浙江工业大学

操作系统课程设计报告

(2023 级)

学生姓名 _____

学生学号 _____

学科(专业) _____

所在学院 _____

提交日期 2026 年 01 月 19 日

目录

1 概述与环境配置 3
1.1 主要任务 3
1.2 环境配置 3
2 需求和可行性分析 3
2.1 需求分析 3
2.1.1 实验二：简单 Shell 实现需求 3
2.1.2 实验四：Pintos 用户程序需求 4
2.2 可行性分析 5
3 实验二：简单 SHELL 的设计与实现 5
3.1 总体设计思路与数据结构 5
3.1.1 总体设计思路 5
3.1.2 核心数据结构 5
3.2 具体功能实现 5
3.2.1 命令解析算法 5
3.2.2 内建命令实现 6
3.2.3 外部程序执行与路径解析 8
3.2.4 输入/输出重定向 8
3.2.5 信号处理与后台运行 9
3.3 遇到的问题及解决方法 10
3.4 流程图 11
3.5 测试结果 12
4 实验四：PINTOS 用户程序的设计与实现 12
4.1 总体设计思路与数据结构 12
4.2 具体功能实现 13
4.2.1 参数传递 13
4.2.2 内存访问安全 14
4.2.3 系统调用处理框架 15
4.2.4 进程控制系统调用 19
4.2.5 文件操作系统调用 23
4.3 遇到的问题及解决方法 25
4.4 测试结果 25
5 总结与展望 25
5.1 总结 25
5.2 不足 26
5.3 展望 26
6 工程能力与目标达成 26
6.1 专业知识与基础理论的深度理解 26
6.2 系统建模与运行机制推演 26
6.3 基本原理的应用与解决方案的多样性 27
6.4 实验方案设计能力 27
6.5 工程表达与沟通交流 27
7 个人感想 27

1 概述与环境配置

1.1 主要任务

为了深入理解操作系统的设计原理与实现机制，本次课程设计中，我选择了“实验二：简单 Shell 实现”与“实验四：Pintos 用户程序”作为主要实践内容。

- 实验二（简单 Shell 实现）：设计并实现一个模拟 Bash 的命令行解释器。通过该实验，深入理解用户态下进程的创建与控制、输入输出重定向原理以及信号处理机制，掌握操作系统提供的标准接口
- 实验四（Pintos 用户程序）：基于 Pintos 操作系统，扩展内核功能以支持用户程序的运行。该任务涉及参数传递、系统调用（进程控制与文件操作）的实现，以及内核对用户内存空间的保护，从而在内核态层面掌握操作系统如何管理与服务用户进程

1.2 环境配置

本次实验采用在本地 Ubuntu18.04 系统下手动搭建开发环境的方式。

1. 系统与依赖安装：执行以下命令安装 gcc、make、qemu 等必要工具

```
sudo apt update
sudo apt install build-essential automake git qemu-system-x86 gdb
```

2. 配置 GitLab SSH 密钥：

- 生成密钥对：

```
git config --global user.name "name"
git config --global user.email "email@example.com"
ssh-keygen -t rsa -C "email@example.com"
```

- 添加公钥到 GitLab：

```
cat ~/.ssh/id_rsa.pub
```

登录 gitlab.etao.net，进入 Settings -> SSH Keys，将上述公钥内容复制粘贴并保存。

3. 获取 Pintos 初始代码：

```
cd ~
git clone https://gitlab.etao.net/zjutosd/group0
```

4. 环境变量控制：

Pintos 的构建与运行脚本位于源码目录的 /pintos/src/utils 下。为了在任意目录下都能直接使用 pintos 命令，需要将其路径加入系统 PATH 变量中。

- 打开配置文件：

```
nano ~/.bashrc
```

- 在文件末尾添加以下内容：

```
export PATH=$PATH:$HOME/group0/pintos/src/utils
```

- 使配置生效：

```
source ~/.bashrc
```

- 验证环境：使用 which pintos 检查路径是否正确，并进入 /pintos/src/threads 目录尝试编译，确保 make 可正常运行

5. 关联个人仓库：

```
cd ~/group0
# 移除原有的 group0 远程连接
git remote remove origin
# 添加个人仓库地址
git remote add origin <repo-url>
# 推送代码
git push -u origin master
```

