一、如何注册一个系统调用

Linux5.1中，如何注册一个系统调用？

在Linux 5.1中，注册系统调用涉及以下步骤：

1. 编写系统调用的C函数：首先，你需要编写一个包含所需功能的C函数作为你的系统调用。确保函数的原型符合系统调用的规范，并注意所需的参数和返回值。
2. 更新系统调用表：在Linux内核中，系统调用表用于映射系统调用的编号与对应的内核函数。你需要更新系统调用表以包含你的新系统调用。这可以通过编辑`arch/<架构>/kernel/syscalls/syscall_64.tbl`文件来完成（`<架构>`是你正在使用的架构，比如x86）。
3. 分配系统调用号：你需要为新系统调用分配一个唯一的系统调用号。可以在`include/linux/syscalls.h`文件中添加一个新的系统调用号定义。
4. 实现系统调用处理函数：在内核源代码中，你需要实现新系统调用的处理函数。这个函数将在内核中被调用，执行你所定义的功能。你可以在`kernel/<架构>/sys.c`文件中添加你的处理函数。
5. 声明系统调用原型：在`include/linux/syscalls.h`文件中，你需要声明你的新系统调用的原型。这将使用户空间程序能够调用该系统调用。
6. 编译和安装内核：完成以上步骤后，你需要重新编译和安装内核，以使新的系统调用生效。这将包括构建内核、安装新内核和引导到新内核。

需要注意的是，修改内核是一项敏感的任务，需要小心操作。在进行这些更改之前，建议你备份你的系统，并在非生产环境中进行测试。

此外，请注意以上步骤仅适用于Linux内核版本为5.1。在其他内核版本中，步骤可能会有所不同。建议参考特定内核版本的文档和源代码来了解正确的步骤。

具体实例

1、更新系统调用表

arch/x86/entry/syscalls/syscall_64.tbl注册系统调用号428 与 系统调用myTest的对应

```
lld@lld-virtual-machine: ~/linux-5.1/arch/x86/entry/syscalls
312	common	kcmp			__x64_sys_kcmp
313	common	finit_module		__x64_sys_finit_module
314	common	sched_setattr		__x64_sys_sched_setattr
315	common	sched_getattr		__x64_sys_sched_getattr
316	common	renameat2		__x64_sys_renameat2
317	common	seccomp			__x64_sys_seccomp
318	common	getrandom		__x64_sys_getrandom
319	common	memfd_create		__x64_sys_memfd_create
320	common	kexec_file_load		__x64_sys_kexec_file_load
321	common	bpf			__x64_sys_bpf
322	64	execveat		__x64_sys_execveat/ptregs
323	common	userfaultfd		__x64_sys_userfaultfd
324	common	membarrier		__x64_sys_membarrier
325	common	mlock2			__x64_sys_mlock2
326	common	copy_file_range		__x64_sys_copy_file_range
327	64	preadv2			__x64_sys_preadv2
328	64	pwritev2		__x64_sys_pwritev2
329	common	pkey_mprotect		__x64_sys_pkey_mprotect
330	common	pkey_alloc		__x64_sys_pkey_alloc
331	common	pkey_free		__x64_sys_pkey_free
332	common	statx			__x64_sys_statx
333	common	io_pgetevents		__x64_sys_io_pgetevents
334	common	rseq			__x64_sys_rseq
# don't use numbers 387 through 423, add new calls after the last
# 'common' entry
424	common	pidfd_send_signal	__x64_sys_pidfd_send_signal
425	common	io_uring_setup		__x64_sys_io_uring_setup
426	common	io_uring_enter		__x64_sys_io_uring_enter
427	common	io_uring_register	__x64_sys_io_uring_register
428	common	myTest			sys_myTest
#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation. The __x32_compat_sys stubs are created
# on-the-fly for compat_sys_*() compatibility system calls if X86_X32
# is defined.
#
512	x32	rt_sigaction		__x32_compat_sys_rt_sigaction
513	x32	rt_sigreturn		sys32_x32_rt_sigreturn
514	x32	ioctl			__x32_compat_sys_ioctl
515	x32	readv			__x32_compat_sys_readv
516	x32	writev			__x32_compat_sys_writev
517	x32	recvfrom		__x32_compat_sys_recvfrom
518	x32	sendmsg			__x32_compat_sys_sendmsg
```

