Course Title: Pseudo Char Driver

Faculty: Rajesh Sola

L&T Technology Services

LTTS GLOBAL ENGINEERING ACADEMY

Agenda

- Simple Driver
- File Operations
- Kernel Data Structures
- Concurrency & IPC
- IOCTL Operations
- Appendix – Misc Device, Proc files
- Summary

Version

Version	Reviewed by	Approved by	Remarks
1.0			

Learning Outcome

- Linux Driver Model
- Registering Char Device Drivers
- Device Id, Major , Minor numbers
- Implement Driver Operations
- Data Structures in Kernel Space
- Handling multiple devices
- Concurrency & IPC in Kernel
  - Mutual Exclusion, Synchronization
- IOCTL Operations
- Misc Devices, Implementing Proc Files

Pre-Requites

- Operating System Basics – Kernel, System Calls, Process Management etc
- Concurrency – multithreading etc
- IPC Concepts and User space Programming (System Calls)
- File handling in user space using system calls – open, read, write, close
- Virtual Memory Concepts - Overview
- Understanding on general data structures, especially Linked List
- Clarity on Blocking vs Non-Blocking calls

Introduction

Major and Minor Numbers

- Major number – Associates with set/family of devices, managed by common driver typically
- Minor number – Distinguish each device
- Static or Dynamic allocation of major, minor numbers
- Device Id – Major + Minor number
- In 32 bit device id ==> 12 bit major number (MSBits) + 20 bit minor numbes (LSBits)
- Macros – MAJOR, MINOR, MAKEDEV

Userspace Interface

- Unix/Linux Philosophy - Everything is a file
- Device special files in /dev directory, managed by devtmpfs mount
- Creation of device file – Manual / APIs
  - mknod command (manual) , device_create API
- Checking attributes of device files – ls, stat commands (or) lstat system call
- File operations in userspace – open, read, write, close
- Kernel configuration - CONFIG_DEVTMPS_MOUNT
- VFS Role
  - In memory i-node (in-core)
  - open file descriptor

Tutorial : Step by Step

Step-1 : Register Char Driver

```
#include <linux/fs.h>

dev_t pdevid;
int ndevices=1;

static int __init psuedo_init(void)
{
  int ret;
  ret=alloc_chrdev_region(&pdevid, 0, ndevices, "pseudo_sample");
  if(ret) {
    printk("Pseudo: Failed to register driver\n");
    return -EINVAL;
  }
  printk("Successfully registered,major=%d,minor=%d\n",
                                MAJOR(pdevid), MINOR(pdevid));
  printk("Pseudo Driver Sample..welcome\n");
  return 0;
}
static void __exit psuedo_exit(void) {
  unregister_chrdev_region(pdevid, ndevices);
  printk("Pseudo Driver Sample..Bye\n");
}
```

Observer assigned major no.

cat /proc/devices
dmesg

Step-2 : Register File Operations

```
#include <linux/cdev.h>

struct cdev cdev;  //global
int ndevices=1;

struct file_operations fops =  {
  .open    = pseudo_open,
  .release = pseudo_close,
  .write   = pseudo_write,
  .read    = pseudo_read
};

//In init
cdev_init(&cdev, &fops);
kobject_set_name(&cdev.kobj,"pdevice%d",i);
ret = cdev_add(&cdev, pdevid, 1);

//In exit
cdev_del(&cdev);
```

```
Testing driver:-

insmod pseudo.ko
mknod /dev/psample c xxx 0
cat /dev/psample
echo "abc" > /dev/psample
#check dmesg on each operation

# or use open, read, write, close
# on /dev/psample

rmmod pseudo
rm /dev/psample
```

