ITMO Advanced OS 2024

Aleksei Romanovskii, PhD,
SPb Research Center (CBG OS Lab)

Lesson 2024.12.11

EAS basic principles

- Task scheduling that considers energy implications
- Decision should be made basing on:
  - System topology
  - E. g. SMP or HMP
  - Power management features
  - CPU Idle states, DVFS
  - Workload for each core
- Work load calculation is basically independent
  - Separate module providing results to EAS

EAS explained 1 of 3

- EAS goal is to minimize energy, while still getting the job done. That is, we want to maximize:
  - performance (ipc) / power(Watt)

- EAS relies on an Energy Model (EM) of the CPUs to select an energy efficient CPU for each task, with a minimal impact on throughput
  - EM is an interface between drivers knowing the power consumed by devices at various performance levels, and the kernel subsystems willing to use that information to make energy-aware decisions.
  - EM considers CPUs only, i.e. no peripherals, GPU or memory. Model data includes power consumption at each P-state and C-state.

EAS explained 2 of 3

- EAS changes the way CFS tasks are assigned to CPUs.
  - When it is time for the scheduler to decide where a task should run (after wake-up), the EM is used to break the tie between several good CPU candidates and pick the one that is predicted to yield the best energy consumption without harming the system's throughput.
  - The predictions made by EAS rely on specific elements of knowledge about the platform's topology, which include the 'capacity' of CPUs, and their respective energy costs.
  - The 'capacity' of a CPU represents the amount of work it can absorb when running at its highest frequency compared to the most capable CPU of the system.
- EAS categorizes processes into four cgroups, being top-app, system-background, foreground, and background.
- Tasks due to be processed are placed into one of these categories, and then the category is given CPU power and the work is delegated over different CPU cores.
  - The top-app is the highest priority of completion, followed by foreground, background, and then system-background.
  - The background technically has the same priority as system-background, but system-background usually also has access to more little cores.

EAS explained 3 of 3

- When waking the device, EAS will choose the core in the shallowest idle state, minimizing the energy needed to wake the device.
- This helps to reduce the required power in using the device, as it will not wake up the large cluster if it doesn't need to.
- Load tracking is also an extremely crucial part of EAS, this information is used to decide frequencies and how to delegate tasks across the CPU, and there are two options:
  - "Per-Entity Load Tracking" (PELT)
  - "Window-Assisted Load Tracking" (WALT)
- WALT is more bursty, with high peaks in CPU frequency while PELT tries to remain more consistent.
- The load tracker doesn't actually affect the CPU frequency, it just tells the system what the CPU usage is at the moment.
- A higher CPU usage requires a higher frequency and so a consistent trait of PELT is that it causes the CPU frequency to ramp up or down slowly.