2、增加一个系统调用myTest，函数头即可

include/linux/syscalls.h

```
extern int __close_fd(struct files_struct *files, unsigned int fd);

/*
 * In contrast to sys_close(), this stub does not check whether the sysc
 * should or should not be restarted, but returns the raw error codes fr
 * __close_fd().
 */
static inline int ksys_close(unsigned int fd)
{
        return __close_fd(current->files, fd);
}

extern long do_sys_open(int dfd, const char __user *filename, int flags,
                        umode_t mode);

static inline long ksys_open(const char __user *filename, int flags,
                             umode_t mode)
{
        if (force_o_largefile())
                flags |= O_LARGEFILE;
        return do_sys_open(AT_FDCWD, filename, flags, mode);
}

extern long do_sys_truncate(const char __user *pathname, loff_t length);

static inline long ksys_truncate(const char __user *pathname, loff_t len
{
        return do_sys_truncate(pathname, length);
}

static inline unsigned int ksys_personality(unsigned int personality)
{
        unsigned int old = current->personality;

        if (personality != 0xffffffff)
                set_personality(personality);

        return old;
}

#endif

asmlinkage long sys_myTest(unsigned long number);
```

3、函数实体，使用宏SYSCALL_DEFINEx(...)或者直接写函数

kernel/sys.c

```
                        s.mem_unit <<= 1;
                        bitcount++;
                }

                s.totalram >>= bitcount;
                s.freeram >>= bitcount;
                s.sharedram >>= bitcount;
                s.bufferram >>= bitcount;
                s.totalswap >>= bitcount;
                s.freeswap >>= bitcount;
                s.totalhigh >>= bitcount;
                s.freehigh >>= bitcount;
        }

        if (!access_ok(info, sizeof(struct compat_sysinfo)) ||
            __put_user(s.uptime, &info->uptime) ||
            __put_user(s.loads[0], &info->loads[0]) ||
            __put_user(s.loads[1], &info->loads[1]) ||
            __put_user(s.loads[2], &info->loads[2]) ||
            __put_user(s.totalram, &info->totalram) ||
            __put_user(s.freeram, &info->freeram) ||
            __put_user(s.sharedram, &info->sharedram) ||
            __put_user(s.bufferram, &info->bufferram) ||
            __put_user(s.totalswap, &info->totalswap) ||
            __put_user(s.freeswap, &info->freeswap) ||
            __put_user(s.procs, &info->procs) ||
            __put_user(s.totalhigh, &info->totalhigh) ||
            __put_user(s.freehigh, &info->freehigh) ||
            __put_user(s.mem_unit, &info->mem_unit))
                return -EFAULT;

        return 0;
}
#endif /* CONFIG_COMPAT */

asmlinkage long sys_myTest(unsigned long number){
        printk("The pid of this process is %d\n", current->pid);
        printk("The argument number is %d\n", number);
        return 0;
}
```

需要安装的依赖

```
sudo apt-get install git fakeroot build-essential ncurses-dev xz-utils libssl-dev bc flex libelf-dev bison
```

内核编译报错问题

```
lld18@lld18-virtual-machine:~/linux-5.1$ sudo make -j8
arning: Cannot use CONFIG_STACK_VALIDATION=y, please install libelf-dev, libelf-devel or elfutils-libelf-devel
  CALL    scripts/atomic/check-atomics.sh
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
nake[1]: *** 没有规则可制作目标“debian/canonical-certs.pem”, 由“certs/x509_certificate_list” 需求。 停止。
nake[1]: *** 正在等待未完成的任务....
Makefile:1051: recipe for target 'certs' failed
nake: *** [certs] Error 2
nake: *** 正在等待未完成的任务....
  CC      kernel/sys.o
  AR      kernel/built-in.a
```

将.config中

```
CONFIG_MODULE_SIG_KEY="certs/signing_key.pem"
CONFIG_SYSTEM_TRUSTED_KEYRING=y
#CONFIG_SYSTEM_TRUSTED_KEYS="debian/canonical-certs.pem"
CONFIG_SYSTEM_TRUSTED_KEYS=""
CONFIG_SYSTEM_EXTRA_CERTIFICATE=y
CONFIG_SYSTEM_EXTRA_CERTIFICATE_SIZE=4096
CONFIG_SECONDARY_TRUSTED_KEYRING=y
CONFIG_SYSTEM_BLACKLIST_KEYRING=y
CONFIG_SYSTEM_BLACKLIST_HASH_LIST=""
CONFIG_BINARY_PRINTF=y
```

