Creating files in sysfs

eMN Technologies, www.emntech.com

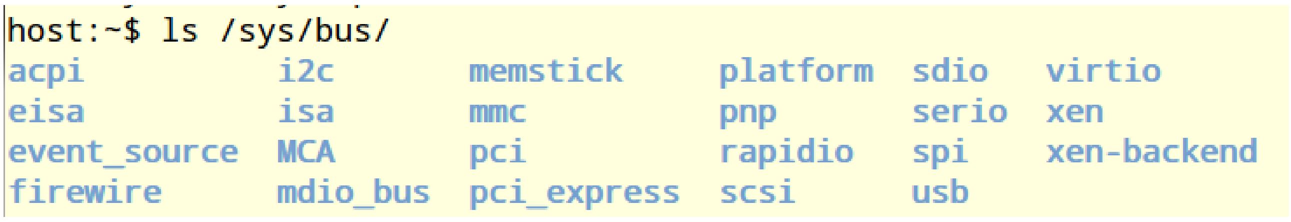
This document will try to explain you how to create files in sysfs pseudo file system. Here we will write a small module which creates files in sysfs.

Before we see the program let's understand a few concepts about sysfs. As you know sysfs is a pseudo file system which contains files that show the devices and device drivers information in Linux. Sysfs is an exact representation of Linux device driver model.

Introduction

As you know, Linux device driver model relies on three components called buses, devices and device drivers that represent a particular device. A bus may have lot of devices on it and each device is attached with a device driver that serves the device. There are system buses like PCI, PCIe, ISA, MCA and so on and there are peripheral buses like USB, I2C, SPI, SCSI and so on. On each bus there are devices. For example on PCI bus you have devices like USB Host controller, I2C Adapter, NIC Card and so on and on USB bus you have devices like pen drive, usb keyboard, usb mouse and so on and on I2C you may have devices like eeprom, sensor devices and so on. So, on PCI bus there are PCI devices, on USB bus there are usb devices and on I2C bus you have i2c devices. For each of these devices you require a device driver that drives that device. For example you have a USB Host controller device driver which drives the USB host controller that is on PCI bus and there is a usb storage device driver which drives USB mass storage devices like pen drives that are attached to the USB Host controller using USB bus.

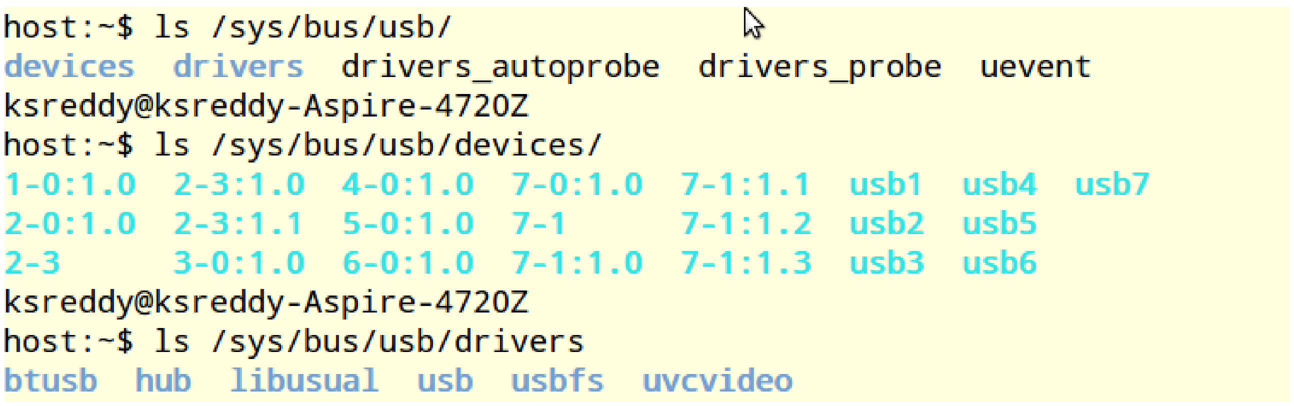
So, main things in Linux device driver model are buses, devices, and device drivers. Sysfs contains a logical representation of these three components. For example there is a directory named 'bus' in sysfs which contains a directory for each bus that is on the system. Like this.



```
host:~$ ls /sys/bus/
acpi          i2c       memstick     platform  sdio   virtio
eisa          isa       mmc          pnp       serio  xen
event_source  MCA       pci          rapidio   spi    xen-backend
firewire      mdio_bus  pci_express  scsi      usb
```

You can see there is a directory for each bus in /sys/bus, like, pci, i2c, usb, scsi and so on.