Load Tracking: what and why

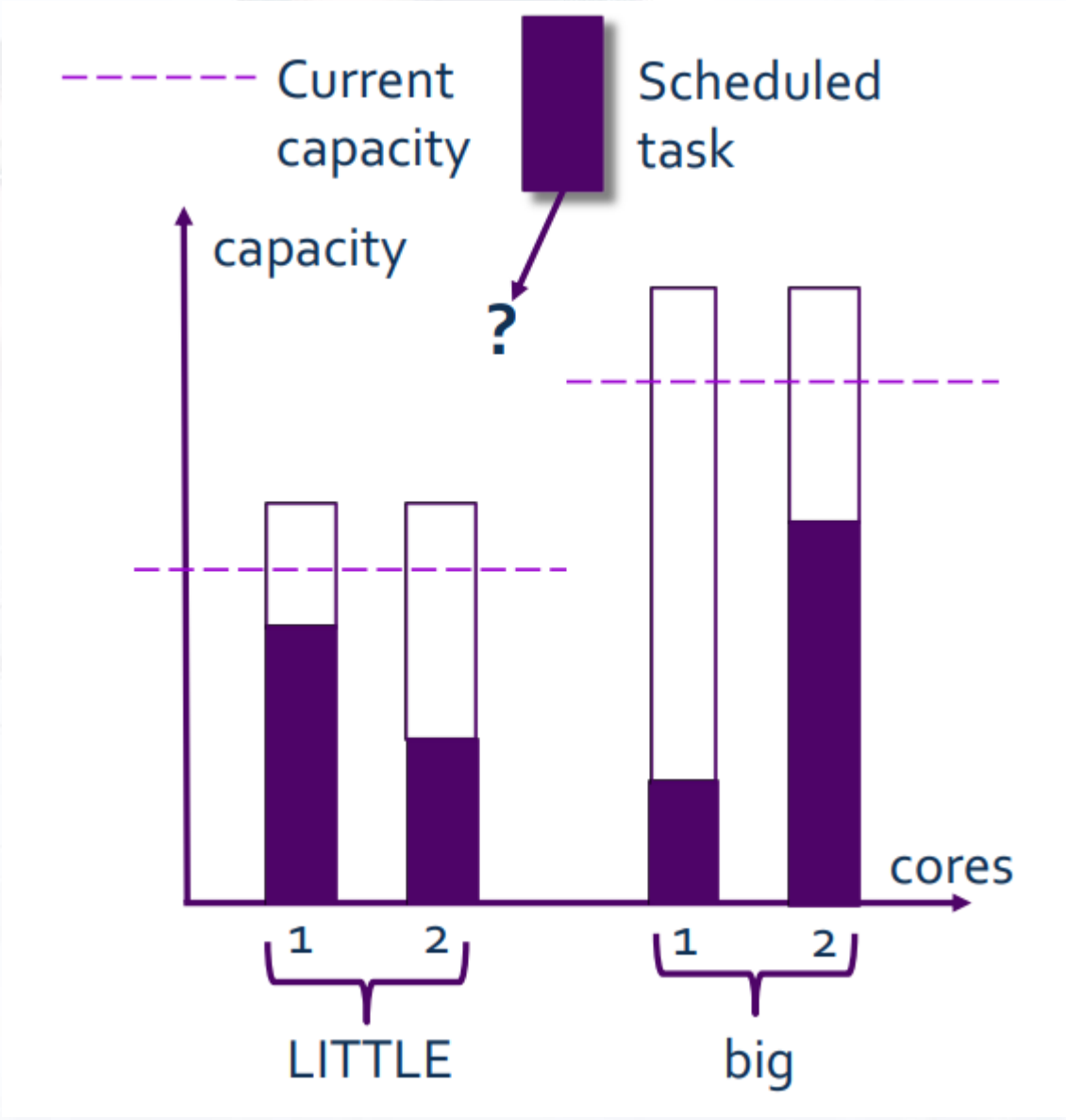
- LT is to track task demand (load) and CPU utilization
  - Classifying a task as "heavy" in mobile world use-cases such as scrolling a UI or browsing a web page where tasks exhibit sporadically heavy load
  - Reclassification of a light task as heavy is also important - for example, a rendering thread may change its demand depending on what is being shown on-screen
- Load balancing
- Task placement
  - Task load and CPU utilization tracking is essential for energy aware scheduling.
  - Heavy tasks can be placed on higher capacity CPUs. Small tasks can be packed on a busy CPU without waking up an idle CPU.
- Frequency guidance
  - CPU frequency governor (schedutil, powersave, ondemand, etc) controls how the CPU raises and lowers its frequency according to the demand.
  - CPU frequency governors like on-demand use a timer and compute the CPU busy time by subtracting the CPU idle time from the timer period.
  - Task migrations are not accounted. If a task execution time is split across two CPUs, governor fails to ramp up the frequency

PELT: Per Entity Load Tracking

- In mainline since kernel 3.8
  - used by mainline CFS
- The main idea is that process can contribute to load even if it is not actually running at the moment
- PELT tracks load on a per-entity (process or task) basis
- Let Li designate the entity's load contribution in period pi
  - Then the total load is L = L0 + L1*q + L2*q^2 + L3*q^3 + ⋯, where q is the decay factor
  - The load is accounted using a decayed geometric series with runnable time in 1 msec period as coefficients.
  - The decay constant is chosen such that the contribution in the 32 msec past is weighted half as strongly as the current contribution.
  - The blocked tasks also contribute to the run-queue load/utilization. The task load is decayed during sleep.

EAS/PELT operation

- Estimate energy
  - E = Pidle * Tidle + Pbusy + Tbusy
- Pick CPU with sufficient spare capacity and smallest energy impact
- Here both LITTLE and big cores have sufficient capacity
- the energy impact is smaller with LITTLE core



Qualcomm HMP scheduler

- Tasks are divided into groups
  - By importance
  - Depending on nice priority
  - By “size”
  - Depending on the calculated load
  - Task may be “big”, ”little” or other
  - Thresholds are parametrized

- Scheduling a task should depend on its properties
  - Task “size” should be defined somehow
  - It’s done basing on task demand calculation

“big” and “small” tasks in HMP

- Small task
  - A periodic task with short execution time
  - Can be easily identified using task average demand

- Big task
  - Task producing high CPU load (parametrized, 90%+)
  - Some heavy tasks HMP doesn’t want to count as big e.g. background threads in Android

- Some tasks are neither big nor small
- Tasks can change their “size” over time

HMP scheduler: task demand

- Task demand Dtask is the contribution of a task's running time to a window
  - Dtask = delta_time*cur_freq /max_possible_freq

  - delta_time - time of task running on a core in a period of time
  - cur_freq - the current frequency of the core this task is running on
  - max_possible_freq is the maximum possible frequency across all cores

- Calculated over N sliding windows (N is a parameter)
  - E. g. the average demand Davg = (D1 + ... + Dn)/N
  - The best result is achieved with D = max{Davg, D1}
  - == Task demand is the maximum of its contribution to the most recently completed window and its average demand over the past N windows.

