./

Learning Report – Kernel Programming



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver. Rel. No.** | **Release Date** | **Prepared. By** | **Reviewed By** | **Approved By** | **Remarks/Revision Details** |
|  |  | Aditya Mewara |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Document History**

Table of Contents

[1. WHAT DO YOU MEAN BY KERNEL? 4](#_Toc68326545)

# WHAT DO YOU MEAN BY KERNEL?

* A kernel is the central part of an operating system. It manages the operations of the computer and the hardware, most notably memory and CPU time
* It decides which process should be allocated to processor to execute and which process should be kept in main memory to execute.

**There are three types of kernels:**

* 1. **A monolithic kernel**
* It is one of types of kernel where all operating system services operate in kernel space. It has dependencies between systems components. It has huge lines of code which is complex
* Advantage

It has good performance.

* Disadvantage

It has dependencies between system component and lines of code in millions.

* 1. **A micro kernel**
* It is kernel types which has minimalist approach. It has virtual memory and thread scheduling. It is more stable with less services in kernel space. It puts rest in user space.
* Advantage

It is more stable.

* Disadvantage

There are lots of system calls and context switches.

* 1. **Hybrid Kernel**
* It is the combination of both monolithic kernel and microkernel. It has speed and design of monolithic kernel and modularity and stability of microkernel.
* Advantage

It combines both monolithic kernel and microkernel.

* Disadvantage

It is still similar to monolithic kernel.

# WHAT DO YOU MEAN BY MODULES?

* **Modules are** pieces of code that **can** be loaded and unloaded into the **kernel** upon demand.
* They extend the functionality of the kernel without the need to reboot the system.
* The kernel consists of a set of kernel modules that interact with each other, each performing a specific function. Some kernel modules perform software functions exclusively, while others (such as device drivers) control the operation of system hardware components.

1. **Activity QEMU installation**

* QEMU is a generic and open source machine emulator and virtualizer.
* QEMU is used to emulate devices and certain privileged instructions and requires either the KQEMU or KVM kernel modules and the host operating system

**Installing QEMU on ARM based architecture**

* + sudo apt install qemu-system-arm

**Running QEMU by ZImage and vexpress dtb file**

* qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio \ -kernel zImage -dtb vexpress-v2p-ca9.dtb \ -sd rootfs.img -append "console=ttyAMA0 root=/dev/mmcblk0 rw"

1. **Activity TOOLCHAIN Installation**

Installing soft load on ARM Architecture

sudo apt install gcc-arm-linux-gnueabi

1. **Download Kernel Source**

Downloading from linux tar.xz from the source and extract it in a new folder

And then

Obtain the zImage and vexpress dtb file

Linux Commands:

make ARCH=arm mrproper

make ARCH=arm vexpress\_defconfig

1. **Building Kernel Modules:**
   1. **Simple Hello Module:**

* Step 1 : Building the hello.c file and writing the contents
* Step 2: make file and writing the contents ( obj-m += hello.o )
* Cross compile using make
  + make –C ${KSRC} M=${PWD} modules ARCH=arm, CROSS\_COMPILE=arm-linux-gnueabi-
* testing on target
  + sudo mount –o loop,rw,sync rootfs.img /mnt/rootfs
  + sudo cp hello.ko /mnt/rootfs/home/root
  + sudo umount /mnt/rootfs
  1. **Simple hello Module with init and exit function**
* Building the hello.c file and writing the contents
* make file and writing the contents
  + obj-m += hello.o
  + KSRC = (where you have linux tar.xz location)
  + all: make –C ${KSRC} M=${PWD} modules
  + clean: make –C ${KSRC} M=${PWD} clean
* Cross compile using make command
* Testing on the target
  1. **Hello module with parameters**
* Building the hello.c file and writing the contents
* **The contents added to be are:**
* int ndevices=1
* module\_param(ndevices,int,S\_IRUGO);
* make file and writing the contents
* make file and writing the contents
* Now in host i.e QEMU pass the arguments like insmod ndevices = 5 or by default it will be 1
  1. **Module Dependency simple**
* Building the hello.c file and writing the contents
  + - The contents added to be are :
    - The functions and variable are present in the hello.c file
    - EXPORT\_SYMBOL\_GPL(xvar);
    - EXPORT\_SYMBOL\_GPL(sayHello);
    - make file and writing the contents
      * **obj-m += simple.o  
        all:  
         make -C /home/user/eworkspace/kernel\_ws/ksrc M=${PWD} modules ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi-  
        clean:  
         make -C /home/user/eworkspace/kernel\_ws/ksrc M=${PWD} modules ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi-**
      * Now open the emulation using tempboot location
      * run command to print the contents insmod (.ko) file
      * dmesg will display the contents of the file
  1. **Module Dependency sample**
* Building the hello.c file and writing the contents
  + - The contents are added to be are apart from simple
    - **extern int xvar;**
    - **extern void sayHello(void);**
    - Then after importing the module from simple we can use the functions defined in the simple module by printing in the sample module
    - We need to first run the simple module and then sample module so that we can use the functions present in the simple module
  1. **ADDING KCONFIG ENTRIES**
     1. Version 1 for K config entries:

Name a file hello.c in folder mtest

* + - * + config HELLO

tristate "Hello module"

default n

help

A Hello module

* + - * + Now making Makefile for the program:

obj-$(CONFIG\_SIMPLE) += hello.o

* + - * + Now update the make file present in the outside folder that is char folder
        + obj-y += mtest/
        + Add the statement to the outside K config

source "drivers/char/mtest/Kconfig"

* + 1. Version 2 for K Config entries:
       - * Name a file hello.c in folder mtest add into Kconfig blank file
         * menu "My Custom Modules“

config SIMPLE

tristate "Simple module"

default n

help A

Hello module

endmenu

* + 1. Version3 for K config entries:
       - * Name a file hello.c in folder mtest add into Kconfig blank file
         * menuconfig CUSTOM

tristate "My Custom Modules"

select SIMPLE

help "My Custom modules"

if CUSTOM

config SIMPLE

tristate "Simple module"

default n

help

A simple module

config SAMPLE

tristate "Sample module"

depends on SIMPLE

default n

help A

sample module

endif

1. Writing the make file in mtest to be included:
2. obj-$(CONFIG\_CUSTOM) += mtest/

# WHAT DO YOU MEAN BY SYSTEM CALLS?

* A **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
* System call **provides** the services of the operating system to the user programs via Application Program Interface(API).
* Services Provided by System Calls :
  + Process creation and management
  + Main memory management
  + File Access, Directory and File system management
  + Device handling(I/O)
  + Protection
  + Networking

1. **Adding a system call:**
   * We need to add the syscall.h with linkage
     + asmlinkage long sys\_mytestcall(void);
   * Adding syscall number so that kernel can identify by the number:
     + 398 common mytestcall sys\_mytestcall
   * In kernel folder add mysys.c file :
     + kernel/mysys.c
   * Update the kernel/Makefile:

obj-y +=mysys.o

* Write this code in kernel/mysys.c file:

SYSCALL\_DEFINE0(testcall)

{

printk("This is my test call\n");

return 0;

}

Invoking System Call from Userspace:

Method 1: Generic wrapper class

Create a .c file and write this code in that file:

#include<stdio.h>

#include<

#define \_\_NR\_testcall 398

int main()

{

int ret;

ret=syscall(\_\_NR\_testcall);

if(ret<0)

perror(“Testcall”);

return 0;

}

Run the system calls by :

./filename.out

**Pseudo Char Driver:**

Step1 : Register Char Driver

Registering the new device to the system means assigning a [major number](https://www.embhack.com/introduction-to-major-and-minor-number/) to it, during the initialization routine. The major number is provided by the kernel for any character or block device.

Two types of ways of restering a character device driver

1. Statistically registration of character device driver
2. Dynamically registration of character device driver

### Statistically registration device driver

When we know the [major number](https://www.embhack.com/introduction-to-major-and-minor-number/) in advance we can register the device using this method.

