NAME

init_module, finit_module − load a kernel module

SYNOPSIS

int init_module(void ****module_image*, unsigned long** *len***,**
const char ****param_values*);**

int finit_module(int *fd***, const char ****param_values***,**
int *flags***);**

Note: glibc provides no header file declaration of **init_module**() and no wrapper function for **finit_module**(); see NOTES.

DESCRIPTION

init_module() loads an ELF image into kernel space, performs any necessary symbol relocations, initializes module parameters to values provided by the caller, and then runs the module's *init* function. This system call requires privilege.

The *module_image* argument points to a buffer containing the binary image to be loaded; *len* specifies the size of that buffer. The module image should be a valid ELF image, built for the running kernel.

The *param_values* argument is a string containing space-delimited specifications of the values for module parameters (defined inside the module using **module_param**() and **module_param_array**()). The kernel parses this string and initializes the specified parameters. Each of the parameter specifications has the form:

name[**=***value*[**,***value*...]]

The parameter *name* is one of those defined within the module using *module_param*() (see the Linux kernel source file *include/linux/moduleparam.h*). The parameter *value* is optional in the case of *bool* and *invbool* parameters. Values for array parameters are specified as a comma-separated list.

finit_module()

The **finit_module**() system call is like **init_module**(), but reads the module to be loaded from the file descriptor *fd*. It is useful when the authenticity of a kernel module can be determined from its location in the filesystem; in cases where that is possible, the overhead of using cryptographically signed modules to determine the authenticity of a module can be avoided. The *param_values* argument is as for **init_module**().

The *flags* argument modifies the operation of **finit_module**(). It is a bit mask value created by ORing together zero or more of the following flags:

MODULE_INIT_IGNORE_MODVERSIONS
Ignore symbol version hashes.

MODULE_INIT_IGNORE_VERMAGIC
Ignore kernel version magic.

There are some safety checks built into a module to ensure that it matches the kernel against which it is loaded. These checks are recorded when the module is built and verified when the module is loaded. First, the module records a "vermagic" string containing the kernel version number and prominent features (such as the CPU type). Second, if the module was built with the **CONFIG_MODVERSIONS** configuration option enabled, a version hash is recorded for each symbol the module uses. This hash is based on the types of the arguments and return value for the function named by the symbol. In this case, the kernel version number within the "vermagic" string is ignored, as the symbol version hashes are assumed to be sufficiently reliable.

Using the **MODULE_INIT_IGNORE_VERMAGIC** flag indicates that the "vermagic" string is to be ignored, and the **MODULE_INIT_IGNORE_MODVERSIONS** flag indicates that the symbol version hashes are to be ignored. If the kernel is built to permit forced loading (i.e., configured with **CONFIG_MODULE_FORCE_LOAD**), then loading continues, otherwise it fails with the error **ENOEXEC** as expected for malformed modules.

RETURN VALUE

On success, these system calls return 0. On error, −1 is returned and *errno* is set appropriately.

ERRORS

EBADMSG (since Linux 3.7)
: Module signature is misformatted.

EBUSY
: Timeout while trying to resolve a symbol reference by this module.

EFAULT
: An address argument referred to a location that is outside the process's accessible address space.

ENOKEY (since Linux 3.7)
: Module signature is invalid or the kernel does not have a key for this module. This error is returned only if the kernel was configured with **CONFIG_MODULE_SIG_FORCE**; if the kernel was not configured with this option, then an invalid or unsigned module simply taints the kernel.

ENOMEM
: Out of memory.

EPERM
: The caller was not privileged (did not have the **CAP_SYS_MODULE** capability), or module loading is disabled (see */proc/sys/kernel/modules_disabled* in **proc**(5)).

The following errors may additionally occur for **init_module**():

EEXIST
: A module with this name is already loaded.

EINVAL
: *param_values* is invalid, or some part of the ELF image in *module_image* contains inconsistencies.

ENOEXEC
: The binary image supplied in *module_image* is not an ELF image, or is an ELF image that is invalid or for a different architecture.

The following errors may additionally occur for **finit_module**():

EBADF
: The file referred to by *fd* is not opened for reading.

EFBIG
: The file referred to by *fd* is too large.

EINVAL
: *flags* is invalid.

ENOEXEC
: *fd* does not refer to an open file.

In addition to the above errors, if the module's *init* function is executed and returns an error, then **init_module**() or **finit_module**() fails and *errno* is set to the value returned by the *init* function.

VERSIONS

finit_module() is available since Linux 3.8.

CONFORMING TO

init_module() and **finit_module**() are Linux-specific.

NOTES

The **init_module**() system call is not supported by glibc. No declaration is provided in glibc headers, but, through a quirk of history, glibc versions before 2.23 did export an ABI for this system call. Therefore, in order to employ this system call, it is (before glibc 2.23) sufficient to manually declare the interface in your code; alternatively, you can invoke the system call using **syscall**(2).

Glibc does not provide a wrapper for **finit_module**(); call it using **syscall**(2).

Information about currently loaded modules can be found in */proc/modules* and in the file trees under the per-module subdirectories under */sys/module*.

See the Linux kernel source file *include/linux/module.h* for some useful background information.

Linux 2.4 and earlier

In Linux 2.4 and earlier, the **init_module**() system call was rather different:

#include <linux/module.h>

int init_module(const char ***name, struct module ***image**);**

(User-space applications can detect which version of **init_module**() is available by calling **query_module**(); the latter call fails with the error **ENOSYS** on Linux 2.6 and later.)

The older version of the system call loads the relocated module image pointed to by *image* into kernel space and runs the module's *init* function. The caller is responsible for providing the relocated image (since Linux 2.6, the **init_module**() system call does the relocation).

The module image begins with a module structure and is followed by code and data as appropriate. Since Linux 2.2, the module structure is defined as follows:

```
struct module {
    unsigned long         size_of_struct;
    struct module        *next;
    const char           *name;
    unsigned long         size;
    long                  usecount;
    unsigned long         flags;
    unsigned int          nsyms;
    unsigned int          ndeps;
    struct module_symbol *syms;
    struct module_ref    *deps;
    struct module_ref    *refs;
    int                 (*init)(void);
    void                (*cleanup)(void);
    const struct exception_table_entry *ex_table_start;
    const struct exception_table_entry *ex_table_end;
#ifdef __alpha__
    unsigned long gp;
#endif
};
```

All of the pointer fields, with the exception of *next* and *refs*, are expected to point within the module body and be initialized as appropriate for kernel space, that is, relocated with the rest of the module.

SEE ALSO

create_module(2), **delete_module**(2), **query_module**(2), **lsmod**(8), **modprobe**(8)

COLOPHON

This page is part of release 5.02 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.