POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA

Energy and power management in the computing continuum
Prof. William Fornaciari

Power Management at the OS Level

AA 2023 – 2024

Prof. William Fornaciari

<william.fornaciari@polimi.it>

HEAP LAB

Power Management at OS level

Outline

- Basics on power consumption
- Power saving blocks and knobs
- **Power states and ACPI**
- **Operating systems integration (the Linux case)**
- Thermal issues
- Miscellanea
- Conclusions

Main Linux PM features

- Platform-specific code
  - Idle-loop, timekeeping (dynamic tick) & clock tree, latency
- Resource Hibernation Frameworks
  - Switch off unused subsystems
  - Low-power states, C-states, suspension (to RAM) and hibernation (to disk)
- Resource Tuning Frameworks
  - Adapt performances to needs
  - P-states and OPPs
- Main focus on x86 architecture
  - Custom and different PM development for SoC-based embedded systems
  - Increasing emphasys on embedded systems

What is an “Operating Point (OP)” ?

- General concept influenced by
  - The latency/response time
  - The power consumption
  - The availability of a proper hw support
  - The link between hw platform and power management software
  - The usage scenario (embedded, general purpose, HPC)
  - Complexity of the policies that are implementable
  - ....
- A first standardization initiative for Intel, 20-30 years ago
- Embedded world is still very *handcrafted*
- Exotic management policies are appearing, remember that simple is better!

- Does it make sense to manage V and F independently?
- OP → pair of optimal {V,f}

Power states

Active and idle states

- Power saving policies can be applied both when the system is in active or idle state
  - *Active*

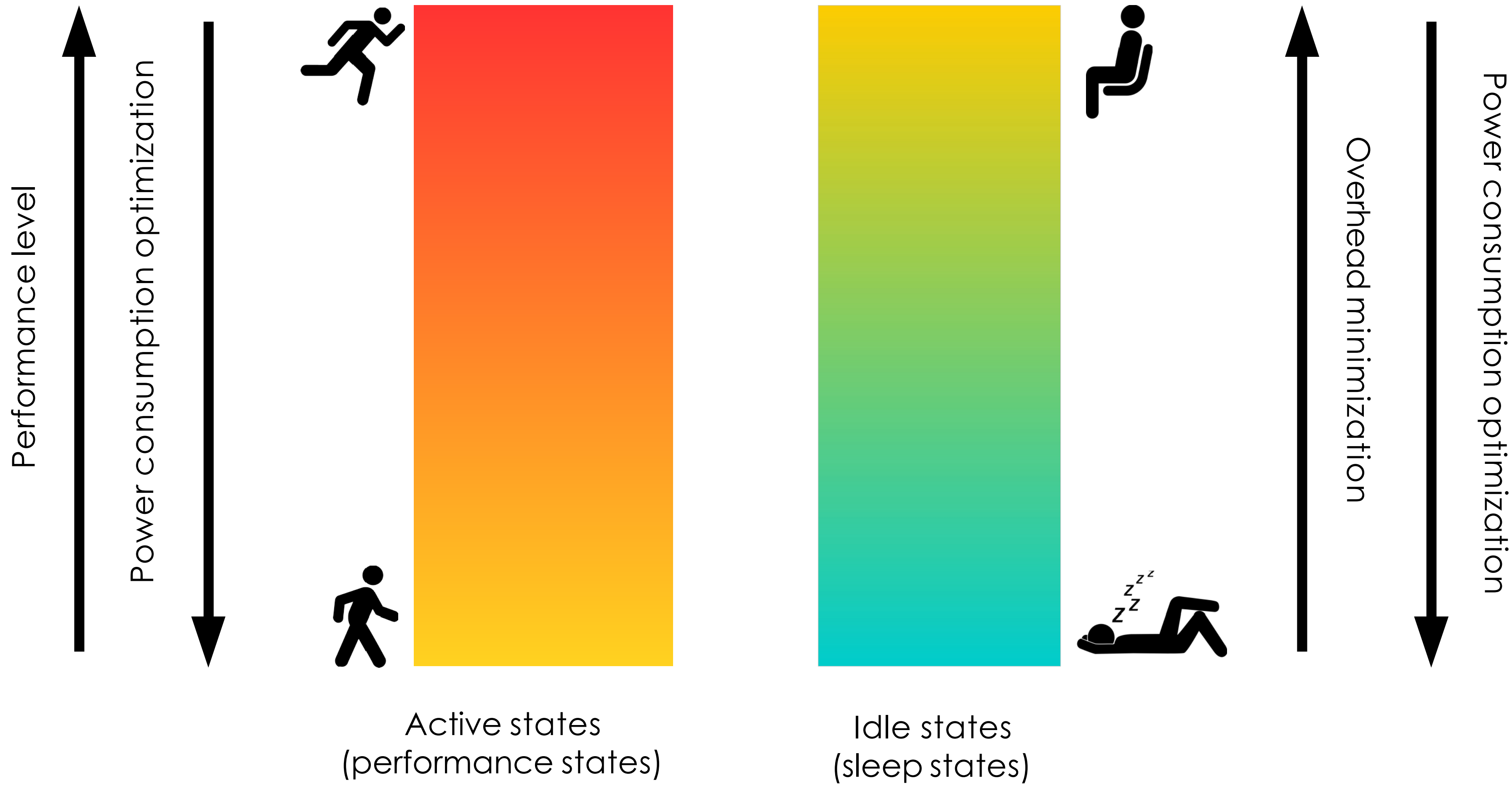
    Put the system in a lower "**performance state**" (or operating point), on the basis of the current level of activity
  - *Idle*

    Put the system into a "**sleep state**", by selectively switching-off / disabling some components, after a period of inactivity

- Power and performance are clearly in trade-off
  - *Active* : reduction of the performance
  - *Idle* : overhead due to wake-up latencies

Power states

Power-performance trade-off in power states



Power states

Power states and power saving

- *Active states (or performance states) allows us to save dynamic power (P_{dyn})*
  - The lower the state, the higher the amount of saved power and the lower the performance delivered by the system
- *Idle states (sleep states) allows us to save static power (P_{sta})*
  - The deeper the state the higher the amount of saved power and the latency of the resume process

- Power states can be de fned at a global system-wide scope or at device level
  - Can we switch-off the display while the CPU is active?
- Suitable standards have been proposed over the years for the defnition of the power states

Advanced Confguration and Power Interface

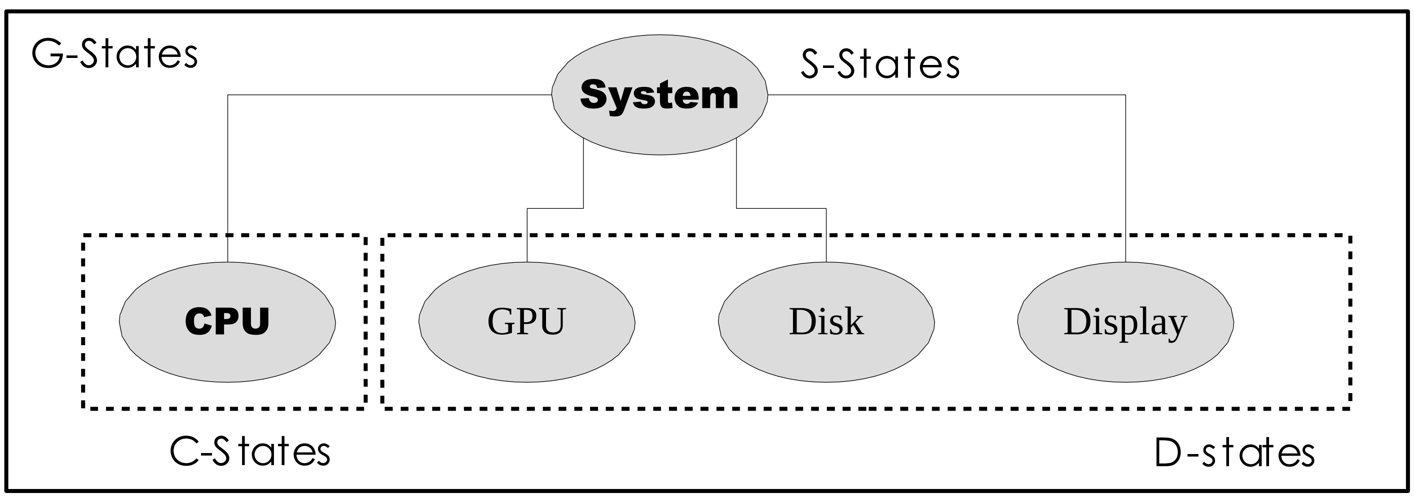
ACPI

- Specifcation introduced in 1996 by Intel, Microsoft and Toshiba
- Replacement of Advanced Power Management (APM) and the Plug-and-Play (PnP) specifcations
  - BIOS-centric approaches
  - The operating system was not aware of the APM activity
- It moves the **power management under the control of the operating system**
- It provides an open standard to discover, confgure and perform power management of hardware components
  - BIOS interfaces (confguration tables, registers, frmware)
  - System, CPU and device power states

Advanced Confguration and Power Interface

Sleep, power and performance states

- ACPI defnes power states at different hierarchy levels
  - Global system state (**G-State**)
  - System-wide sleep states **(S-State)**
  - CPU-level power/sleep states **(C-State)** and performance states **(P-State)**
  - Device level power/sleep states **(D-State)**



Advanced Confguration and Power Interface

Global system states (G-State)

State	Name	Software runs	Description	Power consumption
G0	Working	Yes	• User processes in execution. • Tunable performance levels. • Peripherals powered-on.	Large
G1	Sleeping	No	• No user processes are executed. • System appears off. • Context information are preserved. • Resume without rebooting.	Smaller (depending on the specific S-state)
G2	Soft Off	No	• No code is run. • No context is preserved. • Resume needs a system reboot (can be triggered via wake-on-LAN)	Near to 0
G3	Mechanical Switch Off	No	• No power supply. • No context is preserved. • System must be restarted to come back to G0.	RTC battery

Advanced Confguration and Power Interface

System-wide sleep states (S-States)

- The deeper the state (higher number) the higher the latency required to resume the system.

