lab6实验报告

思考题

Thinking 6.1

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int fildes[2];
char buf[100];
int status;

int main() {
    status = pipe(fildes);

    if(status == -1) {
        printf("error\n");
        exit(EXIT_FAILURE);
    }

    switch(fork()) {
        case -1:
            perror("fork");
            exit(EXIT_FAILURE);
            break;

        case 0: /* 子进程 - 作为管道的写者 */
            close(fildes[0]); /* 关闭不用的读端 */
            write(fildes[1], "Hello world\n", 12); /* 向管道中写数据 */
            close(fildes[1]); /* 写入结束，关闭写端 */
            exit(EXIT_SUCCESS);

        default: /* 父进程 - 作为管道的读者 */
            close(fildes[1]); /* 关闭不用的写端 */
            read(fildes[0], buf, 100); /* 从管道中读数据 */
            printf("parent-process read: %s", buf); /* 打印读到的数据 */
            close(fildes[0]); /* 读取结束，关闭读端 */
            exit(EXIT_SUCCESS);
    }
}
```

Thinking6.2

`dup` 函数中的竞争问题和管道中的 `pp_ref` 竞争问题类似，都是由于 **共享资源的引用计数修改和分配操作的非原子性** 导致的。在多进程环境下，必须确保这些操作是原子的，或者通过锁机制避免竞争。否则，可能会导致文件表项被提前释放、内存访问错误或数据不一致等问题。

Thinking6.3

- **系统调用不一定是原子操作**，其原子性取决于内核实现和具体场景。
- **典型非原子操作**：大文件 `write()`、`fork()` 后的资源竞争、时间获取函数等。
- **开发者需注意**：对关键操作（如管道通信、信号处理）需显式加锁或使用原子API。

反例总结表：

系统调用	非原子场景	风险
`write()`	写入数据超过 `PIPE_BUF`	部分写入，数据不完整
`fork()`	复制文件描述符表	引用计数竞争（如 `pp_ref`）
`gettimeofday()`	多核时钟不同步	时间戳不一致

Thinking6.4

问题1：调整 `pipe_close` 中 `fd` 和 `pipe` 的 `unmap` 顺序是否能解决竞争问题?

原始问题分析

在原始的 `pipe_close` 实现中，如果解除映射的顺序是：

1. 先解除 `pipe` 的映射（`pageref(pipe)--`）。
2. 再解除 `fd` 的映射（`pageref(fd)--`）。

在两次 `unmap` 之间的间隙，会出现：

- `pageref(pipe) == pageref(fd)`（因为 `pipe` 的引用已减1，而 `fd` 的引用尚未减1）。
- 此时若另一个进程检查 `_pipe_is_closed`，会误判为“写端已关闭”（因为条件 `pageref(p[0]) == pageref(pipe)` 被满足）。

解决方案：调整 `unmap` 顺序

如果将顺序调整为：

1. 先解除 `fd` 的映射（`pageref(fd)--`）。
2. 再解除 `pipe` 的映射（`pageref(pipe)--`）。

在两次 `unmap` 之间的间隙，会满足：

- `pageref(pipe) > pageref(fd)`（因为 `fd` 的引用已减1，而 `pipe` 的引用尚未减1）。
- 此时即使另一个进程检查 `_pipe_is_closed`，也不会误判为“写端已关闭”，因为 `pageref(p[0]) == pageref(pipe)` 不成立。

结论

调整 `unmap` 顺序后：

- **在关闭过程中，`pageref(pipe) == pageref(fd)` 的情况永远不会出现。**
- 只有在写端真正关闭时（`fd` 和 `pipe` 的引用均已减1），才会满足 `pageref(p[0]) == pageref(pipe)`。
- 因此，可以避免竞争导致的误判。

问题2：`dup` 函数中是否会出现类似 `close` 的竞争问题?

`dup` 的工作流程