HMP scheduler: task demand scaling

- We already account for difference in maximum frequency
  - Dtask is calculated in regard to maximum frequency across all cores
- We also need to account for higher performance of big cores
  - Dtask,scaled = Dtask * rq->efficiency / max_possible_efficiency
  - Efficiency is a per-run-queue parameter
  - Usually big cores are considered 2x more effective

EAS/WALT 1 of 2

- WALT: Window Assisted Load Tracking. It retains PELT “per-entity” tracking pattern. It implements N-window demand calculation from QHMP
- WALT keeps track of recent N windows of execution for every task.
- Windows where a task had no activity are ignored and not recorded.
  - Windows exist only when the task is on the run-queue or running. This allows rapid reclassification of the same task as heavy after a short sleep.
  - Thus a task does not need to re-execute to gain its demand once more and can be migrated up to a big CPU immediately.
- Task demand is derived from these N samples. Different policies like max(), avg(), max(recent, avg) are available.

EAS/WALT 2 of 2

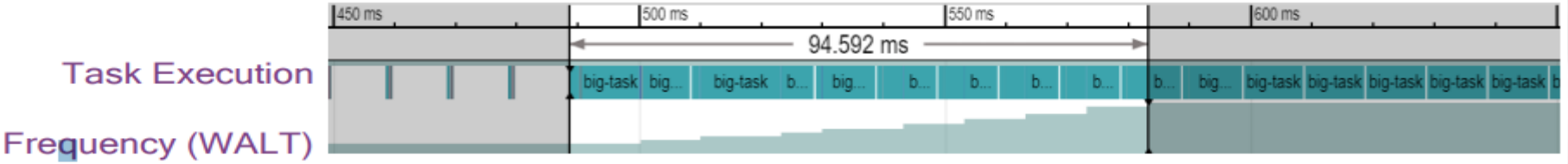
- The wait time of a task on the CPU rq is also accounted towards its demand.
- Task utilization is tracked separately from the demand. The utilization of a task is the execution time in the recently completed window.
- The CPU rq’s utilization is the sum of the utilization of all tasks ran in the recently completed window.
- WALT "forgets" cpu utilization as soon as tasks are taken off of the runqueue, and thus the cpufreq governor can choose to drop frequency after just one window, potentially saving power

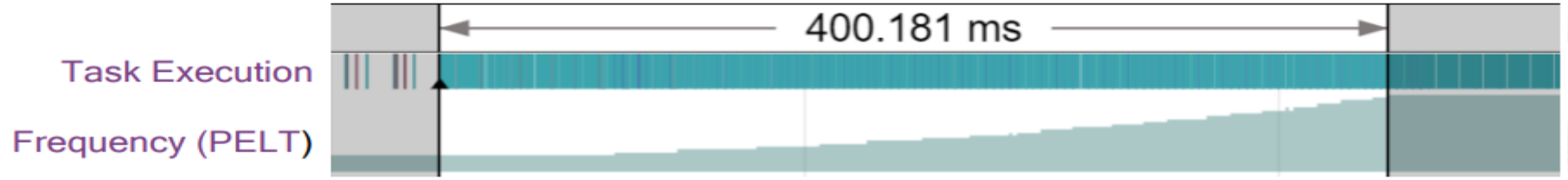
WALT vs PELT 1 of 3

- WALT has more dynamic demand/utilization signal than provided by PELT, which is subjected to the geometric series.
- WALT takes less time and PELT takes more time to detect a heavy task or re-classification of a light task as heavy. Thus the task migration to a higher capacity CPU is delayed. As example, it takes ~138msec for a task with 0 utilization to become a 95% utilization task.
- WALT ignores task when it goes to sleep, PELT decays the utilization of a task when it goes to sleep. As an example, a 100% utilization task would become 10% utilization task just after 100 msec sleep.
- WALT takes less time to ramp frequency up. PELT takes more time to build up the CPU utilization, thus delaying the frequency ramp up.
  - PELT's blocked utilization decay implies that the underlying cpufreq governor would have to delay dropping frequency for longer than necessary

