# 一 cpuidle framework—概述和软件架构

Linux kernel上，Linux系统中，CPU被两类程序占用：一类是进程（或线程），也称进程上下文；另一类是各种中断、异常的处理程序，也称中断上下文。进程的存在，是用来处理事务的，如读取用户输入并显示在屏幕上。而事务总有处理完的时候，如用户不再输入，也没有新的内容需要在屏幕上显示。此时这个进程就 可以让出CPU，但会随时准备回来（如用户突然有按键动作）。同理，如果系统没有中断、异常事件，CPU就不会花时间在中断上下文。 可以深入理解idle进程的创建原理。整体讲解使用的kernel版本是4.4.83。

在Linux kernel中，这种CPU的无所事事的状态，被称作idle状态，而cpuidle framework，就是为了管理这种状态。idle处理的两个重点：

1）idle进程

Idle进程的存在，是为了解决“when idle”的问题。

我们知道，Linux系统运行的基础是进程调度，而所有进程都不再运行时，称作cpu idle。但是，怎么判断这种状态呢？kernel采用了一个比较简单的方法：在init进程（系统的第一个进程）完成初始化任务之后，将其转变为 idle进程，由于该进程的优先级是最低的，所以当idle进程被调度到时，则说明系统的其它进程不再运行了，也即CPU idle了。最终，由idle进程调用idle指令（这里为WFI），让CPU进入idle状态。 调度器如何调度idle进程的，可以看sched_class,划分为不同的调度类，目前存在的调度类如下：

```
kernel/sched/sched.h:1254:extern const struct sched_class stop_sched_class;
kernel/sched/sched.h:1255:extern const struct sched_class dl_sched_class;
kernel/sched/sched.h:1256:extern const struct sched_class rt_sched_class;
kernel/sched/sched.h:1257:extern const struct sched_class fair_sched_class;
kernel/sched/sched.h:1258:extern const struct sched_class idle_sched_class;
```

具体分析如下：

```
/*我们知道，cpu idle thread创建和分类调度类的方式：
rest_init---->init_idle_bootup_task()-----idle->sched_class =
idle_sched_class

其中在sched_init里面也有sched_init----->init_idle(),为其他cpu创建idle thread，并为每个idle thread元素sched_class = idle_sched_class

系统在什么情况下执行idle thread？*/
#define sched_class_highest (&stop_sched_class)
#define for_each_class(class) \
   for (class = sched_class_highest; class; class = class->next)

extern const struct sched_class stop_sched_class;
extern const struct sched_class dl_sched_class;
extern const struct sched_class rt_sched_class;
extern const struct sched_class fair_sched_class;
extern const struct sched_class idle_sched_class;

const struct sched_class stop_sched_class = {
    .next           = &dl_sched_class,
    ...........
}
const struct sched_class dl_sched_class = {
    .next           = &rt_sched_class,
    ...........
}
```

```
const struct sched_class rt_sched_class = {
    .next           = &fair_sched_class,
    ...........
}
const struct sched_class fair_sched_class = {
    .next           = &idle_sched_class,
    ...........
}
/*
 * Simple, special scheduling class for the per-CPU idle tasks:
 */
const struct sched_class idle_sched_class = {
    /* .next is NULL */
    /* no enqueue/yield_task for idle tasks */

    /* dequeue is not valid, we print a debug message there: */
    .dequeue_task       = dequeue_task_idle,
    .check_preempt_curr = check_preempt_curr_idle,
    .pick_next_task     = pick_next_task_idle,
    ...........
}
static struct task_struct *
pick_next_task_idle(struct rq *rq, struct task_struct *prev)
{
    put_prev_task(rq, prev);

    schedstat_inc(rq, sched_goidle);
    return rq->idle;
}
/*这idle进程在启动start_kernel函数的时候调用init_idle函数的时候，把当前进程（0号进程）置为每个rq运行队列的的idle上。
rq->curr = rq->idle = idle; 在init_idle函数里面为每个从cpu创建idle thread并赋值给每个rq里面的curr元素
这里idle就是调用start_kernel函数的进程，是0号进程。*/
```

对于WFI指令，WFI Wakeup events会把CPU从WFI状态唤醒，通常情况下，这些events是一些中断事件，因此CPU唤醒后会执行中断handler，在handler中会wakeup某些进程，在handler返回的时候进行调度，当没有其他进程需要调度执行的时候，调度器会恢复idle进程的执行，当然，idle进程不做什么，继续进入idle状态，等待下一次的wakeup。

2）WFI

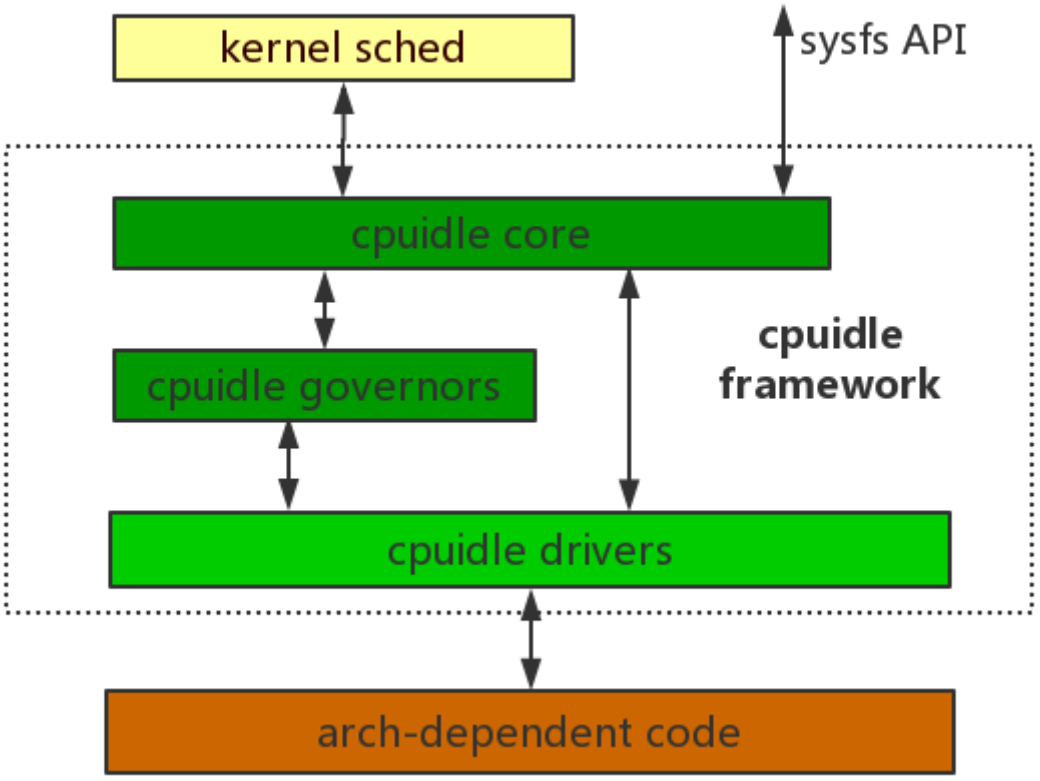
WFI用于解决“how idle”的问题。

一般情况下，ARM CPU idle时，可以使用WFI指令，把CPU置为Wait for interrupt状态。该状态下，至少（和具体ARM core的实现有关）会把ARM core的clock关闭，以节省功耗。但是随着ARM架构的复杂度越来越高，对cpuide的要求等级也不尽相同，对省电的要求也越来越苛刻，因而很多CPU会从“退出时的延迟”和“idle状态下的功耗”两个方面考虑，设计多种idle级别。对延迟较敏感的场合，可以使用低延迟、高功耗的idle；对延迟不敏感的场合，可以使用高延迟、低功耗的idle。由于软件需要根据场景，在恰当的时候，选择一个idle状态，而如何选择则是cpuidle framework需要做的。

**1. 软件架构：**

Linux kernel中，cpuidle framework位于“drivers/cpuidle”文件夹中，包含cpuidle core、cpuidle governors和cpuidle drivers三个模块，再结合位于kernel sched（kernel/sched/idle.c

）中的cpuidle entry，共同完成cpu的idle管理。软件架构如下图




1）kernel schedule模块

位于kernel\sched\idle.c中，负责实现idle线程的通用入口（cpuidle_startup_entry）逻辑，包括idle模式的选择、idle的进入等等。

2）cpuidle core

cpuidle core负责实现cpuidle framework的整体框架，主要功能包括：

根据cpuidle的应用场景，抽象出cpuidle device、cpuidle driver、cpuidle governor三个实体；以函数调用的形式，向上层sched模块提供接口；以sysfs的形式，向用户空间提供接口；向层的cpuidle drivers模块，提供统一的driver注册和管理接口；向下层的governors模块，提供统一的governor注册和管理接口。cpuidle core的代码主要包括：cpuidle.c、driver.c、governor.c、sysfs.c。

