I/O Systems

Operating Systems

Wenbo Shen

Acknowledgement: some pages are based on the slides from Yajin Zhou@ZJU

Review

- Disk: platter, cylinder, sector, track
- Time: positioning time = seek time + rotational latency
- Disk structure, Disk attachment: host- ,network-, storage area network
- Disk scheduling: FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK
- Disk management: physical formatting, partition disk, logical formatting
- RAID: redundant array of inexpensive disk
  - RAID 0: split, RAID 1: mirror, RAID4/RAID 5/RAID 6: block with parity

Overview

Outline

- Overview
- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O Requests to Hardware Operations
- Performance
- Example: Linux tty device

Overview

- I/O management is a major component of OS design and operation
  - important aspect of computer operation
    - I/O device is the way computer to interact with user and other systems
  - I/O devices vary greatly
    - various methods to control them
    - performance varies
    - device drivers encapsulate device details; present an uniform interface
  - new types of devices frequently emerge

I/O Hardware

- Incredible variety of I/O devices
  - storage, communication, human-interface
- Common concepts: signals from I/O devices interface with computer
  - **bus**: an interconnection between components (including CPU)
  - **port**: connection point for device
  - **controller**: component that control the device
    - can be integrated to device or separate circuit board
    - usually contains processor, microcode, private memory, bus controller, etc
- I/O access can use **polling** or **interrupt**

A Typical PC Bus Structure

I/O Hardware

- Some CPU architecture has **dedicated I/O instructions**
  - e.g., x86: in, out, ins, outs
- Devices usually provide registers for data and control I/O of device
  - device driver places (pointers to) commands and data to register
  - registers include **data-in/data-out**, **status**, **control** (or command) register
  - typically 1-4 bytes, or FIFO buffer
- Devices are assigned addresses for registers or on-device memory
  - **direct I/O instructions**
    - to access (mostly) registers
  - **memory-mapped I/O**
    - data and command registers mapped to processor address space
    - to access (large) on-device memory (graphics)

I/O Ports on PCs (Partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Polling

- For each I/O operation:
  - busy-wait if device is busy (status register)
    - Cannot accept any command if busy
  - send the command to the device controller (command register)
  - read status register until it indicates command has been executed
  - read execution status, and possibly reset device status
- Polling requires busy wait
  - reasonable if device is fast; inefficient if device slow

Interrupts

- Polling requires **busy-wait**, inefficient use of CPU resource
- Interrupts can avoid busy-wait
  - device driver (part of OS) send a command to the controller (on device), and return
  - OS can schedule other activities
  - device will interrupt the processor when command has been executed
  - OS retrieves the result by handling the interrupt
- Interrupt-based I/O requires **context switch** at start and end
  - if interrupt frequency is extremely high, context switch wastes CPU time
  - solution: use polling instead
    - example: **NAPI** in Linux enables polling under very high network load

Interrupt-Driven I/O Cycle

Intel Pentium Interrupt Vector Table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Linux vector table on ARM64

- Define vector table
- Set the value to VBAR
  - Vector Base Address Register

```
/*
 * Exception vectors.
 */

        .align  11
ENTRY(vectors)
        ventry  el1_sync_invalid                // Synchronous EL1t
        ventry  el1_irq_invalid                 // IRQ EL1t
        ventry  el1_fiq_invalid                 // FIQ EL1t
        ventry  el1_error_invalid               // Error EL1t

        ventry  el1_sync                        // Synchronous EL1h
        ventry  el1_irq                         // IRQ EL1h
        ventry  el1_fiq_invalid                 // FIQ EL1h
        ventry  el1_error_invalid               // Error EL1h

        ventry  el0_sync                        // Synchronous 64-bit EL0
        ventry  el0_irq                         // IRQ 64-bit EL0
        ventry  el0_fiq_invalid                 // FIQ 64-bit EL0
        ventry  el0_error_invalid               // Error 64-bit EL0

#ifdef CONFIG_COMPAT
        ventry  el0_sync_compat                 // Synchronous 32-bit EL0
        ventry  el0_irq_compat                  // IRQ 32-bit EL0
        ventry  el0_fiq_invalid_compat          // FIQ 32-bit EL0
        ventry  el0_error_invalid_compat        // Error 32-bit EL0
#else
        ventry  el0_sync_invalid                // Synchronous 32-bit EL0
        ventry  el0_irq_invalid                 // IRQ 32-bit EL0
        ventry  el0_fiq_invalid                 // FIQ 32-bit EL0
        ventry  el0_error_invalid               // Error 32-bit EL0
#endif
END(vectors)
```

```
        .section        ".idmap.text", "ax"
__enable_mmu:
        mrs     x1, ID_AA64MMFR0_EL1
        ubfx    x2, x1, #ID_AA64MMFR0_TGRAN_SHIFT, 4
        cmp     x2, #ID_AA64MMFR0_TGRAN_SUPPORTED
        b.ne    __no_granule_support
        ldr     x5, =vectors
        msr     vbar_el1, x5
        msr     ttbr0_el1, x25                  // load TTBR0
        msr     ttbr1_el1, x26                  // load TTBR1
        isb
```

https://elixir.bootlin.com/linux/v4.4/source/arch/arm64/kernel/entry.S
https://elixir.bootlin.com/linux/v4.4/source/arch/arm64/kernel/head.S#L630

