# file.f\_ops.poll

通常的file.f\_ops.poll实现及相关结构体如下

// 通常的file.f\_ops.poll 方法的实现

unsigned int f\_op\_poll (struct file \*filp, struct poll\_table\_struct \*wait)

{

unsigned int mask = 0;

wait\_queue\_head\_t \* wait\_queue;

//1. 通过filep->private\_data 取得指示文件状态变化的wait queue head, 保存到wait\_queue

some\_code();

// 2. 调用poll\_wait 添加回调函数节点

poll\_wait(filp, wait\_queue, wait);

// 3. 取得当前就绪状态保存到mask

some\_code();

return mask;

}

// 通用的poll\_wait 函数, 文件的f\_ops->poll 通常会调用此函数

static inline void poll\_wait(struct file \* filp, wait\_queue\_head\_t \* wait\_address, poll\_table \*p)

{

if (p && p->\_qproc && wait\_address) {

// 调用\_qproc 在wait\_address 上添加节点和回调函数

// 调用 poll\_table\_struct 上的函数指针向wait\_address添加节点, 并设置节点的func

// (如果是select或poll 则是 \_\_pollwait, 如果是 epoll 则是 ep\_ptable\_queue\_proc),

p->\_qproc(filp, wait\_address, p);

}

}

// 文件描述符对应的file结构

struct file {

const struct file\_operations \*f\_op;

spinlock\_t f\_lock;

void \*private\_data; // 文件内部实现细节

#ifdef CONFIG\_EPOLL

/\* Used by fs/eventpoll.c to link all the hooks to this file \*/

struct list\_head f\_ep\_links;

struct list\_head f\_tfile\_llink;

#endif /\* #ifdef CONFIG\_EPOLL \*/

// 其他细节....

};

// 文件操作

struct file\_operations {

// 文件提供给poll/select/epoll

// 获取文件当前状态, 以及就绪通知接口函数

unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);

// 其他方法read write 等... ...

};

// select/poll/epoll 向文件注册就绪后回调节点的接口结构

typedef struct poll\_table\_struct {

// 向wait\_queue\_head 添加回调节点(wait\_queue\_t)的接口函数

poll\_queue\_proc \_qproc;

unsigned long \_key;

} poll\_table;

typedef void (\*poll\_queue\_proc)(struct file \*, wait\_queue\_head\_t \*, struct poll\_table\_struct \*);

// wait\_queue 头节点

struct \_\_wait\_queue\_head {

spinlock\_t lock;

struct list\_head task\_list;

};

typedef struct \_\_wait\_queue\_head wait\_queue\_head\_t;

// wait\_queue 节点

typedef struct \_\_wait\_queue wait\_queue\_t;

struct \_\_wait\_queue {

unsigned int flags;

#define WQ\_FLAG\_EXCLUSIVE 0x01

void \*private;

wait\_queue\_func\_t func;

struct list\_head task\_list;

};

typedef int (\*wait\_queue\_func\_t)(wait\_queue\_t \*wait, unsigned mode, int flags, void \*key);

每当影响文件状态的事件发生时(这通常由IO中断触发，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)。唤醒的相关函数如下

// 调用wait\_queue\_head\_t 上的节点的回调函数(wait\_queue\_t.func)

// 唤醒阻塞线程

void \_\_wake\_up(wait\_queue\_head\_t \*q, unsigned int mode,

int nr\_exclusive, void \*key)

{

unsigned long flags;

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

\_\_wake\_up\_common(q, mode, nr\_exclusive, 0, key);

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

}

static void \_\_wake\_up\_common(wait\_queue\_head\_t \*q, unsigned int mode,

int nr\_exclusive, int wake\_flags, void \*key)

{

wait\_queue\_t \*curr, \*next;

// 遍历并调用func 唤醒线程

list\_for\_each\_entry\_safe(curr, next, &q->task\_list, task\_list) {

unsigned flags = curr->flags;

if (curr->func(curr, mode, wake\_flags, key) &&

(flags & WQ\_FLAG\_EXCLUSIVE) && !--nr\_exclusive) {

break;

}

}

}

// 默认的唤醒函数,poll/select 设置的回调函数会调用此函数唤醒

// 直接唤醒等待队列上的线程,即将线程移到运行队列(rq)

int default\_wake\_function(wait\_queue\_t \*curr, unsigned mode, int wake\_flags,

void \*key)

{

// 这个函数比较复杂, 这里就不具体分析了

return try\_to\_wake\_up(curr->private, mode, wake\_flags);

}

poll 和 select

poll和select的实现基本上是一致的，只是传递参数有所不同，他们的基本流程如下：

复制用户数据到内核空间

估计超时时间

遍历每个文件并调用f\_op->poll 取得文件当前就绪状态, 如果前面遍历的文件都没有就绪,向文件插入wait\_queue节点

如果遍历过程中已经有就绪的文件转到6, 如果遍历过程中有信号产生转到3(重启poll或select), 否则挂起进程等待超时或唤醒

超时或被唤醒后再次遍历所有文件取得每个文件的就绪状态

将所有文件的就绪状态复制到用户空间

清理申请的资源

下面是poll/select共用的结构体及其相关功能:

// select/poll 对poll\_table的具体化实现

struct poll\_wqueues {

poll\_table pt;

struct poll\_table\_page \*table; // 如果inline\_entries 空间不足, 从poll\_table\_page 中分配

struct task\_struct \*polling\_task; // 调用poll 或select 的进程

int triggered; // 已触发标记

int error;

int inline\_index; // 下一个要分配的inline\_entrie 索引

struct poll\_table\_entry inline\_entries[N\_INLINE\_POLL\_ENTRIES];//

};

// 帮助管理select/poll 申请的内存

struct poll\_table\_page {

struct poll\_table\_page \* next; // 下一个 page

struct poll\_table\_entry \* entry; // 指向第一个entries

struct poll\_table\_entry entries[0];

};