当复制一个指向管道的文件描述符时，`dup` 的典型操作包括：

1. 分配一个新的文件描述符 `new_fd`。
2. 增加 `pipe` 的引用计数（`pageref(pipe)++`）。
3. 将 `new_fd` 指向相同的 `pipe`。

可能的竞争场景

假设进程 A 正在执行 `dup`：

1. 进程 A 分配了 `new_fd`，但尚未增加 `pageref(pipe)`。
2. 进程 B 调用 `close` 关闭了原文件描述符 `old_fd`：
    - 如果 `close` 先解除 `old_fd` 的映射（`pageref(old_fd)--`），再解除 `pipe` 的映射（`pageref(pipe)--`）。
    - 在两次 `unmap` 之间，`pageref(pipe)` 可能暂时等于 `pageref(old_fd)`。
3. 进程 A 恢复执行，增加 `pageref(pipe)`，但此时 `pipe` 可能已被释放（因为进程 B 的 `close` 认为 `pageref(pipe)` 已归零）。

类比 `close` 的解决方案

类似 `close` 的竞争问题，`dup` 的竞争可以通过以下方式解决：

1. **确保引用计数的修改是原子的**：
    - 在增加 `pageref(pipe)` 之前，先锁定管道资源。
    - 使用原子操作（如 `atomic_inc`）更新 `pageref(pipe)`。
2. **调整操作顺序**：
    - 在 `dup` 中，先增加 `pageref(pipe)`，再分配 `new_fd`。
    - 这样即使被中断，`pipe` 的引用计数也不会被错误归零。

Thinking6.5

1. 打开文件的过程（Lab5文件系统）

在Lab5中，打开文件的过程主要涉及以下步骤：

- **文件描述符分配**：通过 `fd_alloc()` 分配一个新的文件描述符（`struct Fd`）。
- **文件查找**：调用 `file_open()` 根据路径名查找文件（`struct File`）。
- **建立映射**：将文件内容映射到内存（通过 `file_get_block()` 读取文件块，并建立页表映射）。
- **返回fd**：将文件描述符与进程关联，并返回文件描述符编号。

关键函数：

- `open()`：用户态接口，调用 `fd_alloc()` 和 `file_open()`。
- `file_open()`：解析路径，查找文件。
- `file_get_block()`：读取文件数据块到内存。

注意：文件系统通过块设备驱动（如磁盘）读取数据，并缓存到内存（`struct Block`）。

---

2. 读取并加载ELF文件（Lab1与Lab3）

ELF（Executable and Linkable Format）文件的加载分为两部分：

- **内核加载（Lab1）**：通过 `bootmain()` 读取磁盘上的内核ELF文件，加载到物理内存。
- **用户进程加载（Lab3）**：通过 `load_icode()` 加载用户程序到虚拟内存。

关键步骤：

1. **读取ELF头**：验证魔数（`ELF_MAGIC`），获取程序头表（`struct Proghdr`）。
2. **加载段（`elf_load_seg`）**：
    - 遍历程序头表，找到 `PT_LOAD` 类型的段（代码段、数据段）。
    - 将段内容从文件读取到虚拟内存（`page_insert()` 建立映射）。
3. **处理BSS段**：
    - BSS段在ELF文件中不占用空间（`filesize < memsize`），但需要清零。
    - 在 `load_icode_mapper()` 中，对超出 `filesize` 的部分填充0。

关键函数：

- `elf_load_seg()`：加载单个段到内存。
- `load_icode_mapper()`：实际映射页并处理BSS段。

---

3. BSS段的加载实现（Lab3）

BSS段的特点：

- 在ELF文件中不存储数据（`filesize < memsize`）。
- 加载到内存后需要初始化为0。

实现机制：

1. `elf_load_seg()`：
    - 调用 `load_icode_mapper()` 逐页加载段。
    - 参数：
        - `va`：虚拟地址。
        - `filesz`：段在文件中的大小。
        - `memsz`：段在内存中的大小（包含BSS）。
        - `bin`：文件内容指针。
2. `load_icode_mapper()`：
    - 对每一页：
        - 如果 `filesz > 0`，从文件读取数据到页（`memcpy()`）。
        - 如果 `memsz > filesz`，剩余部分清零（`memset(0)`）。
    - 示例代码：

      c

      复制

      下载