设置为空字符串

二、使用内核socket编程

1、一些数据结构

Sock

include/net/sock.h

```
struct sock {
        /*
         * Now struct inet_timewait_sock also uses sock_common, so please just
         * don't add nothing before this first member (__sk_common) --acme
         */
        struct sock_common      __sk_common;
#define sk_node                 __sk_common.skc_node
#define sk_nulls_node           __sk_common.skc_nulls_node
#define sk_refcnt               __sk_common.skc_refcnt
#define sk_tx_queue_mapping     __sk_common.skc_tx_queue_mapping
#ifdef CONFIG_XPS
#define sk_rx_queue_mapping     __sk_common.skc_rx_queue_mapping
#endif

#define sk_dontcopy_begin       __sk_common.skc_dontcopy_begin
#define sk_dontcopy_end         __sk_common.skc_dontcopy_end
#define sk_hash                 __sk_common.skc_hash
#define sk_portpair             __sk_common.skc_portpair
#define sk_num                  __sk_common.skc_num
#define sk_dport                __sk_common.skc_dport
#define sk_addrpair             __sk_common.skc_addrpair
#define sk_daddr                __sk_common.skc_daddr
#define sk_rcv_saddr            __sk_common.skc_rcv_saddr
#define sk_family               __sk_common.skc_family
#define sk_state                __sk_common.skc_state
#define sk_reuse                __sk_common.skc_reuse
#define sk_reuseport            __sk_common.skc_reuseport
#define sk_ipv6only             __sk_common.skc_ipv6only
#define sk_net_refcnt           __sk_common.skc_net_refcnt
#define sk_bound_dev_if         __sk_common.skc_bound_dev_if
#define sk_bind_node            __sk_common.skc_bind_node
#define sk_prot                 __sk_common.skc_prot
#define sk_net                  __sk_common.skc_net
#define sk_v6_daddr             __sk_common.skc_v6_daddr
#define sk_v6_rcv_saddr __sk_common.skc_v6_rcv_saddr
```

Socket

include/linux/net.h

`socket` 结构体是 Linux 内核中用于表示套接字的数据结构，包含了套接字的各种属性和状态信息。

linux5.1:

```
/**
 *  struct socket - general BSD socket
 *  @state: socket state (%SS_CONNECTED, etc)
 *  @type: socket type (%SOCK_STREAM, etc)
 *  @flags: socket flags (%SOCK_NOSPACE, etc)
 *  @ops: protocol specific socket operations
 *  @file: File back pointer for gc
 *  @sk: internal networking protocol agnostic socket representation
 *  @wq: wait queue for several uses
 */
struct socket {
        socket_state            state;

        short                   type;

        unsigned long           flags;

        struct socket_wq        *wq;

        struct file             *file;
        struct sock             *sk;
        const struct proto_ops  *ops;
};
```

linux5.4:

```
/**
 *  struct socket - general BSD socket
 *  @state: socket state (%SS_CONNECTED, etc)
 *  @type: socket type (%SOCK_STREAM, etc)
 *  @flags: socket flags (%SOCK_NOSPACE, etc)
 *  @ops: protocol specific socket operations
 *  @file: File back pointer for gc
 *  @sk: internal networking protocol agnostic socket representation
 *  @wq: wait queue for several uses
 */
struct socket {
        socket_state            state;

        short                   type;

        unsigned long           flags;

        struct file             *file;
        struct sock             *sk;
        const struct proto_ops  *ops;

        struct socket_wq        wq;
};
```

在Linux 5.1内核中，`proto_ops`结构体用于定义网络协议族中的操作函数（如套接字的操作函数）。在内核源代码中，`proto_ops`结构体的定义通常可以在特定协议族的相关文件中找到。

具体而言，`proto_ops`结构体的定义可能会出现在以下类型的文件中：

1. `net/ipv4/af_inet.c`：这个文件是IPv4协议族（AF_INET）的实现文件。你可以在其中找到`struct proto_ops`结构体的定义，它是IPv4协议族的操作函数表。
2. `net/ipv6/af_inet6.c`：这个文件是IPv6协议族（AF_INET6）的实现文件。类似于IPv4协议族，你可以在其中找到`struct proto_ops`结构体的定义，它是IPv6协议族的操作函数表。
3. 其他特定协议的实现文件：如果你正在寻找某个特定协议族（如RAW、UDP、TCP等）的`proto_ops`结构体定义，你可能需要查找该协议族的实现文件。这些文件通常位于`net/ipv4/`或`net/ipv6/`目录下，具体文件名可能与协议名相关。

