Storage Systems

NPTEL Course

Jan 2012

(Lecture 17)

K. Gopinath

Indian Institute of Science

Accessing Devices/Device Driver

- Many ways to access devices under linux
  - Non-block based devices (“char”) - stream like and control ops
  - Block based devices - Accessed in units of blocks and system caches blocks, FS service provided on top
  - Network devices - mostly accessed in kernel mode
- Devices identified by major, minor numbers
  - Major number used to identify the driver
  - Minor usually treated by driver as a unit identifier
- Kernel expects a non-block device driver to provide following ops
  - Open/Close - Initialization, perms, reference holds
  - Reads/Writes - Do i/o on the device
  - Ioctl - Do some control ops on the device
  - Mmap - To set vma ops on a vma mapping to file
    - vma: virtual memory area

- A character device driver registers itself with
  - register_chrdev(major, "str", ops)
  - unregister_chrdev(major,"str")
- Devices are accessed by user mode programs by accessing special device files
  - Their inode says they are special device files
  - Have major, minor, char/block info
- On open, open method called; on close close() ...
  - Open called only on file object creation, not on dups
  - Close called only on final file object deletion, not on all closes
- Read and write pass pointer to file object which contains current offset
  - Do whatever "offset" means to the device
- lseek - check if offset is legal
- Mmap - check if mapping is legal and set mops

```c
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/vmalloc.h>
#include <linux/string.h>
#include <asm/uaccess.h>
#include <linux/errno.h>
#include "intevts.h"

struct event_t *evtbuf,*nextevt,*lastevt;
int recording=0;
spinlock_t evtbuf_lk;

extern void (*penter_irq)(int irq,int cpu);
extern void (*pleave_irq)(int irq,int cpu);

ssize_t ints_read(struct file *, char *, size_t, loff_t *);
ssize_t ints_write(struct file *, const char *, size_t, loff_t *);
int     ints_open(struct inode *, struct file *);
int     ints_release(struct inode *, struct file *);

static struct file_operations ints_fops = {
        read:       ints_read,
        write:      ints_write,
        open:       ints_open,
        release:    ints_release,
};
```

```c
void enter_irq(int irq,int cpu) {
        int flags;
        spin_lock_irqsave(&evtbuf_lk,flags);
        if(recording && nextevt!=lastevt) {
                rdtscll(nextevt->time);
                nextevt->event=
                        MKEVENT(irq,E_ENTER);
                nextevt->cpu=cpu;
                nextevt++;
        }
        spin_unlock_irqrestore(&evtbuf_lk,flags);
}
void leave_irq(int irq,int cpu) {
        int flags;
        spin_lock_irqsave(&evtbuf_lk,flags);
        if(recording && nextevt!=lastevt) {
                rdtscll(nextevt->time);
                nextevt->event=
                        MKEVENT(irq,E_LEAVE);
                nextevt->cpu=cpu;
                nextevt++;
        }
        spin_unlock_irqrestore(&evtbuf_lk,flags);
}
```

Device Driver Examples

- Pseudo Device Driver “ints”
  - Collects a trace of interrupts in each CPU
    - For each interrupt, trace has time of interrupt, cpu interrupted, interrupt enter or leave, interrupt number
  - When “ints” module loaded, read/write/open/close... ops of device mapped to driver routines. Also,
  - User first writes to /dev/ints to set number of trace entries using the cmd line, and starts trace recording in an internal kernel buffer
  - Next, using the cmd line, user runs a program that reads each entry and prints each trace
- Block Device Driver

```
                evtbuf=(struct event_t *)
    vmalloc(nrents*sizeof(struct event_t));
                if(!evtbuf)
                        return -ENOMEM;
                nextevt=evtbuf;
                lastevt=evtbuf+nrents;

                cli();
                recording=1;
                sti();
                break;
        default: return -EINVAL;
    }

    (*poff)+=size;
    return size;
}

int     ints_open(struct inode *inode, struct file *file)
{ return 0; }

int     ints_release(struct inode *node, struct file *file)
{ return 0; }
```

```
int init_module(void) {

        spin_lock_init(&evtbuf_lk);

        cli();

        penter_irq=enter_irq;

        pleave_irq=leave_irq;

        sti();

        register_chrdev(233, "ints", &ints_fops);

        return 0;

}

void cleanup_module(void) {

        if(recording) {

                cli();

                recording=0;

                sti();

        }



        if(evtbuf) vfree(evtbuf);

        penter_irq=0;

        pleave_irq=0;

        unregister_chrdev(666,"ints");

}
```

Interrupts

- Interrupts can interrupt anytime
  - When cpu is normal user mode (timer ticks)
  - Cpu is in kernel for syscall, or syscall+fault
- An interrupt execution on a CPU can be
  - Interrupted by another different interrupt
  - Or run to completion
- Can raise a soft interrupt
- Process context is resumed only when all interrupts exit
- Interrupt controller + cpu gives the abstraction of irqs, which go out on the bus and can be raised
- Each device driver sets an irq handler and programs the device to interrupt at that irq
- request_irq/free_irq are the functions for irq handler registering