S-State	G-State	Description
S0	G0	System is working.
S1	G1	Known as “Standby”. CPU stops but still powered on. Cache is flushed. RAM still powered on. Devices powered down (where possible).
S2	G1	CPU powered off. Not always implemented.
S3	G1	Known as “Sleep” or “Suspend to RAM” (see later).
S4	G1	Known as “Hibernation” or “Suspend to disk” (see later).
S5	G2	Soft power-off. Basic power on for wake-up events (e.g., wake-on-LAN, GPIO, USB).

ACPI – Advanced Control and Power Interface

Global system management

Power states for single devices

- D0 (Fully on) to D3 (Off))

Performance states (P0-Pn)

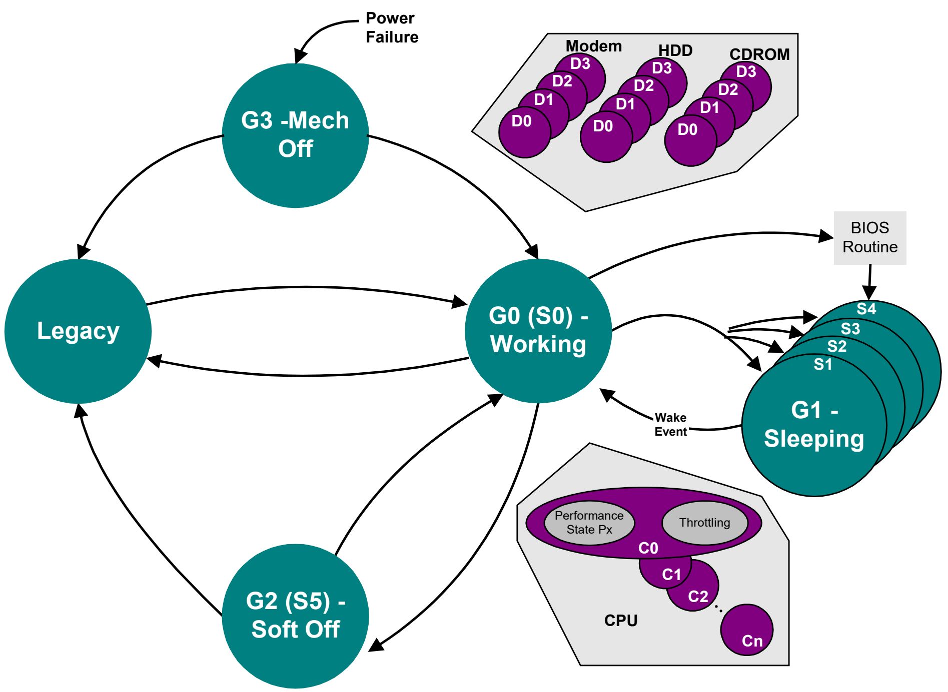
- Defined below C0 for the processor and of D0 for the other devices
- Goal is trading-off performance/power

G1: sleep states (S1-S5)

- differing for consumption, wakeup latency and saved context

G0: sub-states (C0-C3)

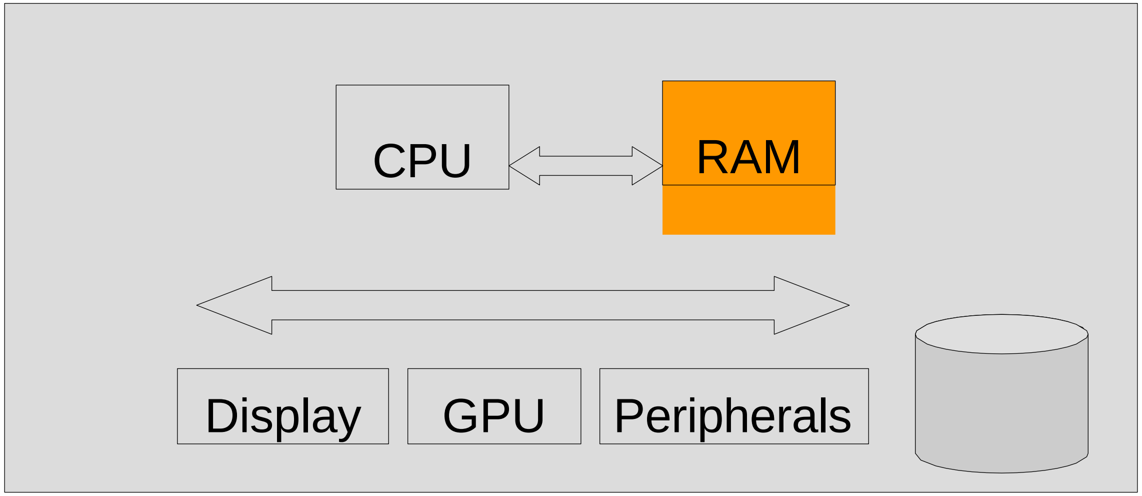
- differing for only the power of processor



Advanced Confguration and Power Interface

Suspend to RAM (STR)

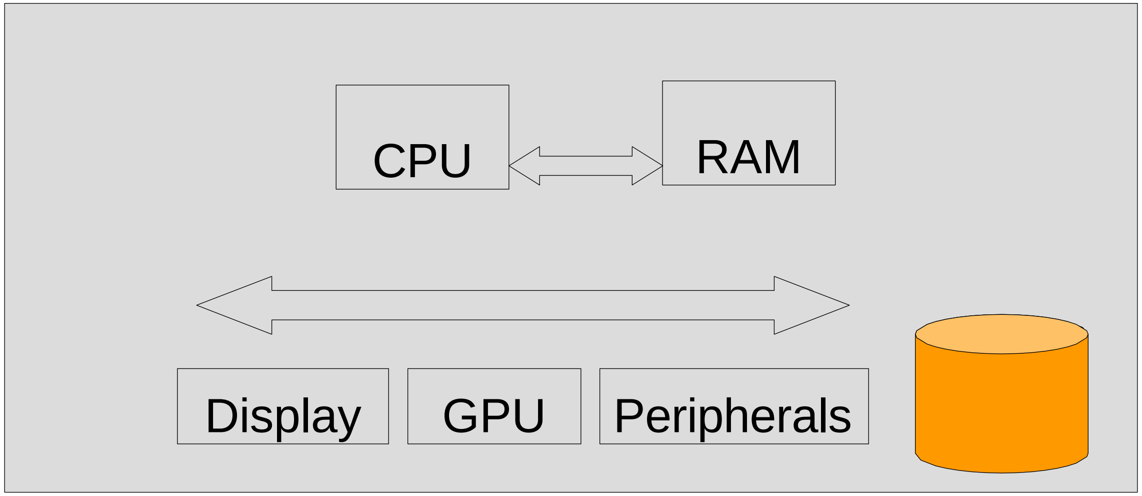
- Low wake up latencies
- Status information like system confguration, open applications, and active fles are stored in main memory (RAM)
  - RAM context and power is retained (properly refreshed)
- I/O and peripherals devices are turned off
- We expect the system to consume a few watts, mainly to power on the RAM



Advanced Confguration and Power Interface

Suspend to Disk (STD)

- Deeper sleep state than “Suspend to RAM”
  - Lower power consumption and much higher wakeup latency
- All the hardware is in off state
- All the devices are switched off
- Platform context is saved on a non-volatile memory device (disk), included the RAM content
- Restore is performed starting from the context in the image saved on the disk



Advanced Confguration and Power Interface

CPU-level sleep states (C-States)

- Idle power-saving states for the CPU
- Implemented though clock-gating or partial power-off
- For multi-core processors C-States can be applied at **package** or at **core** level also
- Actual implementations of the C-States are processor-dependent and can come in a higher number

State	Name	Description
C0	Active	CPU fully active and executing instructions according to a P-state (see later).
C1	Halt	CPU in idle state. Clock frequency scaled down.
C2	Stop-Clock	CPU in idle state. Clock frequency and voltage scaled down.
C3	Sleep	Cache state retained, but cache coherency disabled.

Advanced Confguration and Power Interface

CPU-level sleep states (C-States) on Intel

- From the Intel core-i7-800/core-i5-700 datasheet

Processor Core / Package State Support

State	Description
C0	Active mode, processor executing code.
C1	AutoHALT state.
C1E	AutoHALT state with lowest frequency and voltage operating point.
C3	Execution cores in C3 flush their L1 instruction cache, L1 data cache, and L2 cache to the L3 shared cache. Clocks are shut off to the core.
C6	Execution cores in this state save their architectural state before removing core voltage.

- Wake up latencies usually in the range (0.x – 100 µs)
  - → http: /ena-hpc.org/2014/pdf/paper_06.pdf
  - → http: /dx.doi.org/10.1007/s00450-014-0270-z

Advanced Confguration and Power Interface

CPU-level performance states (P-States)

- Active power saving states for the CPU
  - CPU must be in C-state C0
- Implemented by reducing clock frequency and/or voltage (**DVFS: Dynamic Voltage/Frequency Scaling**)
- For multi-core processors P-States can be applied at **package** or at **core** level also (as for C-States)

State	Description
P0	Full-speed configuration: maximum voltage, frequency and thus performance level.
P1	Reduced speed: voltage and frequency are scaled, releasing lower performance than P0.
Pn	Further levels of voltage and frequency scaling, up to 255 maximum.