Linux vector table on RISC-V64

- Define vector table
- Set the value of handle_exception to stvec
  - Supervisor Trap Vector Base Address Register
- Handle_exception jumps to entries of exception table

```
278         /* Exception vector table */
279 SYM_CODE_START(excp_vect_table)
280         RISCV_PTR do_trap_insn_misaligned
281         ALT_INSN_FAULT(RISCV_PTR do_trap_insn_fault)
282         RISCV_PTR do_trap_insn_illegal
283         RISCV_PTR do_trap_break
284         RISCV_PTR do_trap_load_misaligned
285         RISCV_PTR do_trap_load_fault
286         RISCV_PTR do_trap_store_misaligned
287         RISCV_PTR do_trap_store_fault
288         RISCV_PTR do_trap_ecall_u /* system call */
289         RISCV_PTR do_trap_ecall_s
290         RISCV_PTR do_trap_unknown
291         RISCV_PTR do_trap_ecall_m
292         /* instruciton page fault */
293         ALT_PAGE_FAULT(RISCV_PTR do_page_fault)
294         RISCV_PTR do_page_fault   /* load page fault */
295         RISCV_PTR do_trap_unknown
296         RISCV_PTR do_page_fault   /* store page fault */
297 excp_vect_table_end:
298 SYM_CODE_END(excp_vect_table)
```

```
176 setup_trap_vector:
177         /* Set trap vector to exception handler */
178         la a0, handle_exception
179         csrw CSR_TVEC, a0
180
```

https://elixir.bootlin.com/linux/v6.6.6/source/arch/riscv/kernel/entry.S
https://elixir.bootlin.com/linux/v6.6.6/source/arch/riscv/kernel/entry.S#L21

Interrupts

- Interrupt is also used for exceptions
  - **protection error** for access violation
  - **page fault** for memory access error
  - software interrupt for **system calls**
- Multi-CPU systems can process interrupts concurrently
  - sometimes a CPU may be dedicated to handle interrupts
  - interrupts can also have **CPU affinity**

SMP IRQ Affinity

- Starting with the 2.4 kernel, Linux has gained the ability to assign certain IRQs to specific processors (or groups of processors). This is known as SMP IRQ affinity, and it allows you control how your system will respond to various hardware events. It allows you to restrict or repartition the work load that you server must do so that it can more efficiently do it's job.
  - "balance" out multiple NICs in a multi-processor machine. By tying a single NIC to a single CPU, you should be able to scale the amount of traffic your server can handle nicely.
  - database servers (or servers with lots of disk storage) that also have heavy network loads can dedicate a CPU to their disk controller and assign another to deal with the NIC to help improve response times.

Direct Memory Access

- DMA transfer data directly between I/O device and memory
  - OS only need to issue commands, data transfers bypass the CPU
  - no programmed I/O (one byte at a time), data transferred in large blocks
  - it requires DMA controller in the device or system
- OS issues commands to the DMA controller
  - a command includes: operation, memory address for data, count of bytes…
  - usually it is the pointer of the command written into the command register
  - when done, device interrupts CPU to signal completion

Steps of a DMA Transfer

Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
  - in Linux, devices can be accessed **as files**; low-level **access with ioctl**
- Device-driver layer hides differences among I/O controllers from kernel
  - each OS has its own I/O subsystem and device driver frameworks
  - new devices talking already-implemented protocols need no extra work