2 需求和可行性分析

2.1 需求分析

2.1.1 实验二：简单 Shell 实现需求

本实验旨在设计并实现一个具备基本交互功能的命令行解释器，具体需求如下：

1. 基础命令执行与交互：
   - Shell 需循环运行，显示提示符，读取用户输入的命令行字符串，并对其进行解析
   - 能够通过 fork() 创建子进程，并利用 execv() 加载并运行外部程序。父进程需等待子进程执行完毕后才能接收下一条指令
2. 内建命令支持：
   - pwd：显示当前工作目录的绝对路径
   - cd：切换当前工作目录。支持相对路径与绝对路径；若参数为空或为 ~，则切换至 HOME 环境变量指定的目录
   - exit：退出 Shell 程序
3. 路径解析：
   - 当用户输入不带路径的命令（如 ls ）时，Shell 需读取 PATH 环境变量，遍历其中的目录以查找对应的可执行文件
   - 约束条件：禁止直接使用 execvp，必须手动实现路径搜索逻辑并调用 execv
4. 输入/输出重定向：
   - 支持标准输出重定向符号 >，将程序输出写入指定文件（支持新建或覆盖）
   - 支持标准输入重定向符号 <，将文件内容作为程序输入
   - 需正确处理文件描述符的复制与关闭
5. 信号处理与后台运行：
   - 后台进程：支持在命令末尾添加 & 符号，使程序在后台运行，Shell 无需阻塞等待
   - 内建 wait：等待所有后台进程结束
   - 进程组管理：每个新启动的进程应拥有独立的进程组 ID 。Shell 需通过 tcsetpgrp 将前台进程组设置为当前终端的控制者，确保 CTRL-C (SIGINT) 等信号只发送给前台进程，而不误伤后台进程或 Shell 自身

2.1.2 实验四：Pintos 用户程序需求

本实验要求在 Pintos 内核基础上扩展功能，使其能够支持用户进程的加载、参数传递及系统调用。具体需求如下：

1. 参数传递：
   - 内核在加载用户程序时，需将命令行参数（如 /bin/ls -l foo）分割，并按照 x86 调用约定将参数压入用户栈
   - 栈对齐：必须确保在参数压栈后，栈指针 %esp 是 16 字节对齐的，以满足 System V ABI 标准
   - 正确设置 argv 数组、argc 以及虚假的返回地址
2. 系统调用基础架构：
   - 实现系统调用处理程序，能够捕获用户程序通过 int 0x30 指令发起的中断
   - 能够从用户栈中正确读取系统调用号及参数
3. 进程控制系统调用：
   - exec：创建子进程并执行程序。父进程必须阻塞等待，直到确认子进程成功加载可执行文件后方可返回
   - wait：父进程等待子进程结束并获取其退出状态码。需处理复杂的同步逻辑，包括父进程先于子进程退出、多次等待同一子进程等边界情况
   - exit：终止当前进程，释放资源，并打印格式化的退出信息
4. 文件操作系统调用：
   - 实现 create, remove, open, filesize, read, write, seek, tell, close 等基础文件操作
   - 文件描述符管理：内核需为每个进程维护独立的文件描述符表，将整数 fd 映射到具体的文件结构体
   - 标准输入输出：fd 0 (STDIN) 对应键盘输入，fd 1 (STDOUT) 对应控制台输出
   - 并发安全：需使用全局文件系统锁确保文件操作的线程安全性
5. 内存安全与保护：
   - 指针验证：内核在访问用户传递的指针前，必须验证其指向的地址是否位于用户空间（低于 PHYS_BASE）且已映射物理内存。对于非法指针，应终止进程并释放资源

- 可执行文件保护：在程序运行期间，禁止对其对应的可执行文件进行写入操作（ROX），防止代码被篡改

2.2 可行性分析

1. Shell 实现技术：Linux 系统提供了完善的 C 语言标准库和系统调用接口
   - fork()、exec() 系列函数提供了成熟的进程创建与替换机制
   - dup2() 函数能够高效实现文件描述符的复制，满足重定向需求
   - signal() 和 setpgid() 等接口提供了标准的信号处理与进程组管理方案
2. Pintos 实现技术：
   - Pintos 操作系统提供了线程管理、内存分页和中断处理的基础代码
   - x86 架构的内存分段与分页机制、中断处理流程均有明确的文档支持和硬件规范
   - 实验指导书详细描述了用户栈的布局结构和系统调用的 API 定义
3. 开发环境：
   - 采用 Ubuntu 18.04 LTS (Linux) 作为宿主环境，能够兼容 Pintos 所需的编译与运行工具链
   - 利用 GitLab 平台进行代码托管，便于版本控制和进度管理