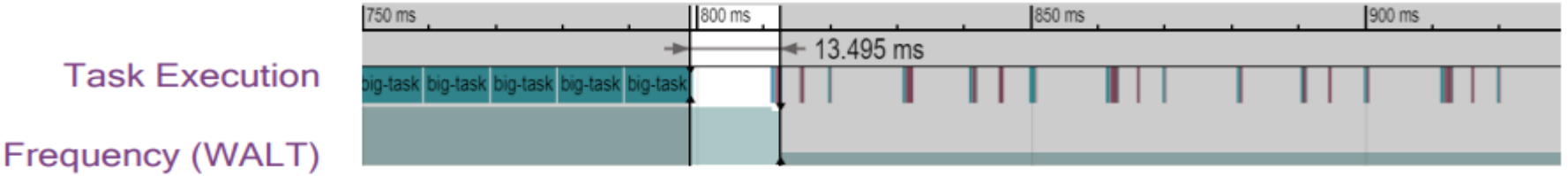
WALT vs PELT 2 of 3

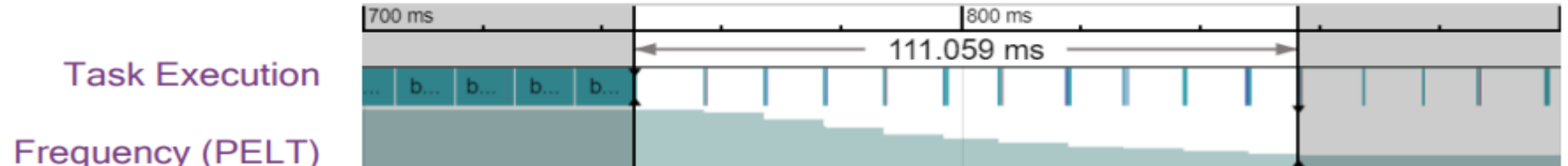
Frequency ramp up





Frequency ramp down





WALT vs PELT 3 of 3

	PELT	WALT
Load tracking	Load is accounted using a geometric series	Load is accounted with a policy that observes past *N* windows
Blocked load/utilization tracking	Load is decayed as part of a runqueue statistic when the task is blocked	Blocked load contribution is removed from runqueue sum/average statistics.
Blocked load restoration	Runqueue statistics include blocked load/utilization at all times	Load contribution is restored to RQ statistics when the task becomes runnable again.

Content

- CPU power management
  - C-states
  - P-states
- Global States and Sleeping states
- OS-controlled P-states

- CPU performance scaling in Linux
- CPUFreq - adjusting CPU frequencies
- CPUFreq scaling governors

- CPU idle time management
- The idle loop
- The idle states
- Idle CPUs and scheduler ticks
- CPUIdle governors

CPU power management

- Two generic ways to manage the power consumption of a processor:
  - powering up/down processor/SoC subsystems
    - remove both dynamic and static currents (sometimes called *power-gating*), or
    - to stop the clock of the core which removes dynamic power consumption only and can be referred to as *clock-gating*.
  - voltage/frequency scaling

- Processor power management technologies are defined in the ACPI specification and are divided into two categories or states:
  - C-states (Core states)
  - P-states (Operating Performance Points, aka OPP)

CPU power management: C-states, P-states

- Different processors support different numbers of C/P-states in which various parts of the CPU are turned off/on.
- C-states describe idle (power saving) states.
- In order to power down a SoC subsystem, that subsystem should not be running anything, so it should be at idle, doing nothing, executing nothing.
- So C-state x, Cx, means one or more subsystems of the CPU is at idle, powered down.
- P-states describe executing (power saving) states.
- The SoC subsystem is actually running but it does not require full performance so the voltage and/or frequency it operates is decreased.
- So P-state x, Px, means the subsystem it refers to (e.g. a CPU core) is operating at a specific (frequency, voltage) pair.

CPU PM: extra ARM terminology

ARM cores typically support several levels of PM:

- **Standby:** the core is left powered-up, but most of its clocks are stopped, or clock-gated. Almost all parts of the core are in a static state and the only power drawn is because of leakage currents and the clocking of the small amount of logic that looks out for the wake-up condition.
- **Retention:** The core state, including the debug settings, is preserved in low-power structures, enabling the core to be at least partially turned off. Changing from low-power retention to running operation does not require a reset of the core.
- **Dormant mode:** the core logic is powered down, but the cache RAMs are left powered up. Often the RAMs are held in a low-power retention state where they hold their contents but are not otherwise functional.
- **Power down:** all data, operating conditions and operating states are lost.
- **Hotplug:** technique that can dynamically switch cores on or off. OS PM has to issue an explicit command to bring a core back online, that is, hotplug a core.