Application I/O Interface

```
root@ubuntu-linux-22-04-02-desktop:/dev# ls -l
total 0
crw-r--r--  1 root      root     10, 235 Dec 12 09:19 autofs
drwxr-xr-x  2 root      root         400 Dec 12 09:19 block
drwxr-xr-x  2 root      root          80 Dec 12 09:19 bsg
crw-rw----  1 root      disk     10, 234 Dec 12 09:19 btrfs-control
drwxr-xr-x  3 root      root          60 Jan  1  1970 bus
lrwxrwxrwx  1 root      root           3 Dec 12 09:19 cdrom -> sr0
drwxr-xr-x  2 root      root        3800 Dec 12 09:19 char
crw--w----  1 root      tty       5,   1 Dec 12 09:19 console
lrwxrwxrwx  1 root      root          11 Dec 12 09:19 core -> /proc/kcore
crw-------  1 root      root     10, 124 Dec 12 09:19 cpu_dma_latency
crw-------  1 root      root     10, 203 Dec 12 09:19 cuse
drwxr-xr-x  6 root      root         120 Dec 12 09:19 disk
drwxr-xr-x  2 root      root          80 Jan  1  1970 dma_heap
drwxr-xr-x  3 root      root         100 Dec 12 09:19 dri
crw-------  1 root      root     10, 126 Dec 12 09:19 ecryptfs
```

I/O Structure

Characteristics of I/O Devices

- Devices vary in many dimensions
  - character-stream or block
  - sequential or random-access
  - synchronous or asynchronous (or both)
  - sharable or dedicated
  - speed of operation
  - read-write, read only, or write only

Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Characteristics of I/O Devices

- Broadly, I/O devices can be grouped by the OS into
  - **block I/O: read, write, seek**
  - **character I/O** (Stream)
  - **memory-mapped file access**
  - **network sockets**
- OSs have usually an escape/back door that passes any IO commands from app to device
  - Linux's **ioctl** call to send commands to a device driver

Block and Character Devices

- Block devices access data in blocks, such as disk drives…
  - commands include **read, write, seek**
  - raw I/O, direct I/O, or file-system access
  - memory-mapped file access possible (e.g., memory-mapped files)
  - DMA
- Character devices include keyboards, mice, serial ports…
  - very diverse types of devices

Network Devices

- Varying enough from block and character to have own interface
  - very different from pipe, mailbox…
- Popular interface for network access is the **socket** interface
  - it separates network protocol from detailed network operation
  - some non-network operations are implemented as sockets
    - e.g., Unix socket

Clocks and Timers

- Clocks and timers can be considered as character devices
  - very important devices as they provide current time, elapsed time, timer
- Normal resolution about 1/60 second, some OS provides higher-resolution ones

Synchronous/Asynchronous I/O

- **Synchronous I/O** includes blocking and non-blocking I/O
  - **blocking I/O**: process suspended until I/O completed
    - easy to use and understand, but may be less efficient
    - insufficient for some needs
  - **non-blocking I/O**: I/O calls return **as much data as available**
    - process does not block, returns whatever existing data (read or write)
    - use select to find if data is ready, then use read or write to transfer data
- **Asynchronous I/O**: process runs while I/O executes,
  - I/O subsystem signals process when I/O completed via signal or callback
  - difficult to use but very efficient

Two I/O Methods

Kernel I/O Subsystem

- **I/O scheduling**
  - to queue I/O requests via per-device queue
  - to schedule I/O for fairness and quality of service
- **Buffering** - store data in memory while transferring between devices
  - to cope with device speed mismatch: receive from network and write to ssd, double buffering
  - to cope with device transfer size mismatch: network buffer reassembly of message
  - to maintain “copy semantics”: write()

Kernel I/O Subsystem

- **Caching**: hold **a copy of data for fast access**
  - key to performance
  - sometimes combined with buffering
    - Buffer in memory is also used as caching for file operations
- **Spooling**: A spool is a buffer that holds the output (device's input) if device can serve only one request at a time
  - i.e., printing
- **Device reservation**: provides exclusive access to a device
  - system calls for allocation and de-allocation
  - watch out for deadlock

Error Handling

- Some OSes try to recover from errors
  - e.g., device unavailable, transient write failures
  - sometimes via retrying the read or write
  - some systems have more advanced error handling
    - track error frequencies, stop using device with high error frequency

- Some OSes just return an error number or code when I/O request fails
  - system error logs hold problem reports

I/O Protection

- OS needs to protect I/O devices
  - e.g., keystrokes can be stolen by a **keylogger** if keyboard is not protected
  - always assume user may attempt to obtain illegal I/O access
- To protect I/O devices:
  - define all I/O instructions to be privileged
    - I/O must be performed via system calls
  - memory-mapped I/O and I/O ports must be protected too

Use System Call to Perform I/O

Kernel Data Structures

- Kernel keeps state info for I/O components
  - e.g., open file tables, network connections, character device state
  - many data structures to track buffers, memory allocation, “dirty” blocks
    - sometimes very complicated
- Some OS uses message passing to implement I/O, e.g., Windows
  - message with I/O information passed from user mode into kernel
  - message modified as it flows through to device driver and back to process

UNIX I/O Kernel Structure

https://elixir.bootlin.com/linux/v6.6.6/source/include/linux/sched.h#L1094
https://elixir.bootlin.com/linux/v6.6.6/source/include/linux/fdtable.h#L49

