## **eventfd是什么**

eventfd是一个用来通知事件的文件描述符。它是内核向用户空间的应用发送通知的机制，可以有效地被用来实现用户空间的事件/通知驱动的应用程序。

## **eventfd接口**

对于eventfd，只有一个系统调用接口

**int** **eventfd**(**unsigned** **int** initval, **int** flags);

创建一个eventfd对象，或者说打开一个eventfd的文件，类似普通文件的open操作。

该对象是一个内核维护的无符号的64位整型计数器。初始化为initval的值。

flags可以以下三个标志位的OR结果：

* EFD\_CLOEXEC：FD\_CLOEXEC，简单说就是fork子进程时不继承，对于多线程的程序设上这个值不会有错的。
* EFD\_NONBLOCK：文件会被设置成O\_NONBLOCK，一般要设置。
* EFD\_SEMAPHORE：（2.6.30以后支持）支持semophore语义的read，简单说就值递减1。

这个新建的fd的操作很简单：

read(): 读操作就是将counter值置0，如果是semophore就减1。

write(): 设置counter的值。

注意，还支持epoll/poll/select操作，当然，以及每种fd都都必须实现的close。

## **eventfd实现高性能消费者线程池**

本实例采用以下设计：生产者创建eventfd并在事件循环中注册事件；消费者线程池中的线程共用一个epoll对象，每个消费者线程并行地进行针对eventfd触发的事件循环的轮询（epoll\_wait）。

**typedef** **struct** thread\_info {

pthread\_t thread\_id;

**int** rank;

**int** epfd;} thread\_info\_t;

**static** **void** **\*producer\_routine**(**void** **\***data) {

**struct** thread\_info **\***p **=** (**struct** thread\_info **\***)data;

**struct** epoll\_event event;

**int** epfd **=** p**->**epfd;

**int** efd **=** **-**1;

**int** ret **=** **-**1;

log("Greetings from [producer-%d]", p**->**rank);

**while** (1) {

sleep(1);

*// create eventfd (no reuse, create new every time)* efd **=** eventfd(1, EFD\_CLOEXEC**|**EFD\_NONBLOCK);

**if** (efd **==** **-**1) handle\_error("eventfd create: %s", strerror(errno));

*// register to poller* event.data.fd **=** efd;

event.events **=** EPOLLIN **|** EPOLLET; *// Edge-Triggered* ret **=** epoll\_ctl(epfd, EPOLL\_CTL\_ADD, efd, **&**event);

**if** (ret **!=** 0) handle\_error("epoll\_ctl");

*// trigger (repeatedly)* write(efd, (**void** **\***)0xffffffff, **sizeof**(uint64\_t));

}}

**static** **void** **\*consumer\_routine**(**void** **\***data) {

**struct** thread\_info **\***c **=** (**struct** thread\_info **\***)data;

**struct** epoll\_event **\***events;

**int** epfd **=** c**->**epfd;

**int** nfds **=** **-**1;

**int** i **=** **-**1;

uint64\_t result;

log("Greetings from [consumer-%d]", c**->**rank);

events **=** calloc(MAX\_EVENTS\_SIZE, **sizeof**(**struct** epoll\_event));

**if** (events **==** NULL) handle\_error("calloc epoll events\n");

**for** (;;) {

nfds **=** epoll\_wait(epfd, events, MAX\_EVENTS\_SIZE, 1000); *// poll every second* **for** (i **=** 0; i **<** nfds; i**++**) {

**if** (events[i].events **&** EPOLLIN) {

log("[consumer-%d] got event from fd-%d", c**->**rank, events[i].data.fd);

*// consume events (reset eventfd)* read(events[i].data.fd, **&**result, **sizeof**(uint64\_t));

close(events[i].data.fd); *// NOTE: need to close here* }

}

}}

**int** **main**(**int** argc, **char** **\***argv[]) {

**struct** thread\_info **\***p\_list **=** NULL, **\***c\_list **=** NULL;

**int** epfd **=** **-**1;

**int** ret **=** **-**1, i **=** **-**1;

*// create epoll fd* epfd **=** epoll\_create1(EPOLL\_CLOEXEC);

**if** (epfd **==** **-**1) handle\_error("epoll\_create1: %s", strerror(errno));

*// producers* p\_list **=** calloc(NUM\_PRODUCERS, **sizeof**(**struct** thread\_info));

**if** (**!**p\_list) handle\_error("calloc");

**for** (i **=** 0; i **<** NUM\_PRODUCERS; i**++**) {

p\_list[i].rank **=** i;

p\_list[i].epfd **=** epfd;

ret **=** pthread\_create(**&**p\_list[i].thread\_id, NULL, producer\_routine, **&**p\_list[i]);

