12.3 多路复用 io-select - 底层原理分析_物联网 / 嵌入式工程师 - 慕课网

> 慕课网慕课教程 12.3 多路复用 io-select – 底层原理分析涵盖海量编程基础技术教程，以图文图表的形式，把晦涩难懂的编程专业用语，以通俗易懂的方式呈现给用户。

- 文件描述符集合在定义时的类型为 fd_set, 在内核中的定义如下:

```
typedef long int __fd_mask;

#define __NFDBITS   (8 * (int) sizeof (__fd_mask))

#define __FD_MASK(d)    ((__fd_mask) (1UL << ((d) % __NFDBITS)))

#define __FD_SETSIZE 1024

typedef struct
{
   __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
} fd_set;
```

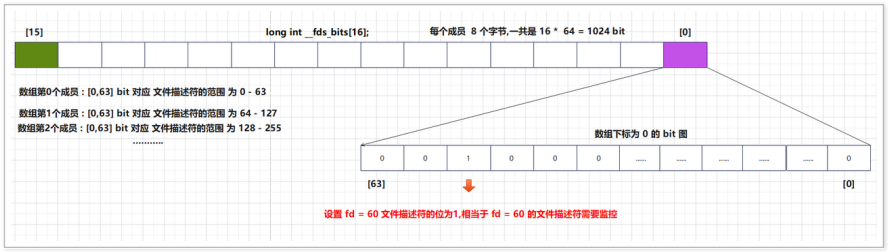
- 数组的类型为 long int 类型,__fd_mask 为 typedef 产生的类型, 在 64 位系统中 long int 的大小为 8 个字节

```
typedef long int __fd_mask;
```

- __NFDBITS 的大小 为 64 ,__FD_SETSIZE 为 1024 ， 经过计算之后，大小为 16

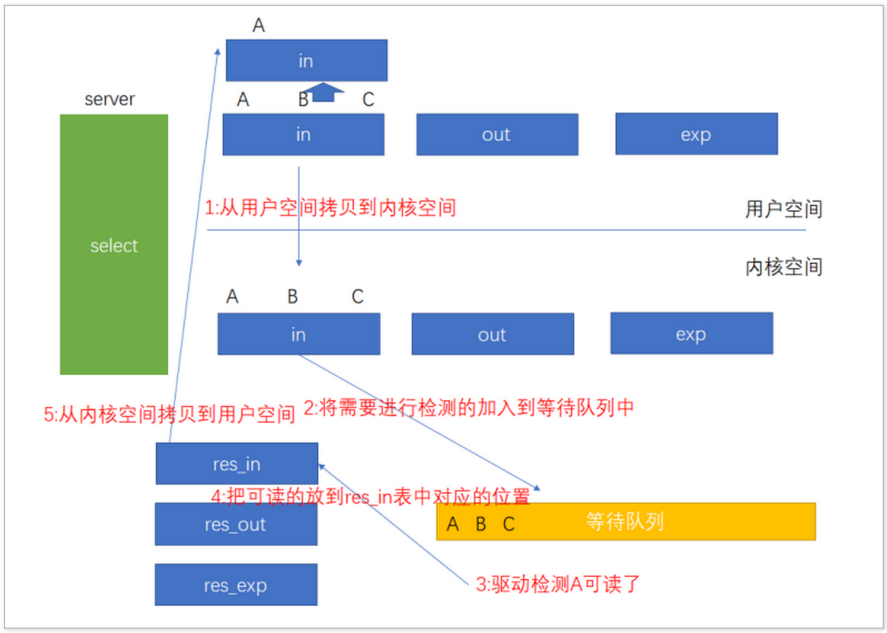
```
long int __fds_bits[16];
```

- 通过上面的计算之后, 数组的大小为 16
- 文件描述符集合的数组最终在存储时，是使用了位图的方式来记录相应的文件描述符，具体原理如下:
    - 数组中没有直接存储文件描述符，而是使用某一位来表示该文件描述符是否需要监控
    - 需要监控的文件描述符需要转成数组的某一个元素的某一位，然后将对应的位设置为 1
    -