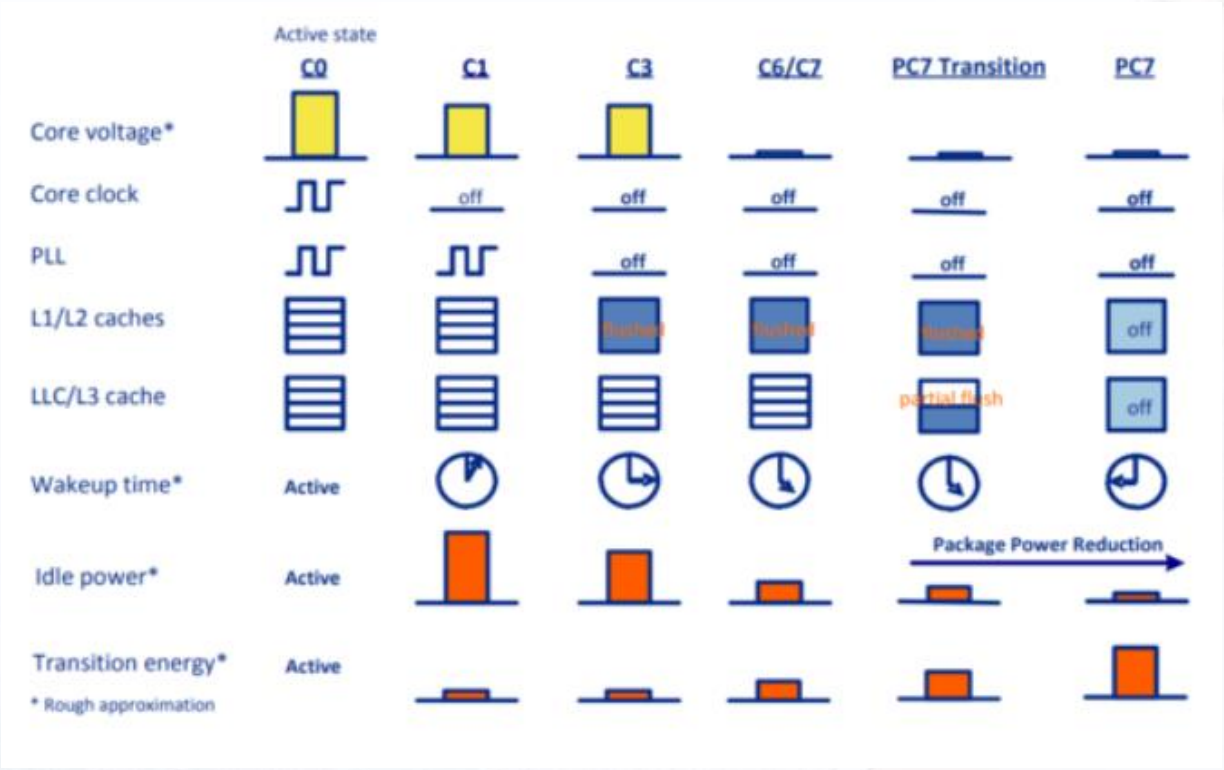
C-states, P-states and CC-states, PC-states

- The states are numbered starting from zero like C0, C1… and P0, P1…
  - The higher the number is the more power is saved.
  - C0 means no power saving by shutting down something, so everything is powered on.
  - P0 means maximum performance, thus maximum frequency, voltage and power used.
- Because most modern CPUs have multiple cores in a single package, C-states are further divided into core C-states (CC-states) and package C-states (PC-states).
- The reason for PC-states is there are other (shared) components in the processor that can also be powered down after all cores using them are powered down (e.g. the shared cache).
- However, as a user or programmer, we cannot control these, since we do not interact with the package directly, but we interact with the individual cores.
- So we can only affect the CC-states directly, PC-states are indirectly affected based on the CC-states of the cores.

Basic C-states (defined by ACPI)

- C0: Active, CPU/Core is executing instructions.
  - P-states are relevant here, CPU/Core may be operating at its maximum performance (thus at P0) or at a lower performance/power (thus at anything other than P0).
- C1: Halt, nothing is being executed, but it can return to C0 instantaneously
  - Since it is not working (but halted), P-states are not relevant for C1 or any Cx other than C0.
- C2: Stop-Clock, similar to C1 but it takes longer time to go back to C0.
- C3: Sleep. It can go back to C0, but it will take considerably longer time.

C-states, sub-systems and energy (Intel)



- Basic timeline of power saving using C-states is:

- The normal operation is at C0.
- First, the clocks of idle core is stopped. (C1)
- Then, the local caches (L1/L2) of the core is flushed and the core is powered down. (C3)
- Then, when all the cores are powered down, the shared cache (L3/LLC) of the package is flushed and at the end the package/whole CPU can be (almost) powered down.
  - almost… because there has to be something powered on to return back to C0.

ACPI Global (G) States and Sleeping (S) States

- GO/S0: Computer is running, not sleeping.

- G1: Sleeping
- G1/S1: Power on Suspend. The system state is preserved, so CPU and CPU Caches are still powered.

- G1/S2: CPU is powered off.
  - So CPU and CPU Caches are lost.
- G1/S3: Standby, Sleep or Suspend to RAM (STR).
  - System RAM remains powered.
- G1/S4: Hibernation or Suspend to Disk.
  - Everything is saved to non-volatile memory (like disk), so RAM and possibly all the system is powered off.

- G2/S5: Soft Off. Like mechanical off but things that can wake up the computer from power save is minimally powered.
  - No state is saved, so a reboot is needed to go back to G0.

- G3: Mechanical Off. The Power Supply Unit is disconnected (e.g. via a power switch).
  - Only the things like real-time clock (RTC) is running because they have their own small batteries.
  - No state is saved, so a reboot is needed to go back to G0.

