Linux内核poll,select,epoll事件通知机制剖析

# 简介

poll, select, epoll 是Linux内核主要的事件通知机制，大多数网络io函数库和服务器程序都对他们进行了封装，了解他们的内部实现，能够更好的了解他们的性能差别。

## 基本用法和注意事项

poll, select, epoll 的基本用法，请参考linux man 手册。

使用当中需要注意的问题:

1. 在使用之前尽可能对他们进行封装，形成统一的接口，这里可以参考任何开源网络服务程序
2. select/pselect中fd\_set能容纳的最大描述符FD\_SETSIZE-1
3. epoll 边缘模式，防止饥饿

等等等等

# 源码剖析

前导

poll/select/epoll的实现都是基于文件提供的poll方法(f\_op->poll)，该方法会在文件对应的一个或多个等待队列上添加回调函数节点(poll\_wait)，并检查文件当前就绪的状态返回给调用者(依赖于文件的实现)。每当影响文件状态的事件发生时(例如网络数据包到达)，文件会遍历等待队列并调用回调函数唤醒等待线程(\_\_wake\_up)。

## 文件操作file\_opertions.poll实现

事实上，文件系统如果想要提供对事件通知机制的支持，就需要提供文件操作(struct file\_opertions)的poll方法(unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*))，并且为每个文件供一个就绪等待队列，当文件就绪时依次调用等待队列上的回调函数发出通知，如果未提供poll方法，则认为该文件一直保持就绪状态(例如，本地文件系统中的文件)。其中struct file 对应于一个打开的文件(也就是文件描述符)，可以通过调用fget(fd)，fget\_xxx(fd)得到。poll\_table\_struct由具体的事件通知机制提供，用来实现具体的通知方法，poll\_table\_struct内部有一个函数指针void (\*qproc)(struct file \*, wait\_queue\_head\_t \*, struct poll\_table\_struct \*)，用来向等待队列中添加节点并设置回调函数。poll的返回值是当前已就绪的操作，可以用来立即判断当前描述符是否可用。

不同的文件poll的实现方法可能不同，但是他们的思想是一样的，实际上这个函数主要做了以下几件事:

1. 根据文件(struct file)的不同，获取文件对应的等待队列(wait\_queue\_head\_t)
2. 如果参数不为空，调用poll\_wait函数，在poll\_wait函数中会回调poll\_table\_struct上的qproc函数，qproc函数会在文件等待队列上插入一个包含等待函数的一个节点，这样当文件就绪时，就会遍历等待队列并调用回调函数发出通知。
3. 返回当前的文件状态。

这里就不解析具体文件系统的实现了，有兴趣的朋友可以自己看一下tcp\_poll，event\_poll\_poll

poll，select，epoll等事件通知机制都是基于file\_operations的poll方法，他们都是在文件的就绪队列上添加节点设置回调函数来实现的。

## poll和select

poll，select的处理流程是

遍历所有的文件描述符，取得file结构体，然后调用file->f\_ops->poll方法，设置等待队列上的回调函数。

然后进程挂起，等待事件发生。只要任何一个文件有事件发生，回调函数就会被调用，在回调函数中会唤醒poll或select的调用进程。

进程唤醒或超时以后会再次遍历所有的文件，逐个检查文件的状态，只是此时不再设置回调函数，而是直接取得文件的就绪状态，然后提交给用户程序。

Epoll的实现与select，poll稍有不同，但也十分类似，

1. 添加描述符，取得file结构，调用file->f\_ops->poll方法，设置等待队列。
2. 文件就绪，调用回调函数，将该文件相关信息添加到epoll的就绪列表中，唤醒epoll进程
3. Epoll\_wait进程，等待超时或唤醒，然后直接返回就绪列表。

以上是简要的分析，下面我们做更加具体的分析。

Epoll 本身也是文件

poll和select的实现

上面已经讲过文件操作上需要有一个poll方法用来支持poll，select，epoll等事件通知机制。

## Poll

// poll ，由用户程序调用

SYSCALL\_DEFINE3(poll, struct pollfd \_\_user \*, ufds, unsigned int, nfds,

long, timeout\_msecs)

{

struct timespec end\_time, \*to = NULL;

int ret;

if (timeout\_msecs >= 0) {

to = &end\_time;

// 设置超时时间

poll\_select\_set\_timeout(to, timeout\_msecs / MSEC\_PER\_SEC,

NSEC\_PER\_MSEC \* (timeout\_msecs % MSEC\_PER\_SEC));

}

// 真正的操作在这里

ret = do\_sys\_poll(ufds, nfds, to);

