`sys_socketcall`: Network systems calls on Linux

Daniel Noé

April 9, 2008

The method used by Linux for system calls is explored in detail in *Understanding the Linux Kernel*. However, the book does not adequately describe the idiosyncrasies of networking system calls. When the BSD Socket interface was added to the kernel seventeen system calls were added at once. The Linux programmers decided to use an additional level of indirection in order to conserve system call numbers. This unusual technique remains in the kernel in many architectures, including i386. Some newer architectures such as x86_64 avoid the extra layer of indirection and use the same technique as other system calls.

In this short article we will explore the `sys_socketcall` system call dispatch method. To match *Understanding the Linux Kernel* all examples here are for Linux Kernel version 2.6.11 on the i386 architecture. We will assume the reader is familiar with the standard system call mechanisms on i386. As you may recall, the kernel contains a large jump table in `arch/i386/kernel/entry.S` which maps system call numbers to functions. By convention these functions begin with `sys_`. But there are no `sys_socket` or `sys_bind` or any other Berkeley Sockets API system calls listed directly in this table.

When a user space library wishes to make a normal system call, it places the corresponding system call number (0-228 currently) in the `eax` register, then performs whatever action is required to switch to the kernel. On older x86 systems this was done with the `int 0x80` instruction. Newer systems provide the `sysenter` instruction which is more efficient. Libraries generally use the vsyscall mechanism, which allows the kernel to automatically select the best entry mechanism. If the system call requires arguments, these are passed in additional registers starting with `ebx`.

Socket calls use work differently. Instead of one system call per userspace call, for socket calls everything is wrapped through the `sys_socketcall` function. This is defined in the `entry.S` jump table as system call 102. The

```
/* Argument list sizes for sys_socketcall */
#define AL(x) ((x) * sizeof(unsigned long))
static unsigned char nargs[18]={AL(0),AL(3),AL(3),AL(3),AL(2),AL(3),
                                AL(3),AL(3),AL(4),AL(4),AL(4),AL(6),
                                AL(6),AL(2),AL(5),AL(5),AL(3),AL(3)};
#undef AL
```

Figure 1: Argument list sizes for sys_socketcall

actual desired user space socket call number (in the range 1-17) is passed in `ebx`. The socket call numbers are shown in Table 1 and are found in `linux/net.h`. Any arguments given to the user space socket call are passed using an "args" array of type `unsigned long`. The address of the first element of this array is placed in `ecx` and a system call to `sys_socketcall` is made via the usual means.

The `sys_socketcall` function is defined in `net/socket.c` and takes two parameters: an integer call number and the `unsigned long` pointer "args". Via the normal system call convention, this function will be called with the value from `ebx` in the first argument and `ecx` in the second. Four local stack variables are declared in `sys_socketcall`. The first is an `unsigned long` array "a" of size 6. Two more `unsigned long` variables "a0" and "a1" are also declared. Finally, an integer "err" is declared to hold return values (if the return value is less than 0 it indicates an error and it is negated and placed in `errno` by user space code).

The first thing `sys_socketcall` does is to check if the call number passed in `ebx` is reasonable. If it is less than 1 or more than `SYS_RECVMSG` (17, the highest call code numerically) then `-EINVAL` is returned ("Invalid argument"). Next, the arguments array is copied from user space to kernel space using the `copy_from_user()` function. The arguments to this function are the kernel stack array `a`, the user space array `args` and the number of bytes to copy.

In order to determine the number of bytes to pass as the third argument of `copy_from_user()`, the code consults the `static unsigned char` array called `nargs[]`, shown in Figure 1. This array is constructed with the help of a macro `AL(x)` which returns `x` times the size of an unsigned long. The `nargs` array contains one entry for each socket call code. It is initialized using the `AL` macro with the number of `unsigned long` arguments of each socket call, respectively. This way the `nargs` array contains the number of bytes in the userspace arguments array for each socket call. The `nargs` array

Userspace function	Call code (from `linux/net.h`)	Kernel function
`socket()`	`SYS_SOCKET`	`sys_socket()`
`bind()`	`SYS_BIND`	`sys_bind()`
`connect()`	`SYS_CONNECT`	`sys_connect()`
`listen()`	`SYS_LISTEN`	`sys_listen()`
`accept()`	`SYS_ACCEPT`	`sys_accept()`
`getsockname()`	`SYS_GETSOCKNAME`	`sys_getsockname()`
`getpeername()`	`SYS_GETPEERNAME`	`sys_getpeername()`
`socketpair()`	`SYS_SOCKETPAIR`	`sys_socketpair()`
`send()`	`SYS_SEND`	`sys_send()`
`sendto()`	`SYS_SENDTO`	`sys_sendto()`
`recv()`	`SYS_RECV`	`sys_recv()`
`recvfrom()`	`SYS_RECVFROM`	`sys_recvfrom()`
`shutdown()`	`SYS_SHUTDOWN`	`sys_shutdown()`
`setsockopt()`	`SYS_SETSOCKOPT`	`sys_setsockopt()`
`getsockopt()`	`SYS_GETSOCKOPT`	`sys_getsockopt()`
`sendmsg()`	`SYS_SENDMSG`	`sys_sendmsg()`
`recvmsg()`	`SYS_RECVMSG`	`sys_recvmsg()`