Step-2 : Dummy File Operations

```
int pseudo_open(struct inode* inode , struct file* file)
{
  printk("Pseudo--open method\n");
  return 0;
}
int pseudo_close(struct inode* inode , struct file* file)
{
  printk("Pseudo--release method\n");
  return 0;
}
ssize_t pseudo_read(struct file * file, char __user * buf , size_t size, loff_t * off)
{
  printk("Pseudo--read method\n");
  return 0;
}
ssize_t pseudo_write(struct file * file, const char __user * buf , size_t size, loff_t * off)
{
  printk("Pseudo--write method\n");
  return -ENOSPC;
}
```

dumpstack usage

Step-3 : Device file creation

```
//Additional Headers
#include <linux/device.h>

struct device *pdev; //global
struct class *pclass; //global

//In init:-

int i=0;
pclass = class_create(THIS_MODULE, "pseudo_class");
//alloc_chrdev_region, cdev_init, cdev_add
pdev = device_create(pclass, NULL, pdevid, NULL, "psample%d",i);

//In exit:-

device_destroy(pclass, pdevid);
class_destroy(pclass);
```

Observe

- device file created in /dev
- sysfs entries for class and device file

Step-4 : Buffer as pseudo device

```
//Additional Headers
#include <linux/slab.h>
#include <linux/uaccess.h>

//Global
unsigned char *pbuffer;
int rd_offset=0;
int wr_offset=0
int buflen=0;

//In init:-
pbuffer = kmalloc(MAX_SIZE, GFP_KERNEL);

//In exit:-
kfree(pbuffer);
```

Step-4 : Implement read, write operations

```
//Write method:-
if(wr_offset >= MAX_SIZE)
{
  printk("buffer is full\n");
  return -ENOSPC;
}
wcount = size;
if(wcount > MAX_SIZE - wr_offset)
  wcount = MAX_SIZE - wr_offset;      //min

ret=copy_from_user(ubuf, pbuffer + wr_offset,
                                  wcount);
if(ret)
{
  printk("copy from user failed\n");
  return -EFAULT;
}
wr_offset+=wcount;
buflen += wcount;
```

```
//Read method:-
if(buflen==0)        //wr_offset-rd_offset==0
{
  printk("buffer is empty\n");
  return 0;
}
rcount = size;
if(rcount > buflen)
  rcount = buflen;      //min of buflen, size

ret=copy_to_user(ubuf, pbuffer + rd_offset,
                                  rcount);
if(ret)
{
  printk("copy to user failed\n");
  return -EFAULT;
}
rd_offset+=rcount;
buflen -= rcount;
```

Kernel Data Structures

Kfifo API

- kfifo_init / kfifo_alloc
- kfifo_free
- kfifo_in
- kfifo_out
- kfifo_len
- kfifo_avail
- kfifo_reset
- kfifo_isempty
- kfifo_is_full

```
//Prototypes in kfifo.h
struct __kfifo {
  unsigned int   in;
  unsigned int   out;
  unsigned int   mask;
  unsigned int   esize;
  void           *data;
};

//Definitions in lib/kfifo.c
```

Step 5 : kfifo usage in Pseudo Driver

```
#include<linux/kfifo.h>

unsigned char *pbuffer;
#define MAX_SIZE 1024

struct kfifo kfifo;

//pseudo_init
pbuffer=kmalloc(MAX_SIZE, GFP_KERNEL);
kfifo_init(&kfifo, pbuffer);
//kfifo_alloc(&kfifo, MAX_SIZE, GFP_KERNEL);

//pseudo_exit
Kfifo_free(kfifo);
```