// 与一个正在poll /select 的文件相关联,

struct poll\_table\_entry {

struct file \*filp; // 在poll/select中的文件

unsigned long key;

wait\_queue\_t wait; // 插入到wait\_queue\_head\_t 的节点

wait\_queue\_head\_t \*wait\_address; // 文件上的wait\_queue\_head\_t 地址

};

下图是 poll/select 实现公共部分的关系图，包喊了与文件直接的关系，以及函数之间的依赖。

下面是poll/select公用的一些函数

// poll\_wqueues 的初始化:

// 初始化 poll\_wqueues , \_\_pollwait会在文件就绪时被调用

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

{

// 初始化poll\_table, 相当于调用基类的构造函数

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

pwq->polling\_task = current;

pwq->triggered = 0;

pwq->error = 0;

pwq->table = NULL;

pwq->inline\_index = 0;

}

// 初始化poll\_table

static inline void init\_poll\_funcptr(poll\_table \*pt, poll\_queue\_proc qproc)

{

pt->\_qproc = qproc;

pt->\_key = ~0UL; /\* all events enabled \*/

}

// wait\_queue设置函数

// poll/select 向文件wait\_queue中添加节点的方法

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); //put\_file() in free\_poll\_entry()

entry->filp = filp;

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

entry->key = p->key;

// 设置回调为 pollwake

init\_waitqueue\_func\_entry(&entry->wait, pollwake);

entry->wait.private = pwq;

// 添加到等待队列

add\_wait\_queue(wait\_address, &entry->wait);

}

// 唤醒函数

// 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;

// 取得文件对应的poll\_table\_entry

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

// 过滤无关事件

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

return 0;

}

// 唤醒

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;

// 将调用进程 pwq->polling\_task 关联到 dummy\_wait

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

smp\_wmb();

pwq->triggered = 1;// 标记为已触发

// 唤醒调用进程

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

}

内存管理poll，select对poll\_table\_entry的申请和释放采用的是类似内存池的管理方式，先使用预分配的空间，预分配的空间不足时，分配一个内存页，使用内存页上的空间。

// 分配或使用已先前申请的 poll\_table\_entry,

static struct poll\_table\_entry \*poll\_get\_entry(struct poll\_wqueues \*p) {

struct poll\_table\_page \*table = p->table;

if (p->inline\_index inline\_entries + p->inline\_index++;

}

if (!table || POLL\_TABLE\_FULL(table)) {

struct poll\_table\_page \*new\_table;

new\_table = (struct poll\_table\_page \*) \_\_get\_free\_page(GFP\_KERNEL);

if (!new\_table) {

p->error = -ENOMEM;

return NULL;

}

new\_table->entry = new\_table->entries;

new\_table->next = table;

p->table = new\_table;

table = new\_table;

}

return table->entry++;

}

// 清理poll\_wqueues 占用的资源

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

{

struct poll\_table\_page \* p = pwq->table;

int i;

for (i = 0; i inline\_index; i++) {

free\_poll\_entry(pwq->inline\_entries + i); // 从文件的等待队列中移除, 并释放文件

}

while (p) {

struct poll\_table\_entry \* entry;

struct poll\_table\_page \*old;

entry = p->entry;

do {

entry--;

free\_poll\_entry(entry);

} while (entry > p->entries);

old = p;

p = p->next;

free\_page((unsigned long) old);

}

}

static void free\_poll\_entry(struct poll\_table\_entry \*entry)

{

// 从等待队列中删除

remove\_wait\_queue(entry->wait\_address, &entry->wait);

fput(entry->filp);

}

poll 的实现

// poll 使用的结构体

struct pollfd {

int fd; // 描述符

short events; // 关注的事件掩码

short revents; // 返回的事件掩码

};

// long sys\_poll(struct pollfd \*ufds, unsigned int nfds, long timeout\_msecs)

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;

// 将相对超时时间msec 转化为绝对时间保存到结构体to 中

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

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

}

// do sys poll

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

// do\_sys\_poll 被信号中断, 重新调用, 对使用者来说 poll 是不会被信号中断的.

if (ret == -EINTR) {

struct restart\_block \*restart\_block;

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

restart\_block->fn = do\_restart\_poll; // 设置重启的函数

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;

}

// ERESTART\_RESTARTBLOCK 不会返回给用户进程,

// 而是会被系统捕获, 然后调用 do\_restart\_poll,

ret = -ERESTART\_RESTARTBLOCK;

}

return ret;

}

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;

/\* 首先使用栈上的空间，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)) {

// 文件描述符数量超过当前进程限制

return -EINVAL;

}

// 复制用户空间数据到内核

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

for (;;) {

walk->next = NULL;

walk->len = len;

if (!len) {

break;

}

// 复制到当前的 entries

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

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

goto out\_fds;

}

todo -= walk->len;

if (!todo) {

break;

}

// 栈上空间不足，在堆上申请剩余部分

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\_wqueues 结构, 设置函数指针\_qproc 为\_\_pollwait

poll\_initwait(&table);

// poll

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

// 从文件wait queue 中移除对应的节点, 释放entry.

poll\_freewait(&table);

// 复制结果到用户空间

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

struct pollfd \*fds = walk->entries;

int j;

for (j = 0; j len; j++, ufds++)

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

goto out\_fds;

}

}

err = fdcount;

out\_fds:

// 释放申请的内存

walk = head->next;

while (walk) {

struct poll\_list \*pos = walk;

walk = walk->next;

kfree(pos);

}

return err;

}

// 真正的处理函数

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) {

// 已经超时就不用添加到文件描述符的等待队列了，

// 直接遍历所有文件描述符, 然后返回

pt = NULL;

timed\_out = 1;

}

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

// 估计进程等待时间，纳秒

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

}

// 遍历文件，为每个文件的等待队列添加唤醒函数(pollwake)

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 会向文件对应的wait queue 中添加节点

// 和回调函数(如果 pt 不为空)

// 并检查当前文件状态并设置返回的掩码

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

// 该文件已经准备好了.

