Kernel command using Linux system calls

Explore the SCI and add your own calls

Level: Intermediate

M. Tim Jones (mtj@mtjones.com), Consultant Engineer, Emulex

21 Mar 2007

Linux® system calls -- we use them every day. But do you know how a system call is performed from user-space to the kernel? Explore the Linux system call interface (SCI), learn how to add new system calls (and alternatives for doing so), and discover utilities related to the SCI.

A **system call** is an interface between a user-space application and a service that the kernel provides. Because the service is provided in the kernel, a direct call cannot be performed; instead, you must use a process of crossing the user-space/kernel boundary. The way you do this differs based on the particular architecture. For this reason, I'll stick to the most common architecture, i386.

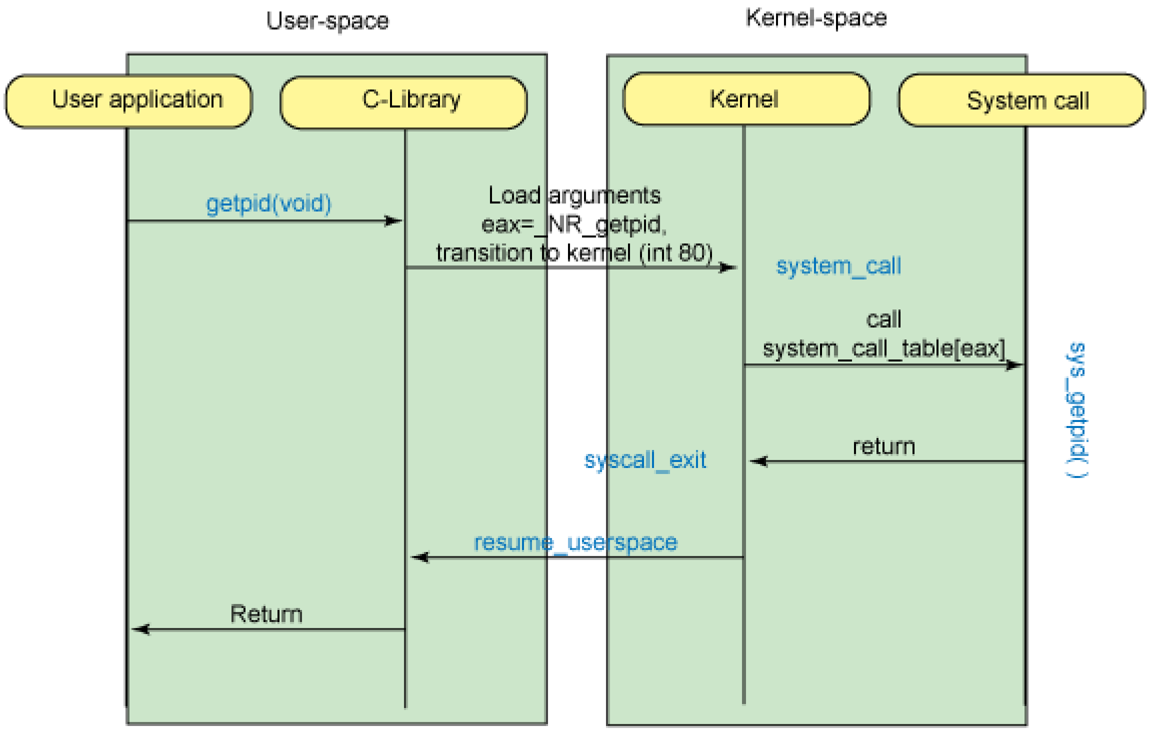
In this article, I explore the Linux SCI, demonstrate adding a system call to the 2.6.20 kernel, and then use this function from user-space. I also investigate some of the functions that you'll find useful for system call development and alternatives to system calls. Finally, I look at some of the ancillary mechanisms related to system calls, such as tracing their usage from a given process.

The SCI

The implementation of system calls in Linux is varied based on the architecture, but it can also differ within a given architecture. For example, older x86 processors used an interrupt mechanism to migrate from user-space to kernel-space, but new IA-32 processors provide instructions that optimize this transition (using `sysenter` and `sysexit` instructions). Because so many options exist and the end-result is so complicated, I'll stick to a surface-level discussion of the interface details. See the Resources at the end of this article for the gory details.

