Kernel APIs, Part 1: Invoking user-space applications from the kernel

Implementation and use of the usermode-helper API

M. Tim Jones
Independent author

16 February 2010

The Linux® system call interface permits user-space applications to invoke functionality in the kernel, but what about invoking user-space applications *from* the kernel? Explore the usermode-helper API, and learn how to invoke user-space applications and manipulate their output.

View more content in this series

Invoking specific kernel functions (system calls) is a natural part of application development on GNU/Linux. But what about going in the other direction, kernel space calling user space? It turns out that there are a number of applications for this feature that you likely use every day. For example, when the kernel finds a device for which a module needs to be loaded, how does this process occur? Dynamic module loading occurs from the kernel through the *usermode-helper* process.

Let's begin with an exploration of usermode-helper, its application programming interface (API), and some of the examples of where this feature is used in the kernel. Then, using the API, you'll build a sample application to better understand how it works and its limitations.

The usermode-helper API

The usermode-helper API is a simple API with a well-known set of options. For example, to create a process from user space, you commonly provide the name of the executable, the options for the executable, and a set of environment variables (refer to the man page for `execve`). The same applies for creating a process from the kernel. But because you're starting the process from kernel space, a few additional options are available.

Kernel version

This article explores the usermode-helper API from the 2.6.27 kernel.

Table 1 shows the core set of kernel functions available in the usermode-helper API.

Table 1. Core functions in the usermode-helper API

API function	Description
`call_usermodehelper_setup`	Prepare a handler for a user-land call
`call_usermodehelper_setkeys`	Set the session keys for a helper
`call_usermodehelper_setcleanup`	Set a cleanup function for the helper
`call_usermodehelper_stdinpipe`	Create a `stdin` pipe for a helper
`call_usermodehelper_exec`	Invoke the user-land call

Also in this table are a couple of simplification functions that encapsulate the kernel functions in Table 2 (requiring a single call instead of multiple calls). These simplification functions are useful for most cases, so you're encouraged to use them, if possible.

Table 2. Simplifications of the usermode-helper API

API function	Description
`call_usermodehelper`	Make a user-land call
`call_usermodehelper_pipe`	Make a user-land call with a pipe `stdin`
`call_usermodehelper_keys`	Make a user-land call with session keys

Let's first walk through the core functions, then explore the capabilities that the simplification functions provide. The core API operates using a handler reference called a `subprocess_info` structure. This structure (which can be found in ./kernel/kmod.c) aggregates all of the necessary elements for a given usermode-helper instance. The structure reference is returned from a call to `call_usermodehelper_setup`. The structure (and subsequent calls) is further configured through calls to `call_usermodehelper_setkeys` (for credentials storage), `call_usermodehelper_setcleanup`, and `call_usermodehelper_stdinpipe`. Finally, once configuration is complete, you can invoke the configured user-mode application through a call to `call_usermodehelper_exec`.

Disclaimer

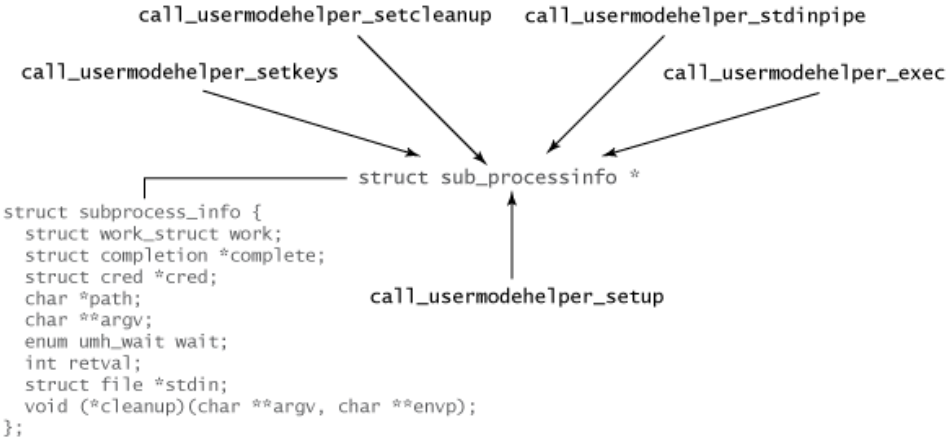
This method provides a necessary function for invoking user-space applications from the kernel. Although there are legitimate uses for this functionality, you should strongly consider whether other implementations are needed. This is one approach, but other approaches are better suited.

