Loadable Filesystem Modules in *janus86*

Janus86 – A Bootloader Integrated kernel

Roshan Sam Thomas, Sibi Antony, Subi Bhaskaran

1 Dynamically Loadable Modules

1.1 Loadable Modules in janus86

One of the remarkable features of *janus86* is the capability to load modules dynamically to the kernel. The loadable module feature provides flexibility in kernel development as the kernel can be compiled independent of the drivers loaded though modules. Dynamically loaded filesystem modules provide access to filesystems without the need for statically compiling them into the kernel. Any new filesystem will be supported without the need to recompile the whole kernel. A typical loadable module has a module header, the standard driver routines, a symbol table and an optional C library.

2 The Filesystem Module

A filesystem module provides the kernel to identify and access a filesystem at runtime. At present *janus86* provides filesystem modules for ext2 and fat32 filesystems. A filesystem module exports three symbols which are root , chdir and loadfile . These are the standard filesystem driver routines that the kernel invokes in order to traverse a filesystem. Whenever a filesystem module is loaded the kernel registers the module in the available drivers list. The kernel looks for the symbol definitions and stores the symbol values in the registered driver list entry. Details of registering a module are explained later.

2.1 The Module Layout

A typical loadable module in *janus86* has the following layout.

Module Header	**Driver Routines**	**Symbol Table**	**Optional C Library**

2.2 The Module header

```
/* structure for a loaded fs module header*/
typedef struct fs_drv_mod_struct {
      unsigned short mod_sign;
      unsigned short mod_size;
      unsigned long load_mem;
      unsigned long start_addr;
      unsigned long sym_tab_addr;
      unsigned char fs_type_byte;
      char mod_name[10];
}fs_drv_mod;
```

mod_sign – Module signature to identify a janus86 module. It carries the value 0xEF86.

mod_size – Size of the loaded module. The size field only informs the kernel about the size of the driver routines and the size of the C library linked with the module is excluded.

load_mem – The physical address at which the module is to be loaded.

start_addr – Start address of the module after the module header.

sym_tab_addr – This field informs the kernel about the location of the symbol table. The symbol table is accessed at the time of registering a module.

fs_type_byte – Informs the kernel about the filesystem type supported by the module.

mod_name – Module name, used to identify a module. The module name is passed as argument while removing a registered module.

2.3 Filesystem module routines

A filesystem module in *janus86* has three standard driver routines which are –

- root - root() initializes a driver to be used with a filesystem. It performs filesystem related initializations such as initializing inode/cluster numbers, LBA values etc.

  ```
  struct fs_properties *(*drv_root) (int start_lba, int drive,
  int inode);
  ```

- chdir - chdir enables to change to a directory with the directory path passed as argument.

  ```
  struct fs_properties* (*drv_chdir) (char *path);
  ```

- loadfile – loads a file to a specified memory location.

```
      int (*drv_loadfile) (char * filename,unsigned char
*load_mem);
```

2.4 Symbol Table

Symbol table holds the values for the symbols exported from the driver module.
The standard symbols used in the filesystem module are root, chdir and loadfile.
The symbol table structure is as shown.

```
/* The symbol table format in the module */
typedef struct fs_sym_tab_struct {
      char sym_name[10];
      unsigned long sym_val;
}fs_sym_tab;
```

sym_name – name of the exported symbol.

sym_val – exported symbol's value.

2.5 Registering a filesystem module

Whenever a new filesystem module is loaded, the module is registered with a list of available drivers. The register_module function is invoked that identifies which module does what and saves their symbol values. The module is checked for the signature 0xEF86. If the loaded module is found to be a valid module a new module entry is created and the symbol values are initialized.

```
new_mod_header = (fs_drv_mod *)kmalloc (sizeof(fs_drv_mod));


new_mod->fs_type_byte   = new_mod_header->fs_type_byte;
new_mod->load_mem       = new_mod_header->load_mem;
strcpy (new_mod->drv_name, new_mod_header->mod_name);
new_mod->sym_root       = new_mod_sym[0]->sym_val;
new_mod->sym_chdir      = new_mod_sym[1]->sym_val;
new_mod->sym_loadfile   = new_mod_sym[2]->sym_val;
```

2.6 Removing a Filesystem Module

Janus86 also provides a facility to remove a registered module. The unregister_module routine deletes the module entry from the list.

```
int unregister_module (char *drv_name)
```

The function accepts the driver name as parameter and searches for this name in the list of registered drivers. If the driver is found the module entry is deleted from the list and the memory is freed.

```
if (strcmp(tmp->drv_name, drv_name) == 0)
{
      fs_drv_list = fs_drv_list->drv_next;
      kfree (tmp);
      return FS_MODULE_REMOVED;
}
```

2.7 Invoking the Driver Routines

Inside the kernel the filesystem section provides an interface similar to that of the standard routines in a filesystem driver. This makes any filesystem driver compatible with the kernel irrespective of the filesystem type. Also, as the filesystem drivers are loaded separately, the kernel size can be further reduced. The fs_root, fs_chdir and fs_loadfile routines initialize the function pointers at runtime, which enables to switch between different filesystems.

- fs_root - Inside the kernel this routine performs two tasks. At first it identifies the partition type byte corresponding to the partition number passed as argument to it. It then looks for the available filesystem drivers that can handle the requested partition. If the driver is found then the function pointers are initialized with the corresponding symbol values.

```
if (tmp->fs_type_byte == part_temp->type_byte)
{
      drv_root = tmp->sym_root;
      drv_chdir = tmp->sym_chdir;
      drv_loadfile = tmp->sym_loadfile;
      fs_prop = (*drv_root)(part_temp->st_lba, 0, 2);
      return FS_DRV_FOUND;
}
```

- fs_chdir – invokes the chdir routine for the filesystem requested through fs_root.

```
struct fs_properties* fs_chdir (char *path)
{
      if (drv_chdir != NULL)
            return ((*drv_chdir)(path));
      else
            return FS_DRV_NOT_FOUND;
}
```

- fs_loadfile – invokes the loadfile routine for the filesystem requested through fs_root

```
int fs_loadfile (char *filename, unsigned char *load_mem)
{
      if (drv_loadfile != NULL)
            return ((*drv_loadfile)(filename, load_mem));
      else
            return FS_DRV_NOT_FOUND;
}
```