Advanced Confguration and Power Interface

Device "D" states

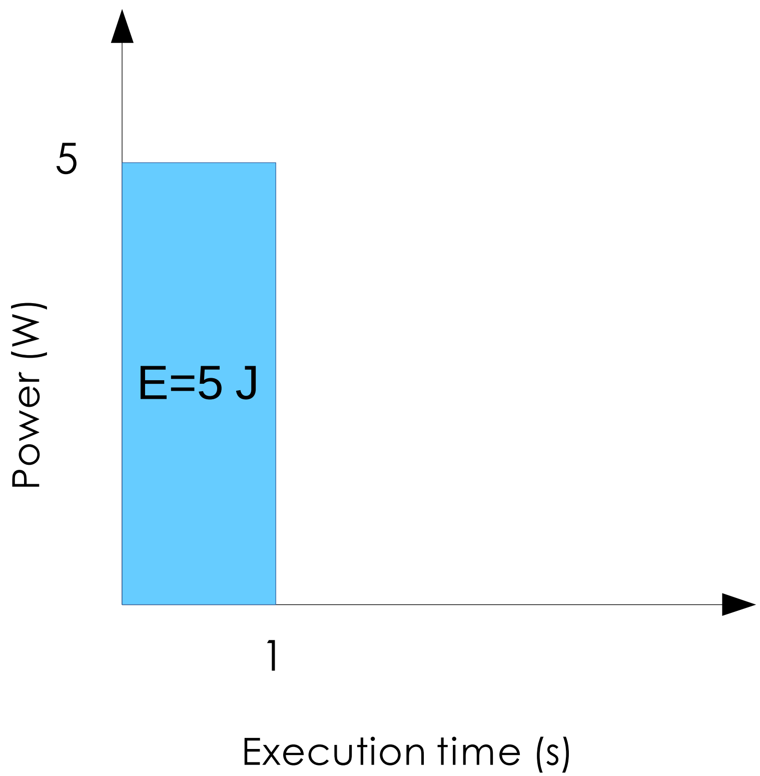
- The device is functional only in D0
- As usual, the deeper the state the higher the power saving and the wake-up latency

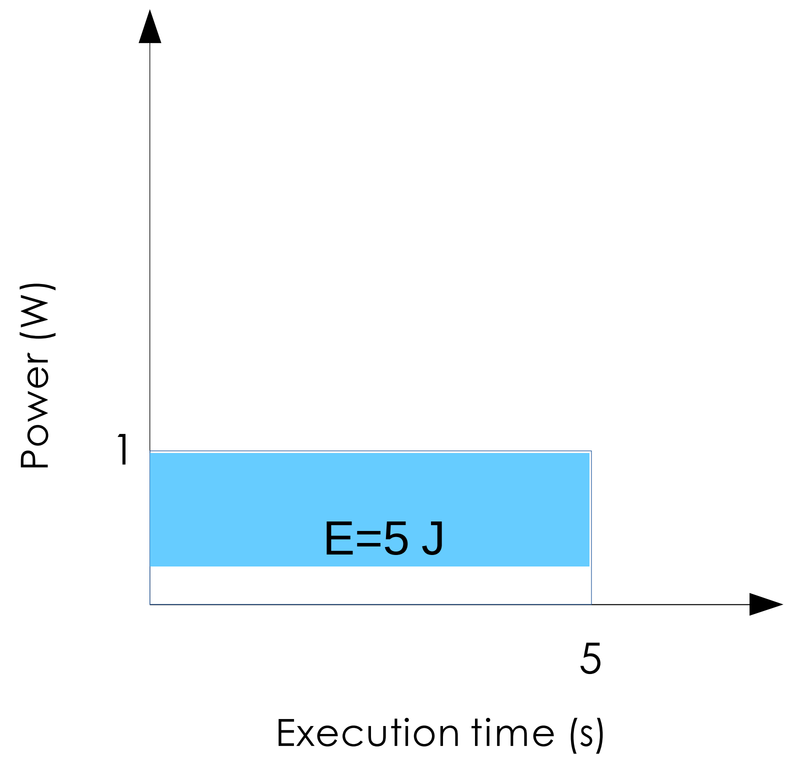
State	Description
D0	Device is fully on
D1	Intermediate sleep state defined by the device
D2	Intermediate (deeper) sleep state defined by the device
D3	Device is powered off

Power management philosophy

Race-to-idle vs Slow-down

- Example: different power consumption profles but equivalent energy consumption
  - Which approach would you go for?

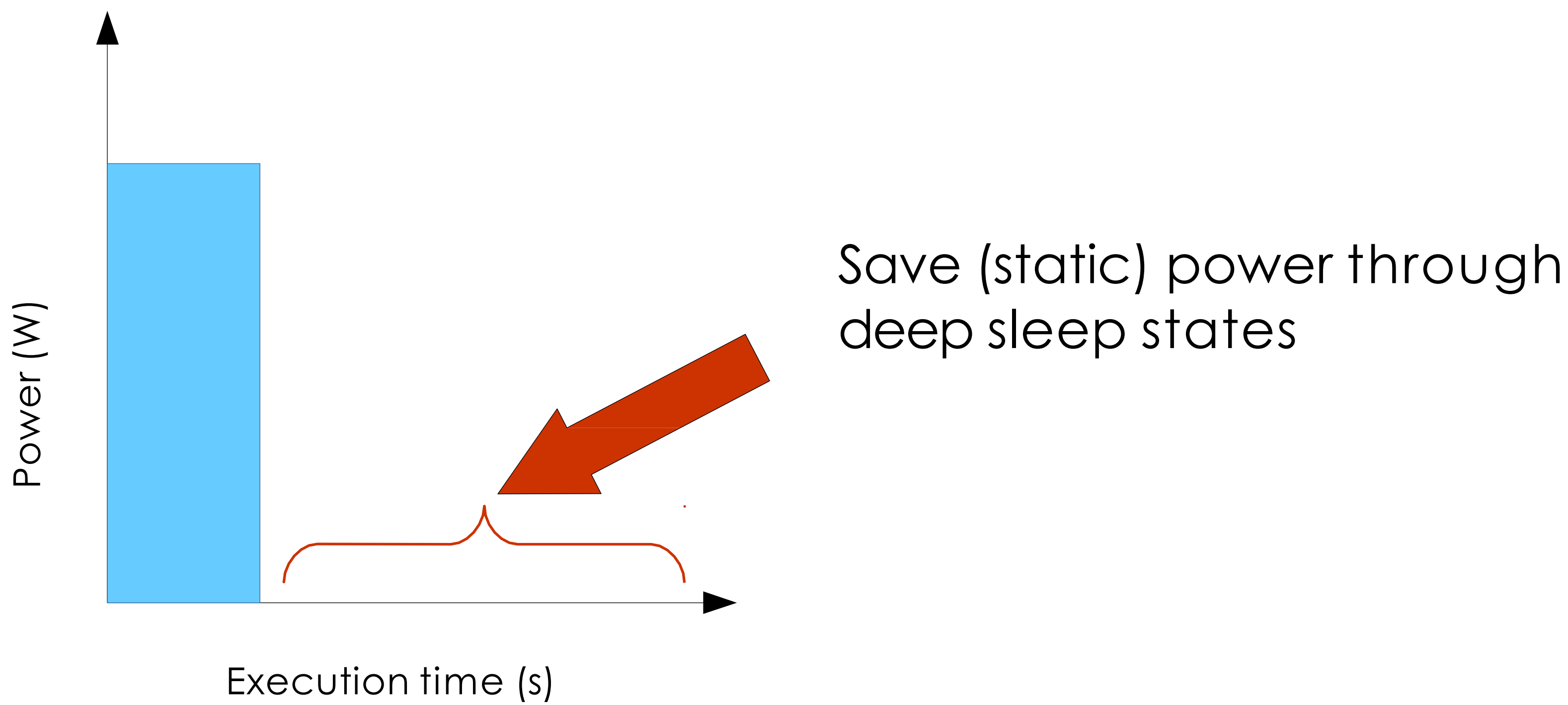




Power management philosophy

Race-to-idle

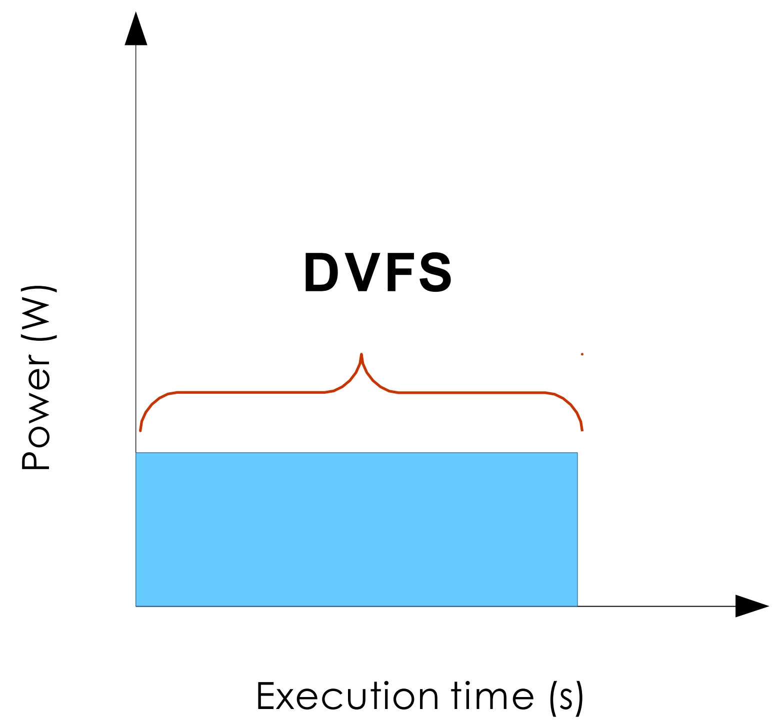
- Complete the execution as fast as possible and go to sleep ASAP (exploit sleep states for save power)
  - Good for HPC domains with poor interactivity
- Used in Intel CPUs, characterized by fast "wake-up from idle" and very good power saving techniques when in idle



Power management philosophy

Slow-down

- Usually adopted on ARM processors
- Save (dynamic) power by exploiting DVFS
  - Adaptive CPU clock frequency / voltage scaling
- More suitable for interactive systems (e.g., smartphone)
  - No much room for exploiting deep sleep states
  - Remember wake-up latencies!