// 错误处理

if (ret == -EINTR) {

struct restart\_block \*restart\_block;

restart\_block = &current\_thread\_info()->restart\_block;

restart\_block->fn = do\_restart\_poll;

restart\_block->poll.ufds = ufds;

restart\_block->poll.nfds = nfds;

if (timeout\_msecs >= 0) {

restart\_block->poll.tv\_sec = end\_time.tv\_sec;

restart\_block->poll.tv\_nsec = end\_time.tv\_nsec;

restart\_block->poll.has\_timeout = 1;

} else

restart\_block->poll.has\_timeout = 0;

ret = -ERESTART\_RESTARTBLOCK;

}

return ret;

}

#define N\_STACK\_PPS ((sizeof(stack\_pps) - sizeof(struct poll\_list)) / \

sizeof(struct pollfd))

int do\_sys\_poll(struct pollfd \_\_user \*ufds, unsigned int nfds,

struct timespec \*end\_time)

{

struct poll\_wqueues table;

int err = -EFAULT, fdcount, len, size;

/\* 使用栈上的空间，节约内存，加速访问 \*/

long stack\_pps[POLL\_STACK\_ALLOC/sizeof(long)];

struct poll\_list \*const head = (struct poll\_list \*)stack\_pps;

struct poll\_list \*walk = head;

unsigned long todo = nfds;

if (nfds > rlimit(RLIMIT\_NOFILE))

return -EINVAL;

// 复制用户空间数据

len = min\_t(unsigned int, nfds, N\_STACK\_PPS);

for (;;) {

walk->next = NULL;

walk->len = len;

if (!len)

break;

if (copy\_from\_user(walk->entries, ufds + nfds-todo,

sizeof(struct pollfd) \* walk->len))

goto out\_fds;

todo -= walk->len;

if (!todo)

break;

// 栈上空间不够，在堆上申请

len = min(todo, POLLFD\_PER\_PAGE);

size = sizeof(struct poll\_list) + sizeof(struct pollfd) \* len;

walk = walk->next = kmalloc(size, GFP\_KERNEL);

if (!walk) {

err = -ENOMEM;

goto out\_fds;

}

}

// 初始化

poll\_initwait(&table);

fdcount = do\_poll(nfds, head, &table, end\_time); // poll

poll\_freewait(&table);

// 复制到用户空间

for (walk = head; walk; walk = walk->next) {

struct pollfd \*fds = walk->entries;

int j;

for (j = 0; j < walk->len; j++, ufds++)

if (\_\_put\_user(fds[j].revents, &ufds->revents))

goto out\_fds;

}

err = fdcount;

out\_fds:

walk = head->next;

while (walk) {

struct poll\_list \*pos = walk;

walk = walk->next;

kfree(pos);

}

return err;

}

void poll\_initwait(struct poll\_wqueues \*pwq)

{

设置函数，和一些变量，\_\_pollwait会在f\_ops->poll中调用

init\_poll\_funcptr(&pwq->pt, \_\_pollwait);

pwq->polling\_task = current;

pwq->triggered = 0;

pwq->error = 0;

pwq->table = NULL;

pwq->inline\_index = 0;

}

static void \_\_pollwait(struct file \*filp, wait\_queue\_head\_t \*wait\_address,

poll\_table \*p)

{

struct poll\_wqueues \*pwq = container\_of(p, struct poll\_wqueues, pt);

struct poll\_table\_entry \*entry = poll\_get\_entry(pwq);

if (!entry)

return;

get\_file(filp);

entry->filp = filp;

entry->wait\_address = wait\_address; // 等待队列

entry->key = p->key;

init\_waitqueue\_func\_entry(&entry->wait, pollwake); // 设置回调

entry->wait.private = pwq;

add\_wait\_queue(wait\_address, &entry->wait); // 添加到等待队列

}

static int do\_poll(unsigned int nfds, struct poll\_list \*list,

struct poll\_wqueues \*wait, struct timespec \*end\_time)

{

poll\_table\* pt = &wait->pt;

ktime\_t expire, \*to = NULL;

int timed\_out = 0, count = 0;

unsigned long slack = 0;

if (end\_time && !end\_time->tv\_sec && !end\_time->tv\_nsec) {

pt = NULL; // 已经超时就不用设置回调函数了，直接遍历返回

timed\_out = 1;

