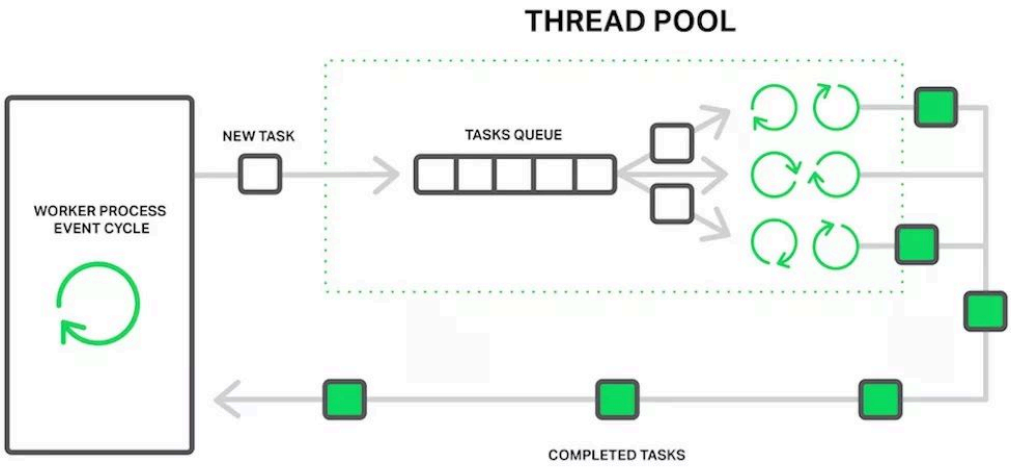
Let events fly - Linux eventfd principle and practice

Posted on 2018-07-31 09:28:35 5.2K 0 report

The article is included in the column: Golang Language Community

Original author: Yang Yang



1. Introduction to eventfd/timerfd

Currently, more and more applications use event-driven methods to implement functions. How to efficiently use system resources to manage and deliver notifications becomes increasingly important. In Linux systems, eventfd is a file descriptor used to notify events, and timerfd is a file descriptor for timer events. Both are mechanisms for the kernel to send notifications to user space applications, and can be effectively used to implement event/notification driven applications in user space.

In short, eventfd is used to trigger event notifications, and timerfd is used to trigger future event notifications.

Developers using eventfd-related system calls need to include header files; for timerfd, it is.

The system call eventfd/timerfd was added to the kernel since Linux 2.6.22, and was originally implemented and maintained by Davide Libenzi.

2. Interface and parameter introduction

eventfd

For eventfd, there is only one system call interface

Code language: javascript copy

```
1int eventfd(unsigned int initval, int flags);
```

Create an eventfd object, or open an eventfd file, similar to the open operation of a normal file.

This object is an unsigned 64-bit integer counter maintained by the kernel. It is initialized to the value of initval.

Flags can be the OR result of the following three flags:

- EFD_CLOEXEC: FD_CLOEXEC, in simple terms, does not inherit when forking a child process. It is not wrong to set this value for a multi-threaded program.
- EFD_NONBLOCK: The file will be set to O_NONBLOCK, which is generally required.
- EFD_SEMAPHORE: (supported after 2.6.30) supports read with semophore semantics. Simply put, the value is decremented by 1.

Li Haibin V

LV.1

Golang Language Com... | Webr

article 1.9K Liked 12.6K

Author's Related Selections

- Getting started with epoll
- In-depth understanding of th
- [Go Language Community]

Table of contents

1. Introduction to eventfd/timerfc
2. Interface and parameter intro
eventfd
timerfd
3. Use Case - Implementing a H
Eventfd corresponding impleme
Timerfd corresponding impleme
4. Typical application scenarios
5. Kernel implementation details
6. Summary
7. References

The operation of this newly created fd is very simple:

read(): The read operation is to set the counter value to 0, and if it is a semophore, it will be reduced by 1.

write(): Sets the value of counter.