Inside these directories there are two directories, *devices* and *drivers* which contain a list of devices that are on a particular bus, and list of device drivers that are attached to the each device on this bus. For example let's take usb bus, it looks like this.



```
host:~$ ls /sys/bus/usb/
devices  drivers  drivers_autoprobe  drivers_probe  uevent
ksreddy@ksreddy-Aspire-4720Z
host:~$ ls /sys/bus/usb/devices/
1-0:1.0  2-3:1.0  4-0:1.0  7-0:1.0  7-1:1.1  usb1  usb4  usb7
2-0:1.0  2-3:1.1  5-0:1.0  7-1      7-1:1.2  usb2  usb5
2-3      3-0:1.0  6-0:1.0  7-1:1.0  7-1:1.3  usb3  usb6
ksreddy@ksreddy-Aspire-4720Z
host:~$ ls /sys/bus/usb/drivers
btusb  hub  libusual  usb  usbfs  uvcvideo
```

And there are many other directories in sysfs, like, *devices*, *class*, *fs*, *dev, kernel* and so on. But this is not the article that explains these all directories, we will have a saperate article on Linux device driver model later, with code walkthrough.

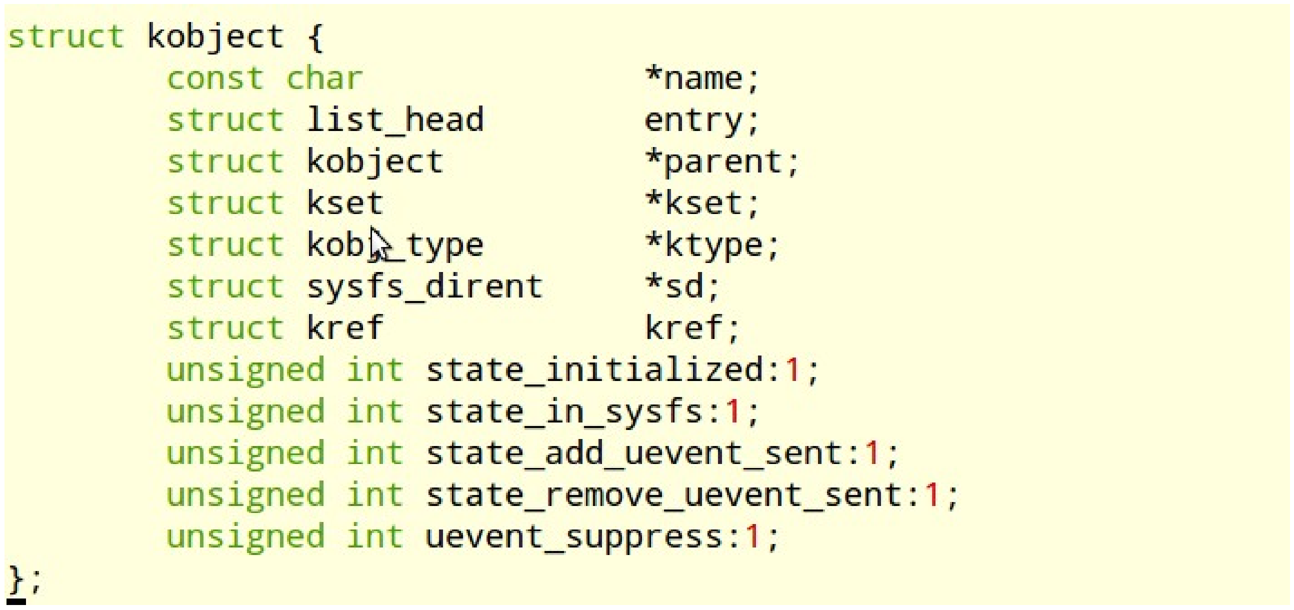
Each directory under sysfs may contain some files called attributes. For example there is an attribute(i.e a file in sysfs) 'uevent_helper' under /sys/kernel directory which can be used to specify the kernel which program has to be invoked when there is a change, for example a new device is added, a device is removed and so on, in the device drivers subsytem(hotpluging).

