# Kernel八问<八> ——

# 第八问：杂项

## SMP下，其他CPUs是怎么启动的？

现有的Linux系统启动（arm）都是单CPU（cpu0）启动，当启动完毕后，多CPU就开始工作了，那么除了cpu0，其他cpu是怎么开始工作的？又是在什么时候开始启动的？接下来让我们来分析一下这个问题。

当系统启动的时候，硬件层面会让一个CPU开始工作，同时阻止其他CPU进一步操作。这是为了避免一些竞态事件发生，同时保证系统初始化的速度（不用加锁）。

一般有三种方法来唤醒CPU继续运行：

1. Spin-table
2. Psci
3. ACPI Parking-protocol

第一种顾名思义跟spin锁一样的机制。

第二种全称是power state coordination interface。这是一种标准的电源管理接口，可以让OS在ARM设备上管理在不同异常级别工作的软件。这种方式比第一种要复杂的多，但是优点是可以控制CPU的行为（suspend，off等）。

第三种利用ACPI来启动CPU的一种方法，跟PSCI的区别最大区别是这种方法是用IPI来唤醒的（关于IPI可以参考《内核八问-第一问》）。利用这种方法启动CPU，我并没有找到。这里也就不再讲述。

下面我们来讲解系统是什么时候、什么方式来控制其他CPU的启动。

Kernel处理的流程是一样的，唯一不同的是唤醒方式。让我们来RTFSC(read the fucking source code)：

我们先讲第一种唤醒方式：代码主要集中在：

init\_bootcpu\_ops();

smp\_init\_cpus();

smp\_build\_mpidr\_hash();

当U-boot（参考u-boot.doc）加载kernel后，core0开始运行初始化程序，在setup\_arch阶段，core0发现还有其他的core在沉睡。“生命不能浪费在睡觉上”core0认为，于是core0就开始初始化其他core。这部分主要集中在smp\_init\_cpus这个函数，init\_bootcpu\_ops是为主核core0添加enable\_method方法处理函数。

Smp\_init\_cpus这个函数非常重要，Let’s go：

if (acpi\_disabled)

of\_parse\_and\_init\_cpus();

else

acpi\_parse\_and\_init\_cpus();

因为K3并不支持ACPI，所以执行of\_parse\_and\_init\_cpus：

static void \_\_init of\_parse\_and\_init\_cpus(void)

{

struct device\_node \*dn;

for\_each\_of\_cpu\_node(dn) {

u64 hwid = of\_get\_cpu\_mpidr(dn);

if (hwid == INVALID\_HWID)

goto next;

if (is\_mpidr\_duplicate(cpu\_count, hwid)) {

pr\_err("%pOF: duplicate cpu reg properties in the DT\n",

dn);

goto next;

}

if (hwid == cpu\_logical\_map(0)) {

if (bootcpu\_valid) {

pr\_err("%pOF: duplicate boot cpu reg property in DT\n",

dn);

goto next;

}

bootcpu\_valid = true;

early\_map\_cpu\_to\_node(0, of\_node\_to\_nid(dn));

continue;

}

if (cpu\_count >= NR\_CPUS)

goto next;

pr\_debug("cpu logical map 0x%llx\n", hwid);

set\_cpu\_logical\_map(cpu\_count, hwid);

early\_map\_cpu\_to\_node(cpu\_count, of\_node\_to\_nid(dn));

next:

cpu\_count++;

}

}

关于DT会在硬件管理译文讲解，这个函数就是根据address-cells指定的大小，读出reg的值，K3address\_cell是1，reg的值就是hwid。

if (hwid == cpu\_logical\_map(0))这里处理的是启动core，Kernel默认启动core就是0。

early\_map\_cpu\_to\_node 这函数是处理NUMA的，K3没有支持NUMA。关于NUMA会在内存管理这节讲解。

\_\_cpu\_logical\_map设置对应的logical HWID。

处理完of\_parse\_and\_init\_cpus后，进行一些必要的检查，执行一个for循环：

for (i = 1; i < nr\_cpu\_ids; i++) {

if (cpu\_logical\_map(i) != INVALID\_HWID) {

if (smp\_cpu\_setup(i))

set\_cpu\_logical\_map(i, INVALID\_HWID);

}

}

根据core数目进行smp\_cpu\_setup。其中的cpu\_logical\_map已经在前面的函数里面设置了。执行：

static int \_\_init smp\_cpu\_setup(int cpu)

{

const struct cpu\_operations \*ops;

if (init\_cpu\_ops(cpu))

return -ENODEV;

ops = get\_cpu\_ops(cpu);

if (ops->cpu\_init(cpu))：

return -ENODEV;

set\_cpu\_possible(cpu, true);

return 0;

}

Init\_cpu\_ops：

int \_\_init init\_cpu\_ops(int cpu)