3 实验二：简单 Shell 的设计与实现

3.1 总体设计思路与数据结构

3.1.1 总体设计思路

Shell 的主循环逻辑采用了以下模式：

- Read: 从标准输入读取一行命令
- Parse: 将字符串解析为参数列表（Tokens）
- Execute: 判断是否为内建命令，是则直接执行，否则创建子进程执行外部程序
- Wait: 根据是否后台运行，决定父进程是否阻塞等待

3.1.2 核心数据结构

为了管理变长的命令行参数，定义了 word_list_t 结构体用于存储解析后的 Token 列表。同时，为了实现内建命令的分发，定义了 builtin_entry_t 结构体建立命令名称与处理函数的映射关系。

```
/* Dynamic list of tokens (words) returned by the tokenizer. */
typedef struct {
    char **items;
    size_t count;
} word_list_t;
typedef int builtin_fn(word_list_t *args);
/* Builtin table */
typedef struct {
    const char *name;
    builtin_fn *fn;
    const char *doc;
} builtin_entry_t;
```

3.2 具体功能实现

3.2.1 命令解析算法

命令解析是 Shell 的基础。为了支持带引号的参数（如 "file name"），在 tokenize_line 函数中实现了一个有限状态机。状态机包含三种状态：

- MODE_NORMAL: 普通模式，遇空格分隔，遇引号切换模式
- MODE_SQ: 单引号模式，保留字面值直到遇到下一个单引号
- MODE_DQ: 双引号模式，支持转义字符，直到遇到下一个双引号

```
/* Tokenize a line into words. */
static word_list_t tokenize_line(const char *line) {
    word_list_t wl = {NULL, 0};
    if (!line) return wl;
```

```
    size_t len = strlen(line);
    char *buf = malloc(len + 1); /* temporary buffer for current token */
    size_t bi = 0;
    enum { MODE_NORMAL, MODE_SQ, MODE_DQ } mode = MODE_NORMAL;
    for (size_t i = 0; i < len; ++i) {
        char c = line[i];
        if (mode == MODE_NORMAL) {
            if (c == '\'') {
                mode = MODE_SQ;
            } else if (c == '"') {
                mode = MODE_DQ;
            } else if (c == '\\') {
                if (i + 1 < len) buf[bi++] = line[++i];
            } else if (isspace((unsigned char)c)) {
                if (bi > 0) {
                    buf[bi] = '\0';
                    wl_append(&wl, strdup(buf));
                    bi = 0;
                }
            } else {
                buf[bi++] = c;
            }
        } else if (mode == MODE_SQ) { /* single-quote: take literally until next ' */
            if (c == '\'') {
                mode = MODE_NORMAL;
            } else {
                buf[bi++] = c;
            }
        } else { /* MODE_DQ, double-quote: supports backslash to escape */
            if (c == '"') {
                mode = MODE_NORMAL;
            } else if (c == '\\') {
                if (i + 1 < len) buf[bi++] = line[++i];
            } else {
                buf[bi++] = c;
            }
        }
        /* guard, should not overflow since buf allocated len+1 */
        if (bi >= len) break;
    }
    if (bi > 0) {
        buf[bi] = '\0';
        wl_append(&wl, strdup(buf));
    }
    free(buf);
    return wl;
}
```

3.2.2 内建命令实现

内建命令直接在当前 Shell 进程中执行，不创建子进程。

1 cd：

- 调用系统调用 chdir() 切换工作目录
- 首先检查参数。如果参数为空或为 ~，则通过 getenv("HOME") 获取用户主目录并跳转；否则跳转到指定路径。若路径无效，打印 perror 错误信息

2 pwd：

- 调用 getcwd() 获取当前工作目录的绝对路径并打印到标准输出

3 exit：

- 用于退出 Shell 程序
- 在退出前，检查是否处于交互模式（interactive_shell）。如果是，则调用 tcsetattr 恢复终端的原始属性（saved_termios），确保不破坏终端状态，最后调用 exit(0)

4 wait

- 用于等待所有后台子进程结束
- 在一个 while 循环中重复调用 wait(&status)，直到返回值为 -1（表示没有子进程剩余）

5 ?:

- 打印帮助信息
- 遍历 builtins 结构体数组，输出所有支持的内建命令及其描述文档

```
static builtin_entry_t builtins[] = {
    { "?", bi_help, "display this help" },
    { "exit", bi_exit, "exit the shell" },
    { "pwd", bi_pwd, "print working directory" },
    { "cd", bi_cd, "change directory" },
    { "wait", bi_wait, "wait for background processes" },
};
/* Print help listing */
static int bi_help(word_list_t *args) {
    (void)args;
    for (size_t i = 0; i < sizeof(builtins)/sizeof(builtins[0]); ++i) {
        printf("%s - %s\n", builtins[i].name, builtins[i].doc);
    }
    return 1;
}
/* Exit the shell */
static int bi_exit(word_list_t *args) {
    (void)args;
    /* restore terminal state if interactive */
    if (interactive_shell) tcsetattr(shell_fd, TCSANOW, &saved_termios);
    exit(0);
}
/* Print working directory */
static int bi_pwd(word_list_t *args) {
    (void)args;
    char cwd[PATH_MAX];
    if (getcwd(cwd, sizeof(cwd))) {
        puts(cwd);
        return 1;
    } else {
        perror("pwd");
        return -1;
    }
}
/* Change current directory */
static int bi_cd(word_list_t *args) {
    const char *target = NULL;
    if (args->count >= 2) target = args->items[1];
    if (target == NULL || strcmp(target, "~") == 0) {
        target = getenv("HOME");
        if (!target) {
            fprintf(stderr, "cd: HOME not set\n");
            return -1;
        }
    }
    if (chdir(target) < 0) {
```

```
        perror("cd");
        return -1;
    }
    return 1;
}
/* Wait for all background children */
static int bi_wait(word_list_t *args) {
    (void)args;
    int status;
    /* reap all children; wait returns -1 when no child exists */
    while (wait(&status) > 0) { }
    return 1;
}
```

3.2.3 外部程序执行与路径解析

对于非内建命令，Shell 调用 run_external 函数。该函数首先通过 fork() 创建子进程。在子进程中，如果不含绝对路径（即不含 /），则需手动解析 PATH 环境变量。

路径解析算法：

1 获取 PATH 环境变量

2 使用 strtok_r 按 : 分割路径

3 利用 snprintf 拼接目录与命令名

4 使用 access(candidate, X_OK) 检查文件是否存在且可执行

5 若找到，调用 execv 执行；否则继续查找

```
        /* search PATH without modifying environment */
        const char *path_env = getenv("PATH");
        if (!path_env) {
            fprintf(stderr, "PATH not found\n");
            _exit(127);
        }
        char *pathdup = strdup(path_env);
        char *saveptr = NULL;
        char *entry = strtok_r(pathdup, ":", &saveptr);
        while (entry) {
            char candidate[PATH_MAX];
            snprintf(candidate, sizeof(candidate), "%s/%s", entry, argv[0]);
            if (access(candidate, X_OK) == 0) {
                execv(candidate, argv);
                perror("execv");
                free(pathdup);
                _exit(127);
            }
            entry = strtok_r(NULL, ":", &saveptr);
        }
        free(pathdup);
        fprintf(stderr, "%s: command not found\n", argv[0]);
        _exit(127);
```

3.2.4 输入/输出重定向

重定向通过在 execv 之前操作文件描述符实现。在构建参数列表时，扫描 < 和 > 符号，并使用 dup2 将文件描述符复制到标准输入（STDIN_FILENO）或标准输出（STDOUT_FILENO）。

```
    for (size_t i = 0; i < argc; ++i) {
        char *tok = wl->items[i];
        if (strcmp(tok, "<") == 0) {
            if (i + 1 < argc) {
                fd_in = open(wl->items[++i], O_RDONLY);
                if (fd_in < 0) { perror("open"); free(argv); return -1; }
            } else {
```

```
            fprintf(stderr, "syntax error: expected filename after '<'\n");
            free(argv); return -1;
        }
    } else if (strcmp(tok, ">") == 0) {
        if (i + 1 < argc) {
            fd_out = open(wl->items[++i], O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if (fd_out < 0) { perror("open"); free(argv); return -1; }
        } else {
            fprintf(stderr, "syntax error: expected filename after '>'\n");
            free(argv); return -1;
        }
    } else {
        argv[ai++] = tok;
    }
}
    /* Apply redirection in the child process. */
    if (fd_in != -1) {
        if (dup2(fd_in, STDIN_FILENO) < 0) { perror("dup2"); _exit(127); }
        close(fd_in);
    }
    if (fd_out != -1) {
        if (dup2(fd_out, STDOUT_FILENO) < 0) { perror("dup2"); _exit(127); }
        close(fd_out);
    }
```