What triggers a CPU core to enter Cx-state

- C0 is entered at boot, when an interrupt happens or a write event occurs on the address monitored by an MWAIT (WFI, WFE in ARM, use DSB to sync data) instruction.

- C1is entered with HLT or MWAIT instructions.

- C3 is entered with MWAIT instruction, then L1 and L2 caches are flushed to LLC and all core clocks are stopped.
  - However, the core maintains its state since it is still powered.

- C6 is entered with MWAIT instruction, then the processor state is saved into a dedicated SRAM and the voltage to core is reduced to zero.
  - At this state, core has no power.
  - When exiting C6, the processor state is restored back from the SRAM.

- C7 and C8 are same as C6 for the core.

OS-controlled P-states

- OS is aware of the P-states, a specific P-state is requested by the OS.
- This means selecting an operating frequency, whereas the voltage is automatically selected by the processor depending on the frequency and other factors.
- After a P-state is requested by writing it to a model specific register, the voltage is transitioned to an automatically calculated value and the clock generator (PLL) locks to the frequency requested.
- All cores in cluster (big.LITTLE) share the same P-state, so it is not possible to set this individually for a core.
- The current P-state / operating point can be read from another model specific register

What is not controlled (by OS, C/P/G/S-states)?

- Intel AMT and ME (there are similar, less known, technologies in AMD and ARM)

- Intel Active Management Technology (AMT): "hardware and firmware technology for remote out-of-band management of personal computers". AMT has many features which includes power management.

- Intel Management Engine (ME) is the hardware part: an isolated and protected coprocessor, embedded as a non-optional part in all Intel chipsets since 2008.
  - The coprocessor is a special 32-bit ARC microprocessor (RISC architecture) that's physically located inside the PCH chipset (or MCH on older chipsets).

- So, Intel x86s hide another CPU that can take over your machine (you can't audit it)
  - Vulnerabilities, backdoors (FBI, CIA, etc)
  - Can be switched off as discovered by Positive Technologies (Russian based company)

CPU performance scaling in Linux 1 of 2

- OS estimates the required CPU capacity, and decides which P-states to put the CPUs into.
- The activity (is NOT scheduling) by which this happens is referred to as CPU performance scaling or CPU frequency scaling (CPUFreq) - because it involves adjusting the CPU clock frequency
- The Linux kernel supports CPU performance scaling by means of the CPUFreq subsystem that consists of three layers of code:
  - core layer;
  - scaling governors;
  - scaling drivers.
- The CPUFreq core provides the common code infrastructure and user space interfaces for all platforms that support CPU performance scaling.
- Scaling governors implement algorithms to estimate the required CPU capacity. As a rule, each governor implements one, possibly parametrized, scaling algorithm.
- Scaling drivers talk to the hardware. They provide scaling governors with information on the available P-states (or P-state ranges in some cases) and access platform-specific hardware interfaces to change CPU P-states as requested by scaling governors.

CPU performance scaling in Linux 2 of 2

- Generally all available scaling governors can be used with every scaling driver.
- That design is based on the observation: information used by performance scaling algorithms for P-state selection can be represented in a platform-independent form in the majority of cases.
- So it should be possible to use the same performance scaling algorithm implemented in exactly the same way regardless of which scaling driver is used.
- Consequently, the same set of scaling governors should be suitable for every supported platform.
- However, that observation may not hold for performance scaling algorithms based on information provided by the hardware itself, for example through feedback registers.
- That information is typically specific to the hardware interface it comes from and may not be easily represented in an abstract, platform-independent way.
- For this reason, CPUFreq allows scaling drivers to bypass the governor layer and implement their own performance scaling algorithms.

CPUFreq policy objects

- In some cases the hardware interface for P-state control is shared by multiple CPUs.
- For example, the same register (or set of registers) is used to control the P-state of multiple CPUs at the same time and writing to it affects all of those CPUs simultaneously.
- Sets of CPUs sharing hardware P-state control interfaces are represented by CPUFreq as ***struct cpufreq_policy*** objects.
- For consistency, ***struct cpufreq_policy*** is also used when there is only one CPU in the given set.
- The CPUFreq core layer maintains a pointer to a ***struct cpufreq_policy*** object for every CPU in the system, including CPUs that are currently offline.
- If multiple CPUs share the same hardware P-state control interface, all of the pointers corresponding to them point to the same ***struct cpufreq_policy*** object.
- CPUFreq uses struct cpufreq_policy as its basic data type and the design of its user space interface is based on the policy concept.