}

if (end\_time && !timed\_out)

slack = select\_estimate\_accuracy(end\_time);// 估计进程等待时间，纳秒

// 遍历文件，为每个文件的等待队列添加回调函数节点

for (;;) {

struct poll\_list \*walk;

for (walk = list; walk != NULL; walk = walk->next) {

struct pollfd \* pfd, \* pfd\_end;

pfd = walk->entries;

pfd\_end = pfd + walk->len;

for (; pfd != pfd\_end; pfd++) {

// do\_pollfd 会检查当前文件状态并设置返回的掩码

if (do\_pollfd(pfd, pt)) {

// 已经有就绪的文件了，不用再设置回调函数了，

// 把后面的遍历一遍返回就可以了

count++;

pt = NULL;

}

}

}

// 该注册的文件都已经注册了回调函数，后面就等事件发生，然后检查了

pt = NULL;

if (!count) {

count = wait->error;

// 信号来了

if (signal\_pending(current))

count = -EINTR;

}

if (count || timed\_out)

break;

// 转换为内核时间

if (end\_time && !to) {

expire = timespec\_to\_ktime(\*end\_time);

to = &expire;

}

// 等吧, 如果有事件发生或超时，接着循环一遍，记录事件状态并计数

// 这次循环将清除前面设置的等待队列上的回调函数

if (!poll\_schedule\_timeout(wait, TASK\_INTERRUPTIBLE, to, slack))

timed\_out = 1;

}

return count;

}

static inline unsigned int do\_pollfd(struct pollfd \*pollfd, poll\_table \*pwait)

{

unsigned int mask;

int fd;

mask = 0;

fd = pollfd->fd;

if (fd >= 0) {

int fput\_needed;

struct file \* file;

// 取得fd对应的文件结构体

file = fget\_light(fd, &fput\_needed);

mask = POLLNVAL;

if (file != NULL) {

mask = DEFAULT\_POLLMASK;

if (file->f\_op && file->f\_op->poll) {

if (pwait) // 等待事件的掩码

pwait->key = pollfd->events | POLLERR | POLLHUP;

// 注册回调函数，并返回当前就绪状态，就绪后会调用pollwake

mask = file->f\_op->poll(file, pwait);

}

mask &= pollfd->events | POLLERR | POLLHUP; // 移除不需要的文件

fput\_light(file, fput\_needed);// 释放文件

}

}

pollfd->revents = mask;

return mask;

}

// 文件就绪后调用，用来唤醒调用进程

static int pollwake(wait\_queue\_t \*wait, unsigned mode, int sync, void \*key)

{

struct poll\_table\_entry \*entry;

entry = container\_of(wait, struct poll\_table\_entry, wait);

if (key && !((unsigned long)key & entry->key))

return 0;

return \_\_pollwake(wait, mode, sync, key);

}

static int \_\_pollwake(wait\_queue\_t \*wait, unsigned mode, int sync, void \*key)

{

struct poll\_wqueues \*pwq = wait->private;

DECLARE\_WAITQUEUE(dummy\_wait, pwq->polling\_task);

smp\_wmb();

pwq->triggered = 1;// 已触发事件

// 唤醒调用进程

return default\_wake\_function(&dummy\_wait, mode, sync, key);

}

## Select

这里就只介绍select的最核心的函数do\_select，外面的几层就不介绍了，select和poll一样也是挨个遍历描述符

int do\_select(int n, fd\_set\_bits \*fds, struct timespec \*end\_time)

{

ktime\_t expire, \*to = NULL;

struct poll\_wqueues table;

poll\_table \*wait;

int retval, i, timed\_out = 0;

unsigned long slack = 0;

rcu\_read\_lock();

retval = max\_select\_fd(n, fds);

rcu\_read\_unlock();

if (retval < 0)

return retval;

n = retval;

poll\_initwait(&table);

wait = &table.pt;

if (end\_time && !end\_time->tv\_sec && !end\_time->tv\_nsec) {

wait = NULL;

timed\_out = 1;

}

if (end\_time && !timed\_out)

slack = select\_estimate\_accuracy(end\_time);

retval = 0;

for (;;) {

unsigned long \*rinp, \*routp, \*rexp, \*inp, \*outp, \*exp;

inp = fds->in; outp = fds->out; exp = fds->ex;

rinp = fds->res\_in; routp = fds->res\_out; rexp = fds->res\_ex;

// 遍历所有的描述符

for (i = 0; i < n; ++rinp, ++routp, ++rexp) {

unsigned long in, out, ex, all\_bits, bit = 1, mask, j;

unsigned long res\_in = 0, res\_out = 0, res\_ex = 0;

const struct file\_operations \*f\_op = NULL;

struct file \*file = NULL;

