Workqueue work worker

-

**4.4 workqueue**

**4.4.1 基本概念（**http://kernel.meizu.com/linux-workqueue.html**）**

Workqueue 是内核里面很重要的一个机制，特别是内核驱动，一般的小型任务 (work) 都不会自己起一个线程来处理，而是扔到 Workqueue 中处理。Workqueue 的主要工作就是用进程上下文来处理内核中大量的小任务。

Workqueue 的主要设计思想：**并行，多个 work 不要相互阻塞；节省资源，多个 work 尽量共享资源 ( 进程、调度、内存 )，不要造成系统过多的资源浪费。**

为了实现的设计思想，workqueue 的设计实现也更新了很多版本。最新的实现叫做CMWQ(Concurrency Managed Workqueue)。关于 workqueue 中几个概念非常容易混淆，大概可以这样来理解：

work ：工作。

workqueue ：一组工作的集合。workqueue 和 work 一对多的关系。

worker ：工人。在代码中 worker 对应一个 work\_thread() 内核线程。

worker\_pool：一组工人的集合。worker\_pool 和 worker 一对多关系。

pwq(pool\_workqueue)：中间人 / 中介，负责建立起 workqueue 和 worker\_pool 之间的关系。workqueue 和 pwq 是一对多的关系，pwq 和 worker\_pool 是一对一的关系。(可以把qwp当做 worker\_pool中的包工头，代理人，专门找事做的)

最终的目的还是把 work( 工作 ) 传递给 worker( 工人 ) 去执行，中间的数据结构和各种关系目的是把这件事组织的更加清晰高效。

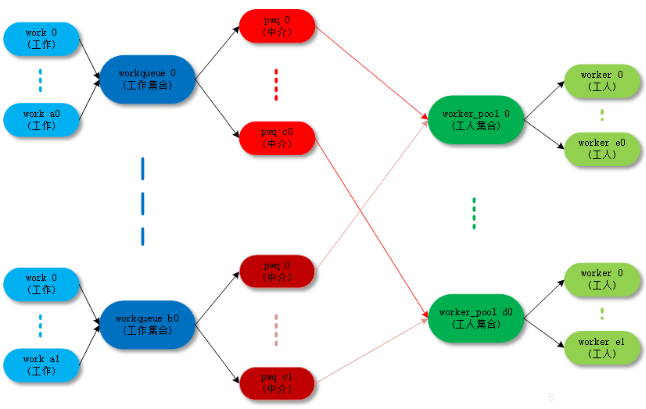
CMWQ的精髓就在worker\_pool里面worker的动态增减管理上 。CMWQ 对 worker\_pool 分成两类：

normal worker\_pool，给通用的 workqueue 使用；

unbound worker\_pool，给 WQ\_UNBOUND 类型的的 workqueue 使用。

默认 work 是在 normal worker\_pool 中处理的。系统的规划是每个 CPU 创建两个 normal worker\_pool：一个 normal 优先级 (nice=0)、一个高优先级 (nice=HIGHPRI\_NICE\_LEVEL)，对应创建出来的 worker 的进程 nice 不一样（创建线程命名为kworker/$cpu：$id）。

unbound worker\_pool 相对应的意思，就是 worker 可以在多个 CPU 上调度的。但是他其实也是绑定的，只不过它绑定的单位不是 CPU 而是 node。所谓的 node 是对 NUMA(Non Uniform Memory Access Architecture) 系统来说的，NUMA 可能存在多个 node，每个 node 可能包含一个或者多个 CPU。同样unbound worker\_pool也是对应创建两个worker\_pool 处理对应的优先级（创建线程命名为kworker/u$cpu：$id）。



**4.4.2 主要数据结构**

Workqueue主要包括如下重要的数据类型。

**Structure Definition**

typedef void (\*work\_func\_t)(struct work\_struct \*work);

struct work\_struct {

atomic\_long\_t data;

struct list\_head entry; // 用于将work挂载到队列上

work\_func\_t func; // 执行函数

};

Work\_struct用于表示一个work, 相对简单，注意这里的data字段，该字段的低比特部分（通常8 bit）为标志位，其他高比特位存放worker\_pool的ID或pool\_workqueue的指针，具体有WORK\_STRUCT\_PWQ标志决定。

而Work通常运行在worker线程中,worker数据结构相对复杂，我们取比较重要的几个字段。

struct worker {

......

struct work\_struct \*current\_work; // 当前处理的work

work\_func\_t current\_func; // 当前执行的work函数

struct pool\_workqueue \*current\_pwq; // 当前所属的pwq

struct list\_head scheduled; // worker调度并准备执行的work队列

struct task\_struct \*task; // 线程对应的task\_struct

struct worker\_pool \*pool; // 当前所属worker\_pool

struct list\_head node; // 对应worker\_pool的集合

unsigned long last\_active； // 最近一次执行的时间戳

unsigned int flags; // worker标志位

int id; // worker id

.......

};

CMWQ提出工作线程池的概念，用于管理worker，其数据结构如下

struct worker\_pool {

spinlock\_t lock; // 保护worker\_pool的同步锁

int cpu; // Normal-绑定cpu, Unbounded - -1

int node; // Unbounded, 所在内存节点

int id;

unsigned int flags;

struct list\_head worklist; // pending状态的work链表

int nr\_workers; // 工作线程数

int nr\_idle; // 处于idle状态的工作线程数

struct list\_head idle\_list; // 处于idle状态的worker 队列

struct list\_head workers; // 所管理的所有worker线程

struct ida worker\_ida; // woker与task的对应关系

struct workqueue\_attrs \*attrs; // worker的属性

atomic\_t nr\_running \_\_\_\_cacheline\_aligned\_in\_smp; //运行线程数

struct rcu\_head rcu; // RCU锁

......