Basic Data structures

Before we write a module to create files in sysfs, let's understand some basic data structures that are required to create files.

Kobject

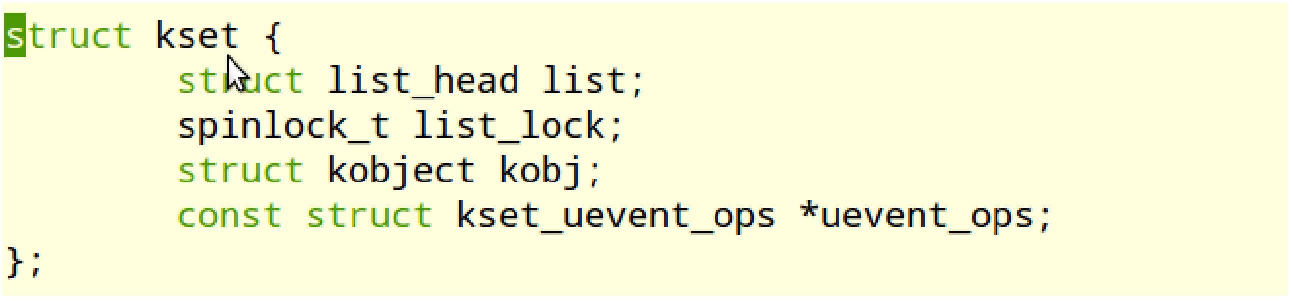
As per the Linux Device driver's model, each directory in sysfs is an object of type struct kobject. This data structure contains infomation like name of the directory, reference count of this kobject, pointer to parent kobject if any, pointer to kset(see below) and so on. This data structure looks like this.



```
struct kobject {
        const char              *name;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct sysfs_dirent     *sd;
        struct kref             kref;
        unsigned int state_initialized:1;
        unsigned int state_in_sysfs:1;
        unsigned int state_add_uevent_sent:1;
        unsigned int state_remove_uevent_sent:1;
        unsigned int uevent_suppress:1;
};
```

Kset

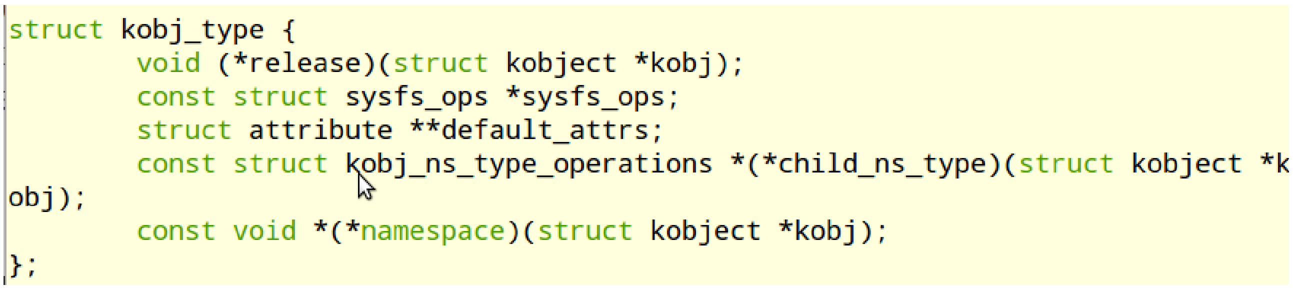
Any kobject should come under a subsystem, let's say 'bus' is a subsystem, 'usb' is a subsystem, 'devices' is a subsystem and so on. Each subsystem is represented with an object of type struct kset. Note that this subsystem is a directory in sysfs, so this subsystem itself is a kobject. So, kset contains an embedded kobject which represents this subsystem in the sysfs. And it also contains a list of kobjects under this kset(i.e under this subsytem).   This structure looks like this.



```
struct kset {
        struct list_head list;
        spinlock_t list_lock;
        struct kobject kobj;
        const struct kset_uevent_ops *uevent_ops;
};
```

