Filesystem

Disclaimer: some slides are adopted from book authors’ slides with permission
Recap

- Blocking, non-blocking, asynchronous I/O
- Data transfer methods
  - Programmed I/O: CPU is doing the I/O
    - Pros
    - Cons
  - DMA: DMA controller is doing the I/O
    - Pros
    - Cons
int main(int argc, char *argv[])
{
    int src_fd, dst_fd; char buf[4100]; int nread, nwrite; char *ptr;

    src_fd = open(argv[1], O_RDONLY);
    dst_fd = open(argv[2], O_WRONLY);
    nread = read(src_fd, buf, sizeof(buf)); ptr = buf;
    while (nread > 0) {
        errno = 0;
        nwrite = write(dst_fd, ptr, nread);
        fprintf(stderr, "nwrite = %d, errno = %d (%s)\n", nwrite, errno, strerror(errno));
        if (nwrite > 0) {
            ptr += nwrite; nread -= nwrite;
        }
    }

    $ sudo ./copyfile /dev/zero /dev/ttyS0
    nwrite = 4100, errno = 0 (Success)
Non-Blocking I/O

int main(int argc, char *argv[])  
{  
    int src_fd, dst_fd; char buf[4100]; int nread, nwrite; char *ptr;

    src_fd = open(argv[1], O_RDONLY);  
dst_fd = open(argv[2], O_WRONLY | O_NONBLOCK);  
nread = read(src_fd, buf, sizeof(buf));  ptr = buf;
while (nread > 0) {
    errno = 0;
    nwrite = write(dst_fd, ptr, nread);
    fprintf(stderr, "nwrite = %d, errno = %d (%s)\n", nwrite, errno, strerror(errno));
    if (nwrite > 0 ) {
        ptr += nwrite; nread -= nwrite;
    }
}
}

$ sudo ./copyfile /dev/zero /dev/ttyS0
nwrite = 4095, errno = 0 (Success)
nwrite = -1, errno = 11 (Resource temporarily unavailable)
nwrite = -1, errno = 11 (Resource temporarily unavailable)
nwrite = 5, errno = 0 (Success)
Recap: Programmed I/O

```c
#define CTRL_BASE_ADDR 0xCE000000

int *io_base = (int *)ioremap_nocache(CRTL_BASE_ADDR, 4096);

// initialize the device (by writing some values to h/w regs)
*io_base = 0x1;
*(io_base + 1) = 0x2;
*(io_base + 2) = 0x3;
...
// wait until the device is ready (bit31 = 0)
while (*io_base & 0x80000000);
```