- 比如当 fd = 60 的成员需要监控，则需要将数组的第 0 个成员的第 [60] bit 设置为 1,
- 当 fd = 64 时，则需要将数组的第 1 个成员的第 [0] bit 设置为 1
- 总结:

- 从上面的文件描述符集合内存管理可以分析出, select 最终只能存储 1024 个文件描述符



- 在 select() 函数中一共需要使用三个文件描述符集合, 分别是
  - in : 读文件描述符集合, 主要包含 需要进行读的文件描述符的集合, 反映在底层实际可以从设备中读取数据
  - out : 写文件描述符集合, 主要包含 需要进行写的文件描述符的集合, 反映在底层实际可以将数据写入到设备中
  - exp : 其他文件描述符集合, 主要包含其他类型的操作的文件描述符集合

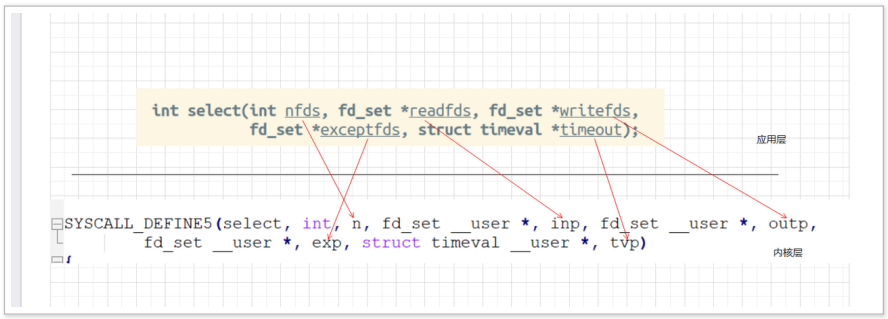
- 一旦调用了 select() 函数, 内核则做了如下事情:
  - 从用户空间将集合的文件描述符拷贝到内核空间
  - 循环遍历 fd_set 中所有的文件描述符, 来检测是否有文件描述符可进行 I/O 操作
    - 如果有文件描述符可进行 I/O 操作, 则设置返回的文件描述符集对应位为 1(res_in,res_out,res_exp)，表示可以进行 I/O 操作跳出循环, 直接返回, 最终会赋值给 in,out,exp 文件描述符集合
    - 如果没有文件描述符可进行 I/O 操作, 则继续循环检测, 如果设置 timeout , 则在超时后返回，此时 select() 函数返回 0

- select() 函数 减少了多进程 / 多线程的开销, 但仍然有很多缺点:
  - 每次调用 select() 函数都需要将 fd 集合拷贝到内核空间, 这个开销在 fd 很多时越大
  - 每次都需要遍历所有的文件描述符集合, 这个开销在 fd 很多时越大
  - 支持的文件描述符只有 1024

```
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;

                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);
        ret = poll_select_copy_remaining(&end_time, tvp, 1, ret);
```

```
        return ret;
}
```