需要注意的是，`proto_ops`结构体的定义可能会因不同的内核版本和配置而有所变化。因此，确保查看与你的内核版本相匹配的代码以获取准确的定义和位置。

sockaddr_in

include/uapi/linux/in.h

`sockaddr_in` 是用于IPv4套接字地址的结构体，其中包含了IP地址和端口号的信息。它是在网络编程中常用的结构体之一，用于表示IPv4地址和端口的组合。

```
/* Structure describing an Internet (IP) socket address. */
#if  __UAPI_DEF_SOCKADDR_IN
#define __SOCK_SIZE__   16              /* sizeof(struct sockaddr)      */
struct sockaddr_in {
  __kernel_sa_family_t  sin_family;     /* Address family               */ /* AF_INET */
  __be16                sin_port;       /* Port number                  */
  struct in_addr        sin_addr;       /* Internet address             */ /* IPv4 address */

  /* Pad to size of `struct sockaddr'. */
  unsigned char         __pad[__SOCK_SIZE__ - sizeof(short int) -
                        sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

sockaddr_in6

include/uapi/linux/in6.h

IPv6对应的协议地址

```
#if __UAPI_DEF_SOCKADDR_IN6
struct sockaddr_in6 {
        unsigned short int      sin6_family;    /* AF_INET6 */
        __be16                  sin6_port;      /* Transport layer port # */
        __be32                  sin6_flowinfo;  /* IPv6 flow information */
        struct in6_addr         sin6_addr;      /* IPv6 address */
        __u32                   sin6_scope_id;  /* scope id (new in RFC2553) */
};
#endif /* __UAPI_DEF_SOCKADDR_IN6 */
```

2、函数定义

sock_create_kern

net/socket.c

`sock_create_kern()` 是一个用于在内核中创建套接字的函数。它允许内核代码创建并配置套接字，以便在内核空间中使用。

```
/**
 *      sock_create_kern - creates a socket (kernel space)
 *      @net: net namespace
 *      @family: protocol family (AF_INET, ...)
 *      @type: communication type (SOCK_STREAM, ...)
 *      @protocol: protocol (0, ...)
 *      @res: new socket
 *
 *      A wrapper around __sock_create().
 *      Returns 0 or an error. This function internally uses GFP_KERNEL.
 */

int sock_create_kern(struct net *net, int family, int type, int protocol, struct socket **res)
{
        return __sock_create(net, family, type, protocol, res, 1);
}
EXPORT_SYMBOL(sock_create_kern);
```

`socket_create`用于用户空间的套接字的创建

```
int sock_create(int family, int type, int protocol, struct socket **res)
{
        return __sock_create(current->nsproxy->net_ns, family, type, protocol, res, 0);
}
```

hton

`hton` 函数是网络编程中的一个常见函数，用于将主机字节序转换为网络字节序。

`hton` 是一个缩写，代表 "Host to Network"，意思是将主机字节序转换为网络字节序。网络字节序是一种统一的字节序，用于在不同主机之间进行数据通信，以保证数据的正确传输。

```
/*
 * The following macros are to be defined by <asm/byteorder.h>:
 *
 * Conversion of long and short int between network and host format
 *      ntohl(__u32 x)
 *      ntohs(__u16 x)
 *      htonl(__u32 x)
 *      htons(__u16 x)
 * It seems that some programs (which? where? or perhaps a standard? POSIX?)
 * might like the above to be functions, not macros (why?).
 * if that's true, then detect them, and take measures.
 * Anyway, the measure is: define only ___ntohl as a macro instead,
 * and in a separate file, have
 * unsigned long inline ntohl(x){return ___ntohl(x);}
 *
 * The same for constant arguments
 *      __constant_ntohl(__u32 x)
 *      __constant_ntohs(__u16 x)
 *      __constant_htonl(__u32 x)
 *      __constant_htons(__u16 x)
 *
 * Conversion of XX-bit integers (16- 32- or 64-)
 * between native CPU format and little/big endian format
 * 64-bit stuff only defined for proper architectures
 *      cpu_to_[bl]eXX(__uXX x)
 *      [bl]eXX_to_cpu(__uXX x)
 *
```

在 Linux 中，相关的字节序转换函数通常位于 `include/linux/byteorder/big_endian.h` 或 `include/linux/byteorder/little_endian.h` 文件中，具体取决于处理器的字节序。

对于大端序（Big Endian）系统，`hton` 函数的定义和相关的字节序转换宏可以在 `include/linux/byteorder/big_endian.h` 文件中找到。

对于小端序（Little Endian）系统，`hton` 函数的定义和相关的字节序转换宏可以在 `include/linux/byteorder/little_endian.h` 文件中找到。

这些头文件中通常包含了一些用于字节序转换的宏，例如 `htonl()` 用于将 32 位无符号整数从主机字节序转换为网络字节序。

