Storage Systems

NPTEL Course

Jan 2012

(Lecture 16)

K. Gopinath

Indian Institute of Science

A Common System Interface: Posix 1

- *access* Tests for file accessibility
- *chdir* Changes current working directory
- *chmod* Changes file mode
- *chown* Changes owner and/or group of a file
- *close* Closes a file
- *closedir* Ends directory read operation
- *creat* Creates a new file or rewrites existing one
- *dup* Duplicates an open file descriptor
- *dup2* Duplicates an open file descriptor
- *execl* Executes a file
- *execle* Executes a file
- *execlp* Executes a file
- *execv* Executes a file
- *execve* Executes a file
- *execvp* Executes a file
- *_exit* Terminates a process
- *fcntl* Manipulates an open file descriptor
- *fdopen* Opens a stream on a file descriptor
- *fork* Creates a process
- *fpathconf* Gets config variable for an open file
- *fstat* Gets file status
- *getcwd* Gets current working directory
- *link* Creates a link to a file
- *lseek* Repositions read/write file offset
- *mkdir* Makes a directory
- *mkfifo* Makes a FIFO special file
- *open* Opens a file
- *opendir* Opens a directory
- *pathconf* Gets config variables for a path
- *pipe* Creates an interprocess channel
- *read* Reads from a file
- *readdir* Reads a directory
- *rename* Renames a file
- *rewinddir* Resets the readdir() pointer
- *rmdir* Removes a directory
- *stat* Gets information about a file
- *umask* Sets the file creation mask
- *unlink* Removes a directory entry
- *utime* Sets file access & modification times
- *write* Writes to a file

POSIX.1b

- *aio_cancel* Tries to cancel an asynchronous op
- *aio_error* Retrieves error status for an asynchronous op
- *aio_read* Asynchronously reads from a file
- *aio_return* Retrieves return status for an asynchronous op
- *aio_suspend* Waits for an asynchronous op to complete
- *aio_write* Asynchronously writes to a file
- *fdatasync* Synchronizes at least the data part of a file with the underlying media
- *fsync* Synchronizes a file with underlying media
- *lio_listio* Performs a list of I/O operations, synchronously or asynchronously

- *mlock* Locks a range of memory
- *mlockall* Locks the entire memory space down
- *mmap* Maps a shared memory object (or possibly another file) into process's addr space
- *mprotect* Changes memory protection on a mapped area
- *msync* Makes a mapping consistent with the underlying object
- *munlock* Unlocks a range of memory
- *munlockall* Unlocks the entire address space
- *munmap* Undo mapping established by mmap

Device Driver

- With each physical device, device driver code manages device hardware
  - brings device into and out of service,
  - sets hardware parameters in the device,
  - transmits data from the kernel to the device,
  - receives data from the device and passes it back to the kernel, and
  - handles device errors

- Diffs betw application programs versus drivers:
  - no main for drivers: driver routines called in response to system calls or other requirements. Switch tables contain starting addresses for principal routines included in all drivers.
  - parallel execution: a driver may receive a request to write data to a disk while waiting for a previous request to complet
    - no new version of driver (and its data structures) for each process => must anticipate & handle contention problems resulting from overlapping I/O reqs processing needed to handle hardware interrupts
  - inefficient driver code can severely degrade overall perf, and driver errors can corrupt or bring down system.

Device specificity

- mem-mapped I/O or I/O space
  - i/o space = all dev regs + frame buffers for mem mapped devices
  - each reg has a well-defined addr that is assigned at boot time using config files used to build system
  - sys may assign a range of addrs to each controller and it may in turn assign it to various devices under it
- programmed I/O (PIO: modems, char terminals, line printers) or DMA (disks/graphics terminals) or DVMA (interacts thru MMU to xfer data to device without going thru mem)

Device interrupts

- kernel code at ipl=0
- if arriving interrupt's ipl <= current ipl of system, blocked
- for each device, a fixed ipl
  - all devices on a contoller typ have same ipl
- some kernel routines incr ipl to block certain interrupts
  - manipulation of disk buffer q => blocks disk interrupts
- set ipl to device's interrupt level
  - while saving current value
  - Use previously saved value when exiting handler
- In some OS, blockable kernel threads instead of ipl

- typ all interrupts invoke a common routine in kernel with some information to identify interrupt
  - saves context, raises ipl to that of interrupt, calls handler; on return, restores context, ret
- identification of interrupt: vectoring? or polling?
  - completely vectored: each device provides interrupt vector # for index
  - only ipl may be available: search linked list of handlers with same ipl
  - vectored may also support linked list of handlers:
    - can add dyn loadable dev drivers on a running system
    - "override" drivers in front of list: trap/handle certain interrupts, rest to default driver
- handler: most important part of system (runs in priority > kernel code)
  - has to be quick and not sleep; also do enough work so that device not idle
  - initiate next I/O req pending before exit

Device Driver

- Monolithic kernel vs microkernel
  - Interfaces different
    - Possibly a loadable module in monolithic
    - Dev driver an appl in case of microkernel
- Device driver requires kernel memory and other resources
  - Also, may be physical memory (for DMA)
  - Or, mapping of memory from/to device
- Kernel needs to issue commands to device
- DDI/DDK (device driver interface/kernel)
  - Isolate device drivers from differing versions of kernel
  - Isolate kernel from hardware details

- #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);
}
```