3）cpuidle drivers

负责idle机制的实现，即：如何进入idle状态，什么条件下会退出，等等。不同的architecture、不同的CPU core，会有不同的cpuidle driver，平台驱动的开发者，可以在cpuidle core提供的框架之下，开发自己的cpuidle driver。代码主要包括：cpuidle-xxx.c，吧比如cpuidle-arm64.c(cpuidle-sprd.c)。

4）cpuidle governors

我们知道Linux kernel的framework有两种比较固定的抽象模式：

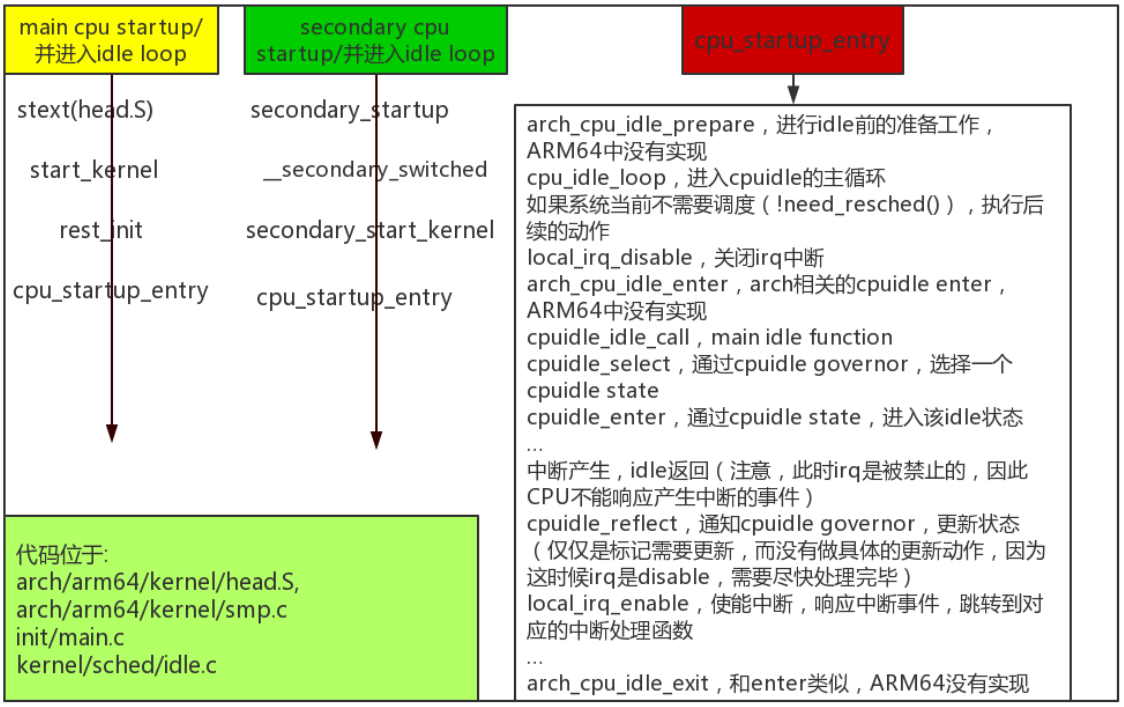
**模式1，provider/consumer模式**：interrupt、clock、timer、regulator等大多数的framework是这种模式。它的特点是，这个硬件模块是为其它一个或多个模块服务的，因而framework需要从对上（consumer）和对下 （provider）两个角度进行软件抽象。

**模式2，driver/governor模式**：本文所描述的cpuidle framework即是这种模式。它的特点是：硬件（或者该硬件所对应的驱动软件）可以提供多种可选“方案”（这里即idle level），“方案”的实现（即机制），由driver负责，但是到底选择哪一种“方案”（即策略），则由另一个模块负责（即这里所说的 governor）。同时cpufreq也是这种模式。

模式2的解释可能有点抽象，把它放到cpuidle的场景里面，就很容易理解了：前面讲过，很多CPU提供了多种idle级别（即上面所说的“方案”），这些idle 级别的主要区别是“idle时的功耗”和“退出时延迟”。cpuidle driver（机制）负责定义这些idle状态（每一个状态的功耗和延迟分别是多少），并实现进入和退出相关的操作。最终，cpuidle driver会把这些信息告诉governor，由governor根据具体的应用场景，决定要选用哪种idle状态（策略）。kernel中，cpuidle governor都位于governors/目录下。

## 2. 软件流程

前面提到过，kernel会在系统启动完成后，在init进程（或线程）中，处理cpuidle相关的事情。大致的过程是这样的：首先需要说明的是，在SMP（多核）系统中，CPU启动的过程是：



# 二、cpuidle framework_cpuidle core架构

## 1. 前言

cpuidle core是cpuidle framework的核心模块，负责抽象出cpuidle device、cpuidle driver和cpuidle governor三个实体，并提供如下功能：

1）向底层的cpuidle driver模块提供cpudile device和cpuidle driver的注册/注销接口。
2）向cpuidle governors提供governor的注册接口。
3）提供全局的cpuidle机制的开、关、暂停、恢复等功能。
4）向用户空间程序提供governor选择的接口。
5）向kernel sched中的cpuidle entry提供cpuidle的级别选择、进入等接口，以方便调用。

根据上面的内容，分析cpuidle framework的实现思路和实现原理。

## 2.主要数据结构

cpuidle core抽象出了cpuidle device、cpuidle driver和cpuidle governor三个数据结构，但是cpuidle device是虚拟出来的设备，意义是什么？

### 2.1 cpuidle_state

在第一章中提到过，cpuidle framework提出的主要背景是，很多复杂的CPU，有多种不同的idle级别。这些idle级别有不同的功耗和延迟，根据场景来决定哪种idle level。 Linux kernel使用struct cpuidle_state结构抽象idle级别（后面将会统称为idle state），如下：

```
struct cpuidle_state {
    char        name[CPUIDLE_NAME_LEN];
    char        desc[CPUIDLE_DESC_LEN];
    unsigned int    flags;
    unsigned int    exit_latency; /* in US */
    int     power_usage; /* in mW */
    unsigned int    target_residency; /* in US */
    bool        disabled; /* disabled on all CPUs */
    int (*enter)    (struct cpuidle_device *dev,
            struct cpuidle_driver *drv,
            int index);
    int (*enter_dead) (struct cpuidle_device *dev, int index);
    /*
     * CPUs execute ->enter_freeze with the local tick or entire timekeeping
     * suspended, so it must not re-enable interrupts at any point (even
     * temporarily) or attempt to change states of clock event devices.
     */
    void (*enter_freeze) (struct cpuidle_device *dev,
                  struct cpuidle_driver *drv,
                  int index);
};
/*name、desc：该idle state的名称和简介；
exit_latency：CPU从该idle state下返回运行状态的延迟，单位为us。它决定了CPU在idle状态和run状态之间切换的效率，如果延迟过大，将会影响系统性能；
power_usage：CPU在该idle state下的功耗，单位为mW；
target_residency：期望的停留时间，单位为us。进入和退出idle state是需要消耗额外的能量的，如果在idle状态停留的时间过短，节省的功耗少于额外的消耗，则得不偿失。governor会根据该字段，结合当前 的系统情况（如可以idle多久），选择idle level；
disabled：表示该idle state在所有CPU上都不可用；
```

```
flags：idle state的特性标志，当前支持如下三个：
        CPUIDLE_FLAG_TIME_VALID，表明CPU停留于该idle state下的时间是可测量的，具体的使用场景，会在介绍governor时详细说明；
        CPUIDLE_FLAG_COUPLED，表明该idle state会同时在多个CPU上起作用，软件 需要特殊处理；
        CPUIDLE_FLAG_TIMER_STOP，表明在该idle state下，timer会停止；
enter：进入该state的回调函数；
enter_dead：CPU长时间不需要工作时（称作offline），可调用该回调函数。
enter_freeze：按照字面意思，应该是执行最后一个级别的idle level，cpu才会回调此函数，但是目前在我们平台上面没有看到对这个回调函数的实现！*/
```