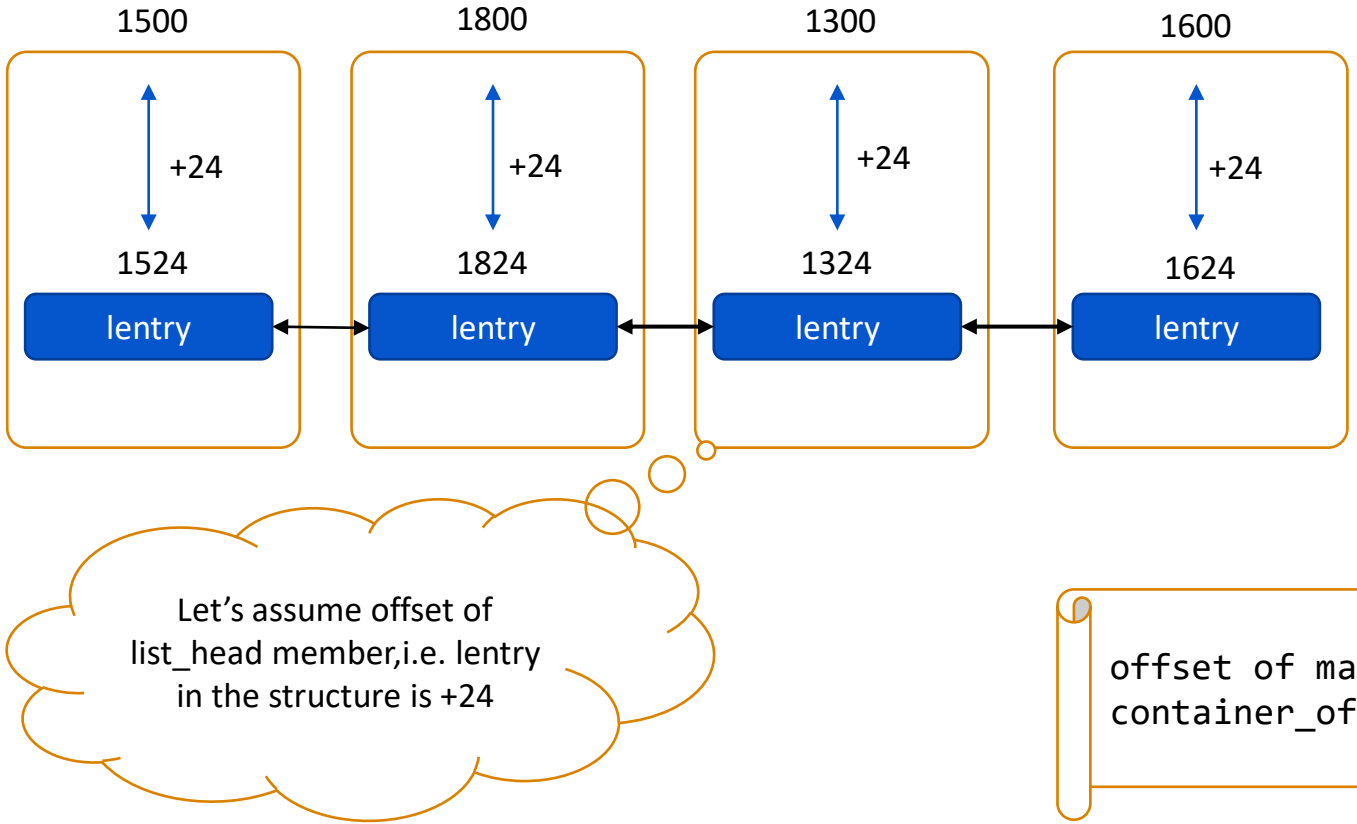
No need to call kfree(pbuffer) in case of kfifo_init kfifo_free will internally release slab memory assigned to underlying buffer

Step-5 : kfifo usage in pseudo driver

```
//Write method:-
if(kfifo_is_full(&myfifo))
{
  printk("buffer is full\n");
  return -ENOSPC;
}
wcount = size;
if(wcount > kfifo_avail(&myfifo);)
  wcount = kfifo_avail(&myfifo);      //min

char *tbuf=kmalloc(wcount, GFP_KERNEL);

ret=copy_from_user(ubuf, tbuf, wcount);
//error handling if copy_form_user
kfifo_in(&myfifo, tbuf, wcount);


kfree(tbuf);
```

```
//Read method:-
if(kfifo_is_empty(&myfifo)) {
  printk("buffer is empty\n");
  return 0;
}

rcount = size;
if(rcount > kfifo_len(&myfifo))
  rcount = kfifo_len(&myfifo);      //min


tbuf = kmalloc(rcount, GFP_KERNEL);
kfifo_out(&myfifo, tbuf, rcount);

ret=copy_to_user(ubuf, tbuf);
//error handling

kfree(tbuf);
```