```
`htons` 和 `hton` 都是网络编程中用于字节序转换的函数，但它们用于不同的数据类型。

1. `htons` 函数：
   • `htons` 是一个缩写，代表 "Host to Network Short"。
   • 用于将 16 位短整型（short）数据从主机字节序转换为网络字节序。
   • `htons` 函数在头文件 `<arpa/inet.h>` 中声明。
   • 示例用法：`uint16_t networkShort = htons(hostShort);`
2. `hton` 函数：
   • `hton` 是一个通用的函数前缀，代表 "Host to Network"。
   • `hton` 后面通常跟随数据类型的缩写，如 `l` 表示 `long`，`s` 表示 `short` 等。
   • 用于将相应数据类型从主机字节序转换为网络字节序。
   • `hton` 函数在不同数据类型对应的头文件中声明，如 `<netinet/in.h>`。
   • 示例用法：`uint32_t networkLong = htonl(hostLong);`

总结来说，`htons` 用于将 16 位短整型数据从主机字节序转换为网络字节序，而 `hton` 是一个通用的函数前缀，用于不同数据类型的主机字节序到网络字节序的转换。
```

in_aton

`in_aton`函数用于将点分十进制（dotted-decimal）格式的IPv4地址转换为32位的网络字节序整数。它的定义位于`include/linux/inet.h`头文件中

```
extern __be32 in_aton(const char *str);
```

kernel_sendmsg

`kernel_sendmsg`函数用于在内核空间中发送消息。在Linux 5.1版本的内核中，`kernel_sendmsg`函数的定义位于`net/socket.c`文件中

```
int sock_sendmsg(struct socket *sock, struct msghdr *msg)
{
	int err = security_socket_sendmsg(sock, msg,
					  msg_data_left(msg));

	return err ?: sock_sendmsg_nosec(sock, msg);
}
EXPORT_SYMBOL(sock_sendmsg);

/**
 *	kernel_sendmsg - send a message through @sock (kernel-space)
 *	@sock: socket
 *	@msg: message header
 *	@vec: kernel vec
 *	@num: vec array length
 *	@size: total message data size
 *
 *	Builds the message data with @vec and sends it through @sock.
 *	Returns the number of bytes sent, or an error code.
 */

int kernel_sendmsg(struct socket *sock, struct msghdr *msg,
		   struct kvec *vec, size_t num, size_t size)
{
	iov_iter_kvec(&msg->msg_iter, WRITE, vec, num, size);
	return sock_sendmsg(sock, msg);
}
EXPORT_SYMBOL(kernel_sendmsg);
```

kernel_recvmsg

`kernel_recvmsg`函数用于在内核空间中接收消息。在Linux 5.1版本的内核中，`kernel_recvmsg`函数的定义位于`net/socket.c`文件中

```c
int sock_recvmsg(struct socket *sock, struct msghdr *msg, int flags)
{
	int err = security_socket_recvmsg(sock, msg, msg_data_left(msg), flags);

	return err ?: sock_recvmsg_nosec(sock, msg, flags);
}
EXPORT_SYMBOL(sock_recvmsg);