**if** (ret **!=** 0) handle\_error("pthread\_create");

}

*// consumers* c\_list **=** calloc(NUM\_CONSUMERS, **sizeof**(**struct** thread\_info));

**if** (**!**c\_list) handle\_error("calloc");

**for** (i **=** 0; i **<** NUM\_CONSUMERS; i**++**) {

c\_list[i].rank **=** i;

c\_list[i].epfd **=** epfd;

ret **=** pthread\_create(**&**c\_list[i].thread\_id, NULL, consumer\_routine, **&**c\_list[i]);

**if** (ret **!=** 0) handle\_error("pthread\_create");

}

*// join and exit* **for** (i **=** 0; i **<** NUM\_PRODUCERS; i**++**) {

ret **=** pthread\_join(p\_list[i].thread\_id, NULL);

**if** (ret **!=** 0) handle\_error("pthread\_join");

}

**for** (i **=** 0; i **<** NUM\_CONSUMERS; i**++**) {

ret **=** pthread\_join(c\_list[i].thread\_id, NULL);

**if** (ret **!=** 0) handle\_error("pthread\_join");

}

free(p\_list);

free(c\_list);

**return** EXIT\_SUCCESS;}

执行过程（2个生产者，4个消费者）：

[1532099804] Greetings from [producer-0]

[1532099804] Greetings from [producer-1]

[1532099804] Greetings from [consumer-0]

[1532099804] Greetings from [consumer-1]

[1532099804] Greetings from [consumer-2]

[1532099804] Greetings from [consumer-3]

[1532099805] [consumer-3] got event from fd-4

[1532099805] [consumer-3] got event from fd-5

[1532099806] [consumer-0] got event from fd-4

[1532099806] [consumer-0] got event from fd-4

[1532099807] [consumer-1] got event from fd-4

[1532099807] [consumer-1] got event from fd-5

[1532099808] [consumer-3] got event from fd-4

[1532099808] [consumer-3] got event from fd-5

^C

注意，推荐在eventfd在打开时设置**NON\_BLOCKING**，并在注册至epoll监听对象时设为**EPOLLET**（尽管一次8字节的read就可以读完整个计数器到用户空间），因为毕竟，只有采用了非阻塞IO和边沿触发，epoll的并发能力才能完全发挥极致。

另外，本实例中的eventfd消费地非常高效，fd号几乎不会超过5（前四个分别为stdin/stdout/stderr/eventpoll），但实际应用中往往在close前会执行一些事务，随着消费者线程的增加，eventfd打开的文件也会增加（这个数值得上限由系统的ulimit -n决定）。然而，eventfd打开、读写和关闭都效非常高，因为它本质并不是文件，而是kernel在内核空间（内存中）维护的一个64位计数器而已。

## **典型应用场景及优势**

在信号通知的场景下，相比pipe有非常大的资源和性能优势。其根本在于counter（计数器）和channel（数据信道）的区别。

第一，是打开文件数量的巨大差别。由于pipe是半双工的传统IPC方式，所以两个线程通信需要两个pipe文件，而用eventfd只要打开一个文件。众所周知，文件描述符可是系统中非常宝贵的资源，linux的默认值也只有1024而已。那开发者可能会说，1相比2也只节省了一半嘛。要知道pipe只能在两个进程/线程间使用，并且是面向连接（类似TCP socket）的，即需要之前准备好两个pipe；而eventfd是广播式的通知，可以多对多的。如上面的NxM的生产者-消费者例子，如果需要完成全双工的通信，需要NxMx2个的pipe，而且需要提前建立并保持打开，作为通知信号实在太奢侈了，但如果用eventfd，只需要在发通知的时候瞬时创建、触发并关闭一个即可。

第二，是内存使用的差别。eventfd是一个计数器，内核维护几乎成本忽略不计，大概是自旋锁+唤醒队列（后续详细介绍），8个字节的传输成本也微乎其微。但pipe可就完全不是了，一来一回数据在用户空间和内核空间有多达4次的复制，而且更糟糕的是，内核还要为每个pipe分配至少4K的虚拟内存页，哪怕传输的数据长度为0。

所以，第一个最佳实践法则：当pipe只用来发送通知（传输控制信息而不是实际数据），放弃pipe，放心地用eventfd，"in all cases"。

另外一个重要优势就是eventfd被设计成与epoll完美结合，比如支持非阻塞的读取等。事实上，二者就是为epoll而生的（但是pipe就不是，它在Unix的史前时代就有了，那时不仅没有epoll连Linux都还没诞生）。应用程序可以在用epoll监控其他文件描述符的状态的同时，可以“顺便“”一起监控实现了eventfd的内核通知机制，何乐而不为呢？