3.2.5 信号处理与后台运行

为了防止 Shell 本身被用户输入的 CTRL-C 终止，采用了进程组管理和终端控制权转移的策略。

- 进程组隔离：子进程调用 setpgid(0, 0) 创建新的进程组，使其独立于 Shell
- 终端控制：如果是前台任务，Shell 通过 tcsetpgrp 将终端控制权移交给子进程组，待子进程结束后收回
- 后台运行：若命令末尾检测到 &，则父进程不执行 waitpid 阻塞，直接打印子进程 PID 并返回

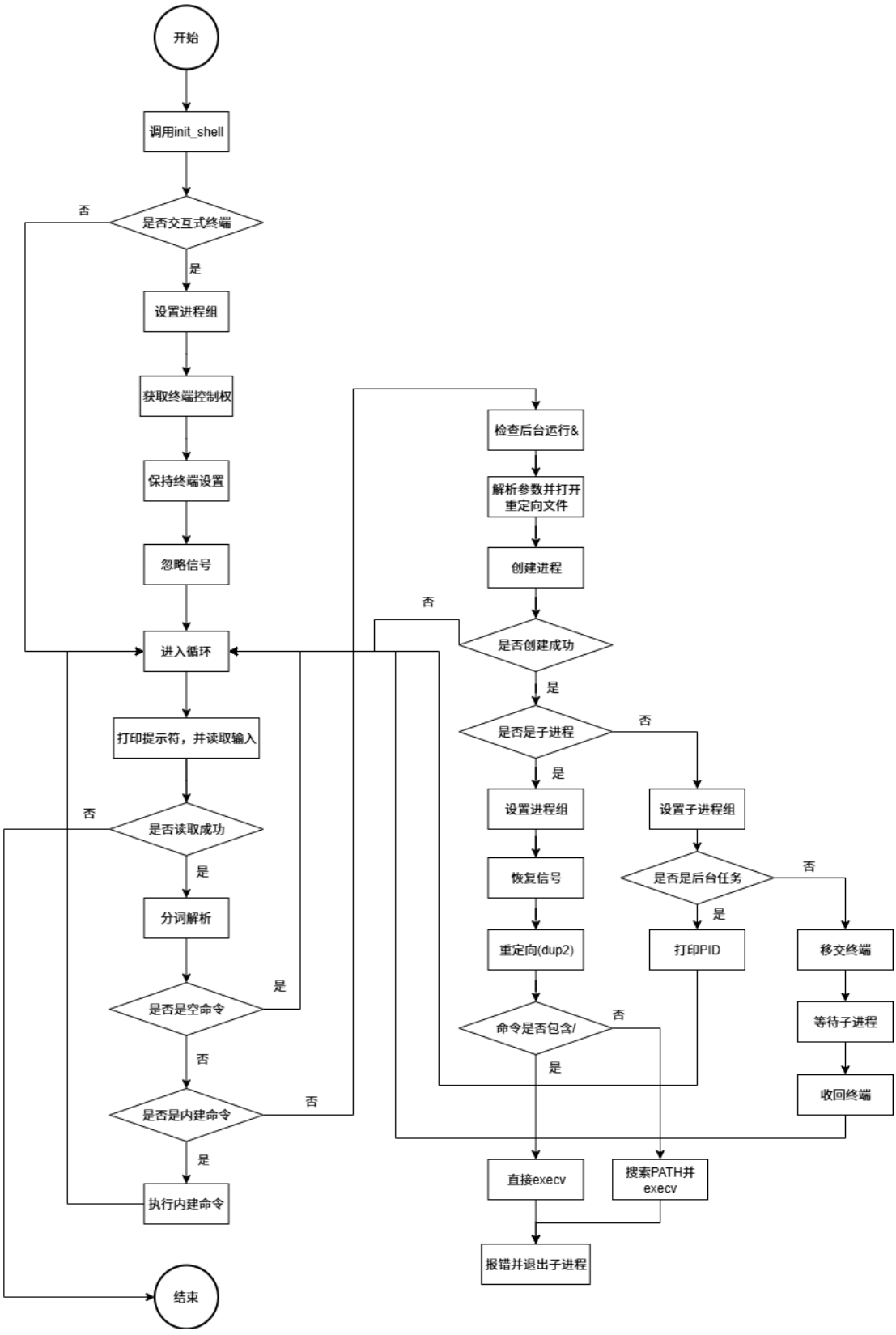
```
/* Setup basic signal handlers for the parent shell. */
static void setup_parent_signals(void) {
    struct sigaction sa;
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);
    sigaction(SIGTTOU, &sa, NULL);
}
    /* Frontend task waiting logic */
    /* set child's pgid so job control works */
    setpgid(child, child);
    if (!background) {
        /* give terminal to child pgid, wait for it, then reclaim terminal */
        tcsetpgrp(shell_fd, child);
        int status = 0;
        if (waitpid(child, &status, WUNTRACED) == -1) {
            perror("waitpid");
            free(argv);
            return -1;
        }
        /* return terminal to shell */
        tcsetpgrp(shell_fd, shell_pgid);
        if (WIFEXITED(status)) {
```

```
            int code = WEXITSTATUS(status);
            free(argv);
            return code;
        } else {
            free(argv);
            return -1;
        }
    } else {
        /* background job: do not wait here */
        printf("[bg %d]\n", (int)child);
        free(argv);
        return 1;
    }
```