/**
 *	kernel_recvmsg - Receive a message from a socket (kernel space)
 *	@sock: The socket to receive the message from
 *	@msg: Received message
 *	@vec: Input s/g array for message data
 *	@num: Size of input s/g array
 *	@size: Number of bytes to read
 *	@flags: Message flags (MSG_DONTWAIT, etc...)
 *
 *	On return the msg structure contains the scatter/gather array passed in the
 *	vec argument. The array is modified so that it consists of the unfilled
 *	portion of the original array.
 *
 *	The returned value is the total number of bytes received, or an error.
 */

int kernel_recvmsg(struct socket *sock, struct msghdr *msg,
		   struct kvec *vec, size_t num, size_t size, int flags)
{
	mm_segment_t oldfs = get_fs();
	int result;

	iov_iter_kvec(&msg->msg_iter, READ, vec, num, size);
	set_fs(KERNEL_DS);
	result = sock_recvmsg(sock, msg, flags);
	set_fs(oldfs);
	return result;
}
EXPORT_SYMBOL(kernel_recvmsg);
```

kernel_bind

net/socket.c

```c
/**
 *	kernel_bind - bind an address to a socket (kernel space)
 *	@sock: socket
 *	@addr: address
 *	@addrlen: length of address
 *
 *	Returns 0 or an error.
 */

int kernel_bind(struct socket *sock, struct sockaddr *addr, int addrlen)
{
	return sock->ops->bind(sock, addr, addrlen);
}
EXPORT_SYMBOL(kernel_bind);
```

kernel_listen

net/socket.c

```c
/**
 *      kernel_listen - move socket to listening state (kernel space)
 *      @sock: socket
 *      @backlog: pending connections queue size
 *
 *      Returns 0 or an error.
 */

int kernel_listen(struct socket *sock, int backlog)
{
        return sock->ops->listen(sock, backlog);
}
EXPORT_SYMBOL(kernel_listen);
```

kernel_accept

net/socket.c

```c
/**
 *      kernel_accept - accept a connection (kernel space)
 *      @sock: listening socket
 *      @newsock: new connected socket
 *      @flags: flags
 *
 *      @flags must be SOCK_CLOEXEC, SOCK_NONBLOCK or 0.
 *      If it fails, @newsock is guaranteed to be %NULL.
 *      Returns 0 or an error.
 */

int kernel_accept(struct socket *sock, struct socket **newsock, int flags)
{
        struct sock *sk = sock->sk;
        int err;

        err = sock_create_lite(sk->sk_family, sk->sk_type, sk->sk_protocol,
                               newsock);
        if (err < 0)
                goto done;

        err = sock->ops->accept(sock, *newsock, flags, true);
        if (err < 0) {
                sock_release(*newsock);
                *newsock = NULL;
                goto done;
        }

        (*newsock)->ops = sock->ops;
        __module_get((*newsock)->ops->owner);

done:
        return err;
}
EXPORT_SYMBOL(kernel_accept);
```

kernel_connect

net/socket.c

```
/**
 *      kernel_connect - connect a socket (kernel space)
 *      @sock: socket
 *      @addr: address
 *      @addrlen: address length
 *      @flags: flags (O_NONBLOCK, ...)
 *
 *      For datagram sockets, @addr is the addres to which datagrams are sent
 *      by default, and the only address from which datagrams are received.
 *      For stream sockets, attempts to connect to @addr.
 *      Returns 0 or an error code.
 */

int kernel_connect(struct socket *sock, struct sockaddr *addr, int addrlen,
                   int flags)
{
        return sock->ops->connect(sock, addr, addrlen, flags);
}
EXPORT_SYMBOL(kernel_connect);
```

kernel_sendpage

net/socket.c

```
/**
 *      kernel_sendpage - send a &page through a socket (kernel space)
 *      @sock: socket
 *      @page: page
 *      @offset: page offset
 *      @size: total size in bytes
 *      @flags: flags (MSG_DONTWAIT, ...)
 *
 *      Returns the total amount sent in bytes or an error.
 */

int kernel_sendpage(struct socket *sock, struct page *page, int offset,
                    size_t size, int flags)
{
        if (sock->ops->sendpage)
                return sock->ops->sendpage(sock, page, offset, size, flags);

        return sock_no_sendpage(sock, page, offset, size, flags);
}
EXPORT_SYMBOL(kernel_sendpage);
```

__get_free_pages

`__get_free_pages`函数用于分配一个物理页面（通常是4KB大小）。在Linux 5.1版本的内核中，`__get_free_pages`函数的定义位于`mm/page_alloc.c`文件中