} \_\_\_\_cacheline\_aligned\_in\_smp;

image2.png

Woker\_pool也是PER CPU 变量，每个CPU 维护两个worker\_pool分别处理低优先级和高优先级的工作线程。所以实际上系统上的worker\_pool总数是一定的，系统上所有的工作队列都对应这些worker\_pool,workqueue与worker\_pool通过pool\_workqueue关联起来。

struct pool\_workqueue {

struct worker\_pool \*pool; // 相关联的woker\_pool

struct workqueue\_struct \*wq; // 相关联的workqueue

int nr\_active; // 活跃的work数

int max\_active; // 可活跃的最大work数

struct list\_head delayed\_works; // 被延迟执行的work链表

struct rcu\_head rcu;

.......

} \_\_aligned(1 << WORK\_STRUCT\_FLAG\_BITS); //根据work标志位对齐

最后是workqueue机制中的workqueue\_struct结构。

struct workqueue\_struct {

struct list\_head pwqs; // 所有pwq的链表

struct list\_head list; // 系统全局管理workqueue的链表

struct mutex mutex; // 保护workqueue的互斥锁

struct list\_head maydays; // 所有rescue的pwq的链表

struct worker \*rescuer; // 所有rescue的worker的链表

struct workqueue\_attrs \*unbound\_attrs;

struct pool\_workqueue \*dfl\_pwq;

char name[WQ\_NAME\_LEN];

struct rcu\_head rcu;

unsigned int flags \_\_\_\_cacheline\_aligned;

struct pool\_workqueue \_\_percpu \*cpu\_pwqs;//per cpu pwq指针 代理

......

};

Work挂入workqueue后，最终需要通过worker\_pool中的worker进行处理，而worker\_pool是系统共享的有限资源，因此必须挑选一个合适的worker\_pool，而后从中选出合适的worker执行。

最后讲一下一些常用的标志位。

|  |  |
| --- | --- |
| WQ\_UNBOUND | WQ不与CPU绑定 |
| WQ\_FREEZABLE | WQ在等待期间是可冻结的 |
| WQ\_MEM\_RECLAIM | WQ处于内存回收的处理 |
| WQ\_HIGHPRI | 高优先级的WQ |
| WQ\_CPU\_INTENSIVE | CPU任务密集的WQ |
| WQ\_SYSFS | WQ对文件系统可见 |
| WQ\_POWER\_EFFICIENT | 考虑能耗问题，从调度和执行上减少消耗，对于BOUND类型，workqueue将是per cpu变量，变量不会进行跨CPU转移 |
| \_\_WQ\_DRAINING | WQ处于draining状态 |
| \_\_WQ\_ORDERED | WQ严格串行执行，不并行 |
| \_\_WQ\_LEGACY | 创建函数的标记 |
| \_\_WQ\_ORDERED\_EXPLICIT | 显式声明是串行执行的 |
| WQ\_MAX\_ACTIVE | 最大活跃WQ数 |
| WQ\_DFL\_ACTIVE | 默认活跃WQ数 |

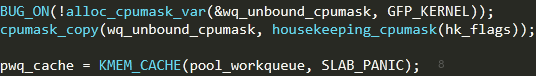
**4.4.3 初始化**

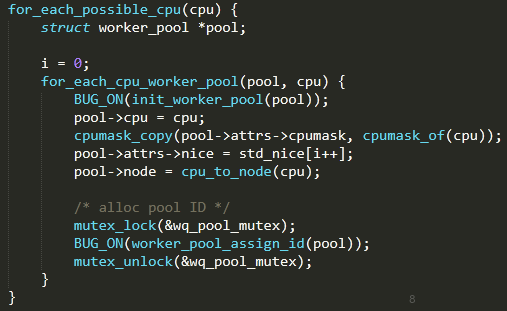
了解了相关数据结构，讲一下workqueue机制的初始化，workqueue的初始化在linux 4.9内核中分为两个阶段：

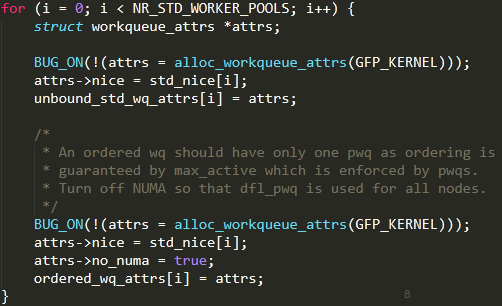
Workqueue\_init\_early: 该阶段进行相关的数据结构的内存分配，同时初始化worker\_pool, 初始化workqueue和work.

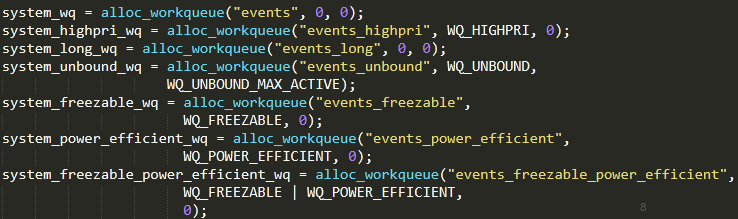
Workqueue\_init：创建worker线程，开始执行和调度work.