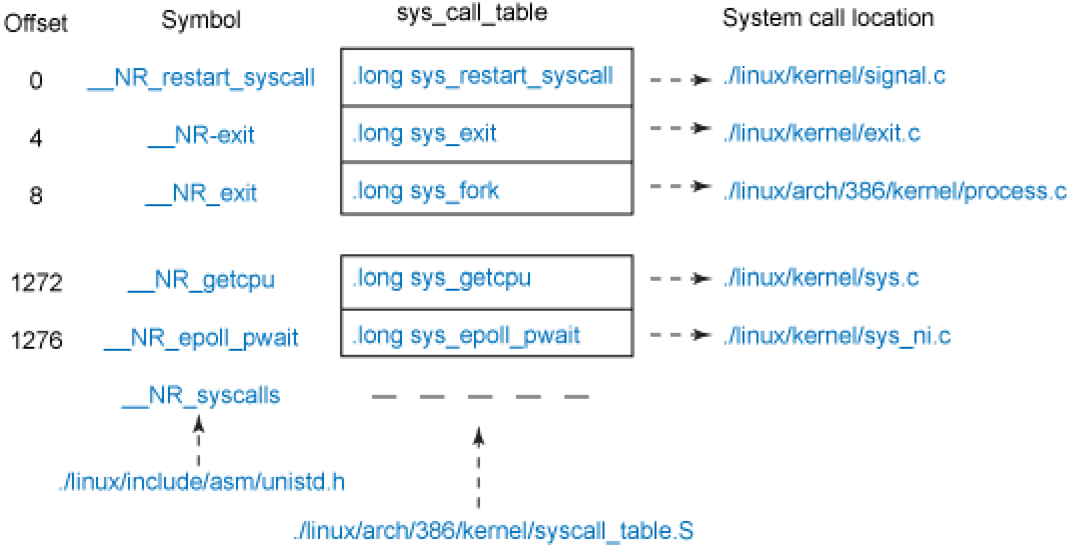
You needn't fully understand the internals of the SCI to amend it, so I explore a simple version of the system call process (see Figure 1). Each system call is multiplexed into the kernel through a single entry point. The eax register is used to identify the particular system call that should be invoked, which is specified in the `C` library (per the call from the user-space application). When the `C` library has loaded the system call index and any arguments, a software interrupt is invoked (interrupt 0x80), which results in execution (through the interrupt handler) of the `system_call` function. This function handles all system calls, as identified by the contents of eax. After a few simple tests, the actual system call is invoked using the `system_call_table` and index contained in eax. Upon return from the system call, `syscall_exit` is eventually reached, and a call to `resume_userspace` transitions back to user-space. Execution resumes in the `C` library, which then returns to the user application.

Figure 1. The simplified flow of a system call using the interrupt method



At the core of the SCI is the system call demultiplexing table. This table, shown in Figure 2, uses the index provided in eax to identify which system call to invoke from the table (`sys_call_table`). A sample of the contents of this table and the locations of these entities is also shown. (For more about demultiplexing, see the sidebar, "System call demultiplexing.")

Figure 2. The system call table and various linkages



Offset	Symbol	sys_call_table	System call location
0	__NR_restart_syscall	.long sys_restart_syscall	./linux/kernel/signal.c
4	__NR-exit	.long sys_exit	./linux/kernel/exit.c
8	__NR_exit	.long sys_fork	./linux/arch/386/kernel/process.c
1272	__NR_getcpu	.long sys_getcpu	./linux/kernel/sys.c
1276	__NR_epoll_pwait	.long sys_epoll_pwait	./linux/kernel/sys_ni.c
	__NR_syscalls		

./linux/include/asm/unistd.h
./linux/arch/386/kernel/syscall_table.S

Adding a Linux system call

Adding a new system call is mostly procedural, although you should look out for a few things. This section walks through the construction of a few system calls to demonstrate their

System call demultiplexing

Some system calls are further demultiplexed by the kernel. For implementation and use by a user-space application.