// 不需要向其他文件的wait queue 中添加唤醒函数了.

count++;

pt = NULL;

}

}

}

// 下次循环的时候不需要向文件的wait queue 中添加节点,

// 因为前面的循环已经把该添加的都添加了

pt = NULL;

if (!count) { // 第一次遍历没有发现ready的文件

count = wait->error;

// 有信号产生

if (signal\_pending(current)) {

count = -EINTR;

}

}

if (count || timed\_out) { // 有ready的文件或已经超时

break;

}

// 转换为内核时间

if (end\_time && !to) {

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

to = &expire;

}

// 等待事件就绪, 如果有事件发生或超时，就再循

// 环一遍，取得事件状态掩码并计数,

// 注意此次循环中, 文件 wait queue 中的节点依然存在

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; // 如果没有 f\_op 或 f\_op->poll 则认为文件始终处于就绪状态.

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

if (pwait) {

pwait->key = pollfd->events | POLLERR | POLLHUP; // 等待事件掩码

}

// 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; // 移除不需要的状态掩码

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

}

}

pollfd->revents = mask; // 更新事件状态

return mask;

}

static long do\_restart\_poll(struct restart\_block \*restart\_block)

{

struct pollfd \_\_user \*ufds = restart\_block->poll.ufds;

int nfds = restart\_block->poll.nfds;

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

int ret;

if (restart\_block->poll.has\_timeout) {

// 获取先前的超时时间

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

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

to = &end\_time;

}

ret = do\_sys\_poll(ufds, nfds, to); // 重新调用 do\_sys\_poll

if (ret == -EINTR) {

// 又被信号中断了, 再次重启

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

ret = -ERESTART\_RESTARTBLOCK;

}

return ret;

}

select 实现

typedef struct {

unsigned long \*in, \*out, \*ex;

unsigned long \*res\_in, \*res\_out, \*res\_ex;

} fd\_set\_bits;

// long sys\_select(int n, fd\_set \*inp, fd\_set \*outp, fd\_set \*exp, struct timeval \*tvp)

SYSCALL\_DEFINE5(select, int, n, fd\_set \_\_user \*, inp, fd\_set \_\_user \*, outp,

fd\_set \_\_user \*, exp, struct timeval \_\_user \*, tvp)

{

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

struct timeval tv;

int ret;

if (tvp) {

if (copy\_from\_user(&tv, tvp, sizeof(tv))) {

return -EFAULT;

}

// 计算超时时间

to = &end\_time;

if (poll\_select\_set\_timeout(to,

tv.tv\_sec + (tv.tv\_usec / USEC\_PER\_SEC),

(tv.tv\_usec % USEC\_PER\_SEC) \* NSEC\_PER\_USEC)) {

return -EINVAL;

}

}

ret = core\_sys\_select(n, inp, outp, exp, to);

// 复制剩余时间到用户空间

ret = poll\_select\_copy\_remaining(&end\_time, tvp, 1, ret);

return ret;

}

int core\_sys\_select(int n, fd\_set \_\_user \*inp, fd\_set \_\_user \*outp,

fd\_set \_\_user \*exp, struct timespec \*end\_time)

{

fd\_set\_bits fds;

void \*bits;

int ret, max\_fds;

unsigned int size;

struct fdtable \*fdt;

//小对象使用栈上的空间,节约内存, 加快访问速度

long stack\_fds[SELECT\_STACK\_ALLOC/sizeof(long)];

ret = -EINVAL;

if (n files);

max\_fds = fdt->max\_fds;

rcu\_read\_unlock();

if (n > max\_fds) {

n = max\_fds;

}

size = FDS\_BYTES(n);

bits = stack\_fds;

if (size > sizeof(stack\_fds) / 6) {

// 栈上的空间不够, 申请内存, 全部使用堆上的空间

ret = -ENOMEM;

bits = kmalloc(6 \* size, GFP\_KERNEL);

if (!bits) {

goto out\_nofds;

}

}

fds.in = bits;

fds.out = bits + size;

fds.ex = bits + 2\*size;

fds.res\_in = bits + 3\*size;

fds.res\_out = bits + 4\*size;

fds.res\_ex = bits + 5\*size;

// 复制用户空间到内核

if ((ret = get\_fd\_set(n, inp, fds.in)) ||

(ret = get\_fd\_set(n, outp, fds.out)) ||

(ret = get\_fd\_set(n, exp, fds.ex))) {

goto out;

}

// 初始化fd set

zero\_fd\_set(n, fds.res\_in);

zero\_fd\_set(n, fds.res\_out);

zero\_fd\_set(n, fds.res\_ex);

ret = do\_select(n, &fds, end\_time);

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

wait = NULL;

timed\_out = 1;

}

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

// 估计需要等待的时间.

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, i 文件描述符

for (i = 0; i = n) {

break;

}

// 不需要监听描述符 i

if (!(bit & all\_bits)) {

continue;

}

// 取得文件结构

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

if (file) {

f\_op = file->f\_op;

// 没有 f\_op 的话就认为一直处于就绪状态

mask = DEFAULT\_POLLMASK;

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

// 设置等待事件的掩码

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

// 获取当前的就绪状态, 并添加到文件的对应等待队列中

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

// 和poll完全一样

}

fput\_light(file, fput\_needed);

// 释放文件

// 检查文件 i 是否已有事件就绪，

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

res\_in |= bit;

retval++;

// 如果已有就绪事件就不再向其他文件的

// 等待队列中添加回调函数

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; // 该添加回调函数的都已经添加了

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

break; // 信号发生，监听事件就绪或超时

}

if (table.error) {

retval = table.error; // 产生错误了

break;

}

// 转换到内核时间

if (end\_time && !to) {

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

to = &expire;

}

// 等待直到超时, 或由回调函数唤醒, 超时后会再次遍历文件描述符

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

to, slack)) {

timed\_out = 1;

}

}

poll\_freewait(&table);

return retval;