{

const char \*enable\_method = cpu\_read\_enable\_method(cpu);

if (!enable\_method)

return -ENODEV;

cpu\_ops[cpu] = cpu\_get\_ops(enable\_method);

if (!cpu\_ops[cpu]) {

pr\_warn("Unsupported enable-method: %s\n", enable\_method);

return -EOPNOTSUPP;

}

return 0;

}

首先从DT里面取得enable\_method， 这里肯定是spin-table，进入cpu\_get\_ops：

static const struct cpu\_operations \* \_\_init cpu\_get\_ops(const char \*name)

{

const struct cpu\_operations \*const \*ops;

ops = acpi\_disabled ? dt\_supported\_cpu\_ops : acpi\_supported\_cpu\_ops;

while (\*ops) {

if (!strcmp(name, (\*ops)->name))

return \*ops;

ops++;

}

return NULL;

}

Kernel选择dt\_supported\_cpu\_ops，这是一个结构体数组，定义为：

static const struct cpu\_operations \*const dt\_supported\_cpu\_ops[] \_\_initconst = {

&smp\_spin\_table\_ops,

&cpu\_psci\_ops,

NULL,

};

其中我们需要的就是smp\_spin\_table\_ops：

const struct cpu\_operations smp\_spin\_table\_ops = {

.name = "spin-table",

.cpu\_init = smp\_spin\_table\_cpu\_init,

.cpu\_prepare = smp\_spin\_table\_cpu\_prepare,

.cpu\_boot = smp\_spin\_table\_cpu\_boot,

};

现在很清楚了，就是根据DT里面定义的enable-method与dt\_supported\_cpu\_ops数组里面的name结构体成员变量进行比较，返回正确的操作结构体。这里明显的就是smp\_spin\_table\_ops。

if (ops->cpu\_init(cpu))：

return -ENODEV;

执行cpu\_init，就是smp\_spin\_table\_cpu\_init，这个函数不展开了，主要就是读取指定的cpu\_relases\_addr，如：

&cpu0 {

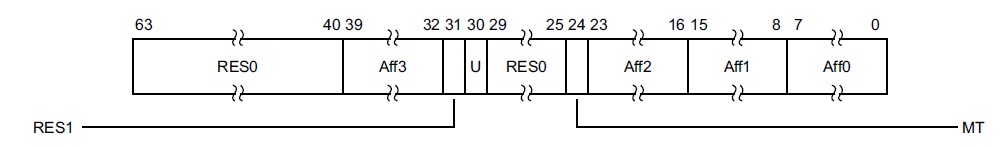
Enable\_method = “spin-table”;

Cpu\_release\_addr = <0x0 0x8000fff8>;

} 读出0x8000fff8，存到cpu\_release\_addr[cpu]里面。

set\_cpu\_possible(cpu, true);

设置core为possible状态（即可能存在）另外还有present（存在，不能干活，正在迷茫中）online（在线了，即将准备工作）active（可以工作了）。这些状态跟调度密切相关。

smp\_build\_mpidr\_hash();设置CPU的亲和性。MPIDR\_EL1

初始化其他core后，core0有开始忙其他事情去了，等到core0建立第一个线程后，pid = kernel\_thread(kernel\_init, NULL, CLONE\_FS);

这个线程参考kernel八问-第七问启动章节，这里我们能够看到该线程会执行

static noinline void \_\_init kernel\_init\_freeable(void)-> smp\_prepare\_cpus(setup\_max\_cpus);

这个函数主要执行：

for\_each\_possible\_cpu(cpu) {

per\_cpu(cpu\_number, cpu) = cpu;

if (cpu == smp\_processor\_id())

continue;

ops = get\_cpu\_ops(cpu);

if (!ops)

continue;

err = ops->cpu\_prepare(cpu);

if (err)

continue;

set\_cpu\_present(cpu, true);

numa\_store\_cpu\_info(cpu);

}

Cpu\_prepare函数就是smp\_spin\_table\_cpu\_prepare，我们来分析一下这个函数：

static int smp\_spin\_table\_cpu\_prepare(unsigned int cpu)

{

\_\_le64 \_\_iomem \*release\_addr;

if (!cpu\_release\_addr[cpu])

return -ENODEV;

release\_addr = ioremap\_cache(cpu\_release\_addr[cpu],

sizeof(\*release\_addr));

if (!release\_addr)

return -ENOMEM;

writeq\_relaxed(\_\_pa\_symbol(secondary\_holding\_pen), release\_addr);

\_\_flush\_dcache\_area((\_\_force void \*)release\_addr,

sizeof(\*release\_addr));

sev();

iounmap(release\_addr);

return 0;

}

这个函数不难理解，但是有个地方会让人困惑，就是其他core是在哪里停下的。让我们来到uboot里面。

#if defined(CONFIG\_ARMV8\_SPIN\_TABLE) && !defined(CONFIG\_SPL\_BUILD)

branch\_if\_master x0, x1, master\_cpu

b spin\_table\_secondary\_jump

#endif

master\_cpu:

bl \_main

spin\_table\_secondary\_jump：

0: wfe

ldr x0, spin\_table\_cpu\_release\_addr

cbz x0, 0b

br x0

这里比较重要的就是wfe这条指令，wfe是wait for event，core等待一个事件的发生，不然就停下来等待。唤醒wfe的指令是sev。

让我们再来梳理一下，所有的core停在wfe上面，然后smp\_spin\_table\_cpu\_prepare把secondary\_holding\_pen入口地址写入release address。调用sev唤醒secondary core。