- 在 select 系统调用中对应的形式参数的值都是由 应用层 select 函数传递过来的

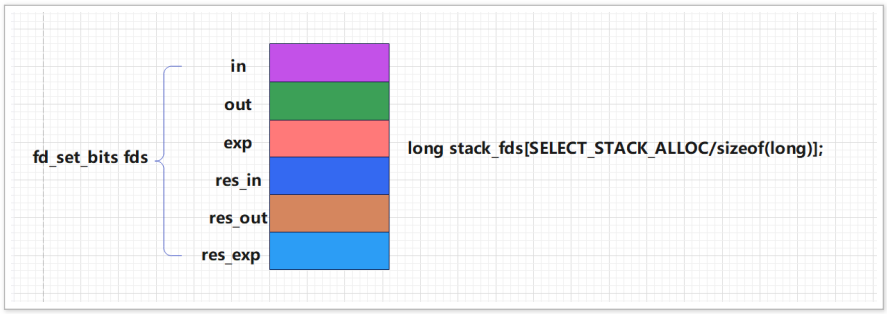


- sys_select 主要调用的核心函数为 core_sys_select, 具体定义如下:

```
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;

        long stack_fds[SELECT_STACK_ALLOC/sizeof(long)];

        ret = -EINVAL;
        if (n < 0)
                goto out_nofds;


        rcu_read_lock();
        fdt = files_fdtable(current->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) {

                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;

        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;
        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 < 0)
                goto out;
        if (!ret) {
                ret = -ERESTARTNOHAND;
                if (signal_pending(current))
                        goto out;
                ret = 0;
        }

        if (set_fd_set(n, inp, fds.res_in) ||
            set_fd_set(n, outp, fds.res_out) ||
```

```
            set_fd_set(n, exp, fds.res_ex))
                ret = -EFAULT;

out:
        if (bits != stack_fds)
                kfree(bits);
out_nofds:
        return ret;
}
```

- 上面的函数具体完成的任务如下:
    - 分配数组空间，用于存储文件描述符
    - ```
      long stack_fds[SELECT_STACK_ALLOC/sizeof(long)];
      ```
    - 计算分配的空间是否足够，如果不够，则需要进一步分配
    - ```
      size = FDS_BYTES(n);
      bits = stack_fds;

      if (size > sizeof(stack_fds) / 6) {

          ret = -ENOMEM;
          bits = kmalloc(6 * size, GFP_KERNEL);
          if (!bits)
             goto out_nofds;
      }
      ```
    - fds 定义的类型 fd_set_bits, 用于管理 6 个集合的空间
    - ```
      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;
      ```
    - fd_set_bits 具体在内核中的定义如下:
    - ```
      typedef struct {
          unsigned long *in, *out, *ex;
          unsigned long *res_in, *res_out, *res_ex;
      } fd_set_bits;
      ```
    - 具体内存管理结构如下:



- 调用 do_select 函数将进行轮询检测，并将就绪的文件描述符保存到结果集合中
- 将结果集合拷贝到应用层集合中

```
    if (set_fd_set(n, inp, fds.res_in) ||
        set_fd_set(n, outp, fds.res_out) ||
        set_fd_set(n, exp, fds.res_ex))
            ret = -EFAULT;
```

```
static inline unsigned long __must_check
set_fd_set(unsigned long nr, void __user *ufdset, unsigned long *fdset)
{
    if (ufdset)
        return __copy_to_user(ufdset, fdset, FDS_BYTES(nr));
    return 0;
}
```