}

epoll

epoll 的实现比poll/select 复杂一些，这是因为

epoll\_wait, epoll\_ctl 的调用完全独立开来,内核需要锁机制对这些操作进行保护，并且需要持久的维护添加到epoll的文件

epoll本身也是文件，也可以被poll/select/epoll监视，这可能导致epoll之间循环唤醒的问题

单个文件的状态改变可能唤醒过多监听在其上的epoll，产生唤醒风暴

epoll各个功能的实现要非常小心面对这些问题，使得复杂度大大增加。

epoll 相关基本结构

// 对应于一个epoll描述符

struct eventpoll {

spinlock\_t lock;

struct mutex mtx;

wait\_queue\_head\_t wq; // sys\_epoll\_wait() 等待在这里

// f\_op->poll() 使用的, 被其他事件通知机制利用的wait\_address

wait\_queue\_head\_t poll\_wait;

/\* 已就绪的需要检查的epitem 列表 \*/

struct list\_head rdllist;

/\* 保存所有加入到当前epoll的文件对应的epitem\*/

struct rb\_root rbr;

// 当正在向用户空间复制数据时, 产生的可用文件

struct epitem \*ovflist;

/\* The user that created the eventpoll descriptor \*/

struct user\_struct \*user;

struct file \*file;

/\*优化循环检查，避免循环检查中重复的遍历 \*/

int visited;

struct list\_head visited\_list\_link;

}

// 对应于一个文件

struct epitem {

// 挂载到eventpoll 的红黑树节点

struct rb\_node rbn;

// 挂载到eventpoll.rdllist 的节点

struct list\_head rdllink;

// 连接到ovflist 的指针

struct epitem \*next;

/\* 文件描述符信息fd + file, 红黑树的key \*/

struct epoll\_filefd ffd;

/\* Number of active wait queue attached to poll operations \*/

int nwait;

// 当前文件的等待队列(eppoll\_entry)列表

// 同一个文件上可能会监视多种事件,

// 这些事件可能属于不同的wait\_queue中

// (取决于对应文件类型的实现),

// 所以需要使用链表

struct list\_head pwqlist;

// 当前epitem 的所有者

struct eventpoll \*ep;

/\* List header used to link this item to the &quot;struct file&quot; items list \*/

struct list\_head fllink;

/\* epoll\_ctl 传入的用户数据 \*/

struct epoll\_event event;

};

struct epoll\_filefd {

struct file \*file;

int fd;

};

// 与一个文件上的一个wait\_queue\_head 相关联

struct eppoll\_entry {

// List struct epitem.pwqlist

struct list\_head llink;

// 所有者

struct epitem \*base;

// 添加到wait\_queue 中的节点

wait\_queue\_t wait;

// 文件wait\_queue 头

wait\_queue\_head\_t \*whead;

};

struct epoll\_event {

\_\_u32 events;

\_\_u64 data;

} EPOLL\_PACKED;

epoll全图

文件系统和eventpoll\_poll的实现

// epoll 文件系统的相关实现

// epoll 文件系统初始化, 在系统启动时会调用

static int \_\_init eventpoll\_init(void)

{

// 限制可添加到epoll的最多的描述符数量

struct sysinfo si;

si\_meminfo(&si);

max\_user\_watches = (((si.totalram - si.totalhigh) / 25) epoll\_create() 和 ep\_eventpoll\_poll

// sys\_epoll\_create

SYSCALL\_DEFINE1(epoll\_create, int, size)

{

if (size file = file;

return fd;

out\_free\_fd:

put\_unused\_fd(fd);

out\_free\_ep:

ep\_free(ep);

return error;

}

// 由于epoll自身也是文件系统，其描述符也可以被poll/select/epoll监视，因此需要实现poll方法。

static const struct file\_operations eventpoll\_fops = {

.release = ep\_eventpoll\_release,

.poll = ep\_eventpoll\_poll,

.llseek = noop\_llseek,

};

static unsigned int ep\_eventpoll\_poll(struct file \*file, poll\_table \*wait)

{

int pollflags;

struct eventpoll \*ep = file->private\_data;

// 插入到wait\_queue

poll\_wait(file, &ep->poll\_wait, wait);

// 扫描就绪的文件列表, 调用每个文件上的poll 检测是否真的就绪,

// 然后复制到用户空间

// 文件列表中有可能有epoll文件, 调用poll的时候有可能会产生递归,

// 调用所以用ep\_call\_nested 包装一下, 防止死循环和过深的调用

// static struct nested\_calls poll\_readywalk\_ncalls;

pollflags = ep\_call\_nested(&poll\_readywalk\_ncalls, EP\_MAX\_NESTS,

ep\_poll\_readyevents\_proc, ep, ep, current);

return pollflags != -1 ? pollflags : 0;

}

static int ep\_poll\_readyevents\_proc(void \*priv, void \*cookie, int call\_nests)

{

return ep\_scan\_ready\_list(priv, ep\_read\_events\_proc, NULL, call\_nests + 1);

}

static int ep\_read\_events\_proc(struct eventpoll \*ep, struct list\_head \*head,

void \*priv)

{

struct epitem \*epi, \*tmp;

poll\_table pt;

init\_poll\_funcptr(&pt, NULL);

list\_for\_each\_entry\_safe(epi, tmp, head, rdllink) {

pt.\_key = epi->event.events;

if (epi->ffd.file->f\_op->poll(epi->ffd.file, &pt) &

epi->event.events) {

return POLLIN | POLLRDNORM;

} else {

// 这个事件虽然在就绪列表中,

// 但是实际上并没有就绪, 将他移除

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

}

}

return 0;

}

防止死循环和过深的递归调用

在epoll的调用过程中所有可能产生递归调用的函数都由函数

static int ep\_call\_nested(struct nested\_calls \*ncalls, int max\_nests,

int (\*nproc)(void \*, void \*, int), void \*priv,

void \*cookie, void \*ctx)；

进行包裹，如果出现死循环或过深的递归调用就会返回错误。

ep\_call\_nested 在每次调用函数nproc之前都向一个全局的链表中插入一个包含当前函数调用上下文ctx(进程，CPU，或epoll文件)和处理的对象标识cookie，然后通过遍历全局链表就可以知道当前递归的深度，并且如果链表中同样的上下文出现了同样的cookie标识cookie则说明产生了死循环。

struct nested\_call\_node {

struct list\_head llink;

void \*cookie; // 函数运行标识, 任务标志

void \*ctx; // 运行环境标识

};

struct nested\_calls {

struct list\_head tasks\_call\_list;

spinlock\_t lock;