The core functions provide you with the greatest amount of control, where the helper functions do more of the work for you in a single call. The pipe-related calls (`call_usermodehelper_stdinpipe` and the helper function `call_usermodehelper_pipe`) create an associated pipe for use by the helper. Specifically, a pipe is created (a file structure in the kernel). The pipe is readable by the user-space application and writable by the kernel side. As of this writing, core dumps are

the only application that can use a pipe with a usermode-helper. In this application (./fs/exec.c `do_coredump()`), the core dump is written through the pipe from kernel space to user space.

The relationship between these functions and the `sub_processinfo` along with the details of the `subprocess_info` structure is shown in Figure 1.

Figure 1. Usermode-helper API relationships



The simplification functions in Table 2 perform the `call_usermodehelper_setup` function and `call_usermodehelper_exec` function internally. The last two calls in Table 2 invoke the `call_usermodehelper_setkeys` and `call_usermodehelper_stdinpipe`, respectively. You can find the source to `call_usermodehelper_pipe` in ./kernel/kmod.c and to `call_usermodehelper` and `call_usermodhelper_keys` in ./include/linux/kmod.h.

Why invoke a user-space application from the kernel?

Let's now look at some of the places in the kernel where the usermode-helper API is put to use. Table 3 doesn't provide an exclusive list of applications but represents a cross-section of interesting uses.

Table 3. Applications of the usermode-helper API in the kernel

Application	Source location
Kernel module loading	./kernel/kmod.c
Power management	./kernel/sys.c
Control groups	./kernel/cgroup.c
Security key generation	./security/keys/request_key.c
Kernel event delivery	./lib/kobject_uevent.c

One of the most straightforward applications of the usermode-helper API is loading kernel modules from kernel space. The function `request_module` encapsulates the functionality of the usermode-helper API and provides a simple interface. In a common usage model, the kernel identifies a device or needed service and makes a call to `request_module` to have the module loaded. Through the usermode-helper API, the module is loaded into the kernel via `modprobe` (the application invoked in user space via `request_module`).

A similar application to module loading is device hot-plugging (to add or remove devices at run time). This feature is implemented with the usermode-helper API, invoking the /sbin/hotplug utility in user space.

An interesting application of the usermode-helper API (via `request_module`) is the textsearch API (./lib/textsearch.c). This application provides a configurable text searching infrastructure in the kernel. This application uses the usermode-helper API through the dynamic loading of search algorithms as loadable modules. In the 2.6.30 kernel release, three algorithms are supported, including Boyer-Moore (./lib/ts_bm.c), a naive finite-state machine approach (./lib/ts_fsm.c), and finally the Knuth-Morris-Pratt algorithm (./lib/ts_kmp.c).

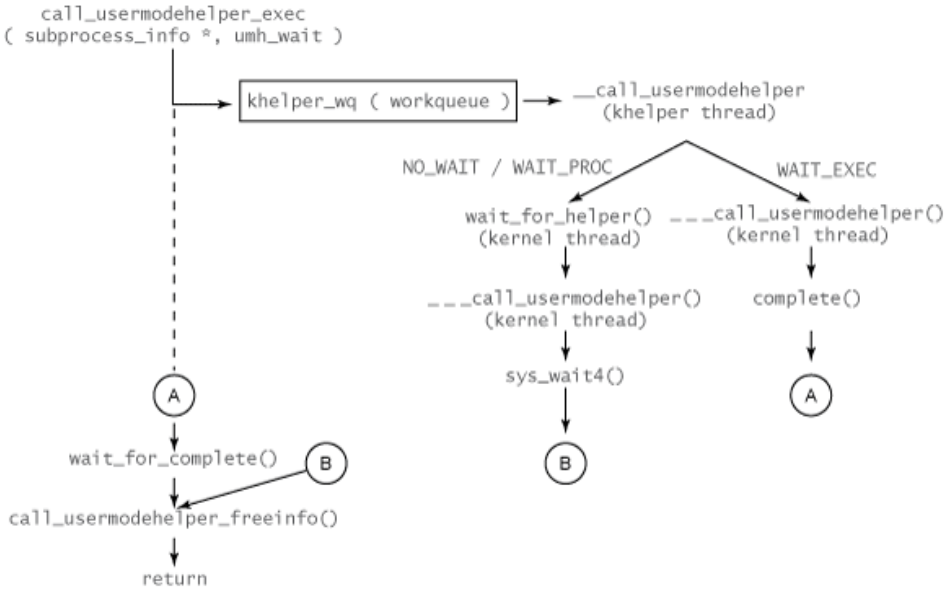
The usermode-helper API also supports Linux in an orderly system shutdown. When a system power-off is necessary, the kernel invokes the /sbin/poweroff command in user space to accomplish it. Other applications are listed in Table 3, with the accompanying source location.