kobj_type

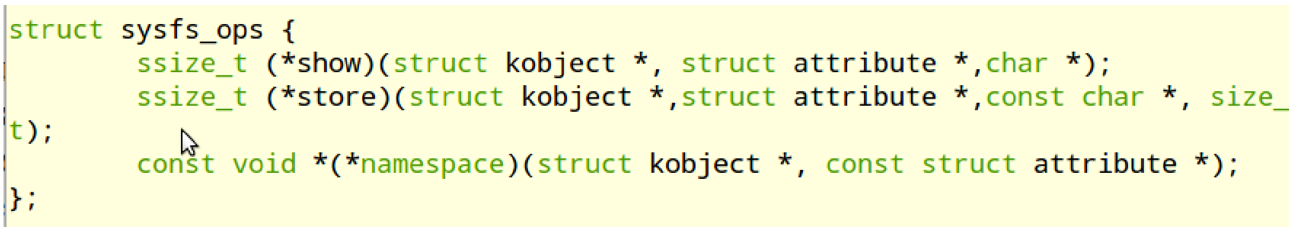
Each kobject should contain a pointer to an object of type kobj_type which points to a list of default attributes(files) under this kobject and sysfs operations that are to be invoked when there is a read or write on a given attribute. This data structure looks like this.



```
struct kobj_type {
        void (*release)(struct kobject *kobj);
        const struct sysfs_ops *sysfs_ops;
        struct attribute **default_attrs;
        const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *k
obj);
        const void *(*namespace)(struct kobject *kobj);
};
```

sysfs_ops

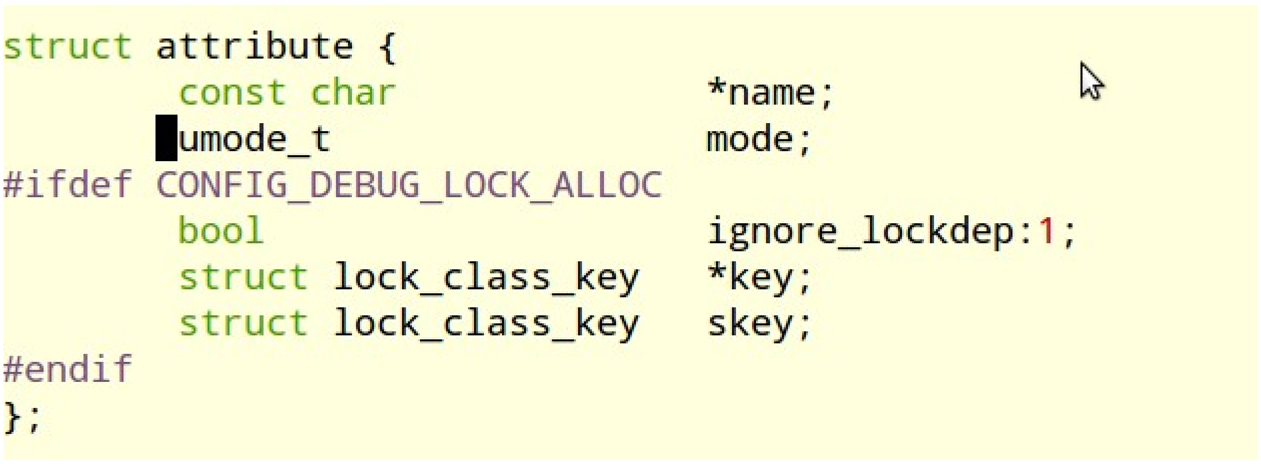
This is a data structure which has two function pointers, *show* and *store*, which will be invoked when there is a read or write on a particular attribute(file).



```
struct sysfs_ops {
        ssize_t (*show)(struct kobject *, struct attribute *,char *);
        ssize_t (*store)(struct kobject *,struct attribute *,const char *, size_
t);
        const void *(*namespace)(struct kobject *, const struct attribute *);
};
```

attribute

Each attribute(file) in sysfs is an object of type struct attribute. This data structure contains name of the attribute and mode of the attribute(file mode). This is the data structure.



```
struct attribute {
        const char                  *name;
        umode_t                     mode;
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        bool                        ignore_lockdep:1;
        struct lock_class_key       *key;
        struct lock_class_key       skey;
#endif
};
```