List implementation in Kernel



List APIs

- LIST_HEAD
- list_add_tail
- list_del
- list_for_each
- list_for_each_safe
- list_for_each_entry
- list_for_each_entry_safe
- list_entry

```
LIST_HEAD(mylist);
```

```
sample_t *pnew = kmalloc(sizeof(sample_t), GFP_KERNEL);
//fill other members of pnew
mylist_addd(&new->lentry, &mylist);
```

List APIs

```
struct list_head *pcur;
list_for_each(pcur, &mylist) {
  ptr = list_entry(pcur, descriptor_t, lentry);
  //access other members of ptr
}
```

```
struct list_head *pcur, *prev;
list_for_each_safe(pcur, prev, &mylist) {
  ptr = list_entry(pcur, descriptor_t, lentry);
  kfree(ptr);
}
```

```
sample_t *ptr;
list_for_each_entry(ptr, &mylist, lentry) {
  //access other members of ptr
}
```

```
sample_t *ptr, *qtr;
list_for_each_entry_safe(ptr, qtr,&mylist,lentry)
{
  kfree(ptr);
}
```

Step-6 : Private Object in Pseudo Driver

```
typedef struct priv_obj {
  struct cdev cdev;
  struct kfifo kfifo;
  strcut device* pdev;
}PRIV_OBJ;

PRIV_OBJ* pobj;

//init:-
pobj=kmalloc(sizeof(PRIV_OBJ), GFP_KERNEL);
//pobj->cdev,pobj->kfifo, pobj->pdev

//exit:-
//pobj->cdev,pobj->kfifo, pobj->pdev
kfree(pobj);
```

In read, write access kfifo as pdev->kfifo

Step-7 : list API usage in Pseudo Driver

```
typedef struct priv_obj {
  struct cdev cdev;
  struct kfifo kfifo;
  strcut device* pdev;
  struct list_head *lentry;
}PRIV_OBJ;

LIST_HEAD(mydevlist);

//init:-
PRIV_OBJ *pobj;          //local, not global this time
pobj=kmalloc(sizeof(PRIV_OBJ), GFP_KERNEL);
//pobj->cdev,pobj->kfifo, pobj->pdev

list_add_tail(&pobj->lentry, &mydevlist);

//exit:-
PRIV_OBJ *ptr, *qtr;
list_for_each_entry_safe(ptr, qtr,&mydevlist,lentry) {
  kfree(ptr);
}
```

Step-7 : Handling multiple Devices

```
//pseudo_open
PRIV_OBJ *pobj = container_of(inode->i_cdev, PRIV_OBJ, cdev);
file->private_data=pobj;
```

```
//pseudo_read, pseudo_write
PRIV_OBJ *pobj = file->private_data;
//access kfifo as pobj->kfifo
```

Write multithread userspace application, where each thread will communication with different devices

IPC in Kernel

Concurrency & IPC Support in Linux Kernel

Concurrency

- Kernel Threads

Locking & Synchronization

- Semaphores
- Mutex
- Spinlocks
- Atomic Operations
- Wait queues

Kernel Threads

```
//Global
static struct task_struct *task1;
static struct task_struct *task2;


//tdemo_init:-
task1=kthread_run(thread_one, NULL, "thread_A");
//kthread_create + wake_up_process
task2=kthread_run(thread_two, NULL, "thread_B");



//tdemo_exit:-
if(task1)
  kthread_stop(task1);
if(task2)
  kthread_stop(task2);
```

```
static int thread_one(void *pargs){
  int i;
  while(!kthread_should_stop())
  {
    printk("Thread A--%d\n",k++);
    msleep(1000);  //ssleep, usleep
  }
  do_exit(0);
  return 0;
}
```

```
//Similarly write code for thread_two
```