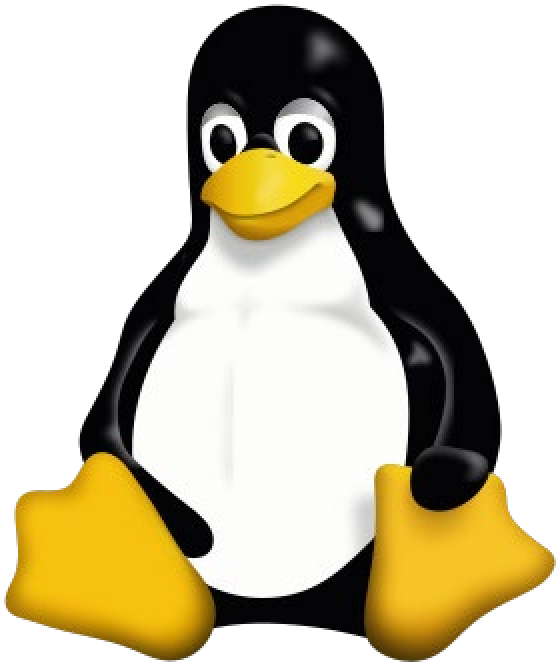


Power management philosophy

Race-to-idle vs Slow-down: who is the winner?

- Very system and application dependent
- We should compare the energy consumption profles
  - Find a break-even point in the P-state selection
  - C-state power saving vs wake up latencies
- Necessary to take into account the other requirements
  - Batch or interactive applications?
  - Any real-time requirement?
- What about thermal management?
  - Race-to-idle leads to higher peak temperature values
  - Thermal management policies could set and upper bound to the highest possible P-state

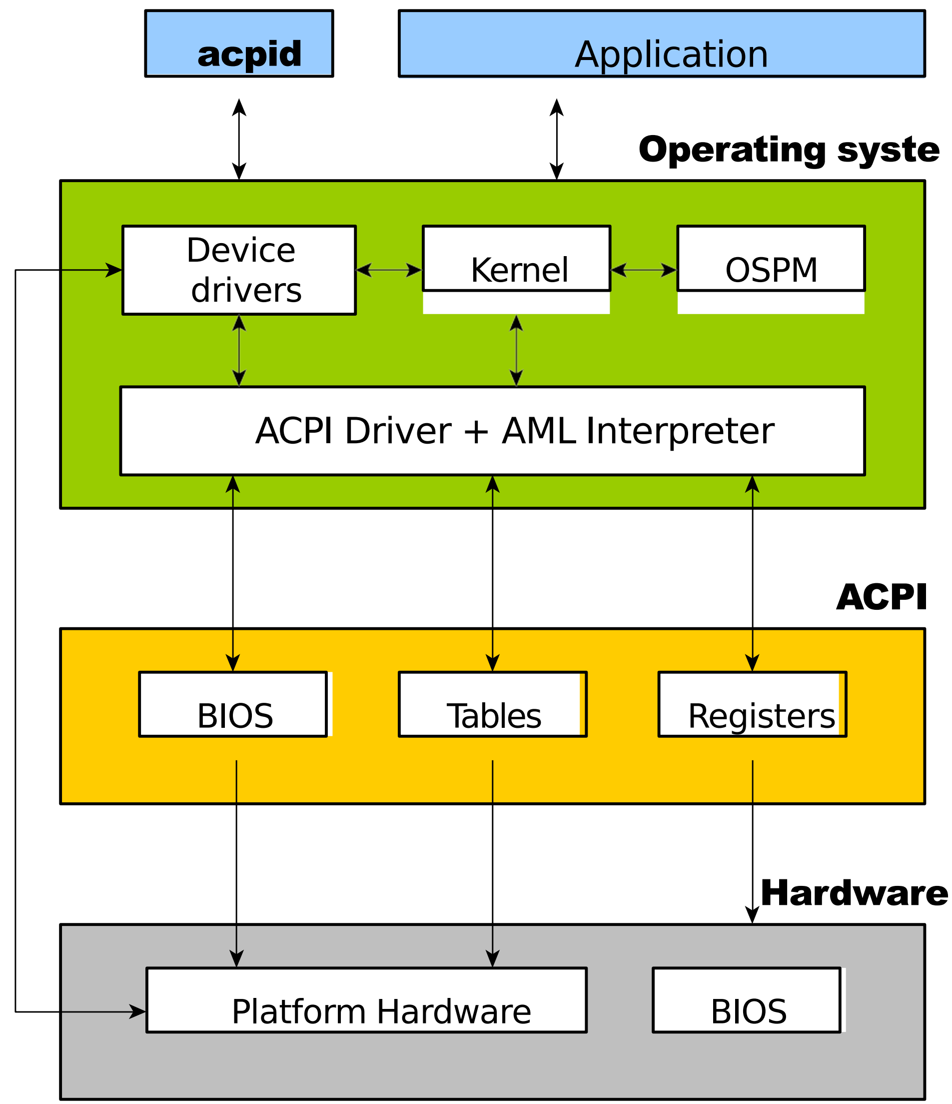
Operating system integration
(the Linux case)



Hardware/Software integration

ACPI integration stack

- ACPI daemon (*acpid*)
  - Listen for ACPI events and launch the action handler
- Operating system
  - The power manager and related policy (OSPM)
  - ACPI subsystem including driver and the interpreter for the ACPI Machine Language (AML)
- For the power management activities, device drivers can exploit ACPI interfaces or not



Hardware/Software integration

ACPI integration stack

- The tools **acpi_listen** intercepts and prints the ACPI events caught by the daemon
  - Laptop lid close/open
  - Battery events
  - Power on/off button presses
  - Display brightness control
  - …

```
$ acpi_listen
button/lid LID close
button/lid LID open
battery PNP0C0A:00 00000081 00000001
button/power PBTN 00000080 00000000 K
PNP0C14:01 000000d0 00000000
video/brightnessdown BRTDN 00000087 00000000
 PNP0C14:01 000000d0 00000000
video/brightnessup BRTUP 00000086 00000000
```

Hardware/Software integration

Device Tree (DT)

- Data structure and language for providing a topological description of the hardware to the operating system
  - SoC, CPUs, busses, GPIO connections, peripheral devices, …
  - **Now, exploited to include power management information too**
- Introduced by Open Firmware for PowerPc and SPARC architercures
  - Now supported by *arm*, *microblaze*, *mips*, *powerpc*, *sparc*, and *x86*
- Structurally it is tree, i.e., an acyclic graph with named nodes
  - A node have an arbitrary number of named properties encapsulating arbitrary data and links
  - A common set of usage conventions, called ‘bindings’, is defned
- A textual representation (*Flattened Device Tree*) is translated into a binary blob (*Device Tree Binary*) passed to the kernel at boot-time
  - **.dtb** fle under /boot

Hardware/Software integration

Device Tree (DT)

- Example: exynos5422-odroidxu3.dts

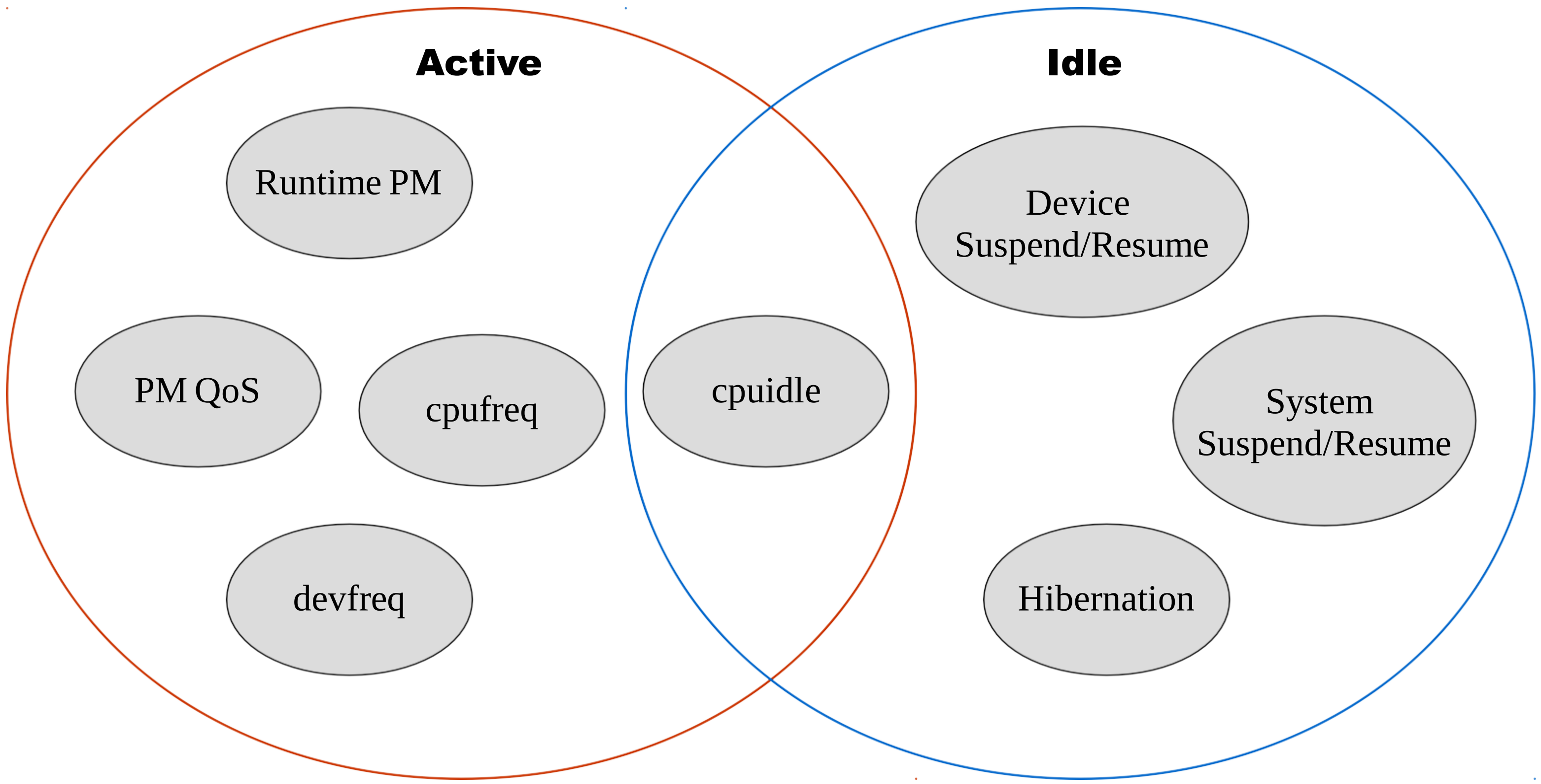
```
/dts-v1/;
#include "exynos5422-odroidxu3-common.dtsi"
#include "exynos5422-odroidxu3-audio.dtsi"
#include "exynos54xx-odroidxu-leds.dtsi"
/ {
	model = "Hardkernel Odroid XU3";
	compatible = "hardkernel,odroid-xu3",
"samsung,exynos5800", "samsung,exynos5";
};

&i2c_0 {
	status = "okay";

	/* A15 cluster: VDD_ARM */
	ina231@40 {
		compatible = "ti,ina231";
		reg = <0x40>;
		shunt-resistor = <10000>;
	};

	/* memory: VDD_MEM */
	ina231@41 {
		compatible = "ti,ina231";
		reg = <0x41>;
		shunt-resistor = <10000>;
	};
```

```
...

	/* GPU: VDD_G3D */
	ina231@44 {
		compatible = "ti,ina231";
		reg = <0x44>;
		shunt-resistor = <10000>;
	};

	/* A7 cluster: VDD_KFC */
	ina231@45 {
		compatible = "ti,ina231";
		reg = <0x45>;
		shunt-resistor = <10000>;
	};
};
...

&usbdrd_dwc3_1 {
	dr_mode = "peripheral";
};
```

