lab5实验报告

思考题

Thinking 5.1

1. 可能引发的问题：

通过 `kseg0` 读写设备时，由于数据会缓存到 CPU Cache 中，可能导致以下问题：

- **数据不一致性**：设备寄存器的值可能不会立即写入实际设备，而是停留在 Cache 中，导致设备行为与预期不符。
    - 例如：向串口发送数据时，若写入的数据被缓存而未实际写入设备，数据可能无法及时发送。
- **读取过时数据**：从设备寄存器读取时，可能直接从 Cache 中获取旧值，而非设备的最新状态。
    - 例如：读取串口状态寄存器时，可能误判设备状态（如“发送完成”标志未更新）。
- **并发问题**：在多核或 DMA 场景下，其他处理器或设备可能直接访问物理设备内存，而 Cache 中的值未同步，导致数据冲突。

2. 对不同设备的影响差异：

不同设备对读写时效性和一致性的要求不同：

- **串口设备（实时性高）**：
    - 需要严格保证读写操作的实时性，写入的数据必须立即到达设备，读取的状态必须是最新值。
    - 若使用缓存（尤其是写回策略），可能导致数据延迟或状态误判，引发通信错误。
    - 通常应通过非缓存地址（如 `kseg1`）或显式禁用缓存（如 `uncached` 操作）访问。
- **IDE 磁盘（块设备，容忍一定延迟）**：
    - 磁盘读写通常以块为单位，且允许一定延迟（如写入缓存后由磁盘控制器异步处理）。
    - 若使用缓存，可能通过写缓冲（Write Buffer）或写合并（Write Combining）优化性能，但仍需注意：
        - DMA 操作时，需确保 Cache 与内存一致（如手动刷 Cache 或使用一致性内存）。
        - 写入磁盘控制命令寄存器时，仍需保证实时性（如通过内存屏障或非缓存写入）。

3. 缓存策略的影响：

- **写回（Write-Back）**：
    - 写入操作仅更新 Cache，延迟写入内存，对设备访问危害最大。
- **写直达（Write-Through）**：
    - 写入操作同步更新 Cache 和内存，但仍可能因 Cache 命中导致读取旧值。
- **解决方案**：
    - 使用非缓存地址（如 `kseg1`）。
    - 显式调用 Cache 刷新指令（如 `sync`、`cache` 指令）。
    - 对内存区域标记为“不可缓存”（通过页表或硬件配置）。

Thinking5.2

一个磁盘块可以存储16个文件控制块

一个目录下最多能有1024个磁盘块，每个磁盘块能够存储16个文件控制块，所以一个目录下最多能有16384个文件

单个文件最大大小为4MB

Thinking5.3

768MB

Thinking5.4

宏定义	值/计算方式	作用
`PTE_DIRTY`	`0x0004`	标记脏页（需写回磁盘）
`SECT_SIZE`	`512`	磁盘扇区大小（字节）
`SECT2BLK`	`BLOCK_SIZE / SECT_SIZE`	每块包含的扇区数
`DISKMAP`	`0x10000000`	磁盘缓存映射起始地址
`DISKMAX`	`0x40000000`	磁盘缓存映射结束地址（最大磁盘 `768MB`）
`BLOCK_SIZE`	`PAGE_SIZE`（`4KB`）	文件系统块大小
`MAXNAMELEN`	`128`	文件名最大长度
`MAXPATHLEN`	`1024`	路径名最大长度
`NDIRECT`	`10`	直接指针数量
`NINDIRECT`	`BLOCK_SIZE / 4`（`1024`）	间接指针块容量
`MAXFILESIZE`	`NINDIRECT * BLOCK_SIZE`（`4MB`）	单个文件最大大小（仅间接部分）
`FILE_STRUCT_SIZE`	`256`	文件控制块大小

Thinking5.5

代码实现

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main() {
    const char *filename = "test_fork_fd.txt";
    int fd = open(filename, O_RDWR | O_CREAT | O_TRUNC, 0666);
    if (fd < 0) {
        perror("open");
```

```
        return 1;
    }

    // 父进程写入数据
    const char *parent_msg = "Parent writes first line.\n";
    write(fd, parent_msg, strlen(parent_msg));
    printf("Parent: wrote data, offset now at %ld\n", lseek(fd, 0,
SEEK_CUR));

    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        return 1;
    } else if (pid == 0) {
        // 子进程读取文件
        char buf[1024];
        ssize_t n = read(fd, buf, sizeof(buf));
        if (n > 0) {
            buf[n] = '\0';
            printf("Child: read from offset %ld: %s", lseek(fd, 0,
SEEK_CUR), buf);
        }
        close(fd);
        return 0;
    } else {
        // 父进程继续写入
        sleep(1); // 确保子进程先读取
        const char *parent_msg2 = "Parent writes second line.\n";
        write(fd, parent_msg2, strlen(parent_msg2));
        printf("Parent: wrote again, offset now at %ld\n", lseek(fd, 0,
SEEK_CUR));

        wait(NULL); // 等待子进程结束
        close(fd);
    }

    return 0;
}
```

程序行为分析

1. **父进程**:
    - 打开文件，写入 `"Parent writes first line.\n"`，偏移量移动到 `24` 字节。
    - 调用 `fork()`，子进程复制父进程的文件描述符 `fd`，但共享同一个打开文件表项。
2. **子进程**:
    - 从当前偏移量（`24`）尝试读取，但文件已无更多数据（因为父进程尚未写入第二行），故 `read()` 返回 `0`。
    - 如果父进程未调用 `sleep(1)`，子进程可能读到部分数据（竞争条件）。
3. **父进程**:
    - 写入 `"Parent writes second line.\n"`，偏移量更新到 `48` 字节。

预期输出

