Linux工作队列实现机制

**工作项、工作队列和工作者线程**

把推后执行的任务叫做工作（work），描述它的数据结构为work\_struct ，

这些工作以队列结构组织成工作队列（workqueue），其数据结构为workqueue\_struct ，

而工作线程就是负责执行工作队列中的工作。系统默认的工作者线程为events。

工作队列(work queue)是另外一种将工作推后执行的形式。工作队列可以把工作推后，交由一个内核线程去执行—这个下半部分总是会在进程上下文执行，但由于是内核线程，其不能访问用户空间。最重要特点的就是工作队列允许重新调度甚至是睡眠。

工作队列和软中断/tasklet中作出选择非常容易。可使用以下规则：

1.如果推后执行的任务需要睡眠，那么只能选择工作队列；

2.如果推后执行的任务需要延时指定的时间再触发，那么使用工作队列，因为其可以利用 timer延时；

3.如果推后执行的任务需要在一个tick之内处理，则使用软中断或tasklet，因为其可以抢占普通进程和内核线程；

4.如果推后执行的任务对延迟的时间没有任何要求，则使用工作队列，此时通常为无关紧要的任务。

实际上，工作队列的本质就是将工作交给内核线程处理，因此其可以用内核线程替换。但是内核线程的创建和销毁对编程者的要求较高，而工作队列实现了内核线程的封装，不易出错，所以我们也推荐使用工作队列。

**工作队列使用**

相关文件：

kernel/include/linux/workqueue.h

Kernel/kernel/workqueue.c

**工作队列的创建**

要使用工作队列，需要先创建工作项，有两种方式：

1）静态创建：

DECLARE\_WORK(name,function); 定义正常执行的工作项

DECLARE\_DELAYED\_WORK(name,function); 定义延后执行的工作项

2）动态创建，运行时创建：

通常在probe()函数中执行下面的操作来初始化工作项：

INIT\_WORK(&work, new\_ts\_work);

INIT\_DELAYED\_WORK(&led\_work,s0340\_ledtime\_scanf);

工作队列待执行的函数原型是：

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

这个函数会由一个工作者线程执行，因此，函数会运行在进程上下文中。默认情况下，允许响应中断，并且不持有任何锁。如果需要，函数可以睡眠。需要注意的是，尽管该函数运行在进程上下文中，但它不能访问用户空间，因为内核线程在用户空间没有相关的内存映射。通常在系统调用发生时，内核会代表用户空间的进程运行，此时它才能访问用户空间，也只有在此时它才会映射用户空间的内存。

创建了工作项之后，在适当的时候可以通过下面的两种方式来提交工作项给工作者线程，通常我们使用的工作队列和工作者线程都是系统初始化时候默认创建的。

**工作队列的调度运行**

schedule\_work(&work)；

&work马上就会被调度，一旦其所在的处理器上的工作者线程被唤醒，它就会被执行。

schedule\_delayed\_work(&delay\_work,delay);

&delay\_work指向的delay\_work直到delay指定的时钟节拍用完以后才会执行。

eg:

schedule\_delayed\_work(&kpd\_backlight\_work,msecs\_to\_jiffies(300));

**默认工作队列和工作者线程创建过程**

系统默认的工作队列名称是：keventd\_wq,默认的工作者线程叫：events/n，这里的n是处理器的编号,每个处理器对应一个线程。比如，单处理器的系统只有events/0这样一个线程。而双处理器的系统就会多一个events/1线程。

默认的工作者线程会从多个地方得到被推后的工作。许多内核驱动程序都把它们的下半部交给默认的工作者线程去做。

默认的工作队列keventd\_wq只有一个，但是其工作者线程在每一个cpu上都有。而标记为singlethread的工作者线程最存在于一个cpu上。

关于默认工作队列keventd\_wq和工作者线程events/n的建立在文件Kernel/kernel/workqueue.c中实现。

Start\_kernel()-->rest\_init(),该函数中创建了两个内核线程kernel\_init和kthreadd，这两个线程都和本文描述的部分有关系，先说说kernel\_init🡺。

kernel\_init()-->do\_basic\_setup()-->init\_workqueues(),该函数中创建了上面提到的默认工作队列和工作者线程。

init\_workqueues()-->

-->hotcpu\_notifier(workqueue\_cpu\_callback,0);

-->keventd\_wq=create\_workqueue("events");

注册的cpu通知链cpu\_chain上的回调函数是workqueue\_cpu\_callback(),

raw\_notifier\_call\_chain()函数用来调用cpu\_chain上的所有回调函数。

这里主要关注的是函数：

create\_workqueue("events");

=============================================================

@kernel/include/linux/workqueue.h

#define \_\_create\_workqueue(name,singlethread,freezeable,rt)/

\_\_create\_workqueue\_key((name),(singlethread),(freezeable),(rt),/NULL,NULL)

#define create\_workqueue(name) \_\_create\_workqueue((name),0,0,0)

#define create\_rt\_workqueue(name) \_\_create\_workqueue((name),0,0,1)

#define create\_freezeable\_workqueue(name) \_\_create\_workqueue((name),1,1,0)

#define create\_singlethread\_workqueue(name) \_\_create\_workqueue((name),1,0,0)

========================================================================

从宏\_\_create\_workqueue的参数可以看出，可以通过传递不同的参数:是否单cpu线程，是否可冻结，是否实时来创建不同类型的工作队列和工作者线程。

work\_struct工作项结构体定义：@kernel/include/linux/workqueue.h

工作队列workqueue\_struct结构体：@kernel/kernel/workqueue.c

分析1：

关键函数\_\_create\_workqueue\_key()分析：

struct workqueue\_struct \*\_\_create\_workqueue\_key(const char \*name,

int singlethread,

int freezeable,

int rt,

struct lock\_class\_key \*key,

const char \*lock\_name)