Usermode-helper API internals

You'll find the source and API for the usermode-helper API in kernel/kmod.c (illustrating its primary use as the kernel-space kernel module loader). The implementation uses `kernel_execve` for the dirty work. Note that `kernel_execve` is the function used to start the `init` process at boot time and does not use the usermode-helper API.

The implementation of the usermode-helper API is quite simple and straightforward (see Figure 2). The work of the usermode-helper begins with the call to `call_usermodehelper_exec` (which is used to kick off the user-space application from a preconfigured `subprocess_info` structure). This function accepts two arguments: the `subprocess_info` structure reference and an enumeration type (whether to not wait, wait for the process to be kicked of, or wait for the entire process to be completed). The `subprocess_info` (or rather, the `work_struct` element of this structure) is then enqueued onto a work structure (`khelper_wq`), which asynchronously performs the invocation.

Figure 2. Internal implementation of the usermode-helper API



When an element is placed onto the `khelper_wq`, the handler function for the work queue is invoked (in this case, `__call_usermodehelper`), which is run through the `khelper` thread. This function begins by dequeuing the `subprocess_info` structure, which contains all of the necessary information for the user-space invocation. The path next depends upon the `wait` variable enumeration. If the requester wants to wait for the entire process to finish, including user-space invocation (`UMH_WAIT_PROC`) or not wait at all (`UMH_NO_WAIT`), then a kernel thread is created from the function `wait_for_helper`. Otherwise, the requester simply wants to wait for the user-space application to be invoked (`UMH_WAIT_EXEC`) but not complete. In this case, a kernel thread is created for `____call_usermodehelper()`.

In the `wait_for_helper` thread, a SIGCHLD signal handler is installed, and another kernel thread is created for `____call_usermodehelper`. But in the `wait_for_helper` thread, a call is made to `sys_wait4` to await termination of the `____call_usermodehelper` kernel thread (indicated by a SIGCHLD signal). The thread then performs any necessary cleanup (either freeing the structures for `UMH_NO_WAIT` or simply sending a completion notification back to `call_usermodehelper_exec()`.

The function `____call_usermodehelper` is where the real work happens for getting the application started in user space. This function begins by unblocking all signals and setting the session key ring. It also installs the `stdin` pipe (if requested). After a bit more initialization, the user-space application is invoked through a call to `kernel_execve` (from kernel/syscall.c), which includes the previously defined `path`, `argv` list (including the user-space application name), and environment. When this process is complete, the thread exits through a call to `do_exit()`.

This process also uses Linux completions, which is a semaphore-like operation. When the `call_usermodehelper_exec` function is invoked, a completion is declared. After the `subprocess_info` structure is placed on the `khelper_wq`, a call is made to `wait_for_completion` (using the completion variable as its only argument). Note that this variable is also stored in the `subprocess_info` structure as the `complete` field. When the child threads want to wake up the

`call_usermodehelper_exec` function, they call the kernel method `complete`, noting the completion variable from the `subprocess_info` structure. This call unlocks the function so that it can continue. You can find the implementation of this API in include/linux/completion.h.

You'll find more information on the usermode-helper API by following the links in the Resources section.

Sample application

Now, let's look at a simple use of the usermode-helper API. You'll first look at the standard API, then learn how to simplify things further using the helper functions.

For this demonstration, you develop a simple loadable kernel module that invokes the API. Listing 1 presents the boilerplate module functions, defining the module entry and exit functions. These two functions are invoked on `modprobe` or `insmod` of the module (module entry function) and `rmmod` of the module (module exit).