Two functions in the kernel for statistical registration of device driver:

* **register\_chrdev()**

int register\_chrdev(unsigned int major, const char \*name, struct file\_operations \*fops);

* **register\_chrdev\_region()**

int register\_chrdev\_region(dev\_t first, unsigned int count, char \*name)

### Dynamically registration of Character Device Driver

In this method, Kernel gives the highest available major number to the device.

1. alloc\_chrdev\_region

The prototype of alloc\_chrdev\_region, is declared in <linux/fs.h>:

int alloc\_chrdev\_region(dev\_t \*dev, unsigned int firstminor, unsigned int count, char \*name);

### Un-registration of character device driver

To deallocate an allocated major number use the ***unregister\_chrdev()*** function. The prototype is given below and the parameters of the function are self-explanatory:

void unregister\_chrdev\_region(dev\_t first, unsigned int count);

Step-2 : Register File Operations

The various operations a driver can perform on the devices it manages.

open device is identified internally by a file structure, and the kernel uses the file\_operations structure to access the driver’s functions.

The structure, defined in <linux/fs.h>, is an array of function pointers. Each file is associated with its own set of functions (by including a field called f\_op that points to a file\_operations structure).

The operations are mostly in charge of implementing the system calls and are thus named *open*, *read*, and so on.

We can consider the file to be an “object” and the functions operating on it to be its “methods,” using object-oriented programming terminology to denote actions declared by an object to act on itself.

ssize\_t (\*write) (struct file \*, const char \*, size\_t, loff\_t \*);

Testing the Device Driver:

First we register the file by using :

insmod pseudo.ko

upload the module by:

mknod /dev/psample c xxx 0

See output by :

cat /dev/psample

write input by target:

echo "abc" > /dev/psample

Check output by:

dmesg

Remove file by:

rmmod filename

See result by:

rm /dev/psample

Step 3: Device file Creation:

The device file allows transparent communication between user-space applications and hardware.

All device files are stored in /dev directory.

Use ls command to browse the directory.

ls -l /dev/

We can create a dive file in two ways.

Manually, Automatically

Manually Creating Device File:

We can create the device file manually by using mknod.

mknod -m

Advantages:

Anyone can create the device file using this method.

You can create the device file even before loading the driver.

Automatically Creating Device File:

The automatic creation of device files can be handled with udev.

Udev is the device manager for the Linux kernel that creates/removes device nodes in the /dev directory dynamically.

Include the header file linux/device.h and linux/kdev\_t.h

Create the struct Class

Create Device with the class which is created by the above step.

Create the class:

It will create a structure under/sys/class/.

struct class \* class\_create (struct module \*owner, const char \*name);

Create Device:

This function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class.

struct device \*device\_create (struct \*class, struct device \*parent, dev\_t dev, const char \*fmt, ...)

Device Destroy:

void device\_destroy (struct class \* class, dev\_t devt);

Step-4: Buffer as pseudo device:

The Z-buffer device is a "pseudo device" in that drawing commands update buffers in memory rather than sending commands to a physical device or file.

To use the Z-buffer as the current graphics device, issue the IDL command:

pbuffer = kmalloc(MAX\_SIZE, GFP\_KERNEL);

Implement read, write operations:

A memory unit stores binary information in groups of bits called words.

 Data input lines provide the information to be stored into the memory, Data output lines carry the information out from the memory.

The control lines Read and write specifies the direction of transfer of data.

Kernel Data Structures:

Kfifo API:

The kernel FIFO implementation, kfifo, is not that widely used and Stefani Seibold would like to see that change

A kfifo is declared using the DECLARE\_KFIFO() macro which can be used inside of a struct or union declaration.

FIFOs declared with with DECLARE\_KFIFO() must be initialized using INIT\_KFIFO().

DECLARE\_KFIFO(name, size)

INIT\_KFIFO(name)

DEFINE\_KFIFO(name, size)

unsigned int kfifo\_in\_rec(struct kfifo \*fifo,

void \*from, unsigned int n, unsigned int recsize)

List implementation in Kernel:

Linked list is contained inside the node, structure of node.

there were multiple implementations of linked lists in the kernel. A single, powerful linked list implementation was needed to remove duplicate code.