所以，第二个最佳实践法则：eventfd配上epoll才更搭哦。

## **内核实现细节**

内核中的数据结构：eventfd\_ctx

该结构除了包括之前所介绍的一个64位的计数器，还包括了等待队列头节点（较新的kernel中还加上了一个kref）。

定义和初始化过程核心代码如下，比较直接：内核malloc，设置count值，创建eventfd的anon\_inode。

**struct** eventfd\_ctx {

wait\_queue\_head\_t wqh;

\_\_u64 count;};

以下为创建eventfd的函数的片段，比较直接。

SYSCALL\_DEFINE2(eventfd2, **unsigned** **int**, count, **int**, flags) {

*// ...* ctx **=** kmalloc(**sizeof**(**\***ctx), GFP\_KERNEL);

**if** (**!**ctx)

**return** **-**ENOMEM;

init\_waitqueue\_head(**&**ctx**->**wqh);

ctx**->**count **=** count;

fd **=** anon\_inode\_getfd("[eventfd]", **&**eventfd\_fops, ctx,

flags **&** (O\_CLOEXEC **|** O\_NONBLOCK));

*// ...*}

等待队列是内核中的重要数据结构，在进程调度、异步通知等多种场景都有很多的应用。其节点结构并不复杂，即自带自旋锁的双向循环链表的节点，如下：

**struct** \_\_wait\_queue\_head {

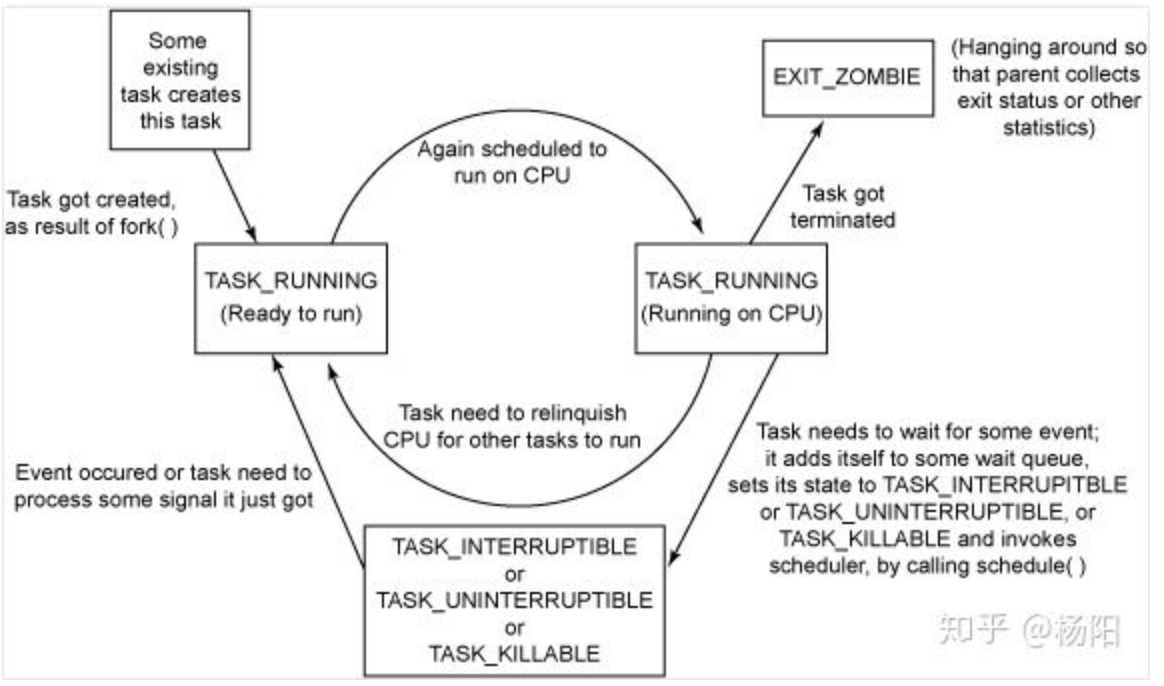
spinlock\_t lock;

**struct** list\_head task\_list;};**typedef** **struct** \_\_wait\_queue\_head wait\_queue\_head\_t;

等待队列中存放的是task（内存中对线程的抽象）的结构。

操作等待队列的函数主要是和调度相关的函数，如：wake\_up和schedule，它们位于sched.c中，前者即唤醒当前等待队列中的task，后者为当前task主动让出CPU时间给等待队列中的其他task。这样，便通过等待队列实现了多个task在运行中（TASK\_RUNNING）和IO等待（TASK\_INTERRUPTABLE）中的状态切换。