Signal Handling

```
static int thread_one(void *pargs){
  int i;
  allow_signal(SIGKILL);
  while(!kthread_should_stop())
  {
    printk("Thread A--%d\n",k++);
    if (signal_pending(task1))
      break;
    msleep(1000);  //ssleep, usleep
  }
  do_exit(0);
  return 0;
}
```

- Do kernel threads will have task_struct and thread_info structures
- What about address space of kernel threads, do mm field in task_struct applicable for them?
- Can you observe kernel threads from userspace using utilities like ps command
- Send SIGKILL to running kernel thread and observer the behavior

Race Conditions & Concurrent Scenarios

```
int val=100;
const int max=10000;
static int thread_one(void *pargs) {
  int i;
  for(i=1;i<=max;i++)
  {
    val++;
    if(kthread_should_stop())
      break;
  }
  do_exit(0);
  return 0;
}

//Similarly write thread_two with val--
```

Scenario-1

```
int val=100;
const int max=20;
static int thread_one(void *pargs) {
  int i;
  for(i=1;i<=max;i++)
  {
     printk("Thread A--%d\n",k++);
     if(kthread_should_stop())
       break;
     msleep(100);
  }
  do_exit(0);
  return 0;
}

//Similarly write thread_two which
//prints max times with some delay.
```

Scenario-2

Semaphore API

- Structsemaphore
- sema_init
- DEFINE_SEMAPHORE
- down_interruptible
- up

- ❑ Locate these APIs in kernel source – prototype and definition
- ❑ Identify other Semaphore APIs
- ❑ Apply semaphore operations for Scenario-1 to avoid race conditions
- ❑ Apply semaphore operations for Scenario-2 to ensure only one for loop (any) executes at a time

Mutex API

- struct mutex
- mutex_init
- DEFINE_MUTEX
- mutex_lock
- mutex_unlock

- ❑ Locate these APIs in kernel source – prototype and definition
- ❑ Identify other Mutex APIs, especially trylock, recursive locks
- ❑ Apply Mutex operations for Scenario-1 to avoid race conditions
- ❑ Apply Mutex operations for Scenario-2 to ensure only one for loop (any) executes at a time

Mutex vs Semaphore, Salient features of Mutex!!

Spinlock API

- spin_lock_init
- DEFINE_SPINLOCK
- spin_lock
- spin_unlock

- ❑ Locate these APIs in kernel source – prototype and definition
- ❑ Identify other Spinlock APIs, especially trylock, recursive locks
- ❑ Apply Spinlock operations for kernel threads
- ❑ Significant usage of spinlocks in Interrupt Handling, Bottom Halves(TBD Later)

Spinlock vs Semaphore/Mutex, when to use which??

Context switching vs Busy Waiting/Spinning?

Wait Queue API

- struct wait_queue_head / wait_queue_head_t
- DEFINE_SPINLOCK
- init_waitqueue_head
- wait_event_interruptible
- wake_up_interruptible

- ❑ Locate these APIs in kernel source – prototype and definition and refer kernel docs
- ❑ Identify other waitqueue APIs,
- ❑ Other IPC techniques built on top of waitqueues
- ❑ Apply Waitqueue operations in Scenario-2, to ensure that always for loop of one kernel thread runs before other

Step-8 : Generate Race Conditions in Pseudo Driver

```
// Consider Step-4 code as baseline, using global buffer as Pseudo
device

//pseudo_write:-
for(i=0;i<wcount;i++) {
  get_user(*(pbuffer+wr_offset),ubuf);
  ++wr_offset;
  ubuf++;
  msleep(100);    //intentional delay to generate race conditions
}
```

In userspace, use two threads to perform concurrent operations on this driver, e.g. one thread may write AAA….A and other thread may write BBB…B

Step-8 : Mutual Exclusion in Pseudo Driver

```
//pseudo_write:-
down_interruptible(&s1); //mutex_lock
for(i=0;i<wcount;i++) {
  get_user(*(pbuffer+wr_offset),ubuf);
  ++wr_offset;
  ubuf++;
  msleep(100);
}
up(&s1);                    //mutex_unlock
```

- When write operation is in progress on behalf of one thread, no other write operation (writing to buffer part) can takes place on behalf of other threads.