```
Parent: wrote data, offset now at 24
Child: read from offset 24:
Parent: wrote again, offset now at 48
```

- **关键现象**：
    - 子进程的 `read()` 从偏移量 `24` 开始（继承自父进程）。
    - 父进程的第二次写入不受子进程读取影响，说明偏移量是共享的。

结论

- **文件描述符**：`fork()` 会复制描述符表，但父子进程的 `fd` 指向**同一个打开文件表项**。
- **文件偏移量**：共享，因此父子进程的读写操作会相互影响偏移量。
- **实际应用**：
    - 若需避免干扰，可在 `fork()` 后重新打开文件（`open()`），或使用 `dup2()` 创建独立描述符。

Thinking5.6

1. `struct File`（文件控制块）

- **作用**：描述文件或目录的元数据，**对应磁盘上的物理实体**（存储在磁盘的 inode 区域）。
- **核心字段**：
    - `f_name`：文件名（`MAXNAMELEN=128`）。
    - `f_size`：文件大小（字节）。
    - `f_type`：文件类型（`FTYPE_REG` 或 `FTYPE_DIR`）。
    - `f_direct[NDIRECT]`：**直接数据块指针**（`NDIRECT=10`，支持 `40KB`）。
    - `f_indirect`：**间接块指针**（指向一个块，存储 `1024` 个指针，支持 `4MB` 文件）。
    - `f_dir`：**内存中**指向父目录的指针（不存盘）。
- **使用场景**：
    - 文件系统读写时，通过 `f_direct` 和 `f_indirect` 定位磁盘块。
    - 目录操作时，`f_dir` 用于快速回溯父目录。

---

2. `struct Fd`（文件描述符元数据）

- **作用**：管理进程打开的文件，**纯内存数据结构**（不存盘）。
- **核心字段**：
    - `fd_dev_id`：文件所属设备 ID（如磁盘、控制台）。
    - `fd_offset`：当前读写偏移量。
    - `fd_omode`：打开模式（`O_RDONLY`、`O_WRONLY` 等）。
    - `fd_sock`：套接字专用字段（非文件时使用）。
- **使用场景**：
    - 进程调用 `open()` 时创建，`close()` 时释放。
    - 通过 `fd_offset` 实现 `read()`/`write()` 的定位。

---

3. `struct Filefd`（文件描述符扩展结构）

- **作用**：组合 `Fd` 和 `File`，用于**文件系统服务进程**（`fs/serv.c`），**纯内存数据**。
- **核心字段**：
  - `f_fd`：内嵌的 `struct Fd`。
  - `f_file`：内嵌的 `struct File`。
  - `f_fileid`：文件唯一 ID（用于服务端查找）。
- **使用场景**：
  - 客户端进程通过 IPC 请求文件操作时，服务端用 `Filefd` 统一管理文件状态。
  - 例如：`fs/serv.c` 中的 `serve_read()` 和 `serve_write()`。

数据结构与物理实体的关系

结构体	存储位置	物理实体对应关系	生命周期
`File`	磁盘/内存	磁盘 inode 块（持久化）	文件存在即存在
`Fd`	内存	进程打开文件表项（临时）	进程打开文件期间
`Filefd`	内存	服务端文件会话（临时）	IPC 请求处理期间

设计框架

1. **磁盘层**：
   - `struct File` 存储在磁盘块中，通过 `f_direct` 和 `f_indirect` 指向数据块。
2. **内存层**：
   - 进程通过 `struct Fd` 管理打开的文件。
   - 文件系统服务进程通过 `struct Filefd` 封装客户端请求。
3. **交互流程**：
   - 客户端 `open()` → 服务端查找 `File` → 返回 `Fd`。
   - 客户端 `read()` → 服务端通过 `Filefd` 定位 `File` 和 `Fd` → 读取磁盘数据。

关键点总结

- `File` **是磁盘文件的元数据**，直接映射到物理块。
- `Fd` **是进程视角的文件句柄**，维护偏移量和打开模式。
- `Filefd` **是服务端的工作结构**，整合 `Fd` 和 `File` 以处理 IPC 请求。
- **父子进程** `fork()` **后**：
  - 共享打开文件表（`Fd` 的 `fd_offset` 会相互影响）。
  - 独立文件描述符表（`fd` 编号可不同）。

Thinking5.7

实线箭头+点表示异步调用

实线箭头不加点表示同步调用

虚线箭头不加点表示同步回调