CPUFreq policy CLI interface

- During the initialization of the kernel, the CPUFreq core layer creates a sysfs directory (kobject) called cpufreq under /sys/devices/system/cpu/.
- That directory contains a policyX subdirectory (where X represents an integer number) for every policy object maintained by the CPUFreq core.
- Each policyX directory is pointed to by cpufreq symbolic links under /sys/devices/system/cpu/cpuY/ (where Y represents an integer that may be different from the one represented by X) for all of the CPUs associated with (or belonging to) the given policy.
- The policyX directories in /sys/devices/system/cpu/cpufreq each contain policy-specific attributes (files) to control CPUFreq behavior for the corresponding policy objects (that is, for all of the CPUs associated with them).

CPUFreq scaling governors 1 of 2

- CPUFreq provides generic scaling governors that can be used with all scaling drivers. Each governor implements a single, possibly parametrized, performance scaling algorithm.

- Scaling governors are attached to policy objects and different policy objects can be handled by different scaling governors at the same time (although that may lead to suboptimal results in some cases).

- The scaling governor for a given policy object can be changed at any time with the help of the scaling_governor policy attribute in sysfs.

- Some governors expose sysfs attributes to control or fine-tune the scaling algorithms implemented by them.
- Those attributes, referred to as governor tunables, can be either global (system-wide) or per-policy, depending on the scaling driver in use.
- If the driver requires governor tunables to be per-policy, they are located in a subdirectory of each policy directory.
- Otherwise, they are located in a subdirectory under /sys/devices/system/cpu/cpufreq/. In either case the name of the subdirectory containing the governor tunables is the name of the governor providing them.

CPUFreq scaling governors 2 of 2

- performance
  - when attached to a policy object, this governor causes the highest frequency, within the scaling_max_freq policy limit, to be requested for that policy.
  - the request is made once at that time the governor for the policy is set to performance and whenever the scaling_max_freq or scaling_min_freq policy limits change after that.

- powersave
  - when attached to a policy object, this governor causes the lowest frequency, within the scaling_min_freq policy limit, to be requested for that policy.
  - the request is made once at that time the governor for the policy is set to powersave and whenever the scaling_max_freq or scaling_min_freq policy limits change after that.

- userspace
  - does not do anything by itself. Instead, it allows user space to set the CPU frequency for the policy it is attached to by writing to the scaling_setspeed attribute of that policy.

- ondemand
  - uses CPU load as a CPU frequency selection metric.
  - In order to estimate the current CPU load, it measures the time elapsed between consecutive invocations of its worker routine and computes the fraction of that time in which the given CPU was not idle.
  - The ratio of the non-idle (active) time to the total CPU time is taken as an estimate of the load.

- schedutil
  - uses CPU utilization data available from the CPU scheduler. It generally is regarded as a part of the CPU scheduler, so it can access the scheduler’s internal data structures directly.

CPUFreq scaling governor schedutil

- It runs entirely in scheduler context, although in some cases it may need to invoke the scaling driver asynchronously when it decides that the CPU frequency should be changed for a given policy
  - that depends on whether or not the driver is capable of changing the CPU frequency from scheduler context
- The actions of this governor for a particular CPU depend on the scheduling class invoking its utilization update callback for that CPU.
- if it is invoked by the CFS/EAS, the governor will use the PELT/ WALT metric for the root control group of the given CPU as the CPU utilization estimate.
- Then, the new CPU frequency to apply is computed in accordance with the formula
  - f = 1.25 * f_0 * util / max
  - where util is the PELT/WALT number, and
  - max is the theoretical maximum of util, and f_0 is either the maximum possible CPU frequency for the given policy, or the current CPU frequency (otherwise).
- This governor also employs a mechanism allowing it to temporarily bump up the CPU frequency for tasks that have been waiting on I/O most recently, called “IO-wait boosting”.
- That happens when the SCHED_CPUFREQ_IOWAIT flag is passed by the scheduler to the governor callback which causes the frequency to go up to the allowed maximum immediately and then draw back to the value returned by the above formula over time.

CPU idle time management 1 of 2

- In CPU idle states the execution of a program is suspended and instructions belonging to it are not fetched from memory or executed
- CPU idle time management operates on CPUs as seen by the *CPU scheduler* (that is the part of the kernel responsible for the distribution of computational work in the system).
- In its view, CPUs are *logical*. That is, they need not be separate physical entities and may just be interfaces appearing to software as individual single-core processors.
- In other words, a CPU is an entity which appears to be fetching instructions that belong to one program from memory and executing them, but it need not work this way physically.