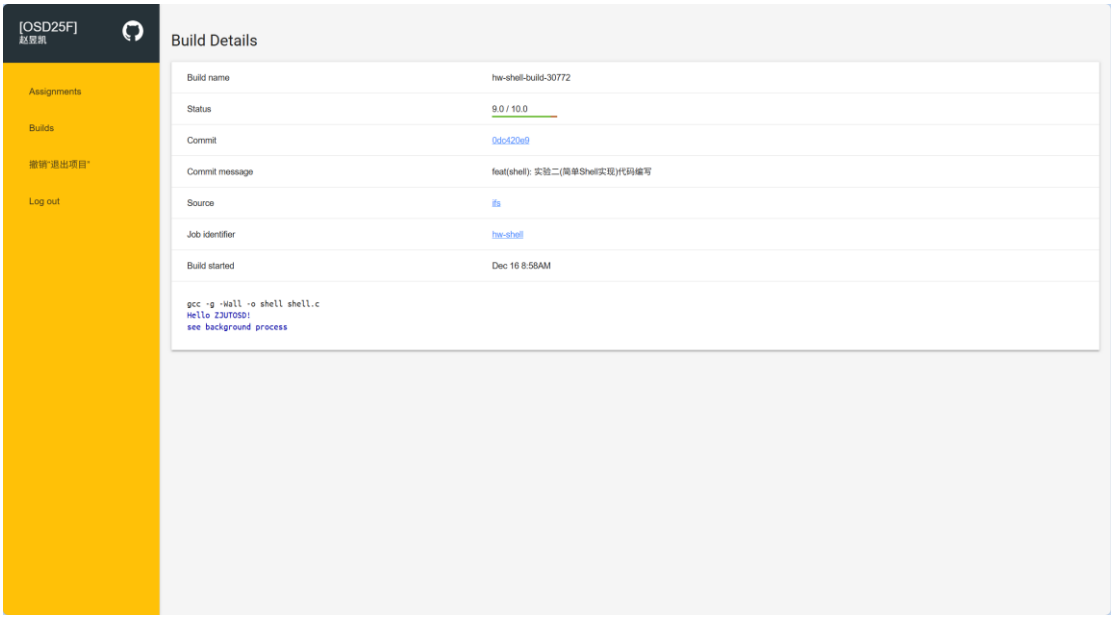
3.3 遇到的问题及解决方法

1. 路径解析时的内存破坏：
   - 在使用 strtok 解析 PATH 环境变量时，后续的函数调用偶尔会出现段错误
   - 分析得知 strtok 会修改原始字符串，且使用静态缓冲区，非线程安全且容易影响上下文
   - 改用 strtok_r 函数，并在解析前使用 strdup 复制环境变量副本，解析完成后释放副本，确保环境变量本身不被破坏
2. Shell 无法接收输入：
   - 在执行完一个前台进程后，Shell 界面卡住，无法响应键盘输入
   - 分析得知当子进程结束后，终端控制权仍停留在已退出的子进程组，Shell 位于后台，尝试读取 TTY 输入时收到了 SIGTTIN 信号导致暂停
   - 在 waitpid 返回后，立即调用 tcsetpgrp(shell_fd, shell_pgid) 将 Shell 重新设为前台进程组
3. 僵尸进程累积：
   - 多次执行后台任务后，系统中出现大量僵尸进程
   - 实现了内建命令 wait。通过 while (wait(&status) > 0); 循环回收所有已终止的子进程资源，防止僵尸进程耗尽系统 PID 资源