```
if (offset < filesz) {
    memcpy(page2kva(page), bin + offset, min(PGSIZE, filesz - offset));
}
if (offset + PGSIZE > filesz) {
    memset(page2kva(page) + (filesz - offset), 0, (offset + PGSIZE) - filesz);
}
```

总结：

- BSS段通过 `memsz > filesz` 标识。
- 加载时显式清零超出 `filesz` 的部分。

---

4. 加载内核进程 vs. 用户进程

对比项	内核加载（Lab1）	用户进程加载（Lab3）
ELF加载函数	`bootmain()`	`load_icode()`
内存映射	直接物理内存	虚拟内存（通过页表）
BSS处理	手动清零（`stab_bin`）	`load_icode_mapper()` 自动清零
权限	内核态（无页表保护）	用户态（受页表限制）

---

5. 实现 `spawn` 函数的提示

`spawn` 是创建用户进程的接口，类似于 `exec`，但需从文件系统加载ELF。可类比 `load_icode()`：

1. **打开文件**：调用 `open()` 获取ELF文件描述符。
2. **读取ELF头**：验证魔数，获取程序头表。
3. **加载段**：
    - 使用 `elf_load_seg()` 加载代码段、数据段。
    - 处理BSS段（`memsz > filesz` 部分清零）。
4. **设置用户栈**：分配栈空间并初始化（如参数压栈）。
5. **启动进程**：设置 `eip` 和 `esp`，切换到用户态。

关键点：

- 复用 `load_icode_mapper()` 处理BSS段。
- 从文件系统读取数据（而非直接内存拷贝）。

总结

1. **文件打开**：通过文件描述符和块设备驱动实现。
2. **ELF加载**：
    - 内核：直接加载到物理内存。
    - 用户进程：通过页表映射到虚拟内存。

3. **BSS段**：通过 `memsz > filesz` 标识，加载时显式清零。
4. `spawn` **实现**：结合文件系统和 `load_icode()` 的逻辑。

Thinking6.6

1. 标准输入/输出的初始化流程

在 MOS 操作系统中，**0 号文件描述符（stdin）和 1 号文件描述符（stdout）的绑定过程**发生在以下关键步骤中：

1. `user/icode.b` **进程的初始化**：
   - 内核启动的第一个用户进程是 `icode.b`（位于 `user/icode.c`），它会调用 `spawn` 创建 `init.b` 进程。
   - `icode.b` 自身并未显式打开控制台设备，但内核在启动用户进程时默认将控制台（`/dev/console`）与 fd 0 和 1 关联。
2. `init.b` **进程显式打开控制台**：
   - `init.b`（位于 `user/init.c`）会**显式调用** `open("/dev/console", O_RDWR)`，返回的文件描述符通常是 0（因为这是第一个打开的文件）。
   - 接着通过 `dup(0)` 复制到 1 号文件描述符，确保 stdin 和 stdout 均指向控制台：

     c

     复制

     下载

     ```
     // user/init.c
     int fd = open("/dev/console", O_RDWR); // fd = 0
     dup(fd); // fd = 1
     dup(fd); // fd = 2 (可选，标准错误 stderr)
     ```
3. `sh.b`（**shell**）**继承文件描述符**：
   - `init.b` 调用 `spawn("sh.b")` 启动 shell 进程。
   - 由于 `spawn` 设置了 `PTE_LIBRARY` 权限位，子进程（`sh.b`）会共享父进程（`init.b`）的文件描述符表，因此 shell 直接继承了 fd 0 和 1，无需重新打开控制台。

2. 关键代码分析

- `open` **和** `dup` **的绑定逻辑**：
  - `open("/dev/console")` 返回的 fd 由文件系统分配，默认从 0 开始（如果无其他打开的文件）。
  - `dup(fd)` 会复制文件描述符，并选择当前最小的空闲 fd（此处为 1）。
- `spawn` **的共享机制**：
  - `spawn` 通过 `PTE_LIBRARY` 共享父进程的页面，包括文件描述符表（`struct Fd *fd_array`），因此子进程直接继承父进程的 fd 0 和 1。

3. 标准输入/输出的“安排”时机

- **显式绑定**：在 `init.b` 中通过 `open + dup` 显式将控制台绑定到 fd 0 和 1。
- **隐式继承**：后续通过 `spawn` 创建的子进程（如 `sh.b`）自动继承这些 fd，无需重复绑定。

4. 流程图解