**Workqueue\_init\_early**

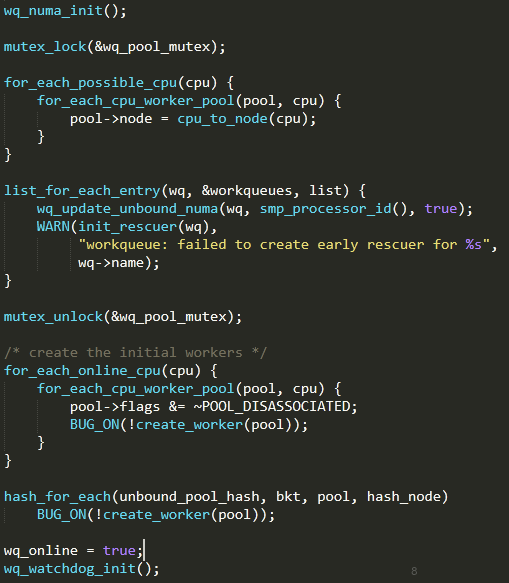








**Workqueue\_init**



#**define** **for\_each\_cpu\_worker\_pool**([pool](http://opengrok.scm.adc.com:8080/source/s?defs=pool&project=SM8250_Q_Master), [cpu](http://opengrok.scm.adc.com:8080/source/s?defs=cpu&project=SM8250_Q_Master)) \

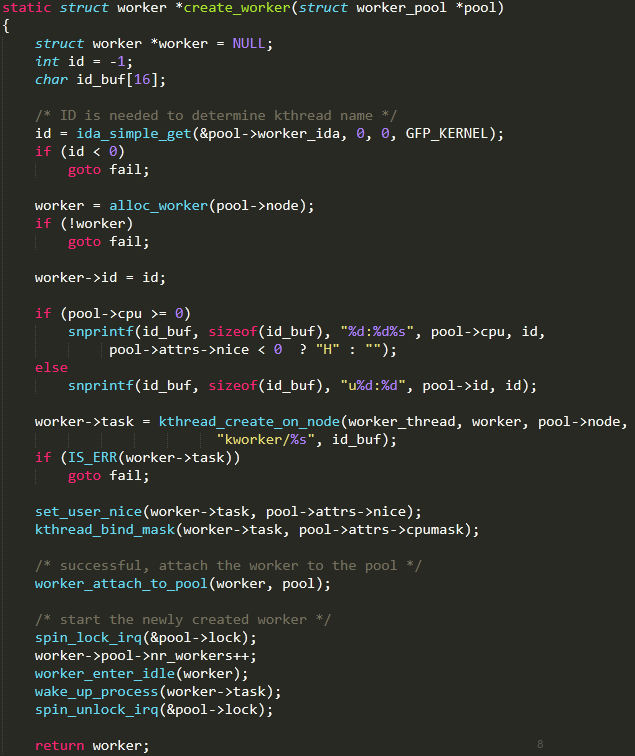
**for** (([pool](http://opengrok.scm.adc.com:8080/source/s?defs=pool&project=SM8250_Q_Master)) = &per\_cpu(cpu\_worker\_pools, [cpu](http://opengrok.scm.adc.com:8080/source/s?defs=cpu&project=SM8250_Q_Master))[0]; \

[(poo](http://opengrok.scm.adc.com:8080/source/s?defs=pool&project=SM8250_Q_Master)l) < &per\_cpu(cpu\_worker\_pools, [cp](http://opengrok.scm.adc.com:8080/source/s?defs=cpu&project=SM8250_Q_Master)u)[**[NR\_STD\_WORKER\_POOL**](http://opengrok.scm.adc.com:8080/source/xref/SM8250_Q_Master/android/kernel/msm-4.19/kernel/workqueue.c)S]; \

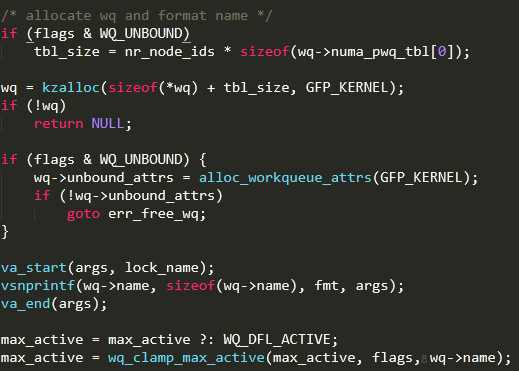
[(poo](http://opengrok.scm.adc.com:8080/source/s?defs=pool&project=SM8250_Q_Master)l)++)

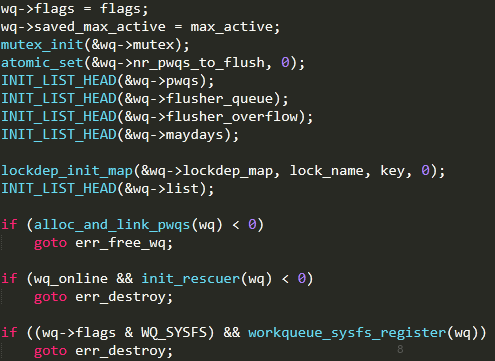
初始化的大致框架如上所示，为了更好认识workqueue的初始化，我们解析两个重要函数，create\_worker与alloc\_workqueue。