Note that epoll/poll/select operations are also supported, and of course, close must be implemented for each fd.

timerfd

For timerfd, there are three system call interfaces involved

Code language: javascript copy

```
1int timerfd_create(int clockid, int flags);int timerfd_settime(int fd, int flags,
2                          const struct itimerspec *new_value,
3                          struct itimerspec *old_value);int timerfd_gettime(int fd, struct iti
```

timerfd_create is used to create a new timerfd object. clockid can specify the type of clock. There are two commonly used ones: CLOCK_REALTIME (real-time clock) or CLOCK_MONOTONIC (monotonically increasing clock). The real-time clock refers to th e system clock, which can be modified manually. The latter monotonically increasing clock will not be affected by the artificial dis continuity of the system clock. Usually the latter is selected. The choice of flags, TFD_CLOEXEC and TFD_NONBLOCK, is mor e direct.

The timerfd_settime function is used to set the expiration time of the timer. The itmerspec structure is defined as follows:

Code language: javascript copy

```
1struct timespec {
2    time_t tv_sec;                /* Seconds */
3    long   tv_nsec;               /* Nanoseconds */};struct itimerspec {
4    struct timespec it_interval;  /* Interval for periodic timer */
5    struct timespec it_value;     /* Initial expiration */};
```

This structure contains two time intervals: it_value refers to the first expiration time, and it_interval refers to the interval time for p eriodic triggering of expiration after the first expiration (if it is set to 0, it is the first expiration).

If old_value is not NULL, the itimerspec structure object pointed to by old_value will be updated with the calling time.

timerfd_gettime(): Returns the current setting value of the timerfd object to the object pointed to by the curr_value pointer.

read(): The semantics of the read operation are: if the timer expires, the number of expiration times is returned, and the result is stored in an 8-byte integer (uint64_6); if it has not expired, it blocks until it expires, or returns EAGAIN (depending on whether N ONBLOCK is set).

In addition, epoll is supported, same as eventfd.

3. Use Case - Implementing a High-Performance Consumer Thread Pool

The producer-consumer design pattern is a common background architecture pattern. This example will implement an event noti fication framework for multiple producers and multiple consumers to illustrate the typical scenario of eventfd/timerfd as a notificat ion implementation in thread communication.

This example adopts the following design: the producer creates eventfd/timerfd and registers the event in the event loop; the thr eads in the consumer thread pool share an epoll object, and each consumer thread polls the event loop triggered by eventfd or ti merfd in parallel (epoll_wait).

Eventfd corresponding implementation

Code language: javascript copy

```
1typedef struct thread_info {
2    pthread_t thread_id;
3    int rank;
4    int epfd;} thread_info_t;static void *consumer_routine(void *data) {
5    struct thread_info *c = (struct thread_info *)data;
6    struct epoll_event *events;
7    int epfd = c->epfd;
8    int nfds = -1;
9    int i = -1;
10   uint64_t result;
11
12   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
            }
        }
    }}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));
    }}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;}
```

Execution process (2 producers, 4 consumers):

Code language: javascript copy

```
1[1532099804] Greetings from [producer-0]
2[1532099804] Greetings from [producer-1]
3[1532099804] Greetings from [consumer-0]
4[1532099804] Greetings from [consumer-1]
5[1532099804] Greetings from [consumer-2]
6[1532099804] Greetings from [consumer-3]
7[1532099805] [consumer-3] got event from fd-4
8[1532099805] [consumer-3] got event from fd-5
9[1532099806] [consumer-0] got event from fd-4
10[1532099806] [consumer-0] got event from fd-4
11[1532099807] [consumer-1] got event from fd-4
12[1532099807] [consumer-1] got event from fd-5
13[1532099808] [consumer-3] got event from fd-4
14[1532099808] [consumer-3] got event from fd-5
15^C
```

The result is as expected (attached: source code link)

Note that it is recommended to set NON_BLOCKING when eventfd is opened, and set it to EPOLLET when registered to the ep oll listener object (although a single 8-byte read can read the entire counter to user space), because after all, only with non-block ing IO and edge triggering can the concurrency capabilities of epoll be fully utilized.

In addition, the eventfd in this example is very efficient in consumption, and the fd number will hardly exceed 5 (the first four are stdin/stdout/stderr/eventpoll respectively). However, in actual applications, some transactions are often executed before closing. As the number of consumer threads increases, the number of files opened by eventfd will also increase (the upper limit of this nu mber is determined by the system's ulimit -n). However, eventfd is very efficient in opening, reading, writing, and closing, becaus e it is not a file in essence, but a 64-bit counter maintained by the kernel in kernel space (in memory).