// 设置描述符的上事件掩码

in = \*inp++; out = \*outp++; ex = \*exp++;

all\_bits = in | out | ex;

if (all\_bits == 0) { // 没有需要监听的事件

i += \_\_NFDBITS;

continue;

}

for (j = 0; j < \_\_NFDBITS; ++j, ++i, bit <<= 1) {

int fput\_needed;

if (i >= n)

break;

if (!(bit & all\_bits))

continue;

file = fget\_light(i, &fput\_needed);

if (file) {

f\_op = file->f\_op;

mask = DEFAULT\_POLLMASK;

if (f\_op && f\_op->poll) {

// 设置等待掩码

wait\_key\_set(wait, in, out, bit);

mask = (\*f\_op->poll)(file, wait); // 设置回调，回调函数

//和poll完全一样

}

fput\_light(file, fput\_needed);

// 检查时候有事件发生，一旦有事件发生就不再添加回调函数

if ((mask & POLLIN\_SET) && (in & bit)) {

res\_in |= bit;

retval++;

wait = NULL;

}

if ((mask & POLLOUT\_SET) && (out & bit)) {

res\_out |= bit;

retval++;

wait = NULL;

}

if ((mask & POLLEX\_SET) && (ex & bit)) {

res\_ex |= bit;

retval++;

wait = NULL;

}

}

}

if (res\_in)

\*rinp = res\_in;

if (res\_out)

\*routp = res\_out;

if (res\_ex)

\*rexp = res\_ex;

cond\_resched();

}

wait = NULL; // 该注册的都注册了，

if (retval || timed\_out || signal\_pending(current))

break; //有信号，事件发生或超时跳出

if (table.error) {

retval = table.error;

break;

}

// 转换到内核时间

if (end\_time && !to) {

expire = timespec\_to\_ktime(\*end\_time);

to = &expire;

}

// 等待，下次循环就直接设置fd\_set就可以了

if (!poll\_schedule\_timeout(&table, TASK\_INTERRUPTIBLE,

to, slack))

timed\_out = 1;

}

poll\_freewait(&table);

return retval;

}

## Epoll

Epoll与poll，select稍有不同，不过基本原理完全一样

// 创建epoll描述符

SYSCALL\_DEFINE1(epoll\_create1, int, flags)

{

int error;

struct eventpoll \*ep = NULL;

/\* Check the EPOLL\_\* constant for consistency. \*/

BUILD\_BUG\_ON(EPOLL\_CLOEXEC != O\_CLOEXEC);

if (flags & ~EPOLL\_CLOEXEC)

return -EINVAL;

// 创建struct eventpoll内部结构

error = ep\_alloc(&ep);

if (error < 0)

return error;

// 取得描述符，设置文件上的file\_operations，用来epoll自身就绪后发出通知

error = anon\_inode\_getfd("[eventpoll]", &eventpoll\_fops, ep,

O\_RDWR | (flags & O\_CLOEXEC));

if (error < 0)

ep\_free(ep);

return error;

}

epoll\_ctl

// epoll

SYSCALL\_DEFINE4(epoll\_ctl, int, epfd, int, op, int, fd,

struct epoll\_event \_\_user \*, event)

{

int error;

int did\_lock\_epmutex = 0;

struct file \*file, \*tfile;

struct eventpoll \*ep;

struct epitem \*epi;

struct epoll\_event epds;

error = -EFAULT;

if (ep\_op\_has\_event(op) &&

copy\_from\_user(&epds, event, sizeof(struct epoll\_event)))

goto error\_return;

error = -EBADF;

file = fget(epfd); // 取得 epfd 对应的文件

if (!file)

goto error\_return;

tfile = fget(fd);// 取得 df对应的文件

if (!tfile)

goto error\_fput;

/\* The target file descriptor must support poll \*/

error = -EPERM;

if (!tfile->f\_op || !tfile->f\_op->poll)

goto error\_tgt\_fput;

error = -EINVAL;

if (file == tfile || !is\_file\_epoll(file)) // 检查文件

goto error\_tgt\_fput;

ep = file->private\_data; // epoll 结构

// 目标文件也是epoll，并且是添加

if (unlikely(is\_file\_epoll(tfile) && op == EPOLL\_CTL\_ADD)) {

mutex\_lock(&epmutex);

did\_lock\_epmutex = 1;

error = -ELOOP;

if (ep\_loop\_check(ep, tfile) != 0)

goto error\_tgt\_fput;

}

mutex\_lock\_nested(&ep->mtx, 0);