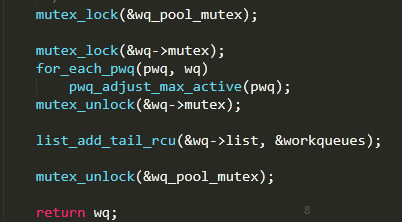
**Create\_worker**



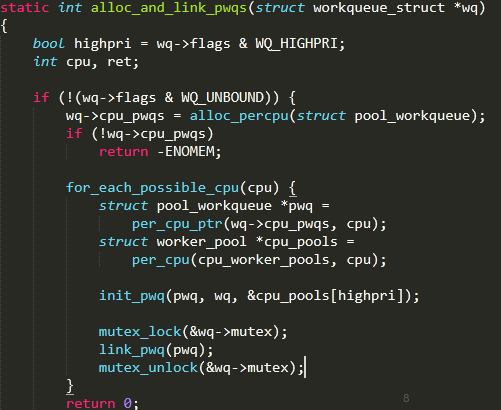
**Alloc\_workqueue**

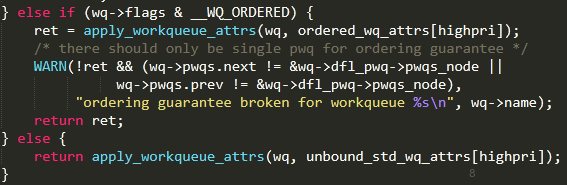






接下来我们在剖析其中与pwq有关的函数alloc\_and\_link\_pwqs。





解析apply\_workqueue\_attrs的调用栈比较深，涉及到大量代码，这里理一下主要逻辑，有兴趣可以自己看。

查找或创建一个属性匹配的默认pwq,而后为每个online的节点创建或查找一个pwq, 更新wq属性

通过RCU机制更新之前创建的workqueue的pwq，以及关联关系

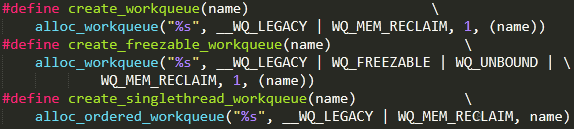
释放RCU的旧数据块

当然这些过程都保证了临界区的保护，所以可以看见许多锁机制的使用。

以上workqueue的初始化过程完毕，本节最后讲一下workqueue提供使用的初始化接口函数，用于创建不同类别的workqueue。

image14.png

image15.png



这些接口无一例外基于\_\_alloc\_workqueue接口，后三个是为了兼容旧接口留下的接口。

**4.4.4 调度**

Linux 内核建议使用系统默认的workqueue,而不是自定义的workqueue, 但work是完全可以自定义的。

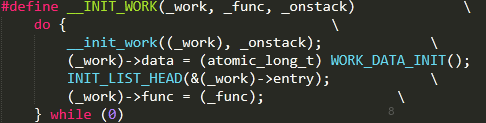
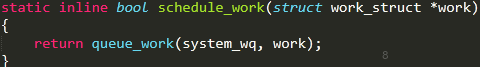


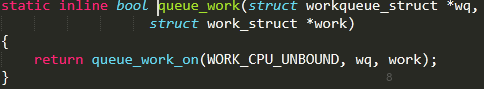
image18.png

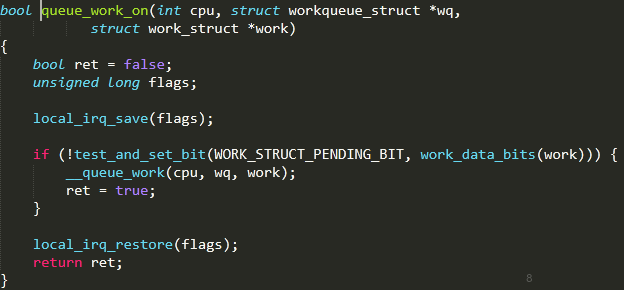
Work创建的函数如上，当然还有其他API，但是基本类似。

Work创建完成后，我们就可以通过schedule\_work将其挂入系统默认的workqueue中。

**Schedule\_work**

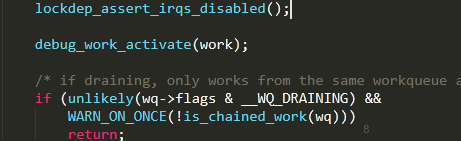




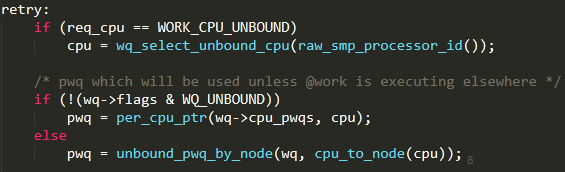


Schdule\_work的调用链比较明显，最终的关键函数为\_\_queue\_work，默认CPU类型为UNBOUND不绑定，同时调度期间关中断，并将work设为pending状态标记进入调度，若已被调度则退出。**采用系统的system\_wq。**