………………………

wq->singlethread = singlethread;

// 创建单模块线程

cwq = init\_cpu\_workqueue(wq, singlethread\_cpu); note -1

//初始化cpu\_workqueue\_struct结构体 cwq

// singlethread\_cpu -- the first cpu in a cpumask

INIT\_LIST\_HEAD(&cwq->worklist);// 初始化工作项列表, 使用时提交的工作项都是 挂接在这个链表上的 // kernel/kernel/workqueue.c

err = create\_workqueue\_thread(cwq, singlethread\_cpu); note 0

p = kthread\_create(worker\_thread, cwq, fmt, wq->name, cpu);

list\_add\_tail(&create.list, &kthread\_create\_list);

// 将新建的kthread挂接到全局的线程链表kthread\_create\_list中

wake\_up\_process(kthreadd\_task);

// kthreadd\_task = find\_task\_by\_pid\_ns(pid, &init\_pid\_ns);

//@ kernel/init/main.c // rest\_init()中初始化，该指针保存的是线程kthreadd的 //task\_struct结构体指针。 唤醒线程kthreadd

wait\_for\_completion(&create.done);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// 等待完成量，我们这里转到kthreadd线程的执行函数中去看一下，这个完成量的唤醒应该是在kthreadd线程中做的，kthreadd线程应该是根据kthread\_create\_list上挂接的kthread\_create\_info结构体来创建特定线程。

// 这部分关于内核线程创建的机制请阅读分析文档：内核线程创建 目录中的相关文件和内核源码。这里不再详细分析。

// 新线程创建ok后，进入了睡眠，然后唤醒了对应的完成量create.done，这边继续执行

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

trace\_workqueue\_creation(cwq->thread, cpu);

start\_workqueue\_thread(cwq, -1); // run this thread

list\_add(&wq->list, &workqueues);

……………………..

kthread\_create()函数通过专门创建线程的内核线程kthreadd创建了公用线程kthread，而在该kthread线程函数中调用其参数传递进来的回调函数threadfn()，这个threadfn()函数就是我们调用kthread\_create()函数时传递进来的第一个参数，第二个参数则是执行回调函数时的参数。

kthread\_create(worker\_thread, cwq, fmt, wq->name, cpu);

//线程创建ok之后，会在线程kthread中调用函数worker\_thread(cwq);

worker\_thread()函数如下，是每一个工作者线程的共用的线程函数。其实工作队列对应的数据结构是workqueue\_struct，而该结构体中包含一个对应cpu的数据结构cpu\_workqueue\_struct，这个数据结构中包含了工作项链表worklist。而所有的工作者线程，只是名字不一样而已，所跑的线程函数都是一样：worker\_thread

static int worker\_thread(void \*\_\_cwq)

{

struct cpu\_workqueue\_struct \*cwq = \_\_cwq;

DEFINE\_WAIT(wait);// 定义一个等待队列项wait

// @ kernel/include/linux/wait.h

if (cwq->wq->freezeable)

set\_freezable();// current->flags &= ~PF\_NOFREEZE;

for (;;) {

prepare\_to\_wait(&cwq->more\_work, &wait, TASK\_INTERRUPTIBLE);

// 可中断睡眠

// 准备进入睡眠等待，wait加入cwq->more\_work等待队列头中，

// 设置非独占进程标志

// 和可中断睡眠标志 @ kernel/kernel/wait.c

if (!freezing(current) &&  !kthread\_should\_stop() && list\_empty(&cwq->worklist))//

     // 当前进程是非冻结状态，当前线程没停止，同时工作项列表为空

     // 的时候进入睡眠让出cpu

schedule();

finish\_wait(&cwq->more\_work, &wait);// 当前线程被唤醒后马上要做的事情

try\_to\_freeze();

if (kthread\_should\_stop())// 检查当前线程是否被要求stop

break;

run\_workqueue(cwq);// 运行工作项中对应的函数

}

return 0;

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

提交工作项:

schedule\_work(struct work\_struct \*work);

queue\_work(keventd\_wq, work);// keventd\_wq工作队列是在函数init\_workqueues()中创 建的

queue\_work\_on(get\_cpu(), wq, work);

//该函数将work工作项提交到当前做该项提交的cpu上的工作队列wq上，如果这个cpu被标记为die，那么可以提交到别的cpu上去执行。返回0，表示该项工作已经提交过，还没执行。非0表示提交成功//

\_\_queue\_work(wq\_per\_cpu(wq, cpu), work);

insert\_work(cwq, work, &cwq->worklist);

list\_add\_tail(&work->entry, head);// 工作项加入链表

wake\_up(&cwq->more\_work);// 唤醒等待在该等待队列头上的所有等 待队列项

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

run\_workqueue

work\_func\_t f = work->func;// 取出工作项函数

cwq->current\_work = work;

list\_del\_init(cwq->worklist.next);// 从链表中删除工作项节点

f(work);

// 执行对应的工作项函数，将work\_struct结构体指针作为参数传递进去

以上内容是通过创建系统默认的工作队列keventd\_wq和工作者线程events/n来分析了其创建过程，提交工作项过程和提交工作后唤醒工作者线程之后的所做的动作

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

可以使用这些接口来创建独立的工作队列和工作者线程来专门为特定的任务服务