};

// 限制epoll 中直接或间接递归调用的深度并防止死循环

// ctx: 任务运行上下文(进程, CPU 等)

// cookie: 每个任务的标识

// priv: 任务运行需要的私有数据

// 如果用面向对象语言实现应该就会是一个wapper类

static int ep\_call\_nested(struct nested\_calls \*ncalls, int max\_nests,

int (\*nproc)(void \*, void \*, int), void \*priv,

void \*cookie, void \*ctx)

{

int error, call\_nests = 0;

unsigned long flags;

struct list\_head \*lsthead = &ncalls->tasks\_call\_list;

struct nested\_call\_node \*tncur;

struct nested\_call\_node tnode;

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

// 检查原有的嵌套调用链表ncalls, 查看是否有深度超过限制的情况

list\_for\_each\_entry(tncur, lsthead, llink) {

// 同一上下文中(ctx)有相同的任务(cookie)说明产生了死循环

// 同一上下文的递归深度call\_nests 超过限制

if (tncur->ctx == ctx &&

(tncur->cookie == cookie || ++call\_nests > max\_nests)) {

error = -1;

}

goto out\_unlock;

}

/\* 将当前的任务请求添加到调用列表\*/

tnode.ctx = ctx;

tnode.cookie = cookie;

list\_add(&tnode.llink, lsthead);

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

/\* nproc 可能会导致递归调用(直接或间接)ep\_call\_nested

\* 如果发生递归调用, 那么在此函数返回之前,

\* ncalls 又会被加入额外的节点,

\* 这样通过前面的检测就可以知道递归调用的深度

\*/

error = (\*nproc)(priv, cookie, call\_nests);

/\* 从链表中删除当前任务\*/

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

list\_del(&tnode.llink);

out\_unlock:

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

return error;

}

epoll\_ctl 实现

// long epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

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))) { // 复制用户空间数据到内核

goto error\_return;

}

/\* Get the "struct file \*" for the eventpoll file \*/

error = -EBADF;

file = fget(epfd); // 取得 epfd 对应的文件

if (!file) {

goto error\_return;

}

tfile = fget(fd); // 取得目标文件

if (!tfile) {

goto error\_fput;

}

/\* 目标文件必须提供 poll 操作 \*/

error = -EPERM;

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

goto error\_tgt\_fput;

}

// 添加自身或epfd 不是epoll 句柄

error = -EINVAL;

if (file == tfile || !is\_file\_epoll(file)) {

goto error\_tgt\_fput;

}

// 取得内部结构eventpoll

ep = file->private\_data;

// EPOLL\_CTL\_MOD 不需要加全局锁 epmutex

if (op == EPOLL\_CTL\_ADD || op == EPOLL\_CTL\_DEL) {

mutex\_lock(&epmutex);

did\_lock\_epmutex = 1;

}

if (op == EPOLL\_CTL\_ADD) {

if (is\_file\_epoll(tfile)) {

error = -ELOOP;

// 目标文件也是epoll 检测是否有循环

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

goto error\_tgt\_fput;

}

} else

{

// 将目标文件添加到 epoll 全局的tfile\_check\_list 中

list\_add(&tfile->f\_tfile\_llink, &tfile\_check\_list);

}

}

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

// 以tfile 和fd 为key 在rbtree 中查找文件对应的epitem

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

error = -EINVAL;

switch (op) {

case EPOLL\_CTL\_ADD:

if (!epi) {

// 没找到, 添加额外添加ERR HUP 事件

epds.events |= POLLERR | POLLHUP;

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

} else {

error = -EEXIST;

}

// 清空文件检查列表

clear\_tfile\_check\_list();

break;

case EPOLL\_CTL\_DEL:

if (epi) {

error = ep\_remove(ep, epi);

} 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 (did\_lock\_epmutex) {

mutex\_unlock(&epmutex);

}

fput(tfile);

error\_fput:

fput(file);

error\_return:

return error;

}

static LIST\_HEAD(visited\_list);

// 检查 file (epoll)和ep 之间是否有循环

static int ep\_loop\_check(struct eventpoll \*ep, struct file \*file)

{

int ret;

struct eventpoll \*ep\_cur, \*ep\_next;

ret = ep\_call\_nested(&poll\_loop\_ncalls, EP\_MAX\_NESTS,

ep\_loop\_check\_proc, file, ep, current);

/\* 清除链表和标志 \*/

list\_for\_each\_entry\_safe(ep\_cur, ep\_next, &visited\_list,

visited\_list\_link) {

ep\_cur->visited = 0;

list\_del(&ep\_cur->visited\_list\_link);

}

return ret;

}

static int ep\_loop\_check\_proc(void \*priv, void \*cookie, int call\_nests)

{

int error = 0;

struct file \*file = priv;

struct eventpoll \*ep = file->private\_data;

struct eventpoll \*ep\_tovisit;

struct rb\_node \*rbp;

struct epitem \*epi;

mutex\_lock\_nested(&ep->mtx, call\_nests + 1);

// 标记当前为已遍历

ep->visited = 1;

list\_add(&ep->visited\_list\_link, &visited\_list);

// 遍历所有ep 监视的文件

for (rbp = rb\_first(&ep->rbr); rbp; rbp = rb\_next(rbp)) {

epi = rb\_entry(rbp, struct epitem, rbn);

if (unlikely(is\_file\_epoll(epi->ffd.file))) {

ep\_tovisit = epi->ffd.file->private\_data;

// 跳过先前已遍历的, 避免循环检查

if (ep\_tovisit->visited) {

continue;

}

// 所有ep监视的未遍历的epoll

error = ep\_call\_nested(&poll\_loop\_ncalls, EP\_MAX\_NESTS,

ep\_loop\_check\_proc, epi->ffd.file,

ep\_tovisit, current);

if (error != 0) {

break;

}

} else {

// 文件不在tfile\_check\_list 中, 添加

// 最外层的epoll 需要检查子epoll监视的文件

if (list\_empty(&epi->ffd.file->f\_tfile\_llink))

list\_add(&epi->ffd.file->f\_tfile\_llink,

&tfile\_check\_list);