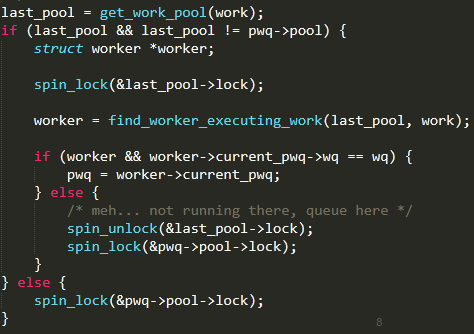
**\_\_queue\_work**



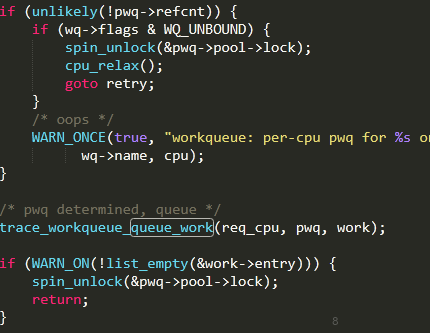
对于draining状态的workqueue,除非work来自同一workqueue，否则立即返回。



对于UNBOUND类型work，wq为其选择一个cpu，对于BOUND类型则使用绑定的CPU，而后取出cpu对应的node的pwq。

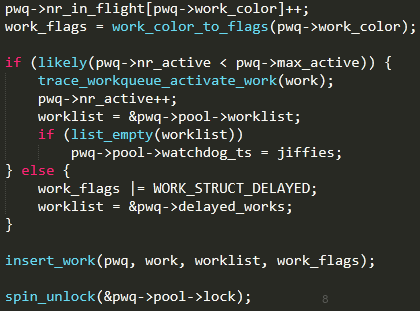


获取work所在的worker\_pool, 而后比较当前分配的pwq所关联的pool是否一致，一致则对pool加锁进入临界区。否则，再获取last\_pool中执行work的worker,如果该线程当前对应的pwq所在wq为调度所在wq(即为chained work)则修正pwq为worker所对应pwq,否则以调度分配的pwq为准。

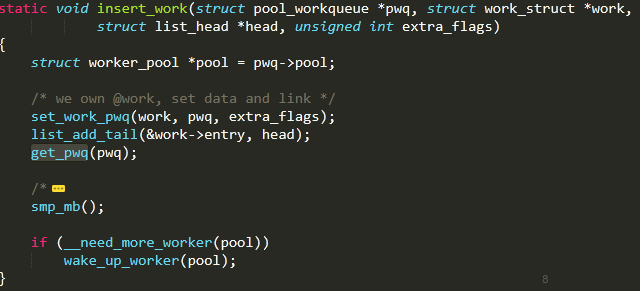


若pwq的引用数为0(未初始化)且WQ属于UNBOUND类型，跳转回retry执行；对于Bound类型直接使用本地CPU对应pwq即可。

若work已经被挂载，说明已被调度，返回；



若当前pwq活跃任务数少于最大限制，将其挂入worklist待执行，否则挂入delayed\_works延迟执行。注意worklist是指针，决定work挂载的去向，这里通过分支判断赋予了不同的list。挂载具体动作由insert\_work完成。



除了设置work的queue和添加到worklist外，对所在pwq增加引用数，此时若需要pool中更多的worker则唤醒worker。

整体而言，schedule\_work函数所做的工作如下：

设置work的WORK\_STRUCT\_PENDING\_BIT标志位，若已设置则返回

寻找合适的CPU和pwq,若是bound类型则直接使用绑定的cpu与pwq

找到pwq后，寻找worker\_pool和对应的worklist

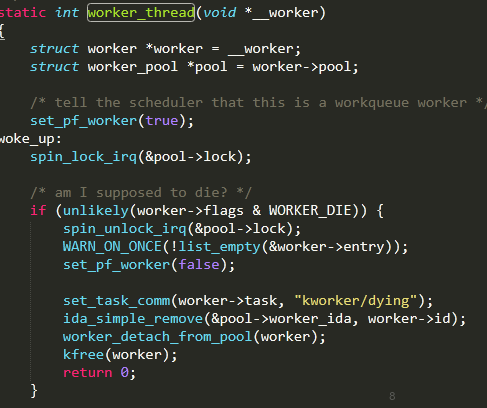
如果当前运行的work数大于0，唤醒一个worker线程

**4.4.5 执行**

回忆我们create\_worker时创建线程时的代码，现在我们看一下worker的执行过程。

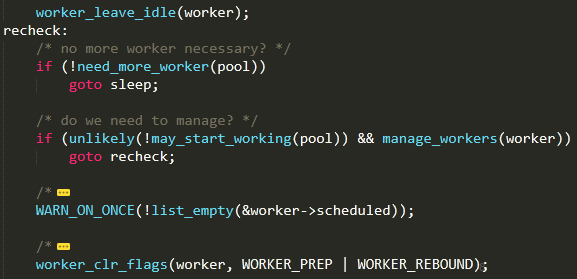
**Worker\_thread**

image28.png



1）设置当前线程的标志位告知线程调度器本线程为worker线程

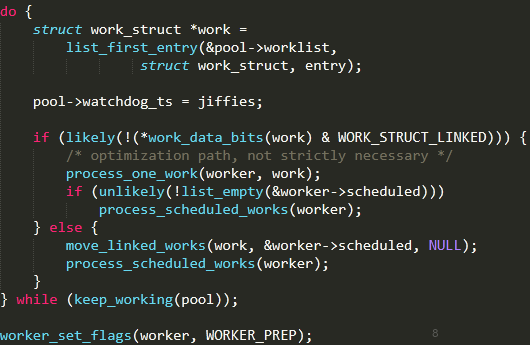
2）若当前线程已经处于僵死态，则从pool中移除并释放内存资源



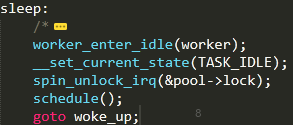
3）worker线程离开idle状态

4）若当前线程没有需要运行的work,跳转至步骤（7）

5）若当前没有闲置的worker但进行了管理，返回recheck



若当前work是linked work，则将其移至scheduled链表再整个处理scheduled链表中的work;否则先处理当前work,然后处理scheduled work。当前worker pool已无正在运行的worker时离开循环