让我们一起复习下，系统中进程的状态转换：



* TASK\_RUNNING: 正在在CPU上运行，或者在执行队列(run queue)等待被调度执行。
* TASK\_INTERRUPTIBLE: 睡眠中等待默写事件出现，task可以被信号打断，一旦接收到信号或显示调用了wake-up，转为TASK\_RUNNING状态。常见于IO等待中。

清楚了task的两种状态以及run queue / wait queue原理，read函数就不难理解了。

以下是read函数的实现：

**static** ssize\_t **eventfd\_read**(**struct** file **\***file, **char** \_\_user **\***buf, size\_t count,

loff\_t **\***ppos){

**struct** eventfd\_ctx **\***ctx **=** file**->**private\_data;

ssize\_t res;

\_\_u64 ucnt;

DECLARE\_WAITQUEUE(wait, current);

**if** (count **<** **sizeof**(ucnt))

**return** **-**EINVAL;

spin\_lock\_irq(**&**ctx**->**wqh.lock);

res **=** **-**EAGAIN;

ucnt **=** ctx**->**count;

**if** (ucnt **>** 0)

res **=** **sizeof**(ucnt);

**else** **if** (**!**(file**->**f\_flags **&** O\_NONBLOCK)) {

\_\_add\_wait\_queue(**&**ctx**->**wqh, **&**wait);

**for** (res **=** 0;;) {

set\_current\_state(TASK\_INTERRUPTIBLE);

**if** (ctx**->**count **>** 0) {

ucnt **=** ctx**->**count;

res **=** **sizeof**(ucnt);

**break**;

}

**if** (signal\_pending(current)) {

res **=** **-**ERESTARTSYS;

**break**;

}

spin\_unlock\_irq(**&**ctx**->**wqh.lock);

schedule();

spin\_lock\_irq(**&**ctx**->**wqh.lock);

}

\_\_remove\_wait\_queue(**&**ctx**->**wqh, **&**wait);

\_\_set\_current\_state(TASK\_RUNNING);

}

**if** (res **>** 0) {

ctx**->**count **=** 0;

**if** (waitqueue\_active(**&**ctx**->**wqh))

wake\_up\_locked(**&**ctx**->**wqh);

}

spin\_unlock\_irq(**&**ctx**->**wqh.lock);

**if** (res **>** 0 **&&** put\_user(ucnt, (\_\_u64 \_\_user **\***) buf))

**return** **-**EFAULT;

**return** res;}

read操作目的是要将count值返回用户空间并清零。ctx中的count值是共享数据，通过加irq自旋锁实现对其的独占安全访问，spin\_lock\_irq函数可以禁止本地中断和抢占，在SMP体系中也是安全的。从源码可以看出，如果是对于（通常的epoll中的，也是上面实例中的）非阻塞读，count大于0则直接返回并清零，count等于0则直接返回EAGAIN。

对于阻塞读，如果count值为0则加入等待队列并阻塞，直到值不为0时（被其他线程更新）返回。阻塞是如何实现的呢？是通过TASK\_INTERRUPTABLE状态下的循环加schedule。注意，schedule前释放了自旋锁，意味着允许其他线程更新值，只要值被更新大于0且又**再次获得cpu时间**，那么就可以跳出循环继续执行而返回了。

考虑一个情景，两个线程几乎同时read请求，那么：两个都会被加入到等待队列中，当第一个抢到自旋锁，返回了大于1的res并重置了count为0，此时它会（在倒数第二个if那里） 第一时间唤醒等待队列中的其他线程，此时第二个线程被调度到，于是开始了自己的循环等待。即实现了：事件只会通知到第一个接收到的线程。

那么问题来了：我们知道在其他线程write后，阻塞的read线程是马上返回的。那么如何能在count置一旦不为0时，等待的调度的阻塞读线程可以尽快地**再次获得cpu时间**，从而继续执行呢？关键在于write函数也有当确认可以成功返回时，主动调用wakeup\_locked的过程，这样就能实现write后立即向等待队列通知的效果了。

write操作与read操作过程非常相似。

关于poll操作的核心代码如下：

*// ...* spin\_lock\_irqsave(**&**ctx**->**wqh.lock, flags);

**if** (ctx**->**count **>** 0)

events **|=** POLLIN;

**if** (ctx**->**count **==** ULLONG\_MAX)

events **|=** POLLERR;

**if** (ULLONG\_MAX **-** 1 **>** ctx**->**count)

events **|=** POLLOUT;

spin\_unlock\_irqrestore(**&**ctx**->**wqh.lock, flags);

在count值大于0时，返回了设置POLLIN标志的事件，使得用户层的应用可以通过epoll监控 eventfd的可读事件状态。