You perform three basic steps to add a new system call to the kernel:

1. Add the new function.
2. Update the header files.
3. Update the system call table for the new function.

Note: This process ignores user-space needs, which I address later.

example, the Berkeley Software Distribution (BSD) socket calls (`socket`, `bind`, `connect`, and so on) are associated with a single system call index (`__NR_socketcall`) but are demultiplexed in the kernel to the appropriate call through another argument. See ./linux/net/socket.c function `sys_socketcall`.

Most often, you create a new file for your functions. However, for the sake of simplicity, I add my new functions to an existing source file. The first two functions, shown in Listing 1, are simple examples of a system call. Listing 2 provides a slightly more complicated function that uses pointer arguments.

Listing 1. Simple kernel functions for the system call example

```
asmlinkage long sys_getjiffies( void )
{
  return (long)get_jiffies_64();
}

asmlinkage long sys_diffjiffies( long ujiffies )
{
  return (long)get_jiffies_64() - ujiffies;
}
```

In Listing 1, two functions are provided for jiffies monitoring. (For more information about jiffies, see the sidebar, "Kernel jiffies.") The first function returns the current jiffies, while the second returns the difference of the current and the value that the caller passes in. Note the use of the `asmlinkage` modifier. This macro (defined in linux/include/asm-i386/linkage.h) tells the compiler to pass all function arguments on the stack.

Listing 2. Final kernel function for the system call example

```
asmlinkage long sys_pdiffjiffies( long ujiffies,
                                  long __user *presult )
{
  long cur_jiffies = (long)get_jiffies_64();
  long result;
  int  err = 0;

  if (presult) {

    result = cur_jiffies - ujiffies;
    err = put_user( result, presult );

  }

  return err ? -EFAULT : 0;
}
```

Listing 2 provides the third function. This function takes two arguments: a `long` and a pointer to a `long` that's defined as `__user`. The `__user` macro simply tells the compiler (through `noderef`) that the pointer should not be dereferenced (as it's not meaningful in the current address space). This function calculates the difference between two jiffies values, and then provides the result to the user through a user-space pointer. The `put_user` function places the result value into user-space at the location that `presult` specifies. If an error occurs during this operation, it will be returned, and you'll likewise notify the user-space caller.

Kernel jiffies

The Linux kernel maintains a global variable called `jiffies`, which represents the number of timer ticks since the machine started. This variable is initialized to zero and increments each timer interrupt. You can read jiffies with the `get_jiffies_64` function, and then convert this value to

For step 2, I update the header files to make room for the new functions in the system call table. For this, I update the header file linux/include/asm/unistd.h with the new system call numbers. The updates are shown in bold in Listing 3.

milliseconds (msec) with `jiffies_to_msecs` or to microseconds (usec) with `jiffies_to_usecs`. The jiffies' global and associated functions are provided in ./linux/include/linux/jiffies.h.

Listing 3. Updates to unistd.h to make room for the new system calls

```
#define __NR_getcpu          318
#define __NR_epoll_pwait     319
#define __NR_getjiffies                320
#define __NR_diffjiffies     321
#define __NR_pdiffjiffies    322

#define NR_syscalls 323
```

Now I have my kernel system calls and numbers to represent them. All I need to do now is draw an equivalence among these numbers (table indexes) and the functions themselves. This is step 3, updating the system call table. As shown in Listing 4, I update the file linux/arch/i386/kernel/syscall_table.S for the new functions that will populate the particular indexes shown in Listing 3.

Listing 4. Update the system call table with the new Functions

```
.long sys_getcpu
.long sys_epoll_pwait
.long sys_getjiffies                   /* 320 */
.long sys_diffjiffies
.long sys_pdiffjiffies
```

Note: The size of this table is defined by the symbolic constant `NR_syscalls`.

At this point, the kernel is updated. I must recompile the kernel and make the new image available for booting before testing the user-space application.

Reading and writing user memory

The Linux kernel provides several functions that you can use to move system call arguments to and from user-space. Options include simple functions for basic types (such as `get_user` or `put_user`). For moving blocks of data such as structures or arrays, you can use another set of functions: `copy_from_user` and `copy_to_user`. Moving null-terminated strings have their own calls: `strncpy_from_user` and `strlen_from_user`. You can also test whether a user-space pointer is valid through a call to `access_ok`. These functions are defined in linux/include/asm/uaccess.h.