}

}

mutex\_unlock(&ep->mtx);

return error;

}

// EPOLL\_CTL\_ADD

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;

// 增加监视文件数

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

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

return -ENOSPC;

}

// 分配初始化 epi

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

return -ENOMEM;

}

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

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

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

epi->ep = ep;

// 初始化红黑树中的key

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

// 直接复制用户结构

epi->event = \*event;

epi->nwait = 0;

epi->next = EP\_UNACTIVE\_PTR;

// 初始化临时的 epq

epq.epi = epi;

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

epq.pt.\_key = event->events;

// 内部会调用ep\_ptable\_queue\_proc, 在文件对应的wait queue head 上

// 注册回调函数, 并返回当前文件的状态

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

// 检查错误

error = -ENOMEM;

if (epi->nwait < 0) { // f\_op->poll 过程出错

goto error\_unregister;

}

// 添加当前的epitem 到文件的f\_ep\_links 链表

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

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

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

// 插入epi 到rbtree

ep\_rbtree\_insert(ep, epi);

/\* now check if we've created too many backpaths \*/

error = -EINVAL;

if (reverse\_path\_check()) {

goto error\_remove\_epi;

}

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

/\* 文件已经就绪插入到就绪链表rdllist \*/

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

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

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

// 通知sys\_epoll\_wait , 调用回调函数唤醒sys\_epoll\_wait 进程

{

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

}

// 先不通知调用eventpoll\_poll 的进程

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

pwake++;

}

}

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

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

if (pwake)

// 安全通知调用eventpoll\_poll 的进程

{

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

}

return 0;

error\_remove\_epi:

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

// 删除文件上的 epi

if (ep\_is\_linked(&epi->fllink)) {

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

}

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

// 从红黑树中删除

rb\_erase(&epi->rbn, &ep->rbr);

error\_unregister:

// 从文件的wait\_queue 中删除, 释放epitem 关联的所有eppoll\_entry

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.

\*/

// TODO:

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

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

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

}

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

// 释放epi

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

return error;

}

#define PATH\_ARR\_SIZE 5

// 在EPOLL\_CTL\_ADD 时, 检查是否有可能产生唤醒风暴

// epoll 允许的单个文件的唤醒深度小于5, 例如

// 一个文件最多允许唤醒1000个深度为1的epoll描述符,

//允许所有被单个文件直接唤醒的epoll描述符再次唤醒的epoll描述符总数是500

//

// 深度限制

static const int path\_limits[PATH\_ARR\_SIZE] = { 1000, 500, 100, 50, 10 };

// 计算出来的深度

static int path\_count[PATH\_ARR\_SIZE];

static int path\_count\_inc(int nests)

{

/\* Allow an arbitrary number of depth 1 paths \*/

if (nests == 0) {

return 0;

}

if (++path\_count[nests] > path\_limits[nests]) {

return -1;

}

return 0;

}

static void path\_count\_init(void)

{

int i;

for (i = 0; i < PATH\_ARR\_SIZE; i++) {

path\_count[i] = 0;

}

}

// 唤醒风暴检查函数

static int reverse\_path\_check(void)

{

int error = 0;

struct file \*current\_file;

/\* let's call this for all tfiles \*/

// 遍历全局tfile\_check\_list 中的文件, 第一级

list\_for\_each\_entry(current\_file, &tfile\_check\_list, f\_tfile\_llink) {

// 初始化

path\_count\_init();

// 限制递归的深度, 并检查每个深度上唤醒的epoll 数量

error = ep\_call\_nested(&poll\_loop\_ncalls, EP\_MAX\_NESTS,

reverse\_path\_check\_proc, current\_file,

current\_file, current);

if (error) {

break;

}

}

return error;

}

static int reverse\_path\_check\_proc(void \*priv, void \*cookie, int call\_nests)

{

int error = 0;

struct file \*file = priv;

struct file \*child\_file;

struct epitem \*epi;

list\_for\_each\_entry(epi, &file->f\_ep\_links, fllink) {

// 遍历监视file 的epoll

child\_file = epi->ep->file;

if (is\_file\_epoll(child\_file)) {

if (list\_empty(&child\_file->f\_ep\_links)) {

// 没有其他的epoll监视当前的这个epoll,

// 已经是叶子了

if (path\_count\_inc(call\_nests)) {

error = -1;

break;

}

} else {

// 遍历监视这个epoll 文件的epoll,

// 递归调用

error = ep\_call\_nested(&poll\_loop\_ncalls,

EP\_MAX\_NESTS,

reverse\_path\_check\_proc,

child\_file, child\_file,

current);

}

if (error != 0) {

break;

}

} else {

// 不是epoll , 不可能吧?

printk(KERN\_ERR "reverse\_path\_check\_proc: "

"file is not an ep!\n");

}

}

return error;

}

static void ep\_poll\_safewake(wait\_queue\_head\_t \*wq)

{

int this\_cpu = get\_cpu();

ep\_call\_nested(&poll\_safewake\_ncalls, EP\_MAX\_NESTS,

ep\_poll\_wakeup\_proc, NULL, wq, (void \*) (long) this\_cpu);

put\_cpu();

}

static int ep\_poll\_wakeup\_proc(void \*priv, void \*cookie, int call\_nests)

{

ep\_wake\_up\_nested((wait\_queue\_head\_t \*) cookie, POLLIN,

1 + call\_nests);

return 0;

}

static inline void ep\_wake\_up\_nested(wait\_queue\_head\_t \*wqueue,

unsigned long events, int subclass)

{

// 这回唤醒所有正在等待此epfd 的select/epoll/poll 等

// 如果唤醒的是epoll 就可能唤醒其他的epoll, 产生连锁反应

// 这个很可能在中断上下文中被调用

wake\_up\_poll(wqueue, events);

}

/\*

\* Removes a "struct epitem" from the eventpoll hash and deallocates

\* all the associated resources.