Timerfd corresponding implementation

The main function is almost the same as the consumer thread implementation, while the producer thread creates timerfd and reg isters it with the event loop.

The it_value of timer is set to 1 second, which means the first trigger is 1 second later; the it_interval is set to 3 seconds, which means it will be triggered again every 3 seconds.

Note that the location of the timerfd_settime function is the same as the previous eventfd write. The two have a similar function o f setting events, except that this time it is a timer event.

Code language: javascript copy

```
1static void *producer_routine(void *data) {
2    struct thread_info *p = (struct thread_info *)data;
3    struct epoll_event event;
4    int epfd = p->epfd;
5    int tfd = -1;
6    int ret = -1;
7    struct itimerspec its;
8    its.it_value.tv_sec = 1;    // initial expiration
9    its.it_value.tv_nsec = 0;
10   its.it_interval.tv_sec = 3; // interval
11   its.it_interval.tv_nsec = 0;
12
13   log("Greetings from [producer-%d]", p->rank);
14   // create timerfd
15   tfd = timerfd_create(CLOCK_MONOTONIC, TFD_CLOEXEC|TFD_NONBLOCK);
16   if (tfd == -1) handle_error("timerfd create: %s", strerror(errno));
17   // register to poller
18   event.data.fd = tfd;
19   event.events = EPOLLIN | EPOLLET;    // Edge-Triggered
20   ret = epoll_ctl(epfd, EPOLL_CTL_ADD, tfd, &event);
21   if (ret != 0) handle_error("epoll_ctl");
22   // register timer expired in future
23   ret = timerfd_settime(tfd, 0, &its, NULL);
24   if (ret != 0) handle_error("timerfd settime");
25   return (void *)0;}
```

Execution process (2 producers, 4 consumers):

Code language: javascript copy

```
 1[1532099143] Greetings from [producer-1]
 2[1532099143] Greetings from [consumer-1]
 3[1532099143] Greetings from [consumer-2]
 4[1532099143] Greetings from [consumer-3]
 5[1532099143] Greetings from [consumer-0]
 6[1532099143] Greetings from [producer-0]
 7[1532099144] [consumer-3] got event from fd-4
 8[1532099144] [consumer-3] got event from fd-5
 9[1532099147] [consumer-3] got event from fd-4
10[1532099147] [consumer-3] got event from fd-5
11[1532099150] [consumer-0] got event from fd-4
12[1532099150] [consumer-0] got event from fd-5
13[1532099153] [consumer-1] got event from fd-4
14[1532099153] [consumer-1] got event from fd-5
15^C
```

As can be seen from the above figure, the two file descriptors fd-4 and fd-5 opened at runtime are timerfd.

The result is as expected (attached: source code link)

4. Typical application scenarios and advantages

Quoting the first sentence of the NOTE paragraph in the eventfs Manual:

> Applications can use an eventfd file descriptor instead of a pipe in all cases where a pipe is used simply to signal events.

In the signal notification scenario, it has a huge resource and performance advantage over pipe. The fundamental reason lies in the difference between counter and channel.

- 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。
- 第三，对于timerfd，还有精准度和实现复杂度的巨大差异。由内核管理的timerfd底层是内核中的hrtimer（高精度时钟定时器），可以精确至纳秒（1e-9秒）级，完全胜任实时任务。而用户态要想实现一个传统的定时器，通常是基于优先队列/二叉堆，不仅实现复杂维护成本高，而且运行时效率低，通常只能到达毫秒级。

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/timerfd，"in all cases"。

另外一个重要优势就是eventfd/timerfd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。

5. 内核实现细节

eventfd在内核源码中，作为syscall实现在内核源码的 fs/eventfd.c下。从Linux 2.6.22版本引入内核，在2.6.27版本以后加入对flag的支持。以下分析参考Linux 2.6.27源码。

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。

代码语言：javascript 复制

```
1struct eventfd_ctx {
2        wait_queue_head_t wqh;
3
```

```
3        __u64 count;};
```