You use the `access_ok` macro to validate a user-space pointer for a given operation. This function takes the type of access (`VERIFY_READ` or `VERIFY_WRITE`), the pointer to the user-space memory block, and the size of the block (in bytes). The function returns zero on success:

```
int access_ok( type, address, size );
```

Moving simple types between the kernel and user-space (such as ints or longs) is accomplished easily with `get_user` and `put_user`. These macros each take a value and a pointer to a variable. The `get_user` function moves the value that the user-space address specifies (`ptr`) into the kernel variable specified (`var`). The `put_user` function moves the value that the kernel variable (`var`) specifies into the user-space address (`ptr`). The functions return zero on success:

```
int get_user( var, ptr );
int put_user( var, ptr );
```

To move larger objects, such as structures or arrays, you can use the copy_from_user and copy_to_user functions. These functions move an entire block of data between user-space and the kernel. The copy_from_user function moves a block of data from user-space into kernel-space, and copy_to_user moves a block of data from the kernel into user-space:

```
unsigned long copy_from_user( void *to, const void __user *from, unsigned long n );
unsigned long copy_to_user( void *to, const void __user *from, unsigned long n );
```

Finally, you can copy a NULL-terminated string from user-space to the kernel by using the strncpy_from_user function. Before calling this function, you can get the size of the user-space string with a call to the strlen_user macro:

```
long strncpy_from_user( char *dst, const char __user *src, long count );
strlen_user( str );
```

These functions provide the basics for memory movement between the kernel and user-space. Some additional functions exist (such as those that reduce the amount of checking performed). You can find these functions in uaccess.h.

Using the system call

Now that kernel is updated with a few new system calls, let's look at what's necessary to use them from a user-space application. There are two ways that you can use new kernel system calls. The first is a convenience method (not something that you'd probably want to do in production code), and the second is the traditional method that requires a bit more work.

With the first method, you call your new functions as identified by their index through the syscall function. With the syscall function, you can call a system call by specifying its call index and a set of arguments. For example, the short application shown in Listing 5 calls your sys_getjiffies using its index.

Listing 5. Using syscall to invoke a system call

```
#include <linux/unistd.h>
#include <sys/syscall.h>

#define __NR_getjiffies               320

int main()
{
  long jiffies;

  jiffies = syscall( __NR_getjiffies );

  printf( "Current jiffies is %lx\n", jiffies );

  return 0;
}
```

As you can see, the syscall function includes as its first argument the index of the system call table to use. Had there been any arguments to pass, these would be provided after the call index. Most system calls include a SYS_ symbolic constant to specify their mapping to the __NR_ indexes. For example, you invoke the index __NR_getpid with syscall as:

```
syscall( SYS_getpid )
```

The `syscall` function is architecture specific but uses a mechanism to transfer control to the kernel. The argument is based on a mapping of `__NR` indexes to `SYS_` symbols provided by /usr/include/bits/syscall.h (defined when the libc is built). Never reference this file directly; instead use /usr/include/sys/syscall.h.

The traditional method requires that you create function calls that match those in the kernel in terms of system call index (so that you're calling the right kernel service) and that the arguments match. Linux provides a set of macros to provide this capability. The `_syscallN` macros are defined in /usr/include/linux/unistd.h and have the following format:

```
_syscall0( ret-type, func-name )
_syscall1( ret-type, func-name, arg1-type, arg1-name )
_syscall2( ret-type, func-name, arg1-type, arg1-name, arg2-type, arg2-name )
```

The `_syscall` macros are defined up to six arguments deep (although only three are shown here).

Now, here's how you use the `_syscall` macros to make your new system calls visible to the user-space. Listing 6 shows an application that uses each of your system calls as defined by the `_syscall` macros.

User-space and __NR constants

Note that in Listing 6 I've provided the `__NR` symbolic constants. You can find these in /usr/include/asm/unistd.h (for standard system calls).