\*/

static int ep\_remove(struct eventpoll \*ep, struct epitem \*epi)

{

int error;

unsigned long flags;

struct file \*file = epi->ffd.file;

/\*

\* Removes poll wait queue hooks. We \_have\_ to do this without holding

\* the "ep->lock" otherwise a deadlock might occur. This because of the

\* sequence of the lock acquisition. Here we do "ep->lock" then the wait

\* queue head lock when unregistering the wait queue. The wakeup callback

\* will run by holding the wait queue head lock and will call our callback

\* that will try to get "ep->lock".

\*/

ep\_unregister\_pollwait(ep, epi);

/\* Remove the current item from the list of epoll hooks \*/

spin\_lock(&file->f\_ep\_lock);

if (ep\_is\_linked(&epi->fllink)) {

ep\_list\_del(&epi->fllink);

}

spin\_unlock(&file->f\_ep\_lock);

/\* We need to acquire the write IRQ lock before calling ep\_unlink() \*/

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

/\* Really unlink the item from the hash \*/

error = ep\_unlink(ep, epi);

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

if (error) {

goto eexit\_1;

}

/\* At this point it is safe to free the eventpoll item \*/

ep\_release\_epitem(epi);

error = 0;

eexit\_1:

DNPRINTK(3, (KERN\_INFO "[%p] eventpoll: ep\_remove(%p, %p) = %d\n",

current, ep, file, error));

return error;

}

/\*

\* Modify the interest event mask by dropping an event if the new mask

\* has a match in the current file status.

\*/

static int ep\_modify(struct eventpoll \*ep, struct epitem \*epi, struct epoll\_event \*event)

{

int pwake = 0;

unsigned int revents;

unsigned long flags;

/\*

\* Set the new event interest mask before calling f\_op->poll(), otherwise

\* a potential race might occur. In fact if we do this operation inside

\* the lock, an event might happen between the f\_op->poll() call and the

\* new event set registering.

\*/

epi->event.events = event->events;

/\*

\* Get current event bits. We can safely use the file\* here because

\* its usage count has been increased by the caller of this function.

\*/

revents = epi->ffd.file->f\_op->poll(epi->ffd.file, NULL);

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

/\* Copy the data member from inside the lock \*/

epi->event.data = event->data;

/\*

\* If the item is not linked to the hash it means that it's on its

\* way toward the removal. Do nothing in this case.

\*/

if (ep\_rb\_linked(&epi->rbn)) {

/\*

\* If the item is "hot" and it is not registered inside the ready

\* list, push it inside. If the item is not "hot" and it is currently

\* registered inside the ready list, unlink it.

\*/

if (revents & event->events) {

if (!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, TASK\_UNINTERRUPTIBLE |

TASK\_INTERRUPTIBLE);

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

pwake++;

}

}

}

}

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

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

if (pwake) {

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

}

return 0;

}

/\*

\* This function unregister poll callbacks from the associated file descriptor.

\* Since this must be called without holding "ep->lock" the atomic exchange trick

\* will protect us from multiple unregister.

\*/

static void ep\_unregister\_pollwait(struct eventpoll \*ep, struct epitem \*epi)

{

struct list\_head \*lsthead = &epi->pwqlist;

struct eppoll\_entry \*pwq;

while (!list\_empty(lsthead)) {

// 从文件poll wait queue 中删除

pwq = list\_first\_entry(lsthead, struct eppoll\_entry, llink);

list\_del(&pwq->llink);

ep\_remove\_wait\_queue(pwq);

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

}

}

epoll\_wait实现

SYSCALL\_DEFINE4(epoll\_wait, int, epfd, struct epoll\_event \_\_user \*, events,

int, maxevents, int, timeout)

{

int error;

struct file \*file;

struct eventpoll \*ep;

/\* The maximum number of event must be greater than zero \*/

if (maxevents <= 0 || maxevents > EP\_MAX\_EVENTS) {

return -EINVAL;

}

/\* Verify that the area passed by the user is writeable \*/

if (!access\_ok(VERIFY\_WRITE, events, maxevents \* sizeof(struct epoll\_event))) {

error = -EFAULT;

goto error\_return;

}

/\* Get the "struct file \*" for the eventpoll file \*/

error = -EBADF;

file = fget(epfd);

if (!file) {

goto error\_return;

}

/\*

\* We have to check that the file structure underneath the fd

\* the user passed to us \_is\_ an eventpoll file.

\*/

error = -EINVAL;

if (!is\_file\_epoll(file)) {

goto error\_fput;

}

/\*

\* At this point it is safe to assume that the "private\_data" contains

\* our own data structure.

\*/

ep = file->private\_data;

/\* Time to fish for events ... \*/

error = ep\_poll(ep, events, maxevents, timeout);

error\_fput:

fput(file);

error\_return:

return error;

}

static int ep\_poll(struct eventpoll \*ep, struct epoll\_event \_\_user \*events,

int maxevents, long timeout)

{

int res = 0, eavail, timed\_out = 0;

unsigned long flags;

long slack = 0;

wait\_queue\_t wait;

ktime\_t expires, \*to = NULL;

if (timeout > 0) {

struct timespec end\_time = ep\_set\_mstimeout(timeout);

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

to = &expires;

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

} else if (timeout == 0) {

/\*

\* Avoid the unnecessary trip to the wait queue loop, if the

\* caller specified a non blocking operation.

\*/

timed\_out = 1;

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

goto check\_events;

}

fetch\_events:

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

// 没有可用的事件

if (!ep\_events\_available(ep)) {

// 设置默认的唤醒函数

init\_waitqueue\_entry(&wait, current);

\_\_add\_wait\_queue\_exclusive(&ep->wq, &wait);

for (;;) {

/\*

\* We don't want to sleep if the ep\_poll\_callback() sends us

\* a wakeup in between. That's why we set the task state

\* to TASK\_INTERRUPTIBLE before doing the checks.