// send data to the device
for (i = 0; i < sizeof(buffer); i++) {
    *(io_base + 0x10) = buffer[i];
    while (*io_base & 0x80000000);
}
```
Programmed I/O (PIO)
Recap: Direct Memory Access

1. Device driver is told to transfer disk data to buffer at address X
2. Device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. Disk controller initiates DMA transfer
4. Disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. When C = 0, DMA interrupts CPU to signal transfer completion
int dad_transfer(struct dad_dev *dev, int write, void *buffer, size_t count)
{
    dma_addr_t bus_addr;

    /* Map the buffer for DMA */
    dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
    dev->dma_size = count;
    bus_addr = dma_map_single(&dev->pci_dev->dev, buffer, count,
                              dev->dma_dir);
    dev->dma_addr = bus_addr;

    /* Set up the device */
    writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
    writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
    writel(dev->registers.addr, cpu_to_le32(bus_addr));
    writel(dev->registers.len, cpu_to_le32(count));

    /* Start the operation */
    writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
    return 0;
}
Storage Subsystem in Linux OS

- User Applications
  - System call Interface
  - Virtual File System (VFS)
    - Filesystem (FAT, ext4, ...)
    - Buffer cache
    - I/O Scheduler
    - Disk driver
  - Inode cache
  - Directory cache

- Kernel

- Hardware
  - SSD (or HDD)
Filesystem

• Definition
  – An OS layer that provides file and directory abstractions on disks

• File
  – User’s view: a collection of bytes (non-volatile)
  – OS’s view: a collection of blocks
    • A block is a logical transfer unit of the kernel (typically block size \(\geq\) sector size)
Filesystem

• File types
  – Executables, DLLs, text, word, ....
  – Filesystems mostly don’t care

• File attributes (metadata)
  – Name, location, size, protection, ...

• File operations
  – Create, read, write, delete, seek, truncate, ...
How to Design a Filesystem?

• What to do?
  – Map disk blocks to each file
  – Need to track free disk blocks
  – Need to organize files into directories

• Requirements
  – Should not waste space
  – Should be fast
Access Pattern

• Sequential access
  – E.g.,) read next 1000 bytes

• Random access
  – E.g.,) Read 10 bytes at the offset 300

• Remember that random access is especially slow in HDD.
File Usage Patterns

• Most files are small
  – .c, .h, .txt, .log, .ico, ...
  – Also more frequently accessed
  – If the block size is too big, it wastes space (why?)

• Large files use most of the space
  – .avi, .mp3, .jpg,
  – If the block size is too small, mapping information can be huge (performance and space overhead)
Disk Allocation

• How to map disk blocks to files?
  – Each file may have very different size
  – The size of a file may change over time (grow or shrink)

• Disk allocation methods
  – Continuous allocation
  – Linked allocation
  – Indexed allocation
Continuous Allocation

- Use continuous ranges of blocks
  - Users declare the size of a file in advance
  - File header: first block #, #of blocks
  - Similar to malloc()

- Pros
  - Fast sequential access
  - easy random access

- Cons
  - External fragmentation
  - difficult to increase
Linked-List Allocation

- Each block holds a pointer to the next block in the file

- Pros
  - Can grow easily

- Cons
  - Bad access perf.
Quiz

• How many disk accesses are necessary for direct access to byte 20680 using linked allocation and assuming each disk block is 4 KB in size?

• Answer: 6 disk accesses.
File Allocation Table (FAT)

- A variation of linked allocation
  - Links are not stored in data blocks but in a separate table FAT[#of blocks]

- Directory entry points to the first block (217)
- FAT entry points to the next block (FAT[217] = 618)
Example: FAT

Disk content

<table>
<thead>
<tr>
<th>Offset (hex)</th>
<th>+0</th>
<th>+2</th>
<th>+4</th>
<th>+6</th>
<th>+8</th>
<th>+A</th>
<th>+C</th>
<th>+E</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x200</td>
<td>0001</td>
<td>0002</td>
<td>FFFF</td>
<td>0104</td>
<td>0205</td>
<td>FFFF</td>
<td>FFFF</td>
<td>000E</td>
<td>FAT[0] ~ FAT[7]</td>
</tr>
<tr>
<td>0x210</td>
<td>0009</td>
<td>000A</td>
<td>FFFF</td>
<td>000C</td>
<td>000D</td>
<td>FFFF</td>
<td>FFFF</td>
<td>0010</td>
<td>FAT[8] ~ FAT[15]</td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Directory entry (stored in different location in disk)

<table>
<thead>
<tr>
<th>File name</th>
<th>...</th>
<th>First block (cluster) no.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project2.pdf</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Q. What are the disk blocks (clusters) of the Project2.pdf file?
A. 8, 9, 10
Indexed Allocation

• Use per-file index block which holds block pointers for the file
  – Directory entry points to a index block (block 19)
  – The index block points to all blocks used by the file

• Pros
  – No external fragmentation
  – Fast random access

• Cons
  – Space overhead
  – File size limit (why?)
Quiz

• Suppose each disk block is 2048 bytes and a block pointer size is 4 byte (32bit). Assume the previously described indexed allocation scheme is used.

• What is the maximum size of a single file?

• Answer
  – \( \frac{2048}{4} \times 2048 = 1,048,576 \) (1MB)