Step-9 : Synchronization in Pseudo Driver

```
//Additional header files
#include<linux/wait.h>

//inside private object
wait_queue_head_t rd_queue;

//init:-
init_waitqueue_head(&pobj->rd_queue);

//pseudo_read:-
if(kfifo_is_empty(&pobj->kfifo))   // //file->f_flags & O_NONBLOCK, -EAGAIN
   wait_event_interruptible(pobj->rd_queue,(kfifo_len(&pobj->kfifo)>0)); //!kfifo_is_empty
   //instead of returning 0, if O_NONBLOCK is not requested from userspace

//pseudo_write:-
wake_up_interruptible(&pobj->rd_queue);   //on writing some data into kfifo
```

Read method blocks when kfifo is empty and unlocks when some data is written to kfifo by write operation

Userspace test code to check synchronization

Step-9 : Synchronization in Pseudo Driver

```
//Additional header files
#include<linux/wait.h>

//inside private object
wait_queue_head_t wr_queue;

//init:-
init_waitqueue_head(&pobj->wr_queue);

//pseudo_write:-
if(kfifo_is_full(&pobj->kfifo))   // //file->f_flags & O_NONBLOCK, -EAGAIN
   wait_event_interruptible(pobj->wr_queue,(kfifo_avail(&pobj->kfifo)>0)); //!kfifo_is_full
   //instead of returning -ENOSPC, if O_NONBLOCK is not requested from userspace

//pseudo_read:-
wake_up_interruptible(&pobj->wr_queue);   //on reading some data from kfifo
```

Write method blocks when kfifo is full and unlocks when some data is retrieved from kfifo by read operation

Userspace test code to check synchronization

IOCTL usage

Step-10 : ioctl

```
#define IOC_MAGIC  'p'
#define MY_IOCTL_LEN          _IO(IOC_MAGIC, 1)
#define MY_IOCTL_AVAIL        _IO(IOC_MAGIC, 2)
#define MY_IOCTL_RESET        _IO(IOC_MAGIC, 3)
```

```
struct file_operations fops =  {
  .open     = pseudo_open,
  .release = pseudo_close,
  .write    = pseudo_write,
  .read     = pseudo_read,
  .unlocked_ioctl    = pseudo_ioctl
};
```

```
static long pseudo_ioctl(struct file *file,
unsigned int cmd, unsigned long arg)
{
  int ret;
  printk("Pseudo--ioctl method\n");
  switch (cmd) {
    case MY_IOCTL_LEN :
        printk("ioctl--kfifo length is %d\n",
                kfifo_len(&myfifo));
        break;
    case MY_IOCTL_AVAIL:
        printk("ioctl--kfifo avail is %d\n",
        kfifo_avail(&myfifo));
        break;
    case MY_IOCTL_RESET:
        printk("ioctl--kfifo got reset\n");
        kfifo_reset(&myfifo);
        break;
  }
  return 0;
}
```

Userspace code

```
//similar header file with magic numbers in
userspace

#define IOC_MAGIC  'p'
#define MY_IOCTL_LEN        _IO(IOC_MAGIC, 1)
#define MY_IOCTL_AVAIL      _IO(IOC_MAGIC, 2)
#define MY_IOCTL_RESET      _IO(IOC_MAGIC, 3)
```

```
fd = open("/dev/psample", O_RDWR);
if(fd<0) {
  perror("open");
  exit(1);
}
ret=ioctl(fd, MY_IOCTL_LEN);
if(ret<0)   {
  perror("ioctl");
  exit(3);
}
ret=ioctl(fd, MY_IOCTL_AVAIL);
if(ret<0)   {
  perror("ioctl");
  exit(3);
}
ret=ioctl(fd, MY_IOCTL_RESET);
if(ret<0) {
  perror("ioctl");
  exit(3);
}
close(fd);
```