CPU idle time management 2 of 2

Generally, three different cases of logical CPUs:

1. If the whole processor can only follow one program at a time, it is a CPU.
   a) In that case, if the hardware is asked to enter an idle state, that applies to the processor as a whole.
2. If the processor is multi-core, each core in it is able to follow at least one program at a time.
   a) The cores need not be entirely independent of each other (they may share caches), but still most of the time they work physically in parallel with each other, so if each of them executes only one program, those programs run mostly independently of each other at the same time.
   b) The entire cores are CPUs in that case and if the hardware is asked to enter an idle state, that applies to the core that asked for it in the first place, but it also may apply to a cluster that the core belongs to
3. Each core in a multi-core processor may be able to follow more than one program in the same time frame
   a) Each core may be able to fetch instructions from multiple locations in memory and execute them in the same time frame, but not necessarily entirely in parallel with each other.
   b) In that case the cores present themselves to software as “bundles” each consisting of multiple individual single-core “processors”, referred to as hardware threads, that each can follow one sequence of instructions.
   c) Then, the hardware threads are CPUs from the CPU idle time management perspective and if the processor is asked to enter an idle state by one of them, the hardware thread (or CPU) that asked for it is stopped, but nothing more happens, unless all of the other hardware threads within the same core also have asked the processor to enter an idle state.
   d) In that situation, the core may be put into an idle state individually or a cluster containing it may be put into an idle state as a whole (if the other cores within the cluster are in idle states already).

The Idle loop

- CPU idle time management subsystem called CPUIdle
- The idle loop code takes two major steps in every iteration of it.
  - First, it calls into a code module referred to as **the governor** that belongs to the CPUIdle to select an idle state for the CPU to ask the hardware to enter.
  - Second, it invokes another code module from the CPUIdle subsystem, called **the driver**, to actually ask the processor hardware to enter the idle state selected by the governor.
- The role of the governor is to find an idle state most suitable for the conditions at hand.
- For this purpose, idle states that the hardware can be asked to enter by logical CPUs are represented in an abstract way independent of the platform or the processor architecture and organized in a linear array.
- That array has to be prepared and supplied by the CPUIdle driver matching the platform the kernel is running on at the initialization time.
- This allows CPUIdle governors to be independent of the underlying hardware and to work with any platforms that the Linux kernel can run on.

The Idle states

- Each idle state present in the array is characterized by two parameters to be taken into account by the governor, the target residency and the (worst-case) exit latency.
  - The target residency is the minimum time the hardware must spend in the given state, including the time needed to enter it (which may be substantial), in order to save more energy than it would save by entering one of the shallower idle states instead.
    - The “depth” of an idle state roughly corresponds to the power drawn by the processor in that state.
  - The exit latency, in turn, is the maximum time it will take a CPU asking the processor hardware to enter an idle state to start executing the first instruction after a wakeup from that state.
- In general the exit latency also must cover the time needed to enter the given state in case the wakeup occurs when the hardware is entering it and it must be entered completely to be exited in an ordered manner.

Idle CPUs and scheduler ticks

- The scheduler tick is a timer that triggers periodically in order to implement the time sharing strategy of the CPU scheduler.
- The currently running task may not want to give the CPU away voluntarily, and the scheduler tick is there to make the switch happen regardless.

- If the tick is allowed to trigger on idle CPUs, it will not make sense for them to ask the hardware to enter idle states with target residencies above the tick period length.
  - Since the time of an idle CPU need not be shared between multiple runnable tasks, the primary reason for using the tick goes away if the given CPU is idle.
  - Consequently, it is possible to stop the scheduler tick entirely on idle CPUs in principle, even though that may not always be worth the effort.

- If the tick is stopped and the wakeup does not occur any time soon, the hardware may spend indefinite amount of time in the shallow idle state selected by the governor, which will be a waste of energy.

- Hence, if the governor is expecting a wakeup of any kind within the tick range, it is better to allow the tick trigger

- The systems that run kernels configured to allow the scheduler tick to be stopped on idle CPUs are referred to as tickless systems and they are generally regarded as more energy-efficient than the systems running kernels in which the tick cannot be stopped.

- If the given system is tickless, it will use the menu governor by default and if it is not tickless, the default CPUIdle governor on it will be ladder.

CPUIdle governor decisions

- There are two types of information that can influence the governor's decisions.
  - The governor knows exactly the time until the closest timer event, because
    - the kernel programs timers and it knows exactly when they will trigger, and
    - it is the maximum time the hardware that the given CPU depends on can spend in an idle state, including the time necessary to enter and exit it.
  - However, the CPU may be woken up by a non-timer event at any time (in particular, before the closest timer triggers) and it generally is not known when that may happen.
  - The governor can only see how much time the CPU actually was idle after it has been woken up (that time will be referred to as the idle duration from now on) and it can use that information somehow along with the time until the closest timer to estimate the idle duration in future.
  - How the governor uses that information depends on what algorithm is implemented by it and that is the primary reason for having more than one governor in the CPUIdle subsystem.
  - There are four CPUIdle governors available, menu, TEO, ladder and haltpoll.

CPUIdle governors

- menu: is the default CPUIdle governor for tickless systems. When invoked to select an idle state for a CPU (i.e. an idle state that the CPU will ask the processor hardware to enter), it attempts to predict the idle duration and uses the predicted value for idle state selection.

- TEO (timer events oriented): is an alternative CPUIdle governor for tickless systems

- ladder and haltpoll: for systems with ticks enabled for CPU idle (not interesting for us)