总结说来，cpuidle state的功能有两个，一个是描述该idle state的特性，二是提供进入该idle state的具体方法。

### 2.2 cpuidle_device

由于cpuidle device是一个虚拟设备，我们可以把它类比为“cpu idle controller”，负责实现cpuidle相关的逻辑。在多核CPU中，每个CPU core，都会对应一个cpuidle device。 kernel使用struct cpuidle_device抽象cpuidle device，如下：

```
struct cpuidle_device {
    unsigned int        registered:1;
    unsigned int        enabled:1;
    unsigned int        cpu;
    int         last_residency;
    struct cpuidle_state_usage  states_usage[CPUIDLE_STATE_MAX];
    struct cpuidle_state_kobj *kobjs[CPUIDLE_STATE_MAX];
    struct cpuidle_driver_kobj *kobj_driver;
    struct cpuidle_device_kobj *kobj_dev;
    struct list_head    device_list;
#ifdef CONFIG_ARCH_NEEDS_CPU_IDLE_COUPLED
    cpumask_t       coupled_cpus;
    struct cpuidle_coupled  *coupled;
#endif
};
/*registered：表示设备是否已经注册到kernel中；
enabled：表示设备是否已经使能；
cpu：该cpuidle device所对应的cpu number；
last_residency：该设备上一次停留在idle状态的时间，单位为us；
state_count：该设备具备的idle state（对应struct cpuidle_state结构）的个数；
states_usage：一个struct cpuidle_state_usage类型的数组，记录了该设备的每个idle state的统计信息，包括是否使能（enable）、设备进入该state的次数（usage）和设备停留在该state的总时间（time，单位为 us）；
kobjs、kobj_driver、kobj_dev：用于组织sysfs；
device_list：用于将该设备添加到一个全局的链表中（cpuidle_detected_devices）；*/
```

由上面的描述可知，cpuidle device和普通常见的device不同，struct cpuidle_device中保存的信息，都是运行时动态创建的信息，不需要driver提供任何额外信息。所以后面会看到每个cpu都可以有自己的cpuidle_device的结构体，可以单独维护自己的idle信息。

### 2.3 cpuidle_driver

cpuidle core使用struct cpuidle_driver抽象cpuidle驱动，如下：

```
struct cpuidle_driver {
    const char      *name;
    struct module       *owner;
    int                     refcnt;
```

```
        /* used by the cpuidle framework to setup the broadcast timer */
    unsigned int            bctimer:1;
    /* states array must be ordered in decreasing power consumption */
    struct cpuidle_state    states[CPUIDLE_STATE_MAX];
    int         state_count;
    int         safe_state_index;
    /* the driver handles the cpus in cpumask */
    struct cpumask      *cpumask;
};
/*bctimer：一个标志，用于指示在cpuidle driver注册和注销时，是否需要设置一个broadcast timer，有关broadcast timer后面再详细介绍；
states、state_count：该driver支持的cpuidle state及其个数。由于struct cpuidle_device中包含了某个state在该设备上是否enable的信息，这里的state应该是所有cpuidle device所支持的state的公倍数。另外，上面的注释很明确，这些states，需要以功耗大小的降序排列；
cpumask：一个struct cpumask结构的bit map指针，用于说明该driver支持哪些cpu core。*/
```

struct cpuidle_driver结构比较简单，由此可知编写cpuidle driver的主要工作量，是定义所支持的cpuidle state，以及state的enter接口。

### 2.4 cpuidle_governor

cpuidle core使用struct cpuidle_governor结构抽象cpuidle governor，如下：

```
struct cpuidle_governor {
    char            name[CPUIDLE_NAME_LEN];
    struct list_head    governor_list;
    unsigned int        rating;
    int   (*enable)     (struct cpuidle_driver *drv,
                    struct cpuidle_device *dev);
    void  (*disable)    (struct cpuidle_driver *drv,
                    struct cpuidle_device *dev);
    int   (*select)     (struct cpuidle_driver *drv,
                    struct cpuidle_device *dev);
    void  (*reflect)    (struct cpuidle_device *dev, int index);
    struct module       *owner;
};
/*name：该governor的名称；
governor_list：用于将该governor添加到一个全局的governors列表中（cpuidle_governors），由此可见系统允许存在多个governor；
rating：governor的级别，正常情况下，kernel会选择系统中rating值最大的governor作为当前governor，这个在governor切换的时候有体现；
enable/disable：governor的enable/disable回调函数，一般会在enable中进行一些初始化操作，在disable中进行反操作；
select：根据当前系统的运行状况，以及各个idle state的特性，选择一个state（即决策）；
reflect：通过该回调函数，可以告知governor，系统上一次所处的idle state是哪个，并update对于的idle info，并尽可能的快速处理，这个函数会增加退出时延。*/
```

## 3. 功能说明

### 3.1 cpuidle_device管理

cpuidle_device管理主要负责cpuidle device注册/注销、使能/禁止，位于drivers/cpuidle/cpuidle.c中，由下面的四个接口实现：

```
extern int cpuidle_register_device(struct cpuidle_device *dev);
//初始化struct cpuidle_device变量
extern void cpuidle_unregister_device(struct cpuidle_device *dev);
```

```
extern int cpuidle_enable_device(struct cpuidle_device *dev);
extern void cpuidle_disable_device(struct cpuidle_device *dev);
```

1）cpuidle_register_device

该接口的内部动作如下：调用__cpuidle_device_init接口，初始化struct cpuidle_device变量，主要是将dev->states_usage、dev->last_residency等状态信息清零；调用__cpuidle_register_device接口，将struct cpuidle_device指针保存在一个全局的per cpu的指针中（cpuidle_devices），同时将它添加到一个全局列表中（cpuidle_detected_devices）；调用cpuidle_add_sysfs接口，添加该设备有关的sysfs文件；调用cpuidle_enable_device接口，使能该设备，（可知，新注册的设备默认是使能的）；调用cpuidle_install_idle_handler接口，install idle handler。所谓的idle handler，其实就是一个内部的全局变量（initialized）。

2）cpuidle_enable_device

我们通过cpuidle_enable_device接口，来理解一下何为idle device的enable？调用cpuidle_get_cpu_driver接口，获取该设备对应的driver，如果设备没有绑定driver，或者当前没有governor（cpuidle_curr_governor为NULL），则不能enable，返回错误；依据driver提供的idle state的个数（drv->state_count），初始化设备的state_count变量（dev->state_count）；调用cpuidle_add_device_sysfs接口注册cpuidle device的sysfs文件，注意和前面的cpuidle_add_sysfs不同；如果current governor提供了enable接口，调用之；置位设备的enabled标志（dev->enabled = 1），同时将全局变量enabled_devices加 1，cpuidle_install_idle_handler/cpuidle_uninstall_idle_handler就是利用enabled_devices全局变量设置或者清零initialized标志的。initialized用来作为cpuidle是否可用的一个条件，具体使用会在cpuidle_idle_call函数里面作为判断是否可以继续idle流程。由前面的描述可知，在多核系统中，每个CPU会对应一个cpuidle device，因此cpuidle_register_device每被执行一次，就会注册一个不同的设备。如果cpuidle需要将它们分别记录下来，就要借助per-CPU变量，以cpuidle_devices指针为例，其声明和定义分别如下：

```
DEFINE_PER_CPU(struct cpuidle_device *, cpuidle_devices);//这是一个指针，方便后面使用。
DEFINE_PER_CPU(struct cpuidle_device, cpuidle_dev);/*这个才是真正的变量，即cpuidle_dev是一个cpuidle_device结构体
在cpuidle_register函数中，实现了变量的赋值：*/
    for_each_cpu(cpu, drv->cpumask) {
        device = &per_cpu(cpuidle_dev, cpu);
        device->cpu = cpu;
```

### 3.2 cpuidle driver管理

cpuidle driver管理位于drivers/cpuidle/driver.c中，主要负责cpuidle driver的注册、获取等逻辑的实现。

cpuidle_register_driver用于注册一个cpuidle driver，它主要完成如下内容：
注册之前，调用者需要提供详细的states信息（drv->states\drv->state_count）。进行一些合法性检查，包括idle state的个数是否大于零、cpuidle功能是否使能等等；
调用__cpuidle_driver_init初始化driver的内部数据，包括drv->refcnt、drv->cpumask、drv->bctimer等。调用__cpuidle_set_driver，将该driver“注册”到系统。如果drv->bctimer为1，则调用on_each_cpu_mask，在每个CPU上，执行一次cpuidle_setup_broadcast_timer，设置broadcast timer；最后，调用poll_idle_init，为那些具有POLL idle state的平台，注册默认的poll idle state。

1）cpuidle driver注册的意义

从本质上讲，cpuidle driver是一个“driver”，它驱动的对象是cpuidle device，也即CPU。在多核系统（SMP）中，会有多个CPU，也就有多个cpuidle device。如果这些device的idle功能相同，那么一个cpuidle driver就可以驱动这些device。否则，则需要多个driver才能驱动。