Step 10 : Another IOCTL Operation

```
struct pseudo_stat {
  int len;
  int avail;
};

#define MY_IOCTL_PSTAT       _IOR(IOC_MAGIC, 4, struct pseudo_stat)
```

```
static long pseudo_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
  switch(cmd) {
    case MY_IOCTL_PSTAT:
      printk("ioctl--kfifo statistrics\n");
      stat.len=kfifo_len(&myfifo);
      stat.avail=kfifo_avail(&myfifo);
      ret=copy_to_user( (char __user*)arg, &stat, sizeof(pseudo_stat));
      if(ret) {
        printk("error in copy_to_user\n");
        return -EFAULT;
      }
      break;
  }
  return 0;
}
```

Userspace Test Code for IOCTL

```
//open
struct pseudo_stat stat;
ret=ioctl(fd, MY_IOCTL_PSTAT, &stat);
if(ret<0) {
  perror("ioctl");
  exit(4);
}
//print stat.len, stat.avail
//close
```

Appendix

Miscellaneous Devices

```
#include <linux/miscdevice.h>

static struct file_operations fops = {
  .open    = pseudo_open,
  .release = pseudo_close,
  .read    = pseudo_read,
  .write   = pseudo_write,
};
static struct miscdevice pseudo_char_misc = {
  .minor = MISC_DYNAMIC_MINOR,
  .name  = "pseudo_misc_char",
  .fops  = &fops,
};

//pseudo_misc_init
ret = misc_register(&pseudo_char_misc);

//pseudo_misc_exit
misc_deregister(&pseudo_char_misc);
```

Ref:-
https://www.kernel.org/doc/html/v4.14/driver-api/misc_devices.html
https://github.com/opersys/circular-driver

Simple Proc File

```
static struct proc_dir_entry *pentry;
static struct proc_dir_entry *pdir;

struct file_operations fops={
  .open    = pdemo_open,
  .release = pdemo_close,
  .read    = pdemo_read,
  .write   = pdemo_write,
};

//pdemo_init:-
pdir=proc_mkdir("ptest",NULL);
//if(pdir==NULL) …
pentry=proc_create("psample",0666,pdir,&fops);
//if(pentry==NULL) …
proc_set_user(pentry, KUIDT_INIT(0), KGIDT_INIT(0));
proc_set_size(pentry,80);

//pdemo_exit:-
remove_proc_entry("psample",pdir);
remove_proc_entry("ptest",NULL);
```

- Implement empty open, read, write, close operations, like Step-2 of Pseudo driver

Sequential Proc File – single_open

```
static struct proc_dir_entry *pentry;
static struct proc_dir_entry *pdir;

ssize_t custom_proc_show(struct seq_file* m, void* p)
{
  seq_printf(m,"psingle--custom show,dummy_var=%d\n",100);
  seq_printf(m,"current pid=%d\n",current->pid);
  return 0;
}

int pdemo_open (struct inode * inode, struct file * file)
{
  return single_open(file,custom_proc_show,NULL);
}

struct file_operations fops={
  .open    = pdemo_open,
  .release = single_release,
  .read    = seq_read,
  .write   = seq_write,
};
```

Assignment/Coding Tasks based on Proc

- Implement simple read, write operations on Proc file
- Implement a proc file which does simple echo operation, i.e. a string written to proc file, same will be read back. You may consider any other string operation like reverse, toggle the case.
- Implement a proc file, on read operation print all possible attributes of invoking process (like system call task given earlier)
- Implement a proc file, which prints pid, ppid of all processes on read operation (like system call task given earlier)
- Implement a proc file, which prints minimal information about available CPUs (minimal version of /proc/cpuinfo), Hint:- for_each_cpu / cputype.h / read_cupid
- Implement a proc file as follows
  - init method – append n entries to list
  - read method – traverse the list and print (export data to userspace)
  - write method – append one more element to list
  - exit method – traverse and delete list nodes

Thank You !

L&T Technology Services