- Irqs can get routed to any processor, and usually the interrupt controller balances this load
- An irq will run only on one processor at a time.
  - Different irqs can run concurrently
  - Irqs can serialize with other processors with spinlocks
- disable_irq(irq) disables this irq only and returns only after this irq has completed if executing
- cli/sti disables/sets interrupts on all processors and returns only after all currently executing interrupts are completed on all processors

- Processes and interrupts are concurrent
  - Need mutual exclusion and synchronization
- Spin locks allow only one processor to own a critical section
  - Interrupts blocked on local cpu, others spin
- Waiting for events - synchronization
  - We usually sleep till event occurs
  - Sleep locks, condition variables or semaphores
  - Interruptible vs Non-interruptible sleep
  - Mutex serves synch as well in some cases

Access to user memory in driver

- User memory(buf ptr) to be touched only with
  - get_user/put_user, copy_from/to_user
  - Does kernel/user address space check
  - "traps" faults so that -EFAULT can be returned
  - Can sleep, so must be concurrent safe even on a single processor
- Use space on stack carefully, only <8K
  - No recursion, no huge arrays, call by val struct

```
ssize_t ints_read(struct file *file, char *buf, size_t size, loff_t *poff) {
        int bufsize;
        if(!evtbuf) return -EINVAL;
        if(recording) {
                cli();
                recording=0;
                sti();
        }
        bufsize=MIN(sizeof(struct event_t)*
                        (nextevt-evtbuf)-*poff, size);
        if(bufsize) {
                if(copy_to_user(buf,((char *)evtbuf)
                                        +*poff, bufsize))
                        return -EFAULT;
        }
        (*poff)+=bufsize;
        return bufsize;
}
ssize_t ints_write(struct file *file, const char *buf, size_t size, loff_t
          *poff) {
        char c;
        char kbuf[32];
        int ret,nrents,strsize;
if(get_user(c,buf) || size<2)
        return -EFAULT;
switch(c) {
        case 's':
        case 'S':
                strsize=MIN(sizeof(kbuf),size-1);
                if(copy_from_user(kbuf,buf+1,strsize))
                        return -EFAULT;
                kbuf[strsize]=0;
                ret=sscanf(kbuf,"%d",&nrents);
                if(ret!=1 || !nrents)
                        return -EINVAL;
                if(recording) {
                        cli();
                        recording=0;
                        sti();
                }
                if(evtbuf) {
                        vfree(evtbuf);
                        evtbuf=0;
                }
```

User code

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <assert.h>
#include <stdlib.h>
#include "intevts.h"

#define rdtscll(val) \
    __asm__ __volatile__("rdtsc" : "=A" (val))
#define STACKNR         32

int fd1,fd2;
int cpustack[2][STACKNR],stktop[2]={-1,-1};
unsigned long long cpustart[2];

void push(int cpu,int val) {
        assert(stktop[cpu]<STACKNR-1);
        cpustack[cpu][++stktop[cpu]]=val;
}

int pop(int cpu) {
        assert(stktop[cpu]>=0);
        return cpustack[cpu][stktop[cpu]--];
}

int peek(int cpu) {

        assert(stktop[cpu]>=0);

        return cpustack[cpu][stktop[cpu]];

}

void die(char *func) {

        perror(func);

        exit -1;

}

void closefiles(void) {

        close(fd1);

        close(fd2);

}

int initfiles(void) {

        fd1=open("out.cpu1",O_TRUNC|O_CREAT|O_WRONLY,0666);

        if(fd1<0)  die("open");

        fd2=open("out.cpu2",O_TRUNC|O_CREAT|O_WRONLY,0666);

        if(fd2<0) die("open");

}
```

```c
void output(int cpu,long long x, int y) {
      char buffer[32];
      sprintf(buffer,"%lld %d\n",x,y+(cpu?0:20));
       if(cpu) write(fd1,buffer,strlen(buffer));
       else write(fd2,buffer,strlen(buffer));
}

main() {
       int ret,fd=0;
       struct event_t e;
       if(!initfiles()) return -1;

       push(0,-1);
       push(1,-1);

     while((ret=read(fd,&e,sizeof(e)))==sizeof(e)) {
     printf("Event record:%lld,%d - [%s,%d]\n",
     e.time, e.cpu,
     EVTTYP(e.event)==0 ? "E_ENTER":"E_LEAVE",
     EVTNR(e.event));
     assert(e.cpu==1||e.cpu==0);
            if(!cpustart[e.cpu]) {
                  cpustart[e.cpu]=e.time;
                  e.time=0;
            }
            else {  e.time=e.time-cpustart[e.cpu];
                    e.time=e.time*1000*1000/
                              (1263*1000*1000);
            }
            if(EVTTYP(e.event)==E_ENTER) {
                  output(e.cpu,e.time,peek(e.cpu));
                  output(e.cpu,e.time,EVTNR(e.event));
                  push(e.cpu,EVTNR(e.event));
            }
            else if(EVTTYP(e.event)==E_LEAVE) {
                  assert(EVTNR(e.event)==peek(e.cpu));
                  output(e.cpu,e.time,peek(e.cpu));
                  pop(e.cpu);
                  output(e.cpu,e.time,peek(e.cpu));
            }
            else assert(0);
      }
      closefiles();
}
```