基于上面的事实，cpuidle core提供一个名称为“CONFIG_CPU_IDLE_MULTIPLE_DRIVERS”的配置项，用于设置是否需要多个cpuidle driver。如果没有使能这个配置项，说明所有CPU的idle功能相同，一个cpuidle driver即可。此时cpuidle driver的注册，就是将driver保存在一个名称为cpuidle_curr_driver的全局指针中，且只能注册一次。同理，cpuidle driver的获取，就是返回这个指针的值。可以看到这个配置宏的使用如下：

```
#ifdef CONFIG_CPU_IDLE_MULTIPLE_DRIVERS
//per cpu has a struct cpuidle_driver
static DEFINE_PER_CPU(struct cpuidle_driver *, cpuidle_drivers);
/**
 * __cpuidle_get_cpu_driver - return the cpuidle driver tied to a CPU.
 * @cpu: the CPU handled by the driver
 * Returns a pointer to struct cpuidle_driver or NULL if no driver has been
 * registered for @cpu.
 */
static struct cpuidle_driver *__cpuidle_get_cpu_driver(int cpu)
{
    return per_cpu(cpuidle_drivers, cpu);
}

static inline int __cpuidle_set_driver(struct cpuidle_driver *drv)
{
    int cpu;
    for_each_cpu(cpu, drv->cpumask) {

        if (__cpuidle_get_cpu_driver(cpu)) {
            __cpuidle_unset_driver(drv);
            return -EBUSY;
        }
        per_cpu(cpuidle_drivers, cpu) = drv;
    }
    return 0;
}
#else
static inline struct cpuidle_driver *__cpuidle_get_cpu_driver(int cpu)
{
    return cpuidle_curr_driver;
}

static inline int __cpuidle_set_driver(struct cpuidle_driver *drv)
{
    if (cpuidle_curr_driver)
        return -EBUSY;
    cpuidle_curr_driver = drv;
    return 0; }
```

2）broadcast timer功能

前面2.1小节提供过cpuidle state中的“CPUIDLE_FLAG_TIMER_STOP” flag。当cpuidle driver的idle state中有state设置了这个flag时，说明对应的CPU在进入idle state时，会停掉该CPU的local timer，此时Linux kernel的clock event framework便不能再依赖本CPU的local timer。针对这种情况，设计者会提供一个broadcast timer，该timer独立于所有CPU运行，并可以把tick广播到每个CPU上，因而不受idle state的影响。因此，如果cpuidle state具有STOP

TIMER的特性的话，需要在driver注册时，调用clock events提供的notify接口（clockevents_notify），告知clock events模块，打开broadcast timer。如下：

```
static int __cpuidle_register_driver(struct cpuidle_driver *drv)
{
……..
    if (drv->bctimer)
        on_each_cpu_mask(drv->cpumask, cpuidle_setup_broadcast_timer,
                 (void *)1, 1);
……..
    return 0;
}
static void __cpuidle_driver_init(struct cpuidle_driver *drv)
{
……..
    /*
     * Look for the timer stop flag in the different states, so that we know
     * if the broadcast timer has to be set up.  The loop is in the reverse
     * order, because usually one of the deeper states have this flag set.
     */
    for (i = drv->state_count - 1; i >= 0 ; i--) {
        if (drv->states[i].flags & CPUIDLE_FLAG_TIMER_STOP) {
            drv->bctimer = 1;
            break;
        }
    }
}
```

__cpuidle_driver_init接口负责检查所有的idle state，如果有state设置了CPUIDLE_FLAG_TIMER_STOP flag，则置位drv->bctimer变量，表示需要开启broadcast timer；然后在__cpuidle_register_driver中，根据drv->bctimer的状态，调用cpuidle_setup_broadcast_timer接口，打开具体CPU上的broadcat timer。

**3.3 cpuidle governor管理**

governor管理位于drivers/cpuidle/governor.c中，包括governor的注册、获取和切换功能，分别由下面三个接口实现：

```
int cpuidle_switch_governor(struct cpuidle_governor *gov)
{
    struct cpuidle_device *dev;
    if (gov == cpuidle_curr_governor)
        return 0;
    cpuidle_uninstall_idle_handler();
    if (cpuidle_curr_governor) {
        list_for_each_entry(dev, &cpuidle_detected_devices, device_list)
            cpuidle_disable_device(dev);
        module_put(cpuidle_curr_governor->owner);
    }
    cpuidle_curr_governor = gov;
    if (gov) {
        if (!try_module_get(cpuidle_curr_governor->owner))
            return -EINVAL;
        list_for_each_entry(dev, &cpuidle_detected_devices, device_list)
            cpuidle_enable_device(dev);
        cpuidle_install_idle_handler();
        printk(KERN_INFO "cpuidle: using governor %s\n", gov->name);
    }
```

```
        return 0;
    }
    int cpuidle_register_governor(struct cpuidle_governor *gov)
    {
                ……..
        mutex_lock(&cpuidle_lock);
        if (__cpuidle_find_governor(gov->name) == NULL) {
            ret = 0;
            list_add_tail(&gov->governor_list, &cpuidle_governors);
            if (!cpuidle_curr_governor ||
                cpuidle_curr_governor->rating < gov->rating)
                cpuidle_switch_governor(gov);
        }
        mutex_unlock(&cpuidle_lock);
        return ret;
    }
```

cpuidle_register_governor接口的逻辑比较简单，将governor保存到一个全局链表（cpuidle_governors）中，并判断新注册governor的rating是否大于当前governor（保存在cpuidle_curr_governor指针中），如果大于，则将新注册governor切换为当前governor；Cpuidle_switch_governor用于切换当前的governor，需要注意的是，切换之前，要disable所有的device（cpuidle_disable_device），并在切换之后打开。

### 3.4 cpuidle整体的管理功能

整体的管理功能位于drivers/cpuidle/cpuidle.c中，负责如下事项：

1）cpuidle framework的初始化，由cpuidle_init实现

```
    static inline void latency_notifier_init(struct notifier_block *n)
    {
        pm_qos_add_notifier(PM_QOS_CPU_DMA_LATENCY, n);
    }
    /**
     * cpuidle_init - core initializer
     */
    static int __init cpuidle_init(void)
    {
        int ret;
        if (cpuidle_disabled())
            return -ENODEV;

        ret = cpuidle_add_interface(cpu_subsys.dev_root);
        if (ret)
            return ret;
        latency_notifier_init(&cpuidle_latency_notifier);
        return 0;
    }
```

主要完成：调用cpuidle_add_interface接口，添加CPU global sysfs attributes；调用latency_notifier_init接口，添加一个pm qos notifier，以便当系统有新CPU/DMA的latency需求时，通知到cpuidle framework并将所有进入idle的cpu wakeup。

2）idle state的select和enter

由下面两个接口实现：

```
    extern int cpuidle_select(struct cpuidle_driver *drv,struct cpuidle_device
    *dev);
```

```
extern int cpuidle_enter(struct cpuidle_driver *drv, struct cpuidle_device
*dev, int index);
```

cpuidle_select的实现非常简单，让当前使用的governor回调select函数。cpuidle_enter接口根据state index，进入指定的state，调用state的enter函数，并记录相关的统计信息即可。

3）cpuidle driver注册的简单接口

由cpuidle_register接口实现，主要目的是在简单的平台上（所有cpuidle device的功能一样），省去cpuidle device的注册过程（由cpuidle core帮忙实现）。driver只需要定义各个idle state，并通过cpuidle_register注册cpuidle driver即可。arm64平台就是使用这个方式。

剩下的内容就是供用户空间操作和查看的sys接口了，具体source code：driver/cpuidle/sysfs.c

# 三 CPUIDLE framework_ARM64通用idle驱动

### 1. 前言

下文sharkL3 /pike2项目的cpuidle driver为例，说明怎样在cpuidle framework的框架下，编写cpuidle driver。

### 2. arm64_idle_init

ARM64 generic CPU idle driver的代码没有找到，但pike2的cpuidle driver应该是基于此实现的，内容和形式差不多，code位于“drivers/cpuidle/cpuidle-sprd.c”中，它的入口函数是arm64_idle_sprd_init：

```
static const struct of_device_id arm_idle_state_match[] __initconst = {
    { .compatible = "arm,idle-state",
      .data = arm_enter_idle_state },
    { },
};
static int __init arm_idle_sprd_init(void)
{
    int  ret;
    struct cpuidle_driver *drv = &arm_idle_driver;
    /*
     * Initialize idle states data, starting at index 1.
     * This driver is DT only, if no DT idle states are detected (ret == 0)
     * let the driver initialization fail accordingly since there is no
     * reason to initialize the idle driver if only wfi is supported.
     */
/*会解析dts，获取当前cpu支持多少个cpuidle state*/
    ret = dt_init_idle_driver(drv, arm_idle_state_match, 1);
            ……..
    }
#ifdef CONFIG_ARM64
    int cpu;
    /*
     * Call arch CPU operations in order to initialize
     * idle states suspend back-end specific data
     */
    for_each_possible_cpu(cpu) {
        ret = cpu_init_idle(cpu);
                    ……..
```

```
        }
    #endif
        ret = cpuidle_register(drv, NULL);
                ……...
    }
```