Power management in Linux

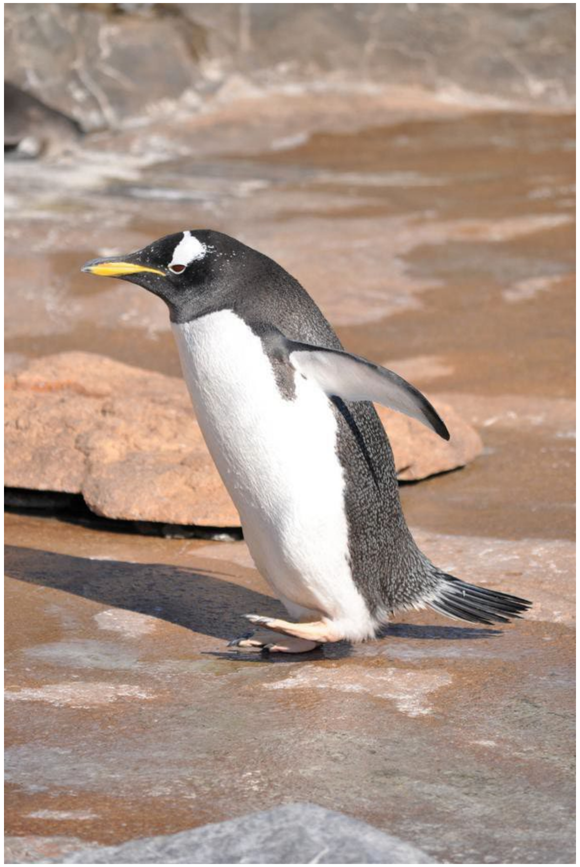
Power management frameworks

- In the Linux kernel, the OSPM is implemented through a set of internal frameworks
  - (Partial) distinction between frameworks for power management when the system or device is active vs idle



Power management in Linux

Active power management



Active power management in Linux

cpuidle

- Both an active and idle power management framework
- Manage the state transitions of the CPU between C-states
  - C-states defned in the device tree with latency information
- Suitable governors (*menu*, *ladder*) implement the transition policies based on…
  - Latency limitations imposed by the QoS framework
  - Entry/exit time, minimum residency time (worth to save power)
  - Next predictable event (timers)
  - Heuristics based on the CPU load
- The governor execution is triggered by the scheduler, when the “idle task” is scheduled
  - No active tasks to schedule on the current CPU
- The CPU driver is responsible for the actual C-state selection
  - For example **intel_idle** driver uses the special MWAIT instruction to inform the CPU about switching in idle state

Active power management in Linux

cpuidle

- Example from a Vexpress SoC with big.LITTLE ARM CPU

```
cpus {
	cpu0: cpu@100 {
		device_type = "cpu";
		compatible = "arm,cortex-a15";
		...
		cpu-idle-states = <&CLUSTER_SLEEP_BIG>;
	};

	idle-states: {
		CLUSTER_SLEEP_BIG: cluster-sleep-big {
			compatible = "arm,idle-state";
			local-timer-stop;
			entry-latency-us = <1000>;
			exit-latency-us = <700>;
			min-residency-us = <2000>;
		};
		...
	}
}
```

Active power management in Linux

cpuidle

- Such information can be retrieved via a sysfs interface, along with some runtime statistics
  - A **state<N>** directory is created for each C-state

```
$ tree -L 2 /sys/devices/system/cpu/cpu*/cpuidle
/sys/devices/system/cpu/cpu*/cpuidle
├── state0
│   ├── desc
│   ├── disable
│   ├── latency
│   ├── name
│   ├── power
│   ├── residency
│   ├── time
│   └── usage
├── state1
├── state2
├── state3
...
```

Active power management in Linux

Operating Performance Point (OPP) library

- A library through which drivers and frameworks can manage the set of **<frequency, voltage>** pairs (also called P-states), supported by the devices and the SoC power domains
  - Usually, specifed in the *device tree*
  - Exploited by the DVFS frameworks (e.g., cpufreq) to navigate through the set and select the performance point
- For each OPP, a **struct dev_pm_opp** object is created
- Provides lookup functions for retrieving OPPs on the basis of a target frequency (mix, max, exact,...)
- Provides functions to dynamically enable/disable OPPs on the basis, for instance, of *thermal management* policies

Active power management in Linux

Operating Performance Point (OPP) library

- Example from the DT of the Samsung Exynos5422 SoC

```
cpus {
        cpu0: cpu@100 {
                device_type = "cpu";
                compatible = "arm,cortex-a7";
                ...
                operating-points-v2 = <&cluster_a7_opp_table>;
                ...
        };
        ...
}
```

```
soc: soc {
        cluster_a7_opp_table: opp_table1 {
                compatible = "operating-points-v2";
                opp-shared;
                opp@1300000000 {
                        opp-hz = /bits/ 64 <1300000000>;
                        opp-microvolt = <1275000>;
                        clock-latency-ns = <140000>;
        };
                opp@120000000 { ... };
                opp@110000000 { ... };
                ...
```

<1.30GHz,
1.275V>

Active power management in Linux

cpufreq

- The framework responsible of performing CPU DVFS (P-state selection)
- Several **governors** available, implementing specifc DVFS policies

Governor name	Description	Note
performance	Requires to set the highest frequency value.	scaling_max_freq could be set as upper bound.
powersave	Requires to set the lowest frequency value.	scaling_min_freq could be set as lower bound.
userspace	Leave the explicit frequency setting to the user.	scaling_setspeed attribute exposed for this.
schedutil	Tight interaction with the scheduler. The frequency is selected according to the CPU utilization.	Used with recent Android-based devices, in conjunction with the Energy-Aware Scheduler.
ondemand	Requires to set the frequency on the basis of the CPU load (active time)	Often the default option.
conservative	Similar to ondemand, but the selection of frequency is performed in a progressive manner.	Thought for battery-powered devices in general.

Active power management in Linux

cpufreq

- CPU drivers are then responsible for the actual P-state setting
  - **intel_pstate** driver (enabled by default on modern Intel-based systems) bypasses the governors and implements its own scaling policy based on information retrieved from specifc registers (Intel Machine-Specifc Registers, MSR)
- User-space interface via sysfs, allows the user to select and confgure the current governor

```
$ tree -L 2 /sys/devices/system/cpu/cpu*/cpufreq
/sys/devices/system/cpu/cpu0/cpufreq
├── afected_cpus
├── cpuinfo_cur_freq
├── cpuinfo_max_freq
├── cpuinfo_min_freq
├── cpuinfo_transition_latency
├── related_cpus
├── scaling_available_governors
├── scaling_cur_freq
├── scaling_driver
├── scaling_governor
├── scaling_max_freq
├── scaling_min_freq
└── scaling_setspeed
/sys/devices/system/cpu/cpu1/cpufreq
 ...
```

Active power management in Linux

devfreq

- It is the cpufreq equivalent framework for devices
- If the driver can monitor the activity of the device, it can run agovernor for managing the DVFS
  - A devfreq structure is defned to set the governor, the QoS constraints and collect statistics

```
struct devfreq {
        ...
        struct device dev;
        struct devfreq_dev_profle *profle;
        const struct devfreq_governor *governor;
        ...
        struct devfreq_dev_status last_status;
        ...
        struct  dev_pm_qos_request  user_min_freq_req;
        struct dev_pm_qos_request user_max_freq_req;
        unsigned long scaling_min_freq;
        unsigned long scaling_max_freq;
        ...
        struct devfreq_stats stats;
        ...
};
```