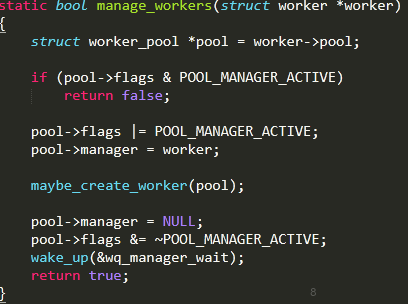


Worker重新进入闲置状态，设置current状态为IDLE,释放自旋锁进行线程调度

当且仅当调度期间worker线程可被抢占\中断，单独完成后，返回wokeup处忙等自旋锁直至获得锁再次开始执行。

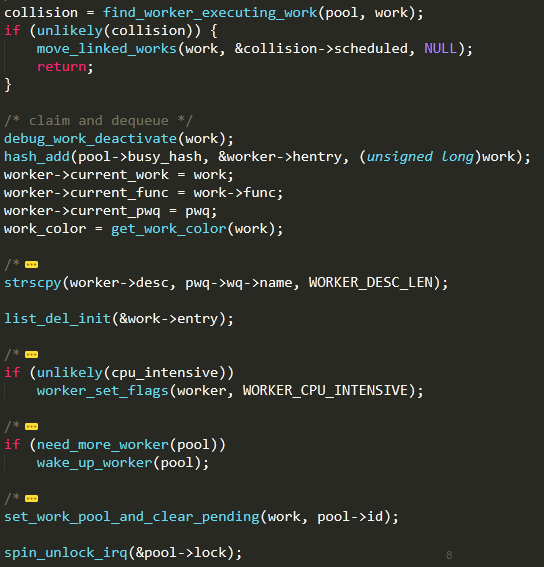
注意一下manage\_worker与process\_one\_work的实现。

**Manage\_workers**



若当前woker所属pool正在被管理，则返回false，否则设置标志位, 根据需要可能创建线程，然后清除标志位，返回true

**Process\_one\_work**



若当前work已在其他线程运行，将其移入到scheduled list并返回

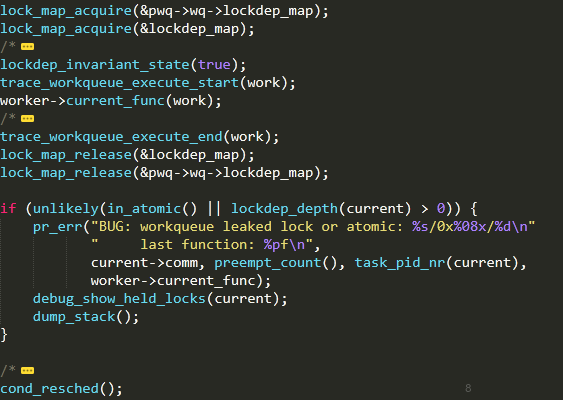
将当前work添加到pool的busy\_hash中（说明该work急需执行，同时将work分派给当前worker线程使其立即执行

若WQ是CPU密集的，将worker也设置为CPU密集的

若当前没有运行线程，则唤醒一个worker线程

确认上述执行完毕后，清除worker的pending位

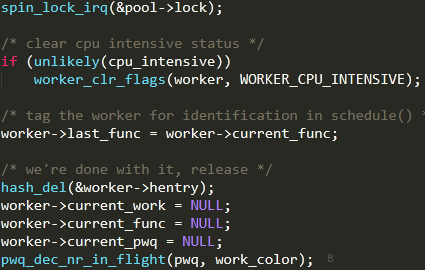
开中断，之前属于不可中断的原子操作，此时说明workqueue执行work期间可被中断



正式执行任务worker->current\_func(work)