由该函数的执行过程，可以看出cpuidle driver的实现过程，包括：

1）静态定义一个struct cpuidle_driver变量（这里为arm_idle_driver）并填充必要的字段：

```
    static struct cpuidle_driver arm_idle_driver = {
        .name = "arm_idle_sprd",
        .owner = THIS_MODULE,
        /*
         * State at index 0 is standby wfi and considered standard
         * on all ARM platforms. If in some platforms simple wfi
         * can't be used as "state 0", DT bindings must be implemented
         * to work around this issue and allow installing a special
         * handler for idle state index 0.
         */
        .states[0] = {
            .enter                  = arm_enter_idle_state,
            .exit_latency           = 1,
            .target_residency       = 1,
            .power_usage        = UINT_MAX,
            .name                   = "WFI",
            .desc                   = "ARM WFI",
        }
    };
    /*该driver的名称为“arm_idle_sprd”，会体现在sysfs中体现；
    对于state0，driver将其初始化为：exit latency和target residency均为1（最小值），
    power usage为整数中的最大值。由此可以看出，这些信息不是实际信息（因为driver不可能知道所
    有ARM平台的WFI相关的信息），而是相对信息，其中的含义是：所有其它的state，exit latency和
    target residency都会比state0大，power usage都会比state0小，讨巧的设计。*/
```

2）初始化其它的idle states（从state1开始），从解析函数看，必须从dts获取，否则失败。

3）对每一个cpu，调用cpu_init_idle接口，初始化用于支持cpuidle的、和cpu suspend有关的功能。

4）调用cpuidle_register，将cpuidle driver注册到cpuidle core中。

### 3. dt_init_idle_driver

dt_init_idle_driver函数用于从DTS（sc7731e.dtsi）中解析出cpuidle states的信息，并初始化arm64_idle_driver中的states数组。在分析这个函数之前，先看一下cpuidle相关的DTS源文件是怎么写的？

```
    idle-states {
                            ……...
            LIGHT_SLEEP: light_sleep {
                compatible = "arm,idle-state";
                entry-latency-us = <20>;
                exit-latency-us = <10>;
                min-residency-us = <50>;
                local-timer-stop;
            };
            HEAVY_SLEEP: heavy_sleep {
                compatible = "arm,idle-state";
                entry-latency-us = <400>;
```

```
                exit-latency-us = <700>;
                min-residency-us = <1200>;
                local-timer-stop;
            };
        };
    /*cpuidle有关的DTS信息，从属于cpus node的，先看最后面的idle-states node，它负责定义该ARM平台支持的所有的idle states，每个子node就是一个state：各个state定义都以“arm,idle-state”标识；entry-latency-us、exit-latency-us、min-residency-us分别定义了改idle state的几个重要参数，local-timer-stop对应CPUIDLE_FLAG_TIMER_STOP flag。这些信息会被dt_init_idle_driver解析出来，并保存在arm64_idle_driver的state数组中。
    在每个cpu的node中，通过cpu-idle-states字段，指明该CPU支持的idle states，可以有多个。同时，在解析过程中，会为每个state指定enter回调函数，但是对于arm64平台来说，统一使用arm_enter_idle_state接口。*/
```

**4. cpu_init_idle**

cpuidle功能的支持，需要依赖CPU的、和电源管理有关的底层代码实现。对ARM64来说，kernel将这些底层代码抽象为一个系列的操作函 数集（struct cpu_operations，具体可参考arch/arm64/kernel/cpu_ops.c）。以ARM64为例，ARM document规定了一种PSCI（Power State Coordination Interface）接口，它由firmware实现，用于电源管理有关的操作，如IDLE相关的、SMP相关的、Hotplug相关的、等等。对cpuidle来说，需要在cpuidle driver注册之前，调用cpu_init_idle，该函数会根据当前使用的操作函数集，调用其中的cpu_init_idle回调函数，进行 idle相关的初始化操作。对应的函数为cpu_psci_cpu_init_idle。

**5. arm_enter_idle_state**

idle state的enter函数用于使CPU进入指定的idle state，如下以pike2为例：pike2只有L_SLEEP和H_SLEEP两种状态。

```
#ifdef CONFIG_ARM64  //不是arm64也差不多的enter函数
static int arm_enter_idle_state(struct cpuidle_device *dev,struct cpuidle_driver *drv, int idx)
{
    int ret = 0;
    struct timeval start_time, end_time;
    long usec_elapsed;
    struct device_node *np = of_find_node_by_name(NULL , "idle-states");
    if (cpuidle_debug)
        do_gettimeofday(&start_time);
    switch (idx) {
    case STANDBY:
        cpu_do_idle();
        break;
    case L_SLEEP:
        light_sleep_en(np);
        cpu_do_idle();
        light_sleep_dis(np);
        break;
    case CORE_PD:
        light_sleep_en(np);
        cpu_pm_enter();
        ret = cpu_suspend(idx);
        cpu_pm_exit();
        light_sleep_dis(np);
        break;
```

```
        case CLUSTER_PD:
            light_sleep_en(np);
            cpu_pm_enter();
            cpu_cluster_pm_enter();
            ret = cpu_suspend(idx);
            cpu_cluster_pm_exit();
            cpu_pm_exit();
            light_sleep_dis(np);
            break;
        case TOP_PD:
            light_sleep_en(np);
            cpu_pm_enter();
            cpu_cluster_pm_enter();
            ret = cpu_suspend(idx);
            cpu_cluster_pm_exit();
            cpu_pm_exit();
            light_sleep_dis(np);
            break;
        default:
            cpu_do_idle();
            WARN(1, "[CPUIDLE]: NO THIS IDLE LEVEL!!!");
        }
        if (cpuidle_debug) {
            do_gettimeofday(&end_time);
            usec_elapsed = (end_time.tv_sec - start_time.tv_sec) * 1000000 +
                (end_time.tv_usec - start_time.tv_usec);
            pr_info("[CPUIDLE] Enter idle state: %d ,usec_elapsed = %ld \n",
                idx, usec_elapsed);
        }
        return ret ? -1 : idx;
    }
    #endif
    //根据不同的state idx执行特定的save power和退出save power mode！
```

下面是我们cpu根据不同产品选择不同的idle state level：

```
    #ifndef CPUIDLE_SPRD_HEADER
    #define CPUIDLE_SPRD_HEADER
    enum {
        STANDBY = 0,  /* WFI */
        L_SLEEP,      /* Light Sleep, WFI & DDR Self-refresh & MCU_SYS_SLEEP */
        H_SLEEP,        /* HEAVY/Doze Sleep */
        CORE_PD,      /* Core power down & Lightsleep */
    #ifdef CONFIG_ARM64
        CLUSTER_PD,   /* Cluster power down & Lightsleep */
        TOP_PD,       /* Top Power Down & Lightsleep */
    #endif
    };
    //Level越高，save power就越多，但是退出时延就多，对于快速响应的task的性能就差。
```

对于如何选择进入哪个idle state，则需要governor支援。

# 四 cpuidle framework_menu/sprd governor

## 1. 前言

本文以menu governor为例，进一步理解cpuidle framework中governor的概念，并学习governor的实现方法。在当前的kernel中，有2个governor，分别为ladder和menu（蜗蜗试图理解和查找，为什么会叫这两个名字，暂时还没有答案）。ladder在periodic timer tick system中使用，menu在tickless system中使用。现在主流的系统，出于电源管理的考量，大多都是tickless system。另外，menu governor会利用pm qos framework，在选择策略中加入延迟容忍度（Latency tolerance）的考量。

**2. 背后的思考**

下面的内容，来源于drivers/cpuidle/governors/menu.c中的注释。governor的主要职责，是根据系统的运行情况，选择一个合适的idle state。具体的算法，需要基于下面两点考虑：

1）切换的代价

进入C state的目的，是节省功耗，但CPU在C state和normal state（P-state）之间切换，是要付出功耗上面的代价的。这最终会体现在idle state的target_residency字段上。

idle driver在注册idle state时，要非常明确state切换的代价，基于该代价，CPU必须在idle state中停留超过一定的时间（target_residency）才是划算的。