```
内核启动
  |
  ↓
启动 user/icode.b
  |
  ↓
spawn → init.b
  |       |
  |       ├─ open("/dev/console") → fd = 0
  |       |
  |       ├─ dup(0) → fd = 1
  |       |
  |       ↓
  |       └──── spawn → sh.b（继承 fd 0/1）
```

5. 总结

- **关键步骤**：`init.b` 中显式调用 `open` 和 `dup` 将控制台绑定到 fd 0 和 1。
- **共享机制**：`spawn` 通过 `PTE_LIBRARY` 共享父进程的 fd 表，使得子进程（如 shell）无需重新打开控制台即可使用标准输入/输出。
- **设计意义**：这种机制保证了进程间文件描述符的一致性，同时避免了重复初始化（如每次启动 shell 都重新打开控制台）。

thinking6.7

- **MOS Shell**：
  - 内置命令（如 `cd`）由 shell 直接处理，通过系统调用修改当前进程状态。
  - 外部命令（如 `ls`）通过 `spawn` 创建子进程执行。
- **Linux `cd` 为内置命令的原因**：
  1. **进程隔离性**：子进程无法修改父进程的 `cwd`。
  2. **必要性**：必须由 shell 直接调用 `chdir()`。
  3. **性能**：避免无意义的子进程创建开销。

关键设计原则：

- 若命令需修改 shell 自身状态或高频调用，应设计为内置命令。
- 若命令是独立功能（如文件操作），可设计为外部命令。

thinking6.8

1. 命令解析：`ls.b | cat.b > motd`

这条命令由三部分组成：

1. `ls.b`：列出当前目录的文件。
2. `|`（**管道**）：将 `ls.b` 的输出作为 `cat.b` 的输入。
3. `> motd`（**重定向**）：将 `cat.b` 的输出写入文件 `motd`。

2. 进程创建（`spawn`）

在 MOS 中，管道和重定向的实现会涉及以下进程创建：

1. `ls.b` **进程**：
    - Shell 调用 `spawn("ls.b")` 创建子进程执行 `ls.b`。
    - 其标准输出（stdout）被重定向到管道的写端。
2. `cat.b` **进程**：
    - Shell 调用 `spawn("cat.b")` 创建子进程执行 `cat.b`。
    - 其标准输入（stdin）从管道的读端读取数据。
    - 其标准输出（stdout）被重定向到文件 `motd`。
3. **Shell 自身的管道设置**：
    - Shell 需调用 `pipe()` 创建管道，并将 `ls.b` 和 `cat.b` 的输入输出绑定到管道两端。

观察到的 `spawn` 次数：

- **2 次**：分别对应 `ls.b` 和 `cat.b` 的创建。

3. 进程销毁

进程销毁发生在以下情况：

1. `ls.b` **完成执行**：
    - `ls.b` 输出结束后，关闭管道写端，进程退出。
2. `cat.b` **完成执行**：
    - `cat.b` 读取完管道数据后，关闭管道读端，进程退出。
3. **Shell 回收子进程**：
    - Shell 通过 `wait()` 或类似机制回收 `ls.b` 和 `cat.b` 的退出状态。

观察到的进程销毁次数：

- **2 次**：分别对应 `ls.b` 和 `cat.b` 的退出。

4. 完整流程