以下为创建eventfd的函数的片段，比较直接。

代码语言：javascript 复制

```
1SYSCALL_DEFINE2(eventfd2, unsigned int, count, int, flags) {
2        // ...
3    ctx = kmalloc(sizeof(*ctx), GFP_KERNEL);
4    if (!ctx)
5        return -ENOMEM;
6    init_waitqueue_head(&ctx->wqh);
7    ctx->count = count;
8    fd = anon_inode_getfd("[eventfd]", &eventfd_fops, ctx,
9                  flags & (O_CLOEXEC | O_NONBLOCK));
10       // ...}
```

稍提一下，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：

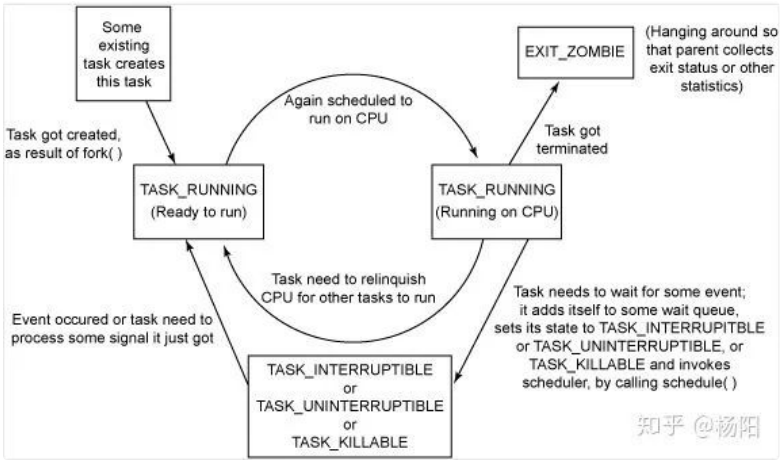
代码语言：javascript 复制

```
1struct __wait_queue_head {
2    spinlock_t lock;
3    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：

代码语言：javascript 复制

```
1static ssize_t eventfd_read(struct file *file, char __user *buf, size_t count,
2                loff_t *ppos){
3    struct eventfd_ctx *ctx = file->private_data;
4    ssize_t res;
5    __u64 ucnt;
6    DECLARE_WAITQUEUE(wait, current);
7
8    if (count < sizeof(ucnt))
9        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，不在此展开。