```c
/*
 * Common helper functions. Never use with __GFP_HIGHMEM because the returned
 * address cannot represent highmem pages. Use alloc_pages and then kmap if
 * you need to access high mem.
 */
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
{
	struct page *page;

	page = alloc_pages(gfp_mask & ~__GFP_HIGHMEM, order);
	if (!page)
		return 0;
	return (unsigned long) page_address(page);
}
EXPORT_SYMBOL(__get_free_pages);
```

__get_user_pages

mm/gup.c

```c
/**
 * __get_user_pages() - pin user pages in memory
 * @tsk:	task_struct of target task
 * @mm:		mm_struct of target mm
 * @start:	starting user address
 * @nr_pages:	number of pages from start to pin
 * @gup_flags:	flags modifying pin behaviour
 * @pages:	array that receives pointers to the pages pinned.
 *		Should be at least nr_pages long. Or NULL, if caller
 *		only intends to ensure the pages are faulted in.
 * @vmas:	array of pointers to vmas corresponding to each page.
 *		Or NULL if the caller does not require them.
 * @nonblocking: whether waiting for disk IO or mmap_sem contention
 *
 * Returns number of pages pinned. This may be fewer than the number
 * requested. If nr_pages is 0 or negative, returns 0. If no pages
 * were pinned, returns -errno. Each page returned must be released
 * with a put_page() call when it is finished with. vmas will only
 * remain valid while mmap_sem is held.
 *
```

```c
static long __get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
		unsigned long start, unsigned long nr_pages,
		unsigned int gup_flags, struct page **pages,
		struct vm_area_struct **vmas, int *nonblocking)
```

每一个返回的页面在用完后都必须用put_page调用来释放

EXPORT_SYMBOL

`EXPORT_SYMBOL`是一个宏定义，用于在Linux内核模块中将一个符号（函数、变量或其他对象）导出为可供其他内核模块或内核代码使用的全局符号。

在Linux内核中，符号（函数、变量等）默认情况下是私有的，即只能在定义该符号的模块内部使用。如果一个内核模块希望将某个符号暴露给其他模块使用，可以使用`EXPORT_SYMBOL`宏将该符号导出。

导出符号后，其他模块或内核代码可以使用`extern`关键字声明并使用该符号。

以下是`EXPORT_SYMBOL`宏的使用示例：

c Copy code

```c
// 在一个内核模块中导出一个函数
EXPORT_SYMBOL(my_function);

// 在一个内核模块中导出一个变量
EXPORT_SYMBOL(my_variable);
```

`EXPORT_SYMBOL`宏通常用于内核模块编程，使得不同的模块可以在需要时共享函数、变量和其他资源。通过导出和导入符号，模块之间可以相互通信、共享功能，并提高内核的可扩展性和灵活性。

需要注意的是，正确使用`EXPORT_SYMBOL`需要遵循一些规范和最佳实践，包括确保导出符号的稳定性和适当的命名空间管理。在内核模块开发中，建议参考内核源代码和文档以了解更多关于`EXPORT_SYMBOL`的详细 Regenerate response

call_usermodehelper

kernel/umh.c

调用`call_usermodehelper_setup` + `call_usermodehelper_exec`，都在umh.c里面

`call_usermodehelper_exec()` 函数用于在内核空间调用用户空间程序，并传递参数给该程序。它接受一个指向 `subprocess_info` 结构体的指针作为参数，该结构体包含了要执行的程序路径、参数列表等信息。

```c
/**
 * call_usermodehelper() - prepare and start a usermode application
 * @path: path to usermode executable
 * @argv: arg vector for process
 * @envp: environment for process
 * @wait: wait for the application to finish and return status.
 *        when UMH_NO_WAIT don't wait at all, but you get no useful error back
 *        when the program couldn't be exec'ed. This makes it safe to call
 *        from interrupt context.
 *
 * This function is the equivalent to use call_usermodehelper_setup() and
 * call_usermodehelper_exec().
 */
int call_usermodehelper(const char *path, char **argv, char **envp, int wait)
{
        struct subprocess_info *info;
        gfp_t gfp_mask = (wait == UMH_NO_WAIT) ? GFP_ATOMIC : GFP_KERNEL;

        info = call_usermodehelper_setup(path, argv, envp, gfp_mask,
                                         NULL, NULL, NULL);
        if (info == NULL)
                return -ENOMEM;

        return call_usermodehelper_exec(info, wait);
}
EXPORT_SYMBOL(call_usermodehelper);
```