Listing 6. Using the _syscall macro for user-space application development

```

#include <stdio.h>
#include <linux/unistd.h>
#include <sys/syscall.h>

#define __NR_getjiffies                320
#define __NR_diffjiffies     321
#define __NR_pdiffjiffies    322

_syscall0( long, getjiffies );
_syscall1( long, diffjiffies, long, ujiffies );
_syscall2( long, pdiffjiffies, long, ujiffies, long*, presult );

int main()
{
  long jifs, result;
  int err;

  jifs = getjiffies();

  printf( "difference is %lx\n", diffjiffies(jifs) );

  err = pdiffjiffies( jifs, &result );

  if (!err) {
    printf( "difference is %lx\n", result );
  } else {
    printf( "error\n" );
  }

  return 0;
}
```

Note that the `__NR` indexes are necessary in this application because the `_syscall` macro uses the `func-name` to construct the `__NR` index (`getjiffies` -> `__NR_getjiffies`). But the result is that you can call your kernel functions using their names, just like any other system call.

Alternatives for user/kernel interactions

System calls are an efficient way of requesting services in the kernel. The biggest problem with them is that it's a standardized interface. It would be difficult to have your new system call added to the kernel, so any additions are likely served through other means. If you have no intent of mainlining your system calls into the public Linux kernel, then system calls are a convenient and efficient way to make kernel services available to user-space.

Another way to make your services visible to user-space is through the /proc file system. The /proc file system is a virtual file system for which you can surface a directory and files to the user, and then provide an interface in the kernel to your new services through a file system interface (read, write, and so on).

Tracing system calls with strace

The Linux kernel provides a useful way to trace the system calls that a process invokes (as well as those signals that the process receives). The utility is called `strace` and is executed from the command line, using the application you want to trace as its argument. For example, if you wanted to know which system calls were invoked during the context of the `date` command, type the following command:

```
strace date
```

The result is a rather large dump showing the various system calls that are performed in the context of a `date` command call. You'll see the loading of shared libraries, mapping of memory, and -- at the end of the trace -- the emitting of the date information to standard-out:

```
...
write(1, "Fri Feb  9 23:06:41 MST 2007\n", 29Fri Feb  9 23:06:41 MST 2007) = 29
munmap(0xb747a000, 4096)     = 0
exit_group(0)                          = ?
$
```

This tracing is accomplished in the kernel when the current system call request has a special field set called `syscall_trace`, which causes the function `do_syscall_trace` to be invoked. You can also find the tracing calls as part of the system call request in ./linux/arch/i386/kernel/entry.S (see `syscall_trace_entry`).

Going further

System calls are an efficient way of traversing between user-space and the kernel to request services in the kernel-space. But they are also tightly controlled, and it's much easier simply to add a new /proc file system entry to provide the user/kernel interactions. When speed is important, however, system calls are an ideal way to squeeze the greatest performance out of your application. See Resources to dig even further into the SCI.

Share this...

Digg this story

Post to del.icio.us

Slashdot it!

Resources

Learn

- In "Access the Linux kernel using the /proc filesystem" (developerWorks, March 2006), learn how to develop kernel code that uses the /proc file system for user-space/kernel communication.

- Read "Sysenter Based System Call Mechanism in Linux 2.6" from Manugarg to get a detailed look at the system call gate between the user-space application and the kernel. This paper focuses on the transition mechanisms provided in the 2.6 kernel.
- This paper details the assembly language linkages between the user-space and the kernel.
- The GNU C Library (glibc) is the standard library for GNU C. You'll find the glibc for Linux and also for numerous other operating systems. The GNU C Library follows numerous standards, including the ISO C 99, POSIX, and UNIX98. You can find more information about it at the GNU Project.
- The Linux syscalls man page gives a complete list of system calls available in Linux.
- Wikipedia provides an interesting perspective on system calls, including history and typical implementations.
- While a bit dated, a Kernel application program interface (API) is provided that documents many of the kernel functions available for general (in-kernel) use. This includes the user-space memory-management functions as well as many others.
- In the developerWorks Linux zone, find more resources for Linux developers.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

M. Tim Jones is an embedded software architect and the author of *GNU/Linux Application Programming*, *AI Application Programming*, and *BSD Sockets Programming from a Multilanguage Perspective*. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.