The linked-list code is declared in <linux/list.h> and the data structure is simple:

struct list\_head {

struct list\_head \*next

struct list\_head \*prev;

};

A list\_head by itself is worthless; it is normally embedded inside your own structure:

struct my\_struct {

struct list\_head list;

unsigned long dog;

void \*cat;

}

IPC in Kernel:

IPC mechanisms as implemented in the Linux 2.4 kernel. It is organized into four sections.

Semaphors:

The functions described in this section implement the user level semaphore mechanisms. Note that this implementation relies on the use of kernel splinlocks and kernel semaphores. To avoid confusion, the term "kernel semaphore" will be used in reference to kernel semaphores. All other uses of the word "sempahore" will be in reference to the user level semaphores.

a semaphore is based on a variable.

binary semaphore;

normal semaphore.

Semaphore API

semaphore API is located in the include/linux/semaphore.h header file.

the semaphore mechanism is represented by the following structure.

struct semaphore {

raw\_spinlock\_t lock;

unsigned int count;

struct list\_head wait\_list;

};

in the Linux kernel. The semaphore structure consists of three fields:

lock - spinlock for a semaphore data protection;

count - amount available resources;

wait\_list - list of processes which are waiting to acquire a lock.

#define DEFINE\_SEMAPHORE(name) \

struct semaphore name = \_\_SEMAPHORE\_INITIALIZER(name, 1)

Mutex:

Mutex is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The Mutex is a locking mechanism that makes sure only one thread can acquire the Mutex at a time and enter the critical section.

wait (mutex);

…..

Critical Section

…..

signal (mutex);

Spin Locks:

The most basic primitive for locking is spinlock.

static DEFINE\_SPINLOCK(xxx\_lock);

unsigned long flags;

spin\_lock\_irqsave(&xxx\_lock, flags);

... critical section here ..

spin\_unlock\_irqrestore(&xxx\_lock, flags);

Documentation/memory-barriers.txt

(5) LOCK operations.

(6) UNLOCK operations.

reader-writer spinlocks:

If your data accesses have a very natural pattern where you usually tend

to mostly read from the shared variables, the reader-writer locks

(rw\_lock) versions of the spinlocks are sometimes useful.

rwlock\_t xxx\_lock = \_\_RW\_LOCK\_UNLOCKED(xxx\_lock);

unsigned long flags;

read\_lock\_irqsave(&xxx\_lock, flags);

read\_unlock\_irqrestore(&xxx\_lock, flags);

write\_lock\_irqsave(&xxx\_lock, flags);

write\_unlock\_irqrestore(&xxx\_lock, flags);

Wait Queue API:

A wait queue is used to wait for someone to wake you up when a certain condition is true. They must be used carefully to ensure there is no race condition. You declare a wait\_queue\_head\_t, and then processes which want to wait for that condition declare a wait\_queue\_t referring to themselves, and place that in the queue.

Declaring

You declare a wait\_queue\_head\_t using the DECLARE\_WAIT\_QUEUE\_HEAD() macro, or using the init\_waitqueue\_head() routine in your initialization code.

Step-8 : Generate Race Conditions in Pseudo Driver:

A race condition is a concurrency problem that may occur inside a critical section. A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

Two Types of Race Conditions

Race conditions can occur when two or more threads read and write the same variable according to one of these two patterns:

Read-modify-write

Check-then-act

IOCTL usage:

The **ioctl**() system call manipulates the underlying device

parameters of special files. In particular, many operating

characteristics of character special files (e.g., terminals) may

be controlled with **ioctl**() requests. The argument *fd* must be an

open file descriptor.

**#include <sys/ioctl.h>**

The second argument is a device-dependent request code. The

third argument is an untyped pointer to memory. It's

traditionally char \*argp (from the days before void \* was valid

C), and will be so named for this discussion.

An ioctl() request has encoded in it whether the argument is an

in parameter or out parameter, and the size of the argument argp

in bytes. Macros and defines used in specifying an ioctl()

request are located in the file <sys/ioctl.h>. See NOTES.

Driver Model:

The Linux Kernel Driver Model is a unification of all the disparate driver models that were previously used in the kernel.

It is intended to augment the bus-specific drivers for bridges and devices by consolidating a set of data and operations into globally accessible data structures.