Active power management in Linux

PM QoS (Quality-of-Service) Interface

- User and kernel-space interface for setting performance goals by drivers, subsystems and applications
  - Goals in terms of latency (µs)
- Two different classes, with specifc request types
  - **System-wide PM QoS**
    - CPU-DMA latency
    - Network latency
    - Network throughput
    - Memory bandwidth
  - **Device-specifc PM QoS**
    - Resume latency
    - Latency tolerate

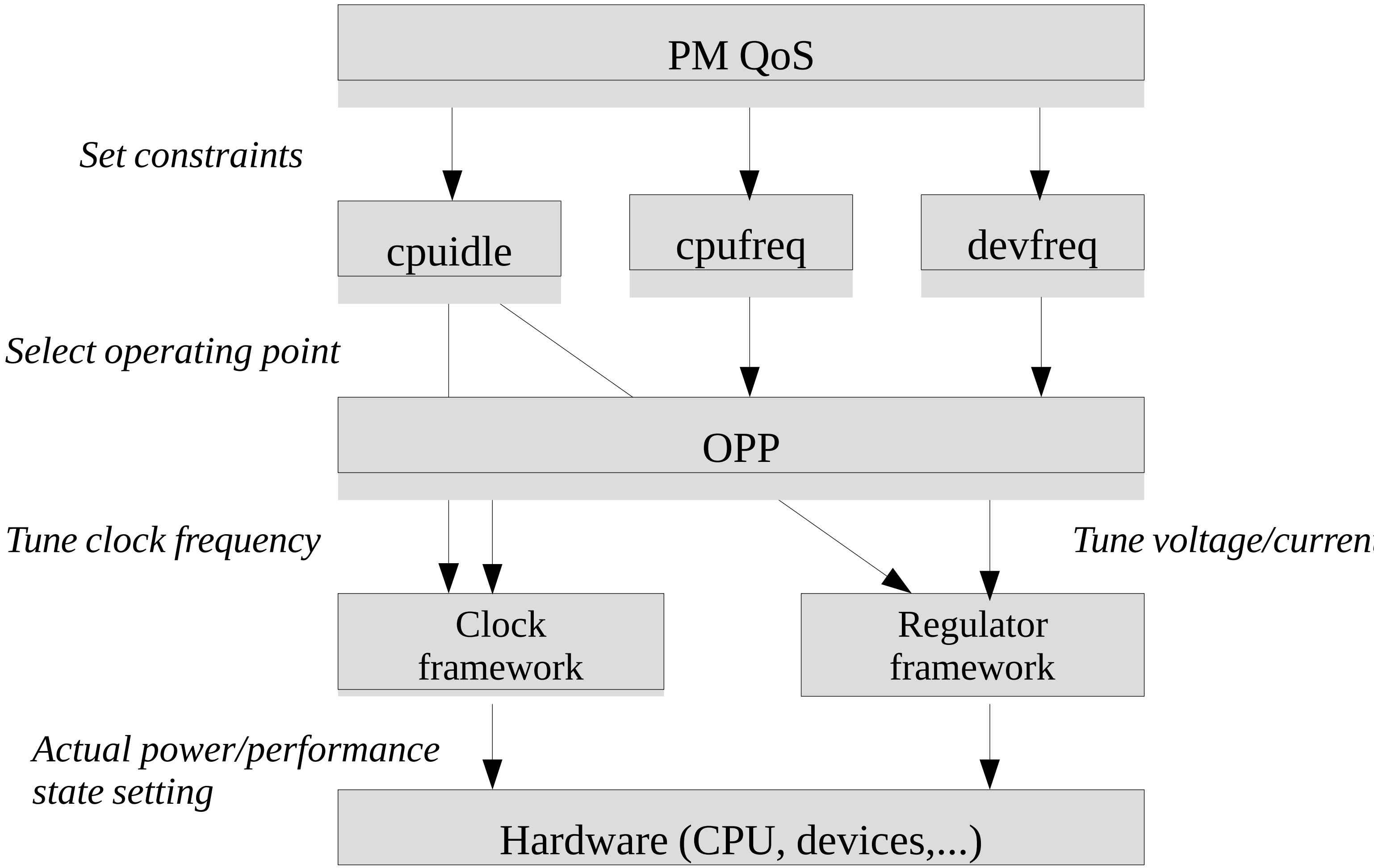
Active power management in Linux

PM QoS (Quality-of-Service) Interface

- Processes and kernel threads register QoS requests
- They introduce constraints on the PM frameworks governors (cpufreq, cpuidle, devfreq,…) for the selection of C-state, P-state and D-state
- Such constraints affects also "idle" power management state transitions (see later)

Active power management in Linux

Frameworks hierarchy



Active power management in Linux

PowerTop

- Administration tool for profling power/performance states transitions
  - It parses the **/sys** flesystem interfaces

PowerTOP 2.8 Overview Idle stats Frequency stats Device stats Tunab

Summary: 7748,2 wakeups/second, 264,4 GPU ops/seconds, 0,0 VFS ops/sec and 190,

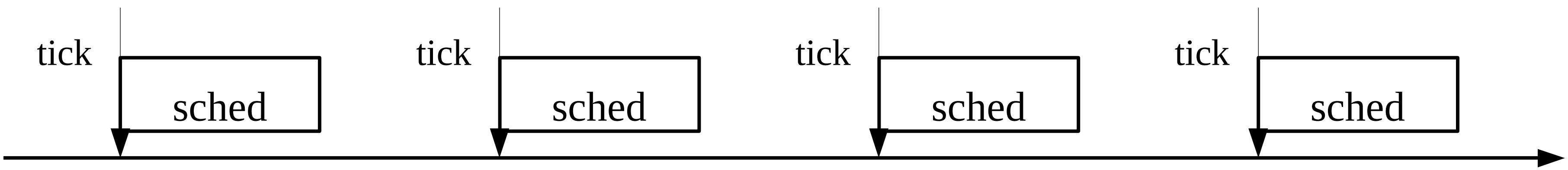
Usage	Events/s	Category	Description
4,6 ms/s	1198,6	Interrupt	[279] nvme0q2
125,6 ms/s	809,0	Process	/usr/lib/virtualbox/Virtu
3,4 ms/s	765,4	Interrupt	[281] nvme0q4
863,8 ms/s	243,1	Process	/usr/lib/libreofce/prog
1,4 ms/s	438,8	Process	[i915/signal:0]
7,8 ms/s	436,0	Interrupt	[6] tasklet(softirq)
77,1 ms/s	382,2	Process	[kswapd0]
1,9 ms/s	387,8	Interrupt	[284] nvme0q7
3,8 ms/s	330,3	Timer	hrtimer_wakeup
3,9 ms/s	289,4	Process	[kworker/1:1H]
21,5 ms/s	185,5	Process	/usr/bin/pulseaudio --sta
1,7 ms/s	294,1	Interrupt	[285] nvme0q8
324,5 ms/s	138,2	Process	/opt/google/chrome/chrome
2,9 ms/s	256,0	Timer	tick_sched_timer
191,9 ms/s	115,0	Process	/usr/lib/xorg/Xorg -core
101,2 ms/s	109,5	Process	compiz
631,0 µs/s	148,4	Process	[rcu_sched]

<ESC> Exit | <TAB> / <Shift + TAB> Navigate |

Active power management in Linux

Scheduling clock ticks confguration

- We can minimize the CPU activity also by passing from a periodic invocation of the OS scheduler, to an event-based approach (*tickless*)
- Kernel confguration options:
  - **HZ_PERIODIC** – Periodic invocation of the scheduler



  - **NO_HZ_IDLE** - Omit scheduling clock ticks on idle
  - **NO_HZ_FULL** - Omit scheduling clock ticks on idle or with a single runnable task

Power management in Linux

Idle power management

Idle power management in Linux

RuntimePM

- Each device driver registers the set of power management callback functions by proving a **struct dev_pm_ops** data structure
- Idleness controlled by the device driver based on the activity
- Devices are independent →no device can prevent another one from suspending
- User-space is not directly involved

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

Idle power management in Linux

RuntimePM

- A drivers should perform explicit *active* or *idle* requests through the following framework functions
  - A reference counting mechanism triggers the callbacks invocation
  - Go active

    Increment reference counter

```
int pm_runtime_get();
int pm_runtime_get_sync();
```

  - Go to idle

    Decrement reference counter

```
int pm_runtime_put();
int pm_runtime_put_sync();
int pm_runtime_put_autosuspend();  // defer the suspension
```

Idle power management in Linux

RuntimePM

- Usage counter == 0
  - Call dev→**runtime_suspend()**

    Prepare for suspension

    Save the device context

    Enable the wake up procedure
- Usage counter == 1
  - Call dev→**runtime_resume()**

    Restore the context

- Per-device PM QoS requests can affect RuntimePM!
  - Some requests could prevent some devices from being suspended

Idle power management in Linux

GenPD (generic power domain)

- Allows us to group the devices in power domains
  - We can apply a power saving action on a power domain scale instead of a single device / functional unit

    Override PM callbacks
  - Has some latency implications!
- The power domains are registered through **struct generic_pm_domain** objects provided by the device drivers

```
struct generic_pm_domain {
...
	int (*power_off) (struct generic_pm_domain *domain);
	int (*power_on) (struct generic_pm_domain *domain);
	int (*set_performance_state)(struct generic_pm_domain
	*genpd, unsigned int state);
	...
}
```

Idle power management in Linux

GenPD (generic power domain)