因此governor在选择C state时，需要预测出CPU将要在C state中的停留时间，并和备选idle state的target_residency字段比较，选取满足“停留时间 > target_residency”的state。

2）系统的延迟容忍度

备选的的C state中，功耗和退出延迟是一对不可调和的矛盾，电源管理的目标，是在保证延迟在系统可接受的范围内的情况下，尽可能的节省功耗。idle driver在注册idle state时，会提供两个信息：CPU在某个state下的功耗（power_usage）和退出该state的延迟（exit_latency）。那么如果知道系统当前所能容忍的延迟（简称latency_req），就可以在所有exit_latency小于latency_req的state中，选取功耗最小的那个。因此，governor算法就转换为获取系统当前的latency_req，而这正是pm qos的特性。基于上面的考量，menu governor的主要任务就转化为两个：

1. 根据系统的运行情况，预测CPU将在C state中停留的时间（简称predicted_us）；
2. 借助pm qos framework，获取系统当前的延迟容忍度（简称latency_req）。

任务1，menu governor从如下几个方面去达成：

上面讲过，menu governor用于tickless system，即menu将“距离下一个tick来临的时间（由next timer event测量，简称next_timer_us）”作为基础的predicted_us。当然这个基础的predicted_us是不准确的，因为在这段时间内，随时都可能产生除next timer event之外的其它wakeup event。为了使预测更准确，有必要加入一个校正因子（correction factor），该校正因子基于过去的实际predicted_us和next_timer_us之间的比率，例如，如果wakeup event都是在预测的next timer event时间的一半时产生，则factor为0.5。另外，为了更精确，menu使用动态平均的factor。

此外，对不同范围 的next_timer_us，correction factor的影响程度是不一样的。例如期望50ms和500ms的next timer event时，都是在10ms时产生了wakeup event，显然对500ms的影响比较大。如果计算平均值时将它们混在一起，就会对预测的准确性产生影响，所以计算correction factor的数据时，需要区分不同级别的next_timer_us。同时，系统是否存在io wait，对factor的敏感度也不同。基于这些考虑，menu使用了一组factor（12个），分别用于不同next_timer_us、不同io wait的场景下的的校正。

最后，在有些场合下，next_timer_us的预测是完全不正确的，如存在固定周期的中断时（音频等）。这时menu采用另一种不同的预测方式：统计过去8次停留时间的标准差（stand deviation），如果小于一定的门限值，则使用这8个停留时间的平均值，作为预测值。

任务2，延迟容忍度（latency_req）的估算，menu综合考虑了两种因素，如下：

1）由pm qos获得的，系统期望的，CPU和DMA的延迟需求。这是一个硬性指标。

2） 基于这样一个经验法则：越忙的系统，对系统延迟的要求越高，结合任务1中预测到的停留时间（predicted_us），以及当前系统的CPU平均负荷和 iowaiters的个数（get_iowait_load函数获得），算出另一个延迟容忍度，计算公式（这是一个经验公式）为：

predicted_us / (1 + 2 * loadavg +10 * iowaiters)，后来在android kernel上修改为了

predicted_us / (1 + 10 * iowaiters)

这个公式反映的是退出延迟和预期停留时间之间的比例，loadavg和iowaiters越大，对退出延迟的要求就越高。最后，latency_req的值取上面两个估值的最小值。

**3. Menu_select如何选择合适的idle state？**

首先按照上面一节可知，需要满足两个因素，就可以作为下一个目标idle state

1. 预测的cpuidle时间大于设定的目标idle state的idle驻留时间，即预测idle时间<=target_residency,(predicted_us >= target_residency)
2. 时延的要求，即系统要求的时延不能大于目标idle state的退出时延，即系统要求的latency<=exit_latency,(latency_req >= exit_latency)

只要满足上面两个条件的idle level就可以被选择。

下面就需要来看看这两个条件是如何计算的：

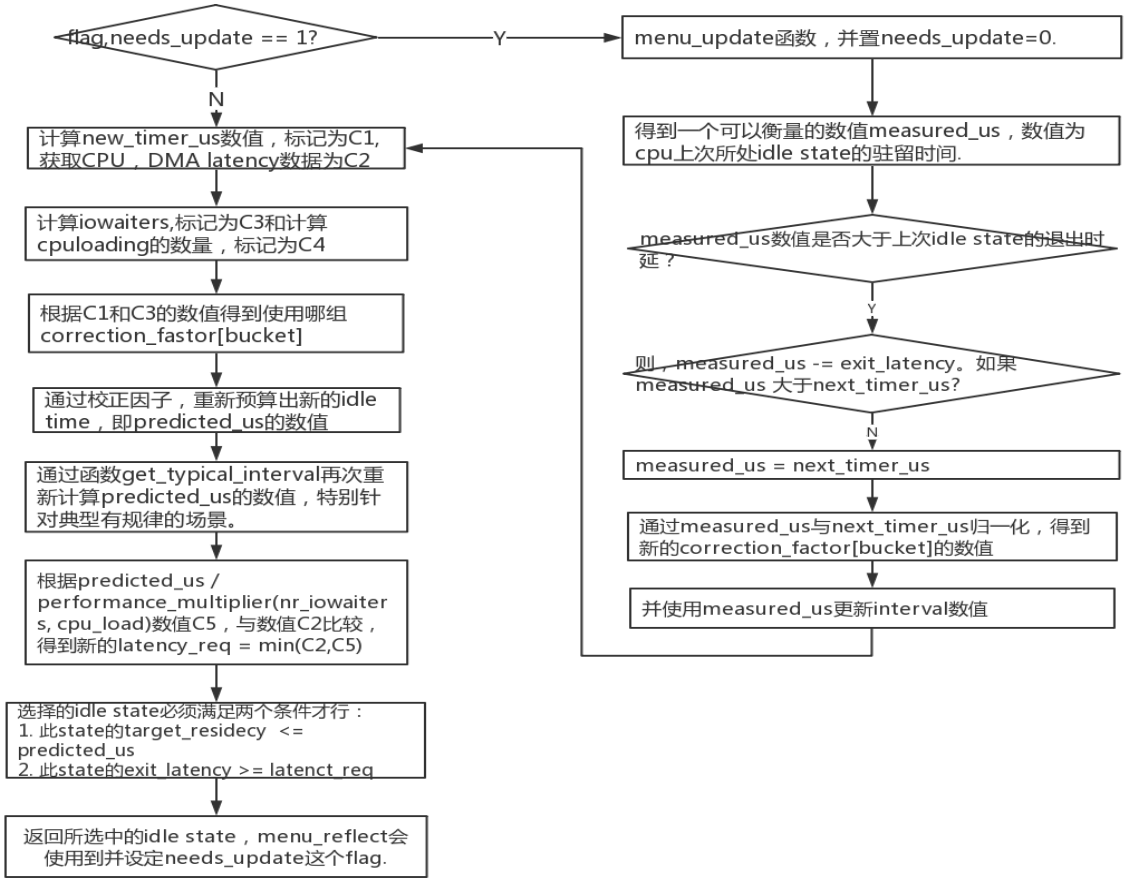
3.1 predicted_us的计算

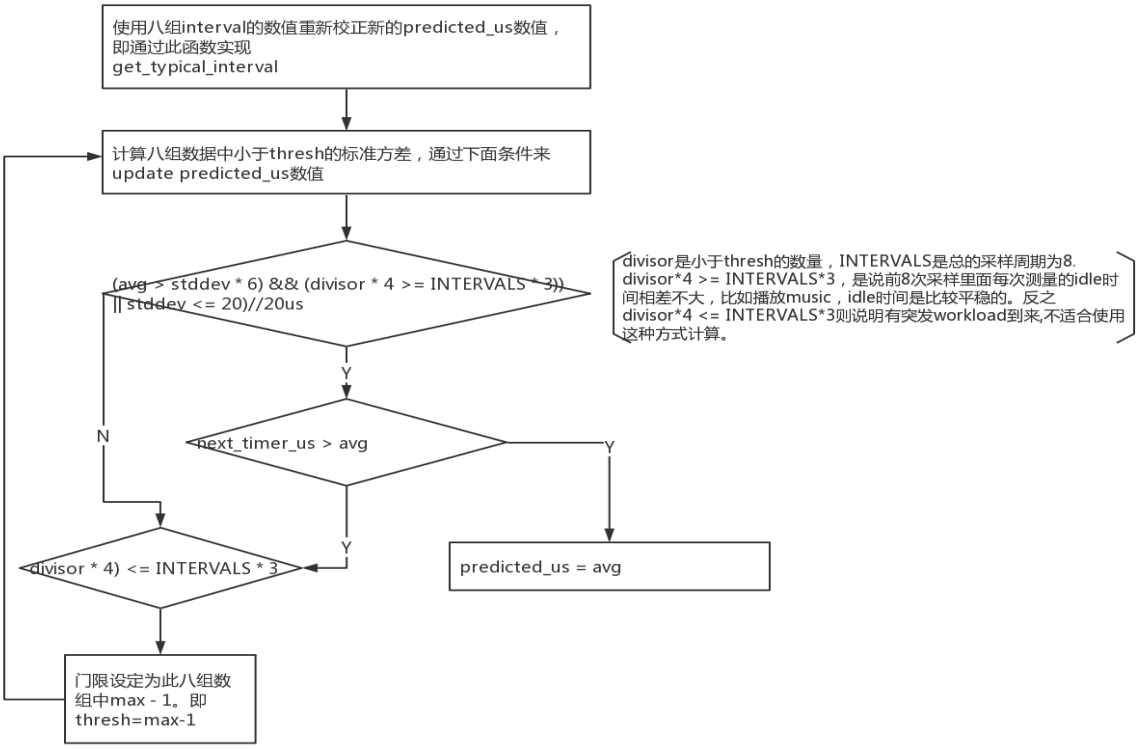
根据struct tick_sched 结构体里面的参数sleep_length的数值得到此cpu 下一个idle的持续时间，由于存在各种系统变化，需要纠正因子来调整此数值，这就使用到了纠正因子如何获取了。

