Storage Systems

NPTEL Course

Jan 2012

(Lecture 18)

K. Gopinath

Indian Institute of Science

Spinlocks & Semaphores

- Shared data betw different parts of code in kernel
  - most common: access to data structures shared between user process context and interrupt context
- In uniprocessor system: mutual excl by setting and clearing interrupts + flags
- SMP: three types of spinlocks: vanilla (basic), read-write, big-reader
  - Read-write spinlocks when many readers and few writers
    - Eg: access to the list of registered filesystems.
  - Big-reader spinlocks a form of read-write spinlocks optimized for very light read access, with penalty for writes
    - limited number of big-reader spinlocks users.
    - used in networking part of the kernel.
- semaphores: Two types of semaphores: basic and read-write semaphores. Different from IPC's
  - Mutex or counting up()& down(); interruptible/ non

Spinlocks: (cont'd)

- A good example of using spinlocks: accessing a data strucuture shared betw a user context and an interrupt handler

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
my_ioctl() {                              // _ioctl: definitely process context!
    spin_lock_irq(&my_lock);    //  and known that interrupts enabled!
    /* critical section */              //  hence, _irq to disable iinterrupts
    spin_unlock_irq(&my_lock);
}
my_irq_handler() {                  // _irq_handler: definitely system (or intr context)
    spin_lock(&lock);           // & hence known that intr disabled!
    /* critical section */         // can use simpler lock
    spin_unlock(&lock);
}
```

  spin_lock: if interrupts disabled or no race with interrupt context

  spin_lock_irq: if interrupts enabled and has to be disabled

  spin_lock_irqsave: if interrupt state not known

- Basic premise of a spin lock: one thread busy-waits on a resource on one processor while another used on another (only true for MP). But code has to work for 1 or more processors. If all threads on 1 processor, if a thread tries to spin lock that is already held by another thread, deadlock.
- Never give up CPU when holding a spinlock!

Top/Bottom Halves

- Top half routines: synch
  - execute in process context
  - may access AS/u-area of calling process
  - may put process to sleep
- Bottom half routines: asynch
  - execute in system context
  - no relation to curr process
  - may not sleep/access AS/u-area of curr proc
- If top half routine is running, has to block interrupts to prevent bottom routines seeing inconsistent data structures

Linux Concurrency Model

- Within appl: clones (incl threads & processes of other syst)
- Inside kernel:
  - Kernel threads: do not have USER context
  - deferrable and interruptible ker funcs:
    - Softirq: reentrant: multiple softirqs of the same type can be run concurrently on several CPUs.
      - No dyn alloc! Have to be statically defined at compile time.
    - Tasklet: multiple tasklets of the same type cannot run concurrently on several CPUs.
      - Dyn alloc OK! Can be allocated and initialized at run time (loadable modules). Impl thru softirqs
    - Bottom Half: multiple bottom halves cannot be run concurrently on several CPUs. No dyn alloc!
      - Impl thru tasklets
- Across HW: IPI

```
#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,
};
```

```
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);
}
```

```
                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");

}
```

```
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");

}
```

```
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();
}
```

Block Devices

- Wide variety
  - Disks, tape, ...
- Drivers provide functionality similar to
  - Open: may bring dev on-line or init data struct
    - set flag for excl use
  - Close
  - Size: determine size of partition
  - Strategy: (bottom half)
    - may reorder requests
    - operates asynch: if dev busy, just Qs it or if process is trying to alloc a free buf, it may have to flush a dirty buf 1st on free list. After write initiated, no further interest
  - Halt: (bottom half)
    - called during shutdown/unloading driver
    - no user context, no interrupts? cannot sleep
- But no read/write routines!
  - Handled by strategy
- Provides support to buffer (“caching”) routines

Buffer Allocation Algs

- getblk: given a device and block number, get the buffer for it locked
- brelse: given a locked buffer, wakeup waiting procs and unlock it
- bread: read a given block into a buffer
- breada: bread + asynch. read ahead
- bwrite: write a given buffer to a block
- Buffer properties
  - No block in 2 different buffers
  - Can be in free list or hash list: Search free list if any buffer needed; hash list if a particular buffer needed
  - Buf alloc safe: allocs during syscall & frees at end
    - Disk drive hw problem: cannot interrupt CPU: buf lost!
    - But no starvation guarantees

Block Layer

- Device Drivers for block devices can use block layer infrastructure
  - Common functionality across all block devices
- Fairly complex to get it right!