- Power domains are defned through the *device tree*
  - Coherently with the physical power domains

```
soc: soc {
...
        mixer: mixer@14450000 {
                compatible = "samsung,exynos5420-mixer";
                ...
                clocks = <&clock CLK_MIXER>, <&clock CLK_HDMI>, <&clock CLK_SCLK_HDMI>;
                clock-names = "mixer", "hdmi", "sclk_hdmi";
                power-domains = <&disp_pd>;
                iommus = <&sysmmu_tv>;
        };

        disp_pd: power-domain@100440C0 {
                compatible = "samsung,exynos4210-pd";
                reg = <0x100440C0 0x20>;
                #power-domain-cells = <0>;
                clocks = <&clock CLK_FIN_PLL>, <&clock CLK_MOUT_USER_ACLK200_DISP1>,
                        <&clock CLK_MOUT_USER_ACLK300_DISP1>,
                        <&clock CLK_MOUT_USER_ACLK400_DISP1>,
                        <&clock CLK_FIMD1>, <&clock CLK_MIXER>;
                clock-names = "oscclk", "clk0", "clk1", "clk2", "asb0" "asb1";
        };
}
```

Idle power management in Linux

GenPD (generic power domain)

- It is based on RuntimePM
  - Reference counting mechanism…
- When all the devices are runtime suspended…
  - Call to genpd->**power_off**()
- When the frst device in the domain is runtime resumed…
  - Call to genpd->**power_on**()
- When a device wants to switch performance state...
  - Defnitely an active power management hook
  - **dev_pm_genpd_set_performance_state()**

Idle power management in Linux

Suspend/Resume: sleep states transitions

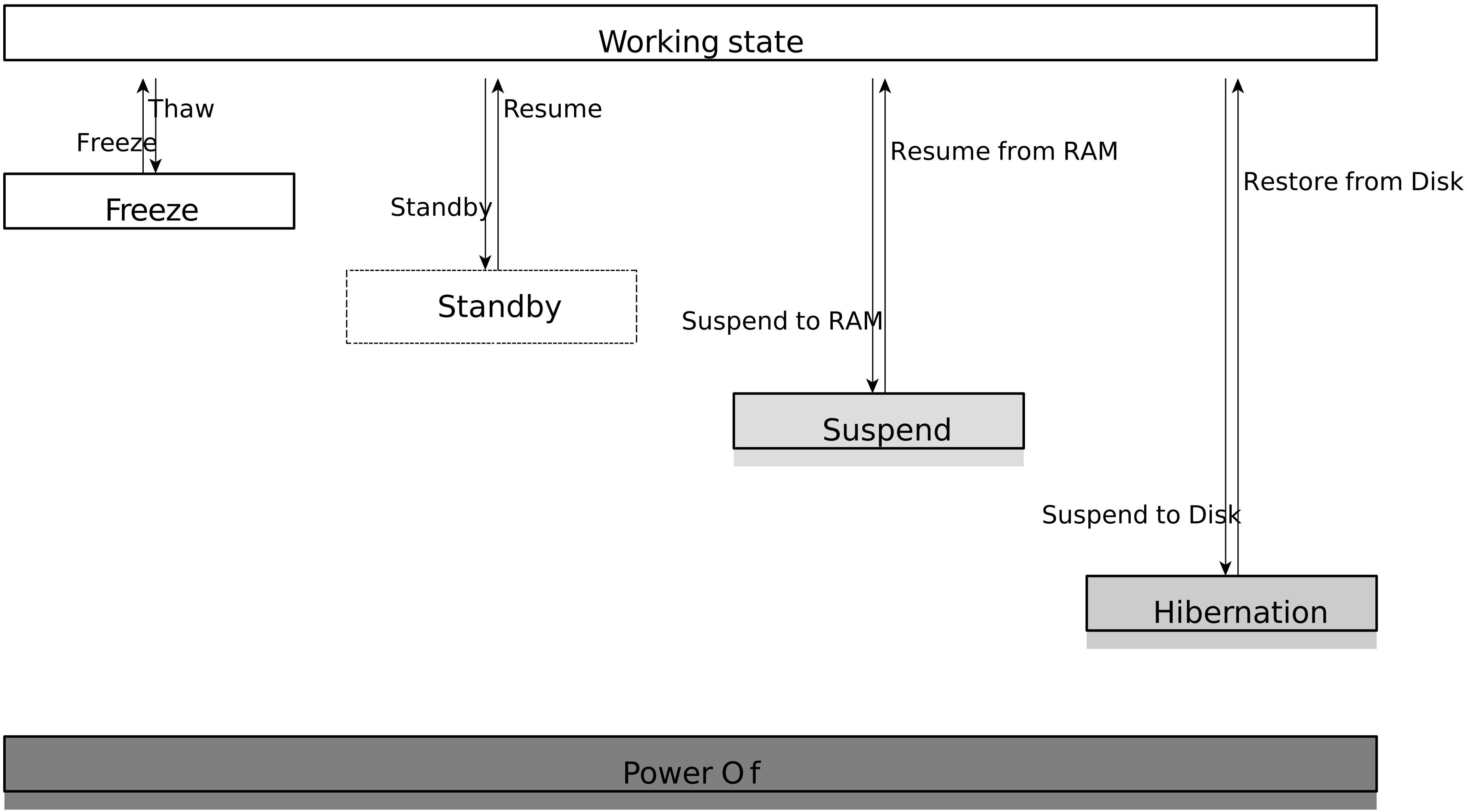
- Triggered by the user (e.g., closure of the laptop lid)
- **Suspend to Idle** (freeze)
  - Prevent user-space processes to execute
  - Put the CPU in the deepest C-state
- **Power on suspend** (standby)
  - Put the non-boot CPUs offine (not always available)
- **Suspend to RAM** (mem)
- **Suspend to Disk (Hibernation)** (disk)

- Check the sleep states support by your system via sysfs

```
$ cat /sys/power/state
freeze mem disk
```

Idle power management in Linux

Suspend/Resume: sleep states transitions



Idle power management in Linux

Suspend/Resume: hooks

- For each device and subsystem, the previously introduced **dev_pm_ops** structure must include pointers to the following functions

```
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);
      ...
}
```

Idle power management in Linux

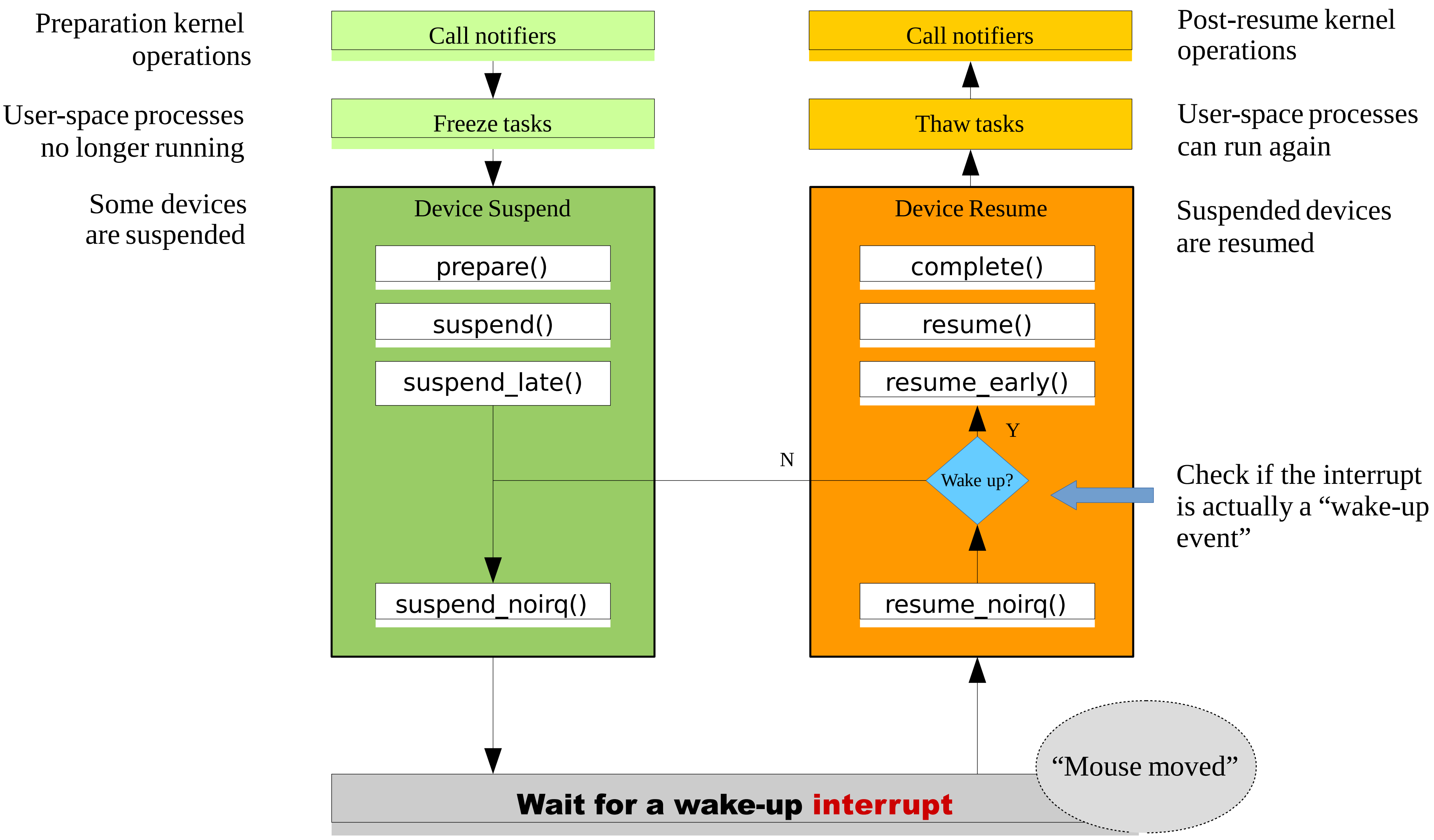
Suspend/Resume: hooks

- The device drivers (and the subsystems) should confgure themselves for waking up
  - The function **device_init_wakeup(dev, bool)** allows the device to inform the PM framework that it is "wake-up capable"
- The kernel provides the function **enable_irq_wake**() to enable the device for receiving system wakeup interrupts
- Once the device has been resumed, the **disable_irq_wake**() does the opposite
  - The PM framework should be notifed, via **pm_wakeup_event**()
- A user-space interface is exposed via *sysfs* for enabling/disabling wakeup capabilities for each device

```
$ ls -l /sys/devices/*/power/wakeup
```