3.2 latency_req的计算

1. 首先获取pm Qos的CPU DMA的latency作为第一个latency_req数值
2. 之后根据cpu loading和iowaiter数量，使用predicted_us来计算得出第二个latency_req，计算方式如下：

   data->predicted_us / performance_multiplier(nr_iowaiters, cpu_load);


flag.needs_update == 1?
Y
menu_update函数，并置needs_update=0.
N
得到一个可以衡量的数值measured_us，数值为cpu上次所处idle state的驻留时间.
计算new_timer_us数值，标记为C1,获取CPU，DMA latency数据为C2
measured_us数值是否大于上次idle state的退出时延？
计算iowaiters,标记为C3和计算cpuloading的数量，标记为C4
Y
则，measured_us -= exit_latency。如果measured_us 大于next_timer_us?
根据C1和C3的数值得到使用哪组correction_fastor[bucket]
N
通过校正因子，重新预算出新的idle time，即predicted_us的数值
measured_us = next_timer_us
通过函数get_typical_interval再次重新计算predicted_us的数值，特别针对典型有规律的场景。
通过measured_us与next_timer_us归一化，得到新的correction_factor[bucket]的数值
根据predicted_us / performance_multiplier(nr_iowaiters, cpu_load)数值C5，与数值C2比较，得到新的latency_req = min(C2,C5)
并使用measured_us更新interval数值
选择的idle state必须满足两个条件才行：
1. 此state的target_residecy <= predicted_us
2. 此state的exit_latency >= latenct_req
返回所选中的idle state，menu_reflect会使用到并设定needs_update这个flag.


使用八组interval的数值重新校正新的predicted_us数值，即通过此函数实现get_typical_interval
计算八组数据中小于thresh的标准方差，通过下面条件来update predicted_us数值
(avg > stddev * 6) && (divisor * 4 >= INTERVALS * 3)) || stddev <= 20)//20us
divisor是小于thresh的数量，INTERVALS是总的采样周期为8.
divisor*4 >= INTERVALS*3，是说前8次采样里面每次测量的idle时间相差不大，比如播放music，idle时间是比较平稳的。反之divisor*4 <= INTERVALS*3则说明有突发workload到来,不适合使用这种方式计算。
Y
N
next_timer_us > avg
Y
Y
divisor * 4) <= INTERVALS * 3
predicted_us = avg
门限设定为此八组数组中max - 1。即thresh=max-1

**4. sprd_select如何选择合适的idle state？**

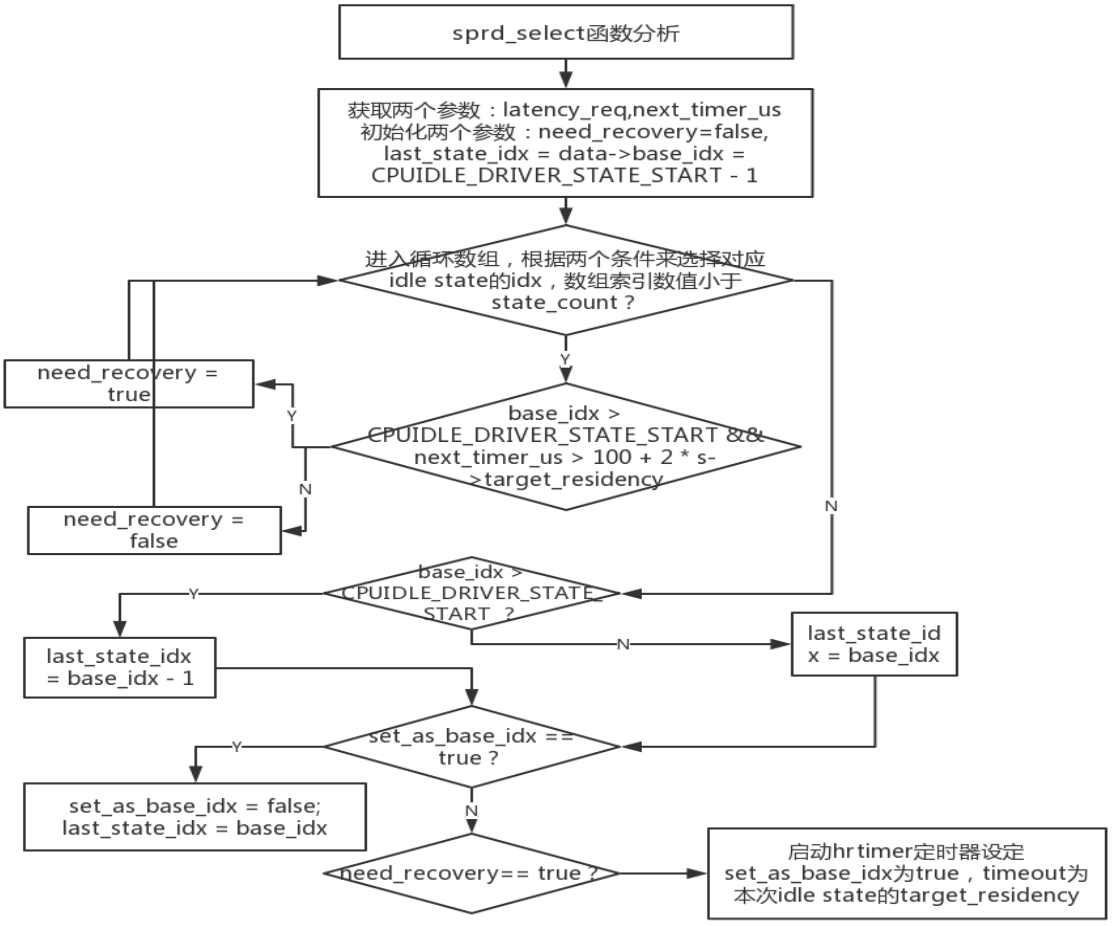
有如下的几个疑问：

- (data->next_timer_us > 100 + 2 * s->target_residency) 这应该是经验数值，为何这样选择呢？
- 第二个疑问：

```
    if (data->base_idx > CPUIDLE_DRIVER_STATE_START)
        data->last_state_idx = data->base_idx - 1;
    else
        data->last_state_idx = data->base_idx;
/*为何这样做呢？如果base_idx为2，last_state_idx会是一样的，为啥要减一呢？目的是啥呢？
*/
```

- 启动hrtimer定时器修改的目的是什么？是当前预估的idle(next_timer_us)时间比较长，我们设定这个hrtimer的目的是，延迟一段时间使cpu进入更深层次的idle state。反之就能够推出2这样做的原因了。

它的流程图如下：



# 五 cpuidle framework_LKML code update review

下载的code base是LKML 4.18.0-rc6。下面通过这些差异来讲解最新主线上面对于cpuidle governor的修改及其出发点。本章按照不同点来分别对比讲解。