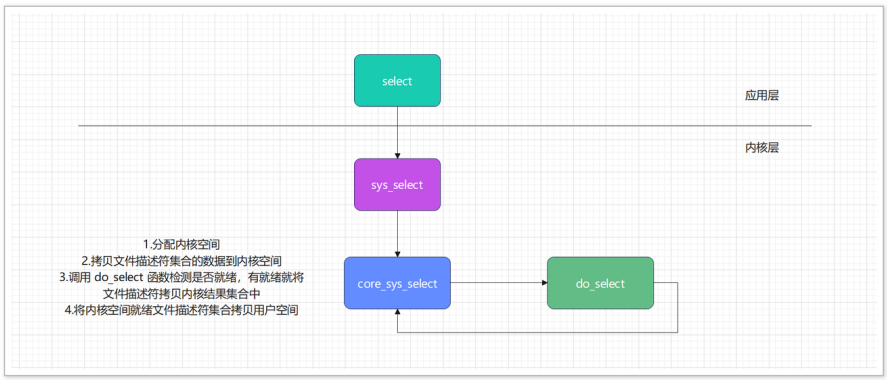
```
int do_select(int n, fd_set_bits *fds, struct timespec *end_time)
{
        ktime_t expire, *to = NULL;
        struct poll_wqueues table;
        poll_table *wait;
        int retval, i, timed_out = 0;
        unsigned long slack = 0;
        unsigned int busy_flag = net_busy_loop_on() ? POLL_BUSY_LOOP : 0;
        unsigned long busy_end = 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->_qproc = 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;
                bool can_busy_loop = false;

                inp = fds->in; outp = fds->out; exp = fds->ex;
                rinp = fds->res_in; routp = fds->res_out; rexp = fds->res_ex;

                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;

                        in = *inp++; out = *outp++; ex = *exp++;
                        all_bits = in | out | ex;
                        if (all_bits == 0) {
                                i += BITS_PER_LONG;
                                continue;
                        }


                        for (j = 0; j < BITS_PER_LONG; ++j, ++i, bit <<= 1) {
                                struct fd f;
                                if (i >= n)
                                        break;
                                if (!(bit & all_bits))
                                        continue;
                                f = fdget(i);
                                if (f.file) {
                                        const struct file_operations *f_op;
                                        f_op = f.file->f_op;
                                        mask = DEFAULT_POLLMASK;

                                        if (f_op->poll) {
                                                wait_key_set(wait, in, out,
                                                             bit, busy_flag);
                                                mask = (*f_op->poll)(f.file, wait);
                                        }
                                        fdput(f);
                                        if ((mask & POLLIN_SET) && (in & bit)) {
                                                res_in |= bit;
                                                retval++;
                                                wait->_qproc = NULL;
                                        }
                                        if ((mask & POLLOUT_SET) && (out & bit)) {
                                                res_out |= bit;
                                                retval++;
                                                wait->_qproc = NULL;
                                        }
                                        if ((mask & POLLEX_SET) && (ex & bit)) {
                                                res_ex |= bit;
                                                retval++;
                                                wait->_qproc = NULL;
                                        }

                                        if (retval) {
                                                can_busy_loop = false;
                                                busy_flag = 0;
```

```
                                } else if (busy_flag & mask)
                                        can_busy_loop = true;

                        }
                }

                if (res_in)
                        *rinp = res_in;
                if (res_out)
                        *routp = res_out;
                if (res_ex)
                        *rexp = res_ex;
                cond_resched();
        }
        wait->_qproc = NULL;
        if (retval || timed_out || signal_pending(current))
                break;
        if (table.error) {
                retval = table.error;
                break;
        }


        if (can_busy_loop && !need_resched()) {
                if (!busy_end) {
                        busy_end = busy_loop_end_time();
                        continue;
                }
                if (!busy_loop_timeout(busy_end))
                        continue;
        }
        busy_flag = 0;


        if (end_time && !to) {
                expire = timespec_to_ktime(*end_time);
                to = &expire;
        }

        if (!poll_schedule_timeout(&table, TASK_INTERRUPTIBLE,
                                   to, slack))
                timed_out = 1;
    }

    poll_freewait(&table);

    return retval;
}
```

- do_select 函数的核心功能
    - 循环遍历文件描述符集合, 并将就绪的文件描述符保存到 结果集合中

```
 for (;;) {
        unsigned long *rinp, *routp, *rexp, *inp, *outp, *exp;
        bool can_busy_loop = false;

        inp = fds->in; outp = fds->out; exp = fds->ex;
        rinp = fds->res_in; routp = fds->res_out; rexp = fds->res_ex;

        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;

                in = *inp++; out = *outp++; ex = *exp++;
                all_bits = in | out | ex;
                if (all_bits == 0) {
                        i += BITS_PER_LONG;
                        continue;
                }

                for (j = 0; j < BITS_PER_LONG; ++j, ++i, bit <<= 1) {
                        struct fd f;
                        if (i >= n)
                                break;
                        if (!(bit & all_bits))
                                continue;
                        f = fdget(i);
                        if (f.file) {
                                const struct file_operations *f_op;
                                f_op = f.file->f_op;
                                mask = DEFAULT_POLLMASK;
                                if (f_op->poll) {
                                        wait_key_set(wait, in, out,
```

```
                                        bit, busy_flag);
                                mask = (*f_op->poll)(f.file, wait);
                        }
                        fdput(f);

                        if ((mask & POLLIN_SET) && (in & bit)) {
                                res_in |= bit;
                                retval++;
                                wait->_qproc = NULL;
                        }

                        if ((mask & POLLOUT_SET) && (out & bit)) {
                                res_out |= bit;
                                retval++;
                                wait->_qproc = NULL;
                        }

                        if ((mask & POLLEX_SET) && (ex & bit)) {
                                res_ex |= bit;
                                retval++;
                                wait->_qproc = NULL;
                        }

                }
        }

        if (res_in)
                *rinp = res_in;
        if (res_out)
                *routp = res_out;
        if (res_ex)
                *rexp = res_ex;

}
```



全文完