I/O Requests to Hardware

- **System resource** access needs to be mapped to hardware
- Consider reading a file from disk for a process:
  - determine device holding file
  - translate name to **device representation**
    - FAT, UNIX: major/minor
  - physically read data from disk into buffer
  - make data available to requesting process
  - return control to process

Life Cycle of An I/O Request

Performance

- I/O is a major factor in system performance:
  - CPU to execute device driver, kernel I/O code
  - context switches due to interrupts
  - data buffering and copying
    - network traffic especially stressful

Network Communications: high context switch

- A remote login from one machine to another
  - Each character typed on the local machine must be transported to the remote machine

Improve Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Use smarter hardware devices
- Balance CPU, memory, bus, and I/O performance for highest throughput
- Move user-mode processes / daemons to kernel threads

Linux IO – Device initialization

- /dev/tty
  - tty_init https://elixir.bootlin.com/linux/v6.6.6/source/drivers/tty/tty_io.c#L3618
  - Create /dev/tty file

```
3618  int __init tty_init(void)
3619  {
3620          register_sysctl_init("dev/tty", tty_table);
3621          cdev_init(&tty_cdev, &tty_fops);
3622          if (cdev_add(&tty_cdev, MKDEV(TTYAUX_MAJOR, 0), 1) ||
3623              register_chrdev_region(MKDEV(TTYAUX_MAJOR, 0), 1, "/dev/tty") < 0)
3624                  panic("Couldn't register /dev/tty driver\n");
3625          device_create(&tty_class, NULL, MKDEV(TTYAUX_MAJOR, 0), NULL, "tty");
```

Linux IO – Device write (write system call)

- Write (echo) to the file reach vfs_write, which eventually calls tty_write

```c
ssize_t vfs_write(struct file *file, const char __user *buf, size_t count,
{
        ssize_t ret;

        if (!(file->f_mode & FMODE_WRITE))
                return -EBADF;
        if (!(file->f_mode & FMODE_CAN_WRITE))
                return -EINVAL;
        if (unlikely(!access_ok(buf, count)))
                return -EFAULT;

        ret = rw_verify_area(WRITE, file, pos, count);
        if (ret)
                return ret;
        if (count > MAX_RW_COUNT)
                count =  MAX_RW_COUNT;
        file_start_write(file);
        if (file->f_op->write)
                ret = file->f_op->write(file, buf, count, pos);
        else if (file->f_op->write_iter)
                ret = new_sync_write(file, buf, count, pos);
        else
                ret = -EINVAL;
```

```c
static const struct file_operations tty_fops = {
        .llseek         = no_llseek,
        .read_iter      = tty_read,
        .write_iter     = tty_write,
        .splice_read    = copy_splice_read,
        .splice_write   = iter_file_splice_write,
        .poll           = tty_poll,
        .unlocked_ioctl = tty_ioctl,
        .compat_ioctl   = tty_compat_ioctl,
        .open           = tty_open,
        .release        = tty_release,
        .fasync         = tty_fasync,
        .show_fdinfo    = tty_show_fdinfo,
};
```

Linux ioctl system call

- ioctl system call

NAME top

```
ioctl - control device
```

SYNOPSIS top

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

int ioctl(int fd, unsigned long request, ...);
```

DESCRIPTION top

```
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.
```

Linux ioctl system call

- ioctl system call implementation
  - ioctl syscall -> vfs_ioctl -> indirect call -> tty_ioctl
  - Leads to many security problems

```
long vfs_ioctl(struct file *filp, unsigned int cmd, unsigned
{
        int error = -ENOTTY;

        if (!filp->f_op->unlocked_ioctl)
                goto out;

        error = filp->f_op->unlocked_ioctl(filp, cmd, arg);
        if (error == -ENOIOCTLCMD)
                error = -ENOTTY;
 out:
        return error;
}
EXPORT_SYMBOL(vfs_ioctl);
```

```
const struct file_operations tty_fops = {
 .llseek          = no_llseek,
 .read_iter       = tty_read,
 .write_iter      = tty_write,
 .splice_read     = generic_file_splice_read,
 .splice_write    = iter_file_splice_write,
 .poll            = tty_poll,
 .unlocked_ioctl  = tty_ioctl,
 .compat_ioctl    = tty_compat_ioctl,
 .open            = tty_open,
 .release         = tty_release,
 .fasync          = tty_fasync,
 .show_fdinfo     = tty_show_fdinfo,
```

Takeaway

- IO hardware
- IO access
  - polling, interrupt
- Device types
- Application I/O Interface
- Kernel IO subsystem