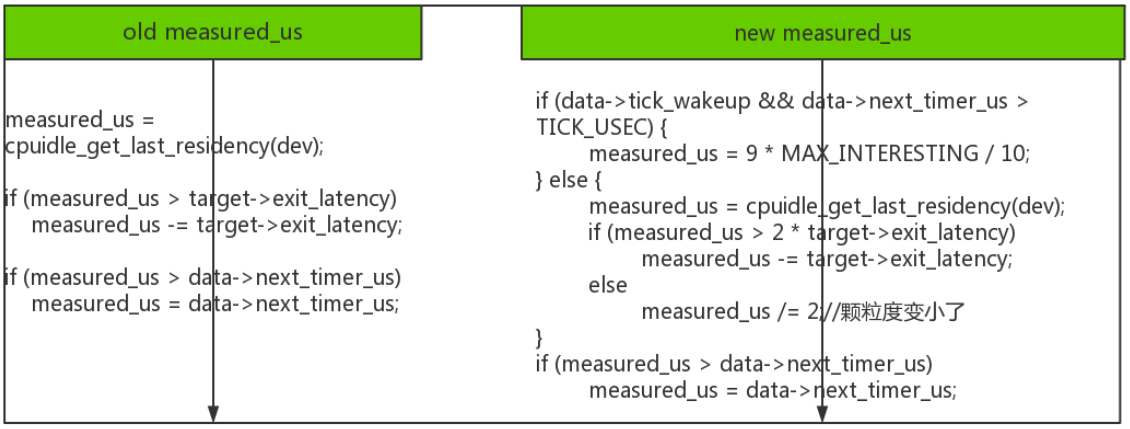
做出了哪些修改：

1. next_timer_us如何得到的？

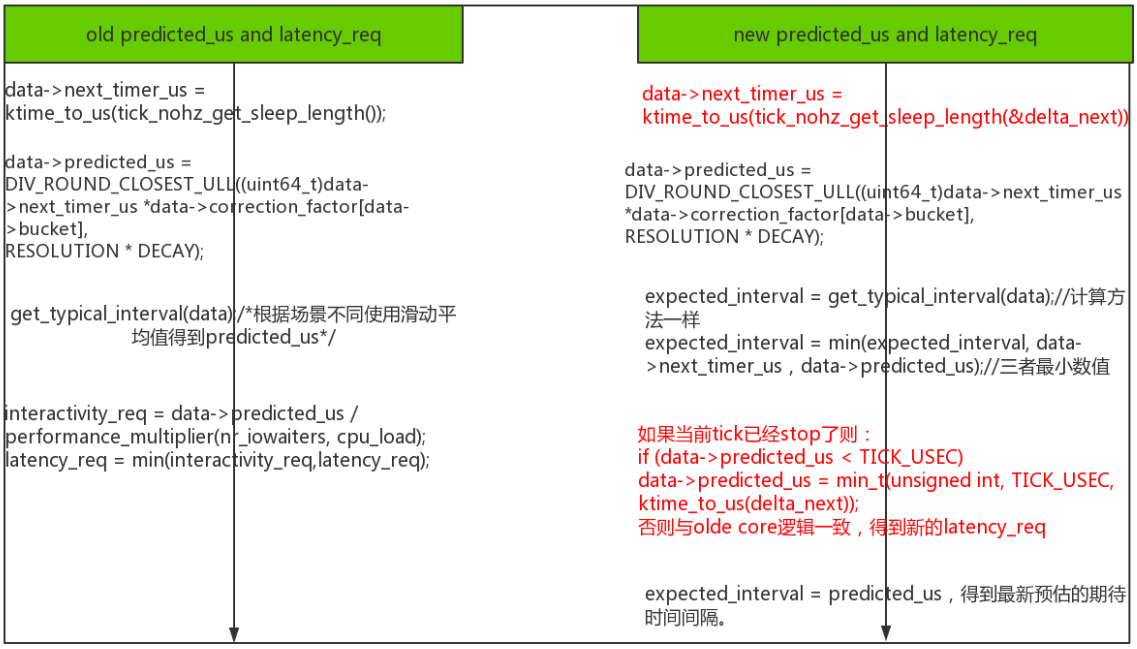

2. tick_wakeup这个标记flag如何使用的？为true表示tick handler已经在运行，在计算measured_us时会被使用到。

tick_wakeup使用在measured_us的update上，即当tick_wakeup==true && next_timer_us大于一个tick时间，则说明系统处于深度idle状态，为了避免predicted_us出现预测错误，预测数值很小，处于浅idle状态，需要合理的设定measured_us时间，帮助预测器下次选择更好的idle state：
#define MAX_INTERESTING 50000
measured_us = 9 * MAX_INTERESTING / 10;
Tick_wakeup数值的修改在每次退出idle时候，函数menu_reflect中修正：
tick_wakeup = tick_nohz_idle_got_tick();

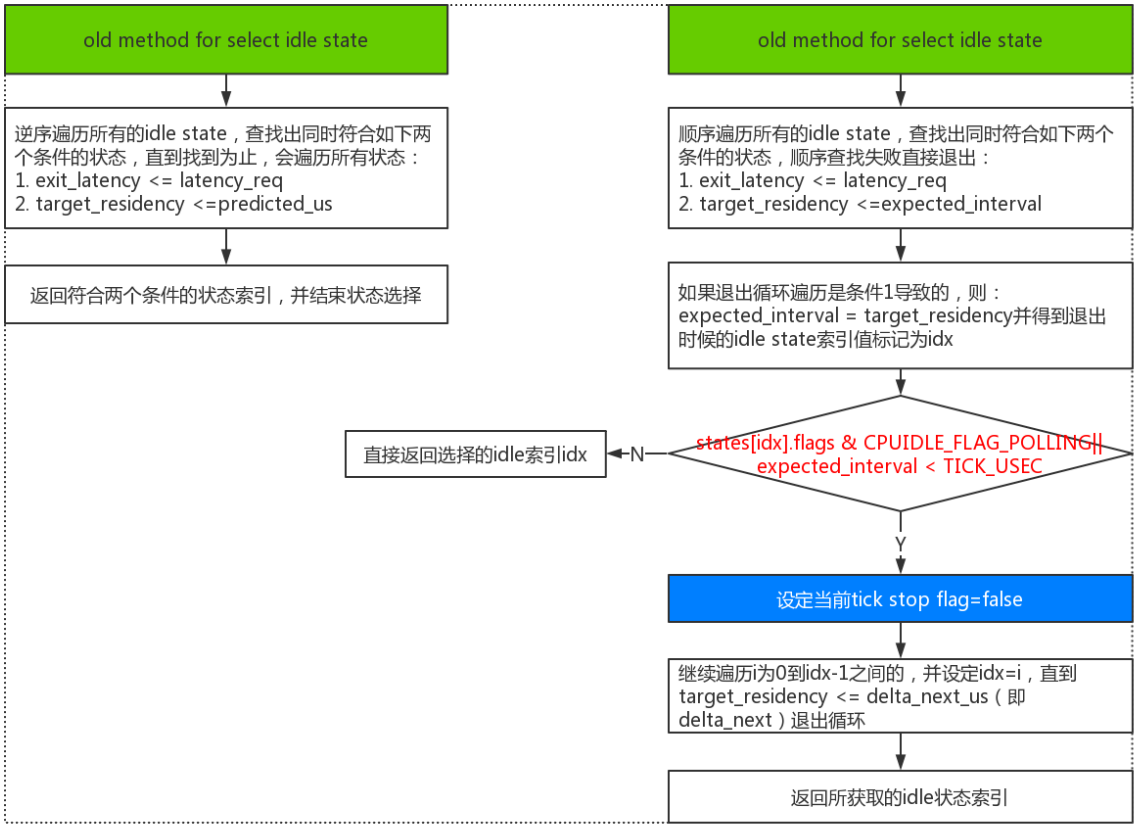
3. measured_us计算方式的改变



4. 如何较为准确的预算出下次idle时间和latency要求




5. 如何选择本次进入哪个idle state




至此，最新主线上面修改的主要cpuidle内容update如上。这里面还有如下若干个问题需要仔细check：

1. 在新的menu governor计算next_timer_us的函数（分能够stop tick还是不能）：
   tick_nohz_get_sleep_length(&delta_next)，存在两个疑问：
   1). delta_next 是clock_event_device 结构体里面的next_event的值
   2). tick_nohz_next_event这个函数不太明白，tick event，clock event区别是什么？
   3). 得到tick next event时间，与hrtimer最近到来时间取最小数值，作为next_timer_us的数值。
2. 目前在PM分支上添加了”**sched: idle: IRQ based next prediction for idle period”相关的patch：**
   **https://lists.linaro.org/pipermail/eas-dev/2015-December/000338.html可以查看最新的。**
3. 还存在一个新的cpuidle governor：irq governor，猜测是与2配套使用的。

对于上面的2和3目前正在学习。但是优先级不高，后面的主要精力会放在EAS和schedule上面，这是块难啃的骨头，加油！！！