Program to Create attributes(files) in sysfs

We have seen basics about sysfs and its data structures. Now let's implement a module which creates a directory(kobject) and two attributes(files) in sysfs. And we need to decide under which directory in sysfs we want to create this new directory and files. You can create a new directory in main sysfs directory(under /sys) itself or you can select any existing directory to create the new directory. But to create a new directory under an existing directory you should know the pointer to the kobject of the existing directory. Let's try to create our new directory under kernel(sys/kernel) directory in sysfs. Kobject of this directory is exported using EXPORT_SYMBOL_GPL in <kernel/ksysfs.c> file, like this.

```
struct kobject *kernel_kobj;
EXPORT_SYMBOL_GPL(kernel_kobj);
```

So, we can use this kernel_kobj as our parent kobject.

Let's assume we want to create a directory named 'mydir' under /sys/kernel and let's assume we want to create two files named 'foo' and 'bar' under 'mydir' (/sys/kernel/mydir). Following are the steps you should follow.

Step1: Create an object of type struct sysfs_ops and define show and store functions.

Step2: Create an object of type struct kobj_type and fill in the sysfs operations

Step3: Create two struct attribute objects one for each file.

Step4: Create an object of type struct kobject.

Step5: Initialize and add the kobject to the device drivers model(to the sysfs).

Step6: Create two files in /sys/kernel/mydir

Step1

Create an object of type struct sysfs_ops and define show and store functions. I did like this.

```
struct sysfs_ops mysysfs_ops = {
            .show = myshow,
            .store = mystore,
};
```

myshow and mystore functions are implemented as follows.

```
static ssize_t myshow(struct kobject *obj,
             struct attribute *attr, char *buf)
{

      if (!strcmp(attr->name, "foo")) {
            strncpy(buf, foo, PAGE_SIZE);
            return strlen(foo);
      } else {
            strncpy(buf, bar, PAGE_SIZE);
            return strlen(bar);
      }
}


static ssize_t mystore(struct kobject *kobj,
                  struct attribute *attr,
                  const char *buf,
                  size_t count)
{
      if (count >= (PAGE_SIZE - 1))
            count = PAGE_SIZE - 1;

      if (!strcmp(attr->name, "foo")) {
            memcpy(foo, buf, count);
            foo[count] = '\0';
      } else {
            memcpy(bar, buf, count);
            bar[count] = '\0';
      }

      return count;
}
```

myshow() function will be called whenever there is a read operation on any of the two files 'foo' and 'bar'. It receives three parameters, pointer to the kobject of our 'mydir' directory, pointer to the struct attribute of the corresponding file(foo's attribute or bar's attribute) and a pointer to a buffer into which we need to write the data. In our case we have taken two buffers named 'foo' and 'bar' which are used to store the data and used to show the data from. These two buffers defined like this.

```
        static char foo[PAGE_SIZE];
        static char bar[PAGE_SIZE];
```

If a read operation is on 'foo' file, we will give data from 'foo' buffer and if there is a read on 'bar' file we will give data from 'bar' buffer. Note that at a time you can give only one PAGE of data, that's why I take buffer size as PAGE_SIZE.

In myshow() function we will check on which file is the read operation by comparing attribute->name with 'foo' and 'bar'.

mystore() function will be called whenever there is a write operation on any of the two files 'foo' and 'bar'. This function receives four parameters, pointer to the kobject of the directory 'mydir', pointer to the struct attribute of the corresponding file(foo's attribute or bar's attribute), pointer to a buffer which contains the user give data and a count parameter which tells number of bytes of data in the given buffer.

We will find out the file on which the read operation is using attribute->name and copy the given data into the corresponding buffer(foo buffer or bar buffer).

Step2

Create an object of type struct kobj_type and fill in the sysfs operations. I did like this.

```
struct kobj_type mykobj_type = {
          .sysfs_ops = &mysysfs_ops,
};
```

Step3 and Step4

Create two struct attribute objects one for each file and create an object of type struct kobject. I did like this.

```
struct mydir {
     struct kobject kobj;
     struct attribute attr[2];
};

struct mydir mydir_object = {
     .attr = {
          {
          .name = "foo",
          .mode = 0666,
          },
          {
           .name = "bar",
           .mode = 0666,
          },
     },
};
```

I created a new structure struct mydir and embedded struct kobject object and two struct attribute objects. And filled in the two attribute object with names of the files and modes of the files. Here mode will tell who can do read/write on these files. I gave read/write permissions for all users.