关于poll操作的核心代码如下：

代码语言：javascript 复制

```
// ...
        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。

6. 本篇小结

通过对eventfd/timerfd的接口和实现的了解，可以看出其不仅功能实用，而且调用方式简单。另外，其实现是非常精巧高效的，构建于内核众多系统基础核心功能之上，为用户态的应用封装了十分高效简单的事件通知机制。

7. 参考资料

Linux 内核源码 https://elixir.bootlin.com/linux/latest/source/fs/eventfd.c

Linux Programmer's Manual eventfd(2) - Linux manual page

linux

Comment

Recommended Reading

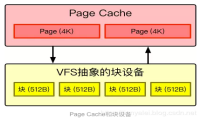
This article explains the concepts, meanings and things behind BIO, NIO and AIO from the perspective of the operating system. This article is mainly divided into 3 parts. The first part explains BIO, NIO and IO multiplexing. The second part explains disk I…

Big and wide · 2018/05/14 4K 0

Let's talk about BIO, NIO and AIO (2)

Cache linux database sql

Disk IO, in simple terms, is the IO that reads hard disks and other devices. Such devices include traditional disks, SSDs, flash memory, CDs, etc. The operating system abstracts them as "block…



Tianya Lei Xiaowu · 2019/07/12 1.4K 0

Ten questions to understand the working principle of Linux epoll

programming algorithm data structure linux

Author: dustinzhou, Tencent IEG operations development engineer epoll is an I/O event notification mechanism unique to Linux. I have long been interested in how epoll can efficiently handle millions of file descriptors. I have recently studied and research…

Tencent Technology Engineering Official Account · 2021/06/03 4K 0

Linux new API signalfd, timerfd, eventfd usage instructions

other

Original article: http://www.cfanz.cn/?c=article&a=read&id=46555 Note that many Linux kernels currently in operation (2013/8/6) may not support! Three new fd

At first sight · 2018/08/10 1.8K 0

Do you really understand the blocking and asynchronous notification mechanisms in the Linux ker…

C language linux data structure

Blocking operation means that when executing device operation, if resources cannot be obtained, the process is suspended until the conditions for operation are met. The suspended process enters the sleep state and is removed from the scheduler's…

Embedded and Linux · 2021/05/20 1.1K 0

Operating system and storage: Analysis of the design and implementation of the new asynchrono…

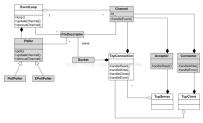
autonomous driving postgresql sql data structure node.js

Author: draculaqian, Tencent backend development engineer Introduction In storage scenarios, we have very high requirements for performance. When selecting the IO technology at the bottom of the storage engine, there may be the following discussion…

Tencent Technology Engineering Official Account · 2021/02/22 2.4K 0

muduo network library learning EventLoop (I): Introduction to event loop class diagram and mudu…

C++ Linux Programming Algorithms



s1mba · 2017/12/28 2K 0

About eventfd, epoll, and inter-thread communication

Linux artificial intelligence

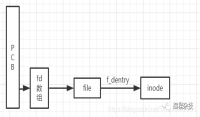
First, let's introduce eventfd 1 #include<sys/eventfd.h> 2 int eventfd(unsigned int initval, int flags); Use this function to create an event object. In order to improve the efficiency of communication between Linux threads, most of them use asynchronous…

Haonan Hacking · 2018/03/08 2.2K 0

Understanding epoll (Part 1) (Based on linux2.6.12.1)

Linux data structures

epoll occupies a large part in current software. Single-threaded event loop software such as nginx and libuv all use epoll. I have analyzed select before, and today I will analyze epoll.

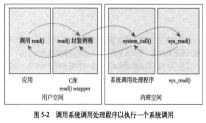


theanarkh · 2020/04/01 1.1K 0

Linux system calls

linux kernel int sys kernel

In modern operating systems, the kernel provides a set of interfaces for user processes to interact with the kernel. These interfaces allow applications to access hardware devices in a limited manne…



Jasonangel · 2022/10/25 9.8K 0

Principle of receiving data from kernel to EPOLL

Programming algorithm Linux socket programming

1. The network card receives the packet when it finds that the MAC address matches. When it finds that the IP address matches, it knows that the upper layer is TCP protocol according to the protocol item in the IP header.

ruochen · 2021/11/25 1.1K 0

Detailed explanation of EPOLL principle

Programming algorithm Linux socket programming

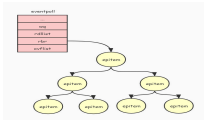
1. The network card receives the packet when it finds that the MAC address matches. When it finds that the IP address matches, it knows that the upper layer is TCP protocol according to the protocol item in the IP header.

Great Inventor · 2021/12/17 2.1K 0

Linux epoll implementation principle

socket programming linux

epoll is a unique IO multiplexing implementation method under the Linux platform. Compared with the traditional select/poll, epoll has greatly improved performance. This article mainly explains the…



Love Cat Big Carp · 2022/10/24 1.7K 0

In-depth understanding of the core principles of Linux C/C++ Timer

Linux C++ Programming Algorithms

I thought that the operating system would have a complete implementation for basic functions like timers. When you need to start a scheduled task, there will be an elegant interface in the following…

sunsky · 2020/12/21 10.7K 0

Libevent event mechanism

Linux programming algorithm

From the previous article "How libevent chooses the underlying implementation", we can see that calling the event_base_new() function initializes the underlying implementation and assigns a value to evsel in the event_base structure. evsel is an eventto…

cpp gas station · 2021/04/16 807 0

Review of Linux backend programming (VI): In-depth understanding of the epoll model

Linux socket programming

Some of you may not know much about select, so let me explain it briefly: Network connection, the server also manages these connected clients through file descriptors. Since it is a server for connection, it is inevitable to receive messages from clients.…

Look, the future · 2021/09/18 667 0

JDK source code analysis NIO implementation

http https network security linux .net

jdk:http://hg.openjdk.java.net/jdk8u/jdk8u60/file/d8f4022fe0cd

Java architects must read · 2021/07/13 1.2K 0