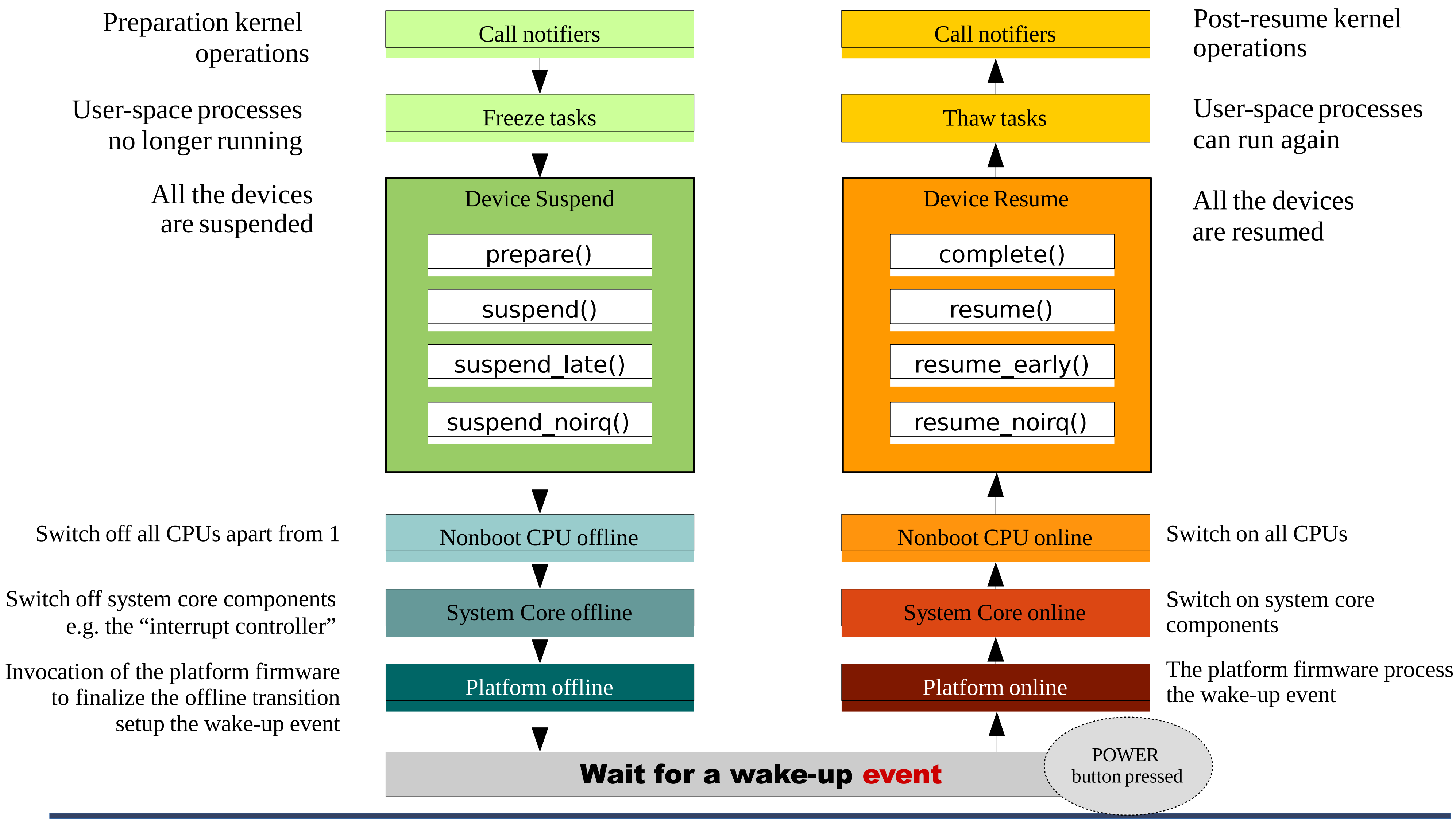
Idle power management in Linux

Suspend-to-Idle



Idle power management in Linux

Suspend-to-RAM



Idle power management in Linux

Suspend-to-Disk (Hibernation)

- *Suspension*
  - The kernel stops all the system activity
  - The kernel creates a snapshot image of memory to be written into persistent storage
  - The snapshot image is written out
  - The system goes into the target low-power state

    (RAM is off too, only a few wakeup devices are on)
- *Restore*
  - A wakeup event triggers the restore (e.g., power button pressure)
  - The platform frmware runs the boot loader which boots a fresh instance of the kernel (*restore kernel*)
  - The restore kernel looks for a hibernation image in the persistent storage and if one is found, it is loaded into memory
  - The restore kernel overwrites itself with the image contents and the execution jumps into the original kernel, stored in the image

    Here, special architecture-specifc low-level code must be executed
  - The *image kernel* restores the system to the pre-hibernation state and allows user space processes to run again

Conclusions

Recap

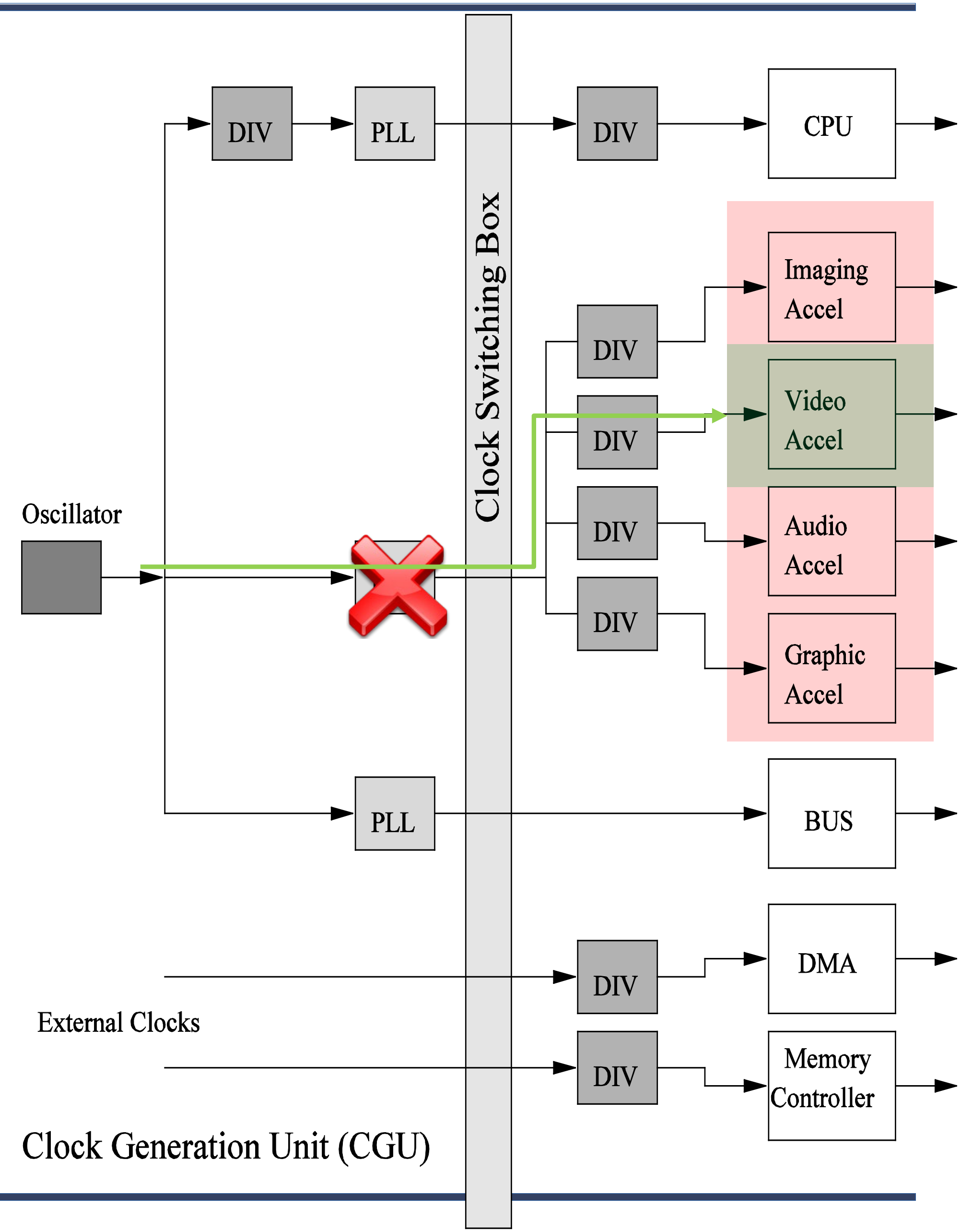
- Nowadays power management frameworks are fundamental components of an operating system
- Power saving opportunities can be exploited when the system is both active and idle state
- The complexity of the modern systems requires to follow a hierarchical approach
  - From the system-wide to the per-device perspective
- Linux provides several frameworks providing hooks for the kernel subsystems and device drivers, along with governors
- Optimal power management strategies should predict power saving opportunities and consider the inter-dependencies among system components

Centralized control for clock distribution

- Track clocks dependencies
- Abstract API specification

  clock rate set/get
- Required device-driver cooperation

  aggressive get/put clocks



Keep track of devices power dependencies

- Optimize regulators usage and efficiency
- Current sinks dependency tracking
- Dynamically control regulator modes

Optimize regulator efficiency

- Depend on current load

e.g. with 10mA load
70% @normal ~ 13mA
90% @idle ~ 11mA

Saving ~ 2mA