Cond\_resched让出cpu（不急着立刻释放本work的状态）

关中断，最后的清理操作属于原子操作

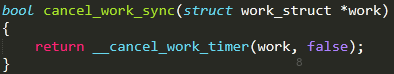


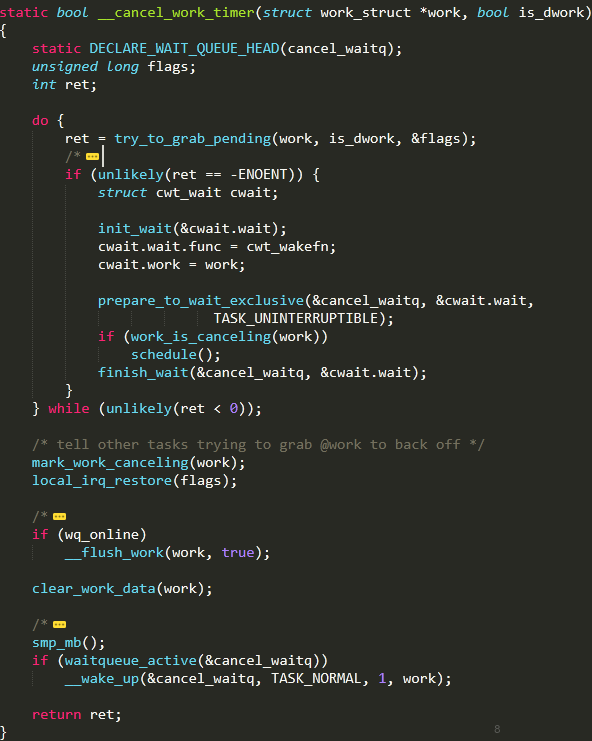
**4.4.6 删除与刷新**

介绍了创建、调度、执行，那么如何删除我们的work并刷新workqueue状态呢？在使用上，可以调用如下的：

Cancel\_work\_sync(work);

**Cancel\_work\_sync**





这里\_\_flush\_work的实际作用在于插入一个barr的task于目标work之后,而后等待barr的完成，当barr完成说明目标task也已完成。即实现了必要的刷新操作。

Tops by wen:

TOP1:

workerPool是一个percpu变量，每个CPU对应有两个worker\_pool ,在系统初始化就已经创建好，worker\_pool数量固定，系统上所有创建的workqueue都对应这些worker\_pool：(NR\_STD\_WORKER\_POOLS:2)

一个针对普通的workqueue，一个针对高优先级workqueue。

worker\_pool 分成两类：

Pool\_workqueue也是一个percpu的变量

即一个workqueue在每个CPU上都有对应的pwq，每创建一个workqueue 就会创建一个percpu的pool\_workqueue, pool\_workqueue对应到已经存在pcpu对应的worker\_pool上。(pool\_workqueue说白了是workqueue在每个cpu上的代理，与对应cpu的worker\_pool对接； 而worker\_pool数量是固定的，pool\_workqueue随着workquque增加而增加)

Schedule\_work后，如果对应cpu的pwq活跃任务数少于最大限制，将其挂入对应worker\_list的worklist待执行，否则挂入pwq的delayed\_works延迟执行,同时pwq增加引用数；

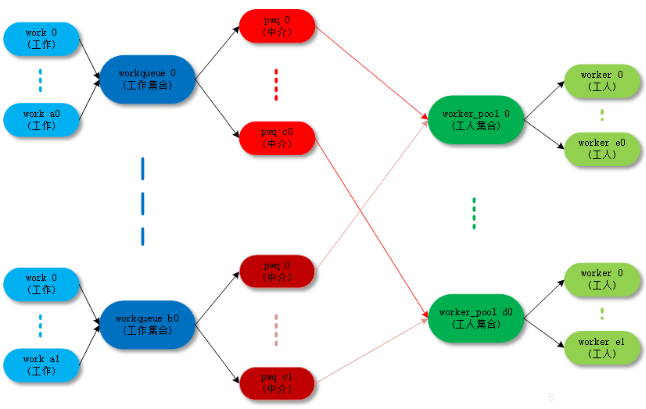
整体而言，schedule\_work工作如下：

设置work的WORK\_STRUCT\_PENDING\_BIT标志位，若已设置则返回

寻找合适的CPU和pwq,若是bound类型则直接使用绑定的cpu与pwq

找到pwq后，寻找worker\_pool和对应的worklist，nr\_activer++

如果当前运行的work数大于0，唤醒一个worker线程



TOP2:

worker\_pool:

worker是具体处理work的对象，系统把worker作为一种资源管理，提出了worker\_pool的概念，一个worker必定会属于某个worker\_pool；

Worker\_pool 两个任务：

1管理worker：

idle\_list 表： 对应空闲的worker (entry)

busy\_hash表：正在执行任务的worker (hentry)

nr\_workers和nr\_idle代表worker和idle worker的数量

2 用户提交的work(对应cpu上各个qwp来的work)

worklis表： list of pending works

static DEFINE\_PER\_CPU\_SHARED\_ALIGNED(struct worker\_pool [NR\_STD\_WORKER\_POOLS],

cpu\_worker\_pools);

Pool\_workqueue:

,workqueue与worker\_pool通过pool\_workqueue关联起来。

struct pool\_workqueue {

struct worker\_pool \*pool; // 相关联的woker\_pool

struct workqueue\_struct \*wq; // 相关联的workqueue

int nr\_active; // 活跃的work数

int max\_active; // 可活跃的最大work数

int nr\_in\_flight[WORK\_NR\_COLORS];

struct list\_head delayed\_works; // 被延迟执行的work链表

struct rcu\_head rcu;

.......

} \_\_aligned(1 << WORK\_STRUCT\_FLAG\_BITS); //根据work标志位对齐

最后是workqueue机制中的workqueue\_struct结构。

struct workqueue\_struct {

struct list\_head pwqs; // 所有pcpu pwq的链表

struct list\_head list; // 系统全局管理workqueue的链表

struct mutex mutex; // 保护workqueue的互斥锁

struct list\_head maydays; // 所有rescue的pwq的链表

struct worker \*rescuer; // 所有rescue的worker的链表

struct workqueue\_attrs \*unbound\_attrs;

struct pool\_workqueue \*dfl\_pwq;

char name[WQ\_NAME\_LEN];

struct rcu\_head rcu;

unsigned int flags \_\_\_\_cacheline\_aligned;

struct pool\_workqueue \_\_percpu \*cpu\_pwqs;//per cpu pwq指针

......