// 在红黑树中查找 目标文件对应的epoll\_item

epi = ep\_find(ep, tfile, fd);

error = -EINVAL;

switch (op) {

case EPOLL\_CTL\_ADD:

if (!epi) {

epds.events |= POLLERR | POLLHUP;

error = ep\_insert(ep, &epds, tfile, fd); // 添加

} else

error = -EEXIST;

break;

case EPOLL\_CTL\_DEL:

if (epi)

error = ep\_remove(ep, epi); // 删除

else

error = -ENOENT;

break;

case EPOLL\_CTL\_MOD: //

if (epi) {

epds.events |= POLLERR | POLLHUP;

error = ep\_modify(ep, epi, &epds);

} else

error = -ENOENT;

break;

}

mutex\_unlock(&ep->mtx);

error\_tgt\_fput:

if (unlikely(did\_lock\_epmutex))

mutex\_unlock(&epmutex);

fput(tfile);

error\_fput:

fput(file);

error\_return:

return error;

}

static int ep\_insert(struct eventpoll \*ep, struct epoll\_event \*event,

struct file \*tfile, int fd)

{

int error, revents, pwake = 0;

unsigned long flags;

long user\_watches;

struct epitem \*epi;

struct ep\_pqueue epq;

user\_watches = atomic\_long\_read(&ep->user->epoll\_watches);

if (unlikely(user\_watches >= max\_user\_watches))

return -ENOSPC;

if (!(epi = kmem\_cache\_alloc(epi\_cache, GFP\_KERNEL)))

return -ENOMEM;

/\* Item initialization follow here ... \*/

INIT\_LIST\_HEAD(&epi->rdllink);

INIT\_LIST\_HEAD(&epi->fllink);

INIT\_LIST\_HEAD(&epi->pwqlist);

epi->ep = ep;

ep\_set\_ffd(&epi->ffd, tfile, fd);

epi->event = \*event;

epi->nwait = 0;

epi->next = EP\_UNACTIVE\_PTR;

/\* Initialize the poll table using the queue callback \*/

epq.epi = epi;

init\_poll\_funcptr(&epq.pt, ep\_ptable\_queue\_proc);

/\*

\* Attach the item to the poll hooks and get current event bits.

\* We can safely use the file\* here because its usage count has

\* been increased by the caller of this function. Note that after

\* this operation completes, the poll callback can start hitting

\* the new item.

\*/

revents = tfile->f\_op->poll(tfile, &epq.pt);

/\*

\* We have to check if something went wrong during the poll wait queue

\* install process. Namely an allocation for a wait queue failed due

\* high memory pressure.

\*/

error = -ENOMEM;

if (epi->nwait < 0)

goto error\_unregister;

/\* Add the current item to the list of active epoll hook for this file \*/

spin\_lock(&tfile->f\_lock);

list\_add\_tail(&epi->fllink, &tfile->f\_ep\_links);

spin\_unlock(&tfile->f\_lock);

/\*

\* Add the current item to the RB tree. All RB tree operations are

\* protected by "mtx", and ep\_insert() is called with "mtx" held.

\*/

ep\_rbtree\_insert(ep, epi);

/\* We have to drop the new item inside our item list to keep track of it \*/

spin\_lock\_irqsave(&ep->lock, flags);

/\* If the file is already "ready" we drop it inside the ready list \*/

if ((revents & event->events) && !ep\_is\_linked(&epi->rdllink)) {

list\_add\_tail(&epi->rdllink, &ep->rdllist);

/\* Notify waiting tasks that events are available \*/

if (waitqueue\_active(&ep->wq))

wake\_up\_locked(&ep->wq);

if (waitqueue\_active(&ep->poll\_wait))

pwake++;

}

spin\_unlock\_irqrestore(&ep->lock, flags);

atomic\_long\_inc(&ep->user->epoll\_watches);

/\* We have to call this outside the lock \*/

if (pwake)

ep\_poll\_safewake(&ep->poll\_wait);

return 0;

error\_unregister:

ep\_unregister\_pollwait(ep, epi);

/\*

\* We need to do this because an event could have been arrived on some

\* allocated wait queue. Note that we don't care about the ep->ovflist

\* list, since that is used/cleaned only inside a section bound by "mtx".

\* And ep\_insert() is called with "mtx" held.

\*/

spin\_lock\_irqsave(&ep->lock, flags);

if (ep\_is\_linked(&epi->rdllink))

list\_del\_init(&epi->rdllink);

spin\_unlock\_irqrestore(&ep->lock, flags);

kmem\_cache\_free(epi\_cache, epi);

return error;

}