Listing 1. Module boilerplate functions

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kmod.h>

MODULE_LICENSE( "GPL" );

static int __init mod_entry_func( void )
{
  return umh_test();
}


static void __exit mod_exit_func( void )
{
  return;
}

module_init( mod_entry_func );
module_exit( mod_exit_func );
```

The use of the usermode-helper API is shown in Listing 2, which you'll explore in detail. The function begins with the declaration of a variety of needed variables and structures. Start with the `subprocess_info` structure, which contains all of the information necessary to perform the user-space invocation. This invocation is initialized when you call `call_usermodehelper_setup`. Next, define your argument list, called `argv`. This list is similar to the `argv` list used in common `c` programs and defines the application (first element of the array) and argument list. A NULL terminator is required to indicate the end of the list. Note here that the `argc` variable (argument count) is implicit, because the length of the `argv` list is known. In this example, the application name is /usr/bin/logger, and its argument is `help!`, which is followed by your terminating NULL. The next required variable is the environment array (`envp`). This array is a list of parameters that define the execution environment for the user-space application. In this example, you define a few typical parameters that are defined for the shell and end with a terminating NULL entry.

Listing 2. Simple usermode_helper API test

```
static int umh_test( void )
{
  struct subprocess_info *sub_info;
  char *argv[] = { "/usr/bin/logger", "help!", NULL };
  static char *envp[] = {
        "HOME=/",
        "TERM=linux",
        "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };

  sub_info = call_usermodehelper_setup( argv[0], argv, envp, GFP_ATOMIC );
  if (sub_info == NULL) return -ENOMEM;

  return call_usermodehelper_exec( sub_info, UMH_WAIT_PROC );
}
```

Next, make a call to `call_usermodehelper_setup` to create your initialized `subprocess_info` structure. Note that you use your previously initialized variables along with a fourth parameter that indicates the GFP mask for memory initialization. Internal to the setup function, there's a call to `kzalloc` (which allocates kernel memory and zeroes it). This function requires either `GFP_ATOMIC` or the `GFP_KERNEL` flag (where the former defines that the call should not sleep and the latter that sleep is possible). After a quick test of your new structure (namely, it's not NULL), continue to make the call using the `call_usermodehelper_exec` function. This function takes your `subprocess_info` structure and an enumeration defining whether to wait (described in the internals section). And that's it! Once the module is loaded, you should see the message in your /var/log/messages file.

You can simplify this process further by using the `call_usermodehelper` API function, which performs the `call_usermodehelper_setup` and `call_usermodehelper_exec` functions together. As shown in Listing 3, this not only removes a function but also removes the need for the caller to manage the `subprocess_info` structure.

Listing 3. An even simpler usermode-helper API test

```
static int umh_test( void )
{
  char *argv[] = { "/usr/bin/logger", "help!", NULL };
  static char *envp[] = {
        "HOME=/",
        "TERM=linux",
        "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };

  return call_usermodehelper( argv[0], argv, envp, UMH_WAIT_PROC );
}
```

Note that in Listing 3, the same requirements exist to set up and make the call (such as initializing the `argv` and `envp` arrays). The only difference here is that the helper function performs the `setup` and `exec` functions.

Going further

The usermode-helper API is an important aspect to the kernel, given its wide and varying use (from kernel module loading, device hot-plugging, and event distribution for udev). Although it's

important to validate genuine applications of the API, it's an important aspect of the kernel to understand and therefore a useful addition to your Linux kernel toolkit.

Resources

Learn

- developerWorks Premium provides an all-access pass to powerful tools, curated technical library from Safari Books Online, conference discounts and proceedings, SoftLayer and Bluemix credits, and more.
- Develop and deploy your next app on the IBM Bluemix cloud platform.
- Little information exists about the usermode-helper API, but the implementation is quite clean and simple to follow. You can review the implementation through LXR (the Linux Cross Referencer—source browser for all source revisions). The two primary files of interest are kmod.c and kmod.h.
- The /proc file system provides a method for communicating between the kernel and user space—namely, though a virtual file system. You can learn more about the /proc file system in "Access the Linux kernel using the /proc filesystem" (developerworks, March 2006).
- The Linux system call interface provides the means for user-space applications to invoke kernel functionality. For more details on Linux system calls, including how to add new system calls, see "Kernel command using system calls" (developerworks, March 2007).
- To illustrate the usermode-helper API, this article uses loadable kernel modules to install test applications into the kernel. To learn more about loadable kernel modules and their implementation, check out "Anatomy of Linux loadable kernel modules" (developerworks, July 2008).
- To learn more about the 2.6 kernel work queue interface, check out this older Linux Journal article from 2003, which provides a good introduction to the API and operation of kernel work queues.
- In the developerWorks Linux zone, find more resources for Linux developers.
- Stay current with developerWorks technical events and Webcasts.
- Follow developerWorks on Twitter.

Get products and technologies

- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

M. Tim Jones

M. Tim Jones is an embedded firmware architect and the author of *Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), 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.