Table 1: Mappings from `switch(call)` in `net/socket.c`: `sys_socketcall()`

is subscripted with the socket call number in order to pass the number of bytes to `copy_from_user()`. See Figure 1 for a listing. Notice that the `AL` macro is undefined after it is used to avoid polluting the namespace.

The next operation is to copy the `a[0] and a[1]` values into the `a0` and `a1` variables. I'm not exactly sure what the point of this is, since it simply provides a different way to refer to these argument values. It may be there in order to provide a hint to the compiler that the first two arguments should be placed into registers.

Finally, `sys_socketcall` enters a large switch statement. It contains one case for each of the seventeen socket call codes listed in Table 1. There is also a default case, but this should never be reached since the call code was already checked before the copy was performed. Each case of the switch statement calls the appropriate kernel function with the correct number of arguments from the `a` array. The return value of the function is placed in `err`, which is returned at the end of the switch statement.

Note that the `sys_socketcall` abstraction described here is not used on

```
struct sockaddr {
  sa_family_t   sa_family;      /* address family, AF_xxx       */
  char          sa_data[14];    /* 14 bytes of protocol address */
};
```

Figure 2: `struct sockaddr` defined in `linux/socket.h`

all architectures. There is a `#ifdef __ARCH_WANT_SYS_SOCKETCALL` which disables the definition of `sys_socketcall` and the `nargs` static array if the architecture elects to directly call the functions from Table 1 instead of using `sys_socketcall`. Note that certain architectures such as x86_64 define this macro even though they don't seem to be using the `sys_socketcall` mechanism. I think this is because the `sys_socketcall` code is used for ia32 compatibility routines in the kernel (for example, 32 bit binaries running on a 64 bit install).

One major data structure is commonly seen in the socket system calls. It is `struct sockaddr`, seen in Figure 2. What seems to be a very simple structure is in fact a placeholder for something much more complex (those used to fancy abstract programming interfaces may even say sinister!). In order to allow the socket system calls to operate on a diverse array of different protocols, the first field of the `struct sockaddr` is a `short int` which represents the address family.

These address families are defined as macros in `linux/socket.h`. Some of them are well known and common (`AF_INET`), some haven't come of age yet (`AF_INET6`), and the sun is setting on some (`AF_SNA`, maintained by the Linux SNA Project, which `socket.h` describes as "nutters!"). By reading the first two bytes of the `struct sockaddr` the address family and thus address format can be determined. This means the same socket interface can be used for many different network protocols.

The `sa_data` field of the structure is marked as 14 bytes, but it doesn't have to be. For example, an IPv6 address is 16 bytes alone, even without additional information required for the socket (such as the port). Typically an address family specific structure such as `sockaddr_in` is allocated and populated. This is then cast to a `struct sockaddr` and passed to the socket system call. The receiving end peeks at the first two bytes of the structure and then knows that the `struct sockaddr` is really a particular protocol specific structure. It can then perform a cast to access the `struct sockaddr` as the protocol specific structure.

This technique seems rather ugly at first glance when compared to mod-

ern object-oriented techniques but there is a certain simplicity to it. It also keeps data compact, which improves cache locality. The only issue is that the casting must be performed manually, and this means it is easy to shoot yourself in the foot. For example, the `struct sockaddr` itself does not have enough space to fit an IPv6 address (but it does have enough for IPv4). It is necessary to allocate space for the larger structure required by IPv6 if you wish to use the same allocation for both IPv6 and IPv4 addresses. The consequences of trying to put an IPv6 address into a 14 byte structure will likely be severe and difficult to debug.

The `sys_socketcall` interface provides an interesting look into the historical development of the Linux kernel. It is hardly necessary to do this kind of indirection out of an attempt to conserve system calls. There have certainly been numerous system calls added since `sys_socketcall` yet the kernel is not close to running out. Yet, the `sys_socketcall` interface remains. It provides negligible overhead and changing the interface would require changes to system libraries. This small example of kernel engineering will remain in the kernel for a long time.