};

\_\_alloc\_workqueue\_key

\_\_alloc\_workqueue\_key函数的一开始有如下的代码：

if ((flags & WQ\_POWER\_EFFICIENT) && wq\_power\_efficient)
        flags |= WQ\_UNBOUND;

在kernel中，有两种线程池，一种是线程池是per cpu的，也就是说，系统中有多少个cpu，就会创建多少个线程池，cpu x上的线程池创建的worker线程也只会运行在cpu x上。另外一种是unbound thread pool，该线程池创建的worker线程可以调度到任意的cpu上去。由于cache locality的原因，per cpu的线程池的性能会好一些，但是对power saving有一些影响。设计往往如此，workqueue需要在performance和power saving之间平衡，想要更好的性能，那么最好让一个cpu上的worker thread来处理work，这样的话，cache命中率会比较高，性能会更好。但是，从电源管理的角度来看，最好的策略是让idle状态的cpu尽可能的保持idle，而不是反复idle，working，idle again。

我们来一个例子辅助理解上面的内容。在t1时刻，work被调度到CPU A上执行，t2时刻work执行完毕，CPU A进入idle，t3时刻有一个新的work需要处理，这时候调度work到那个CPU会好些呢？是处于working状态的CPU B还是处于idle状态的CPU A呢？如果调度到CPU A上运行，那么，由于之前处理过work，其cache内容新鲜热辣，处理起work当然是得心应手，速度很快，但是，这需要将CPU A从idle状态中唤醒。选择CPU B呢就不存在将CPU 从idle状态唤醒，从而获取power saving方面的好处。

了解了上面的基础内容之后，我们再来检视per cpu thread pool和unbound thread pool。当workqueue收到一个要处理的work，如果该workqueue是unbound类型的话，那么该work由unbound thread pool处理并把调度该work去哪一个CPU执行这样的策略交给系统的调度器模块来完成，对于scheduler而言，它会考虑CPU core的idle状态，从而尽可能的让CPU保持在idle状态，从而节省了功耗。因此，如果一个workqueue有WQ\_UNBOUND这样的flag，则说明该workqueue上挂入的work处理是考虑到power saving的。如果workqueue没有WQ\_UNBOUND flag，则说明该workqueue是per cpu的，这时候，调度哪一个CPU core运行worker thread来处理work已经不是scheduler可以控制的了，这样，也就间接影响了功耗。

有两个参数可以控制workqueue在performance和power saving之间的平衡：

1、各个workqueue需要通过WQ\_POWER\_EFFICIENT来标记自己在功耗方面的属性

2、系统级别的内核参数workqueue.power\_efficient。

使用workqueue的用户知道自己在电源管理方面的特点，如果该workqueue在unbound的时候会极大的降低功耗，那么就需要加上WQ\_POWER\_EFFICIENT的标记。这时候，如果没有标记WQ\_UNBOUND，那么缺省workqueue会创建per cpu thread pool来处理work。不过，也可以通过workqueue.power\_efficient这个内核参数来修改workqueue的行为：

**static** **bool** **wq\_power\_efficient** = [IS\_ENABLED](http://opengrok.scm.adc.com:8080/source/s?defs=IS_ENABLED&project=SM8250_R_Master)([CONFIG\_WQ\_POWER\_EFFICIENT\_DEFAULT](http://opengrok.scm.adc.com:8080/source/s?defs=CONFIG_WQ_POWER_EFFICIENT_DEFAULT&project=SM8250_R_Master));

[module\_param\_named](http://opengrok.scm.adc.com:8080/source/s?defs=module_param_named&project=SM8250_R_Master)([power\_efficient](http://opengrok.scm.adc.com:8080/source/s?defs=power_efficient&project=SM8250_R_Master), [**wq\_power\_efficient**](http://opengrok.scm.adc.com:8080/source/xref/SM8250_R_Master/android/kernel/msm-4.19/kernel/workqueue.c), **bool**, 0444);

如果wq\_power\_efficient设定为true，那么WQ\_POWER\_EFFICIENT的标记的workqueue就会强制按照unbound workqueue来处理，即使没有标记WQ\_UNBOUND。Android没有enable这个属性

per cpu的workqueue的pool workqueue对应的线程池也是per cpu的，每个cpu有两个线程池（normal和high priority），因此将pool workqueue和thread pool对应起来是非常简单的事情。对于unbound workqueue，对应关系没有那么直接，如果属性相同，多个unbound workqueue的pool workqueue可能对应一个thread pool。

系统使用哈希表来保存所有的unbound worker thread pool，定义如下：

static DEFINE\_HASHTABLE(unbound\_pool\_hash, UNBOUND\_POOL\_HASH\_ORDER);

在创建unbound workqueue的时候，pool workqueue对应的worker thread pool需要在这个哈希表中搜索，如果有相同属性的worker thread pool的话，那么就不需要创建新的线程池，代码如下：

hash\_for\_each\_possible(unbound\_pool\_hash, pool, hash\_node, hash) {
    if (wqattrs\_equal(pool->attrs, attrs)) { －－－－检查属性是否相同
        pool->refcnt++;
        return pool; －－－－－－－在哈希表找到适合的unbound线程池
    }
}如果没有相同属性的thread pool，那么需要创建一个并挂入哈希表。