\*/

set\_current\_state(TASK\_INTERRUPTIBLE);

if (ep\_events\_available(ep) || timed\_out) {

break;

}

if (signal\_pending(current)) {

res = -EINTR;

break;

}

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

if (!schedule\_hrtimeout\_range(to, slack, HRTIMER\_MODE\_ABS)) {

timed\_out = 1;

}

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

}

\_\_remove\_wait\_queue(&ep->wq, &wait);

set\_current\_state(TASK\_RUNNING);

}

check\_events:

/\* Is it worth to try to dig for events ? \*/

eavail = ep\_events\_available(ep);

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

/\*

\* Try to transfer events to user space. In case we get 0 events and

\* there's still timeout left over, we go trying again in search of

\* more luck.

\*/

if (!res && eavail &&

!(res = ep\_send\_events(ep, events, maxevents)) // 复制事件到用户空间

&& !timed\_out) {

goto fetch\_events;

}

return res;

}

static inline int ep\_events\_available(struct eventpoll \*ep)

{

return !list\_empty(&ep->rdllist) || ep->ovflist != EP\_UNACTIVE\_PTR;

}

struct ep\_send\_events\_data {

int maxevents;

struct epoll\_event \_\_user \*events;

};

static int ep\_send\_events(struct eventpoll \*ep,

struct epoll\_event \_\_user \*events, int maxevents)

{

struct ep\_send\_events\_data esed;

esed.maxevents = maxevents;

esed.events = events;

return ep\_scan\_ready\_list(ep, ep\_send\_events\_proc, &esed, 0);

}

static int ep\_scan\_ready\_list(struct eventpoll \*ep,

int (\*sproc)(struct eventpoll \*,

struct list\_head \*, void \*),

void \*priv,

int depth)

{

int error, pwake = 0;

unsigned long flags;

struct epitem \*epi, \*nepi;

LIST\_HEAD(txlist);

/\*

\* We need to lock this because we could be hit by

\* eventpoll\_release\_file() and epoll\_ctl().

\*/

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

/\*

\* Steal the ready list, and re-init the original one to the

\* empty list. Also, set ep->ovflist to NULL so that events

\* happening while looping w/out locks, are not lost. We cannot

\* have the poll callback to queue directly on ep->rdllist,

\* because we want the "sproc" callback to be able to do it

\* in a lockless way.

\*/

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

// 移动rdllist 到新的链表txlist

list\_splice\_init(&ep->rdllist, &txlist);

ep->ovflist = NULL;

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

/\*

\* Now call the callback function.

\*/

// 调用扫描函数处理txlist

error = (\*sproc)(ep, &txlist, priv);

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

/\*

\* During the time we spent inside the "sproc" callback, some

\* other events might have been queued by the poll callback.

\* We re-insert them inside the main ready-list here.

\*/

// 调用 sproc 时可能有新的事件

for (nepi = ep->ovflist; (epi = nepi) != NULL;

nepi = epi->next, epi->next = EP\_UNACTIVE\_PTR) {

/\*

\* We need to check if the item is already in the list.

\* During the "sproc" callback execution time, items are

\* queued into ->ovflist but the "txlist" might already

\* contain them, and the list\_splice() below takes care of them.

\*/

// epi 不在rdllist, 插入

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

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

}

}

/\*

\* We need to set back ep->ovflist to EP\_UNACTIVE\_PTR, so that after

\* releasing the lock, events will be queued in the normal way inside

\* ep->rdllist.

\*/

ep->ovflist = EP\_UNACTIVE\_PTR;

/\*

\* Quickly re-inject items left on "txlist".

\*/

// 将处理后的 txlist 链接到 rdllist

list\_splice(&txlist, &ep->rdllist);

if (!list\_empty(&ep->rdllist)) {

/\*

\* Wake up (if active) both the eventpoll wait list and

\* the ->poll() wait list (delayed after we release the lock).

\*/

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

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

}

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

pwake++;

}

}

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

mutex\_unlock(&ep->mtx);

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

if (pwake) {

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

}

return error;

}

static int ep\_send\_events\_proc(struct eventpoll \*ep, struct list\_head \*head,

void \*priv)

{

struct ep\_send\_events\_data \*esed = priv;

int eventcnt;

unsigned int revents;

struct epitem \*epi;

struct epoll\_event \_\_user \*uevent;

/\*

\* We can loop without lock because we are passed a task private list.

\* Items cannot vanish during the loop because ep\_scan\_ready\_list() is

\* holding "mtx" during this call.

\*/

// 遍历已就绪链表

for (eventcnt = 0, uevent = esed->events;

!list\_empty(head) && eventcnt < esed->maxevents;) {

epi = list\_first\_entry(head, struct epitem, rdllink);

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

// 获取ready 事件掩码

revents = epi->ffd.file->f\_op->poll(epi->ffd.file, NULL) &

epi->event.events;

/\*

\* If the event mask intersect the caller-requested one,

\* deliver the event to userspace. Again, ep\_scan\_ready\_list()

\* is holding "mtx", so no operations coming from userspace

\* can change the item.

\*/

if (revents) {

// 事件就绪, 复制到用户空间

if (\_\_put\_user(revents, &uevent->events) ||

\_\_put\_user(epi->event.data, &uevent->data)) {

list\_add(&epi->rdllink, head);

return eventcnt ? eventcnt : -EFAULT;

}

eventcnt++;

uevent++;

if (epi->event.events & EPOLLONESHOT) {

epi->event.events &= EP\_PRIVATE\_BITS;

} else if (!(epi->event.events & EPOLLET)) {

/\*

\* If this file has been added with Level

\* Trigger mode, we need to insert back inside

\* the ready list, so that the next call to

\* epoll\_wait() will check again the events

\* availability. At this point, no one can insert

\* into ep->rdllist besides us. The epoll\_ctl()

\* callers are locked out by

\* ep\_scan\_ready\_list() holding "mtx" and the

\* poll callback will queue them in ep->ovflist.

\*/

// 不是边缘模式, 再次添加到ready list,

// 下次epoll\_wait 时直接进入此函数检查ready list,

// 然后发送给用户

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

}

// 如果是边缘模式, 只有当文件状态发生改变时,

// 才文件会再次触发wait\_address 上wait\_queue的回调函数,

}

}

return eventcnt;

}