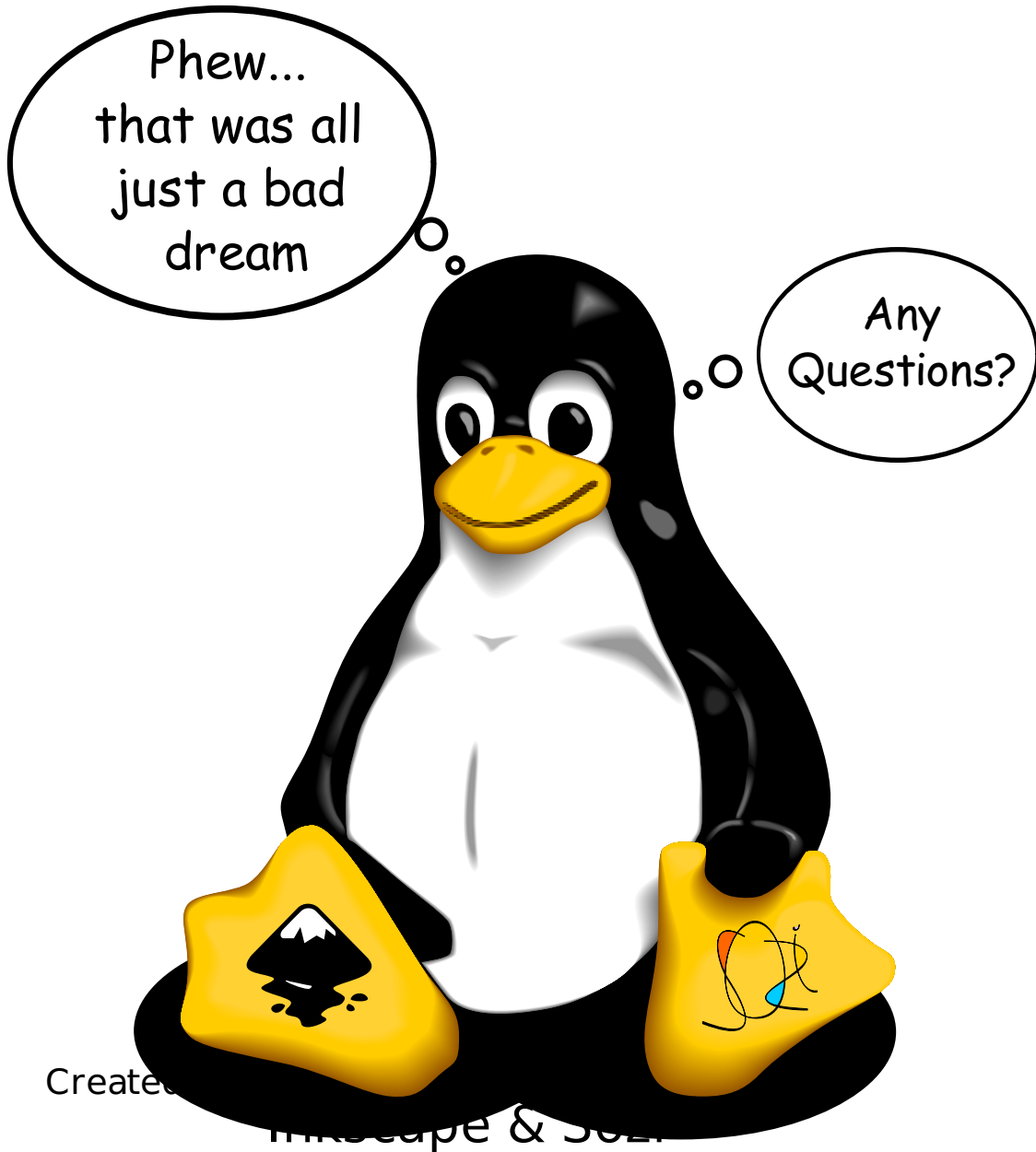
# Idle

# Work in Progress

# Next steps

Enterprise-wide scheduling EAS

- In response effort to these trends efficiency of the scheduler
- Linux 4.14 introduces a new scheduler: Energy Aware Scheduler (EAS)
- EAS is designed to work with the scheduler
- EAS is designed to work with the scheduler
- EAS is designed to work with the scheduler



Created with  
inkscape & so...

Slides under CC-BY-SA 3.0



<http://people.linaro.org/~kevin.hilman/conf/kr2015/>