Step5

Initialize and add the kobject to the device drivers model(to the sysfs). I did like this.

```
        kobject_init(&mydir_object.kobj, &mykobj_type);
        ret = kobject_add(&mydir_object.kobj, kernel_kobj, "mydir");
        if (ret) {
                printk("Error adding kobject\n");
                kobject_put(&mydir_object.kobj);
                return  ret;
        }
```

First I called kobject_init() function to initialize the kobject. Note that I passed pointer to the struct kobj_type object mykobj_type. And finally I called kobject_add() function to add the kobject to the sysfs. This function will add the kobject to the device driver model and creates a directory with the name 'mydir' in sysfs. Parent kobject of this kobject will be kernel_obj, that is /sys/kernel directory. Directory 'mydir' will be created in /sys/kernel/.

Step6

Create two attributes(files) in /sys/kernel/mydir. I did like this.

```
        ret = sysfs_create_file(&mydir_object.kobj, &mydir_object.attr[0]);
        if (ret) {
                printk("Error creating file in sysfs\n");
                kobject_put(&mydir_object.kobj);
                return ret;
        }

        ret = sysfs_create_file(&mydir_object.kobj, &mydir_object.attr[1]);
        if (ret) {
                printk("Error creating file in sysfs\n");
                kobject_put(&mydir_object.kobj);
                return ret;
```

}

I called sysfs_create_file() function to create files in sysfs. We need to pass two parameters to this function, one is the pointer to the kobject of the directory in which these files to be created and pointer to the struct attribute object of the corresponding file.

That's it, we are done with creating directory and files in sysfs.

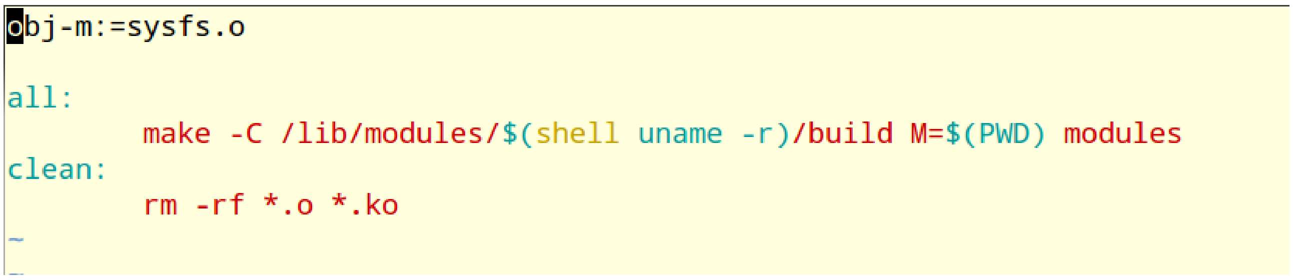
When you want to delete these directories and files, just invoke kobject_put() function which delete the given kobject and it's attributes. You can call this function like this.

```
kobject_put(&mydir_object.kobj);
```

Testing

Compile the module using the following Makefile.

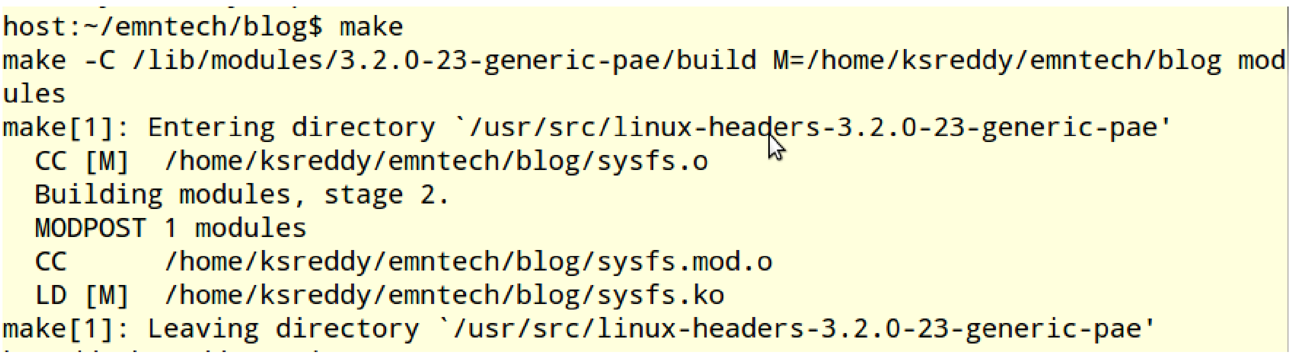
//vim Makefile



```
obj-m:=sysfs.o

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
        rm -rf *.o *.ko
~
```