那么secondary\_holding\_pen做了写什么：

bl el2\_setup // Drop to EL1, w0=cpu\_boot\_mode

bl set\_cpu\_boot\_mode\_flag

mrs x0, mpidr\_el1

mov\_q x1, MPIDR\_HWID\_BITMASK

and x0, x0, x1

adr\_l x3, secondary\_holding\_pen\_release

pen: ldr x4, [x3]

cmp x4, x0

b.eq secondary\_startup

wfe

b pen

前面的bl跳转参考内核八问-第七问，我们主要看adr\_l x3， secondary\_holding\_pen\_release。把secondary\_holding\_pen\_release变量地址放入x3寄存器。读入值到x4寄存器，比较x4与x0的值，如果一样跳转到secondary\_startup。不一样则停在wfe处，等待sev的唤醒。X0的值即MPIDR寄存器里面的AFF1，表则物理core id。

同时函数还调用set\_cpu\_present(cpu, true);把core设置成present状态，上面已经解释过改状态的含义了。

到smp\_init函数：

int num\_nodes, num\_cpus;

idle\_threads\_init();

cpuhp\_threads\_init();

pr\_info("Bringing up secondary CPUs ...\n");

bringup\_nonboot\_cpus(setup\_max\_cpus);

num\_nodes = num\_online\_nodes();

num\_cpus = num\_online\_cpus();

smp\_cpus\_done(setup\_max\_cpus);

idle\_threads\_init： 函数为corex创建一个idle进程，直白一点就是copy init\_task进程。这是一个手动创建的进程，详见kernel八问-第七问。

Cpuhp\_threads\_init： 函数为hotplug cpu创建一个线程，线程主体是smpboot\_thread\_fn

bringup\_nonboot\_cpus(setup\_max\_cpus)： 函数真正激活core的运行：

for\_each\_present\_cpu(cpu) {

if (num\_online\_cpus() >= setup\_max\_cpus)

break;

if (!cpu\_online(cpu))

cpu\_up(cpu, CPUHP\_ONLINE);

}

Present状态已经在前面设置好了，那么执行cpu\_up，因为现在core还不是ONLINE状态。

Cpu\_up函数前面处理的是hotplug方法，我们暂时略过。真正处理的地方是\_cpu\_up里面的

target = min((int)target, CPUHP\_BRINGUP\_CPU);

ret = cpuhp\_up\_callbacks(cpu, st, target);

首先target是CPUHP\_ONLINE，CPUHP\_BRINGUP\_CPU比target小，故target被赋值为CPUHP\_BRINGUP\_CPU。然后进入cpuhp\_up\_callbacks，这里的st是cpuhp\_state，每一个core一个独立的cpuhp\_state。

Cpuhp\_up\_callbacks主要是通过循环执行cpuhp\_hp\_states定义的startup.single或teardown.single函数。起始为CPUHP\_BRINGUP\_CPU。

Bringup\_cpu：

struct task\_struct \*idle = idle\_thread\_get(cpu); 取得当前cpu的idle进程。Idle进程已经在之前创建好，copy init\_task来的。

然后调用\_\_cpu\_up:

secondary\_data.task = idle;

secondary\_data.stack = task\_stack\_page(idle) + THREAD\_SIZE;

update\_cpu\_boot\_status(CPU\_MMU\_OFF);

\_\_flush\_dcache\_area(&secondary\_data, sizeof(secondary\_data));

设置core运行IDLE的task\_struct及栈信息。关MMU。

ret = boot\_secondary(cpu, idle);

启动指定的core。即调用smp\_spin\_table\_cpu\_boot， 这个函数执行下列语句：

write\_pen\_release(cpu\_logical\_map(cpu));

sev();

即写入hwid到secondary\_holding\_pen\_release这个地址。结合前面可以看到

ldr x4, [x3]

cmp x4, x0

b.eq secondary\_startup

开始执行secondary\_startup函数了。

Core0会等待corex的执行是否成功，通过cpu\_running来判断。等到core启动后唤醒等待在cpu\_running上的进程后，判断corex是否是online状态就能知道是否启动成功。

这是在secondary\_start\_kernel里面的：

set\_cpu\_online(cpu, true);

complete(&cpu\_running);

来通知的。现在指定的core就是ONLINE状态了。

接下来继续执行cpuhp\_hp\_states剩余的函数，我们比较关心的active状态呢？

那就是在CPUHP\_AP\_ACTIVE的startup.single里面设置的。



Spin-table基本上就是上述的方法，是不是感觉太low了？

接下来我们来看看比较高级的方式PSCI：