And run make command to compile the module.



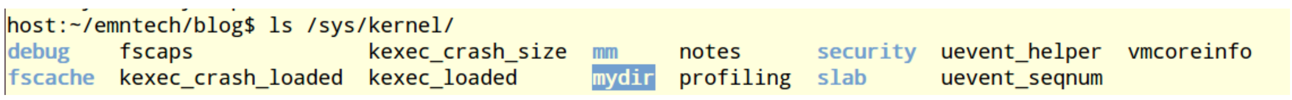
```
host:~/emntech/blog$ make
make -C /lib/modules/3.2.0-23-generic-pae/build M=/home/ksreddy/emntech/blog mod
ules
make[1]: Entering directory `/usr/src/linux-headers-3.2.0-23-generic-pae'
  CC [M]  /home/ksreddy/emntech/blog/sysfs.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/ksreddy/emntech/blog/sysfs.mod.o
  LD [M]  /home/ksreddy/emntech/blog/sysfs.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.2.0-23-generic-pae'
```

Note that I have compiled the module for kernel version **linux-3.2.0**.

Now insert the module using insmod.

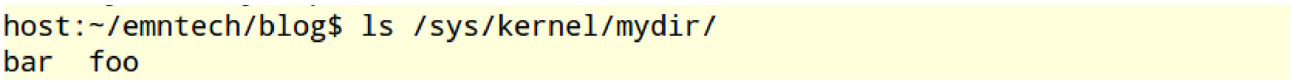
host:~/emntech/blog$ sudo insmod ./sysfs.ko

You should now see a new directory 'mydir' under /sys/kernel, like this.



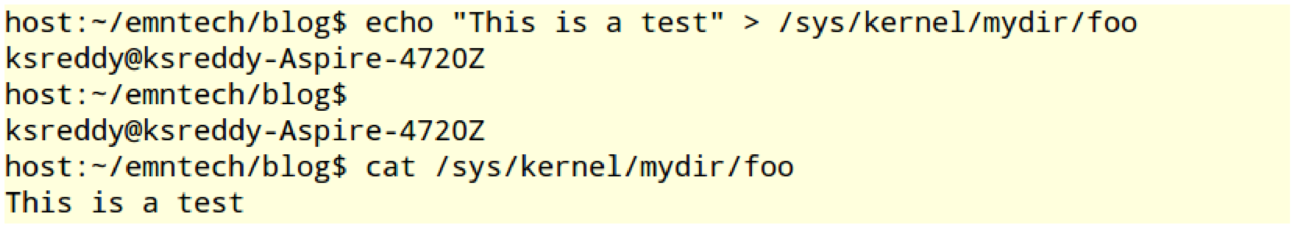
```
host:~/emntech/blog$ ls /sys/kernel/
debug    fscaps             kexec_crash_size  mm     notes      security  uevent_helper  vmcoreinfo
fscache  kexec_crash_loaded kexec_loaded      mydir  profiling  slab      uevent_seqnum
```

And two files 'foo' and 'bar' in 'mydir' like this.



```
host:~/emntech/blog$ ls /sys/kernel/mydir/
bar  foo
```

Now write any message into any of these two files and read it back, like this.



```
host:~/emntech/blog$ echo "This is a test" > /sys/kernel/mydir/foo
ksreddy@ksreddy-Aspire-4720Z
host:~/emntech/blog$
ksreddy@ksreddy-Aspire-4720Z
host:~/emntech/blog$ cat /sys/kernel/mydir/foo
This is a test
```

That's it.

You can download the module here