3.4 流程图


开始
调用init_shell
是否交互式终端
否
是
设置进程组
获取终端控制权
保持终端设置
忽略信号
进入循环
打印提示符，并读取输入
是否读取成功
否
是
分词解析
是否是空命令
是
否
是否是内建命令
否
是
执行内建命令
结束
检查后台运行&
解析参数并打开重定向文件
创建进程
是否创建成功
否
是
是否是子进程
否
是
设置进程组
恢复信号
重定向(dup2)
命令是否包含/
否
是
直接execv
设置子进程组
是否是后台任务
否
是
打印PID
移交终端
等待子进程
收回终端
搜索PATH并execv
报错并退出子进程

3.5 测试结果



4 实验四：Pintos 用户程序的设计与实现

4.1 总体设计思路与数据结构

为了支持多进程管理和系统调用，核心数据结构 struct thread 进行了扩展，引入了进程控制块 (PCB) 的概念以及用于维护父子进程关系和文件描述符的数据结构。

1. 线程结构体拓展：在 struct thread 中添加了用户进程所需的关键字段，用于管理进程状态、子进程列表和打开的文件。

```
struct thread {
  /* Owned by thread.c. */
  tid_t tid;                /* Thread identifier. */
  enum thread_status status; /* Thread state. */
  char name[16];            /* Name (for debugging purposes). */
  uint8_t* stack;           /* Saved stack pointer. */
  int priority;             /* Priority. */
  struct list_elem allelem;  /* List element for all threads list. */
  uint8_t fpu_state[108];
  /* Shared between thread.c and synch.c. */
  struct list_elem elem; /* List element. */
#ifdef USERPROG
  /* Owned by process.c. */
  struct process* pcb; /* Process control block if this thread is a userprog */
  /* Process hierarchy info. */
  tid_t parent_tid;          /* TID of the parent process. */
  struct list child_list;     /* List of child_entry structs. */
  struct child_entry *my_child_entry; /* Pointer to this thread's entry in parent's list. */
  int exit_code;                  /* Exit code to be returned to parent. */

  /* Synchronization for exec system call. */
  struct semaphore load_sema; /* Blocks parent until child loads executable. */
  bool load_success;         /* True if executable loaded successfully. */
  /* File system info. */
  struct list file_list;      /* List of open file descriptors. */
  int next_fd;               /* Next unique file descriptor to allocate. */
  struct file *executable;    /* The running executable file (denies writes). */
#endif
```

```
  /* Owned by thread.c. */
  unsigned magic; /* Detects stack overflow. */
};
```

2. 子进程管理结构：为了实现 wait 系统调用，父进程必须能够获取已退出子进程的状态。定义 struct child_entry 用于在父进程中记录每个子进程的生命周期状态。

```
/* Records information about a child process. */
struct child_entry {
    tid_t tid;                /* Child process TID. */
    int exit_status;          /* Exit status passed to exit(). */
    bool is_exit;             /* True if the child has exited. */
    bool wait;                /* True if the parent has successfully waited. */
    struct semaphore wait_sema; /* Semaphore for parent to wait on child exit. */
    struct list_elem elem;    /* List element for parent's child_list. */
};
```

3. 文件描述符结构：由于 Pintos 默认不提供文件描述符到文件对象的映射。故定义了 struct file_desc 来实现这一映射，每个进程维护自己的 FD 列表。

```
/* Represents an open file descriptor. */
struct file_desc {
    int fd;                   /* File descriptor number. */
    struct file *file;        /* Pointer to the open file. */
    struct list_elem elem;    /* List element for thread's file_list. */
};
```

4.2 具体功能实现

4.2.1 参数传递

参数传递的核心算法在 push_arguments 函数中实现。该函数负责将命令行字符串分割，并按照 x86 调用约定将参数压入用户栈。

1. 解析参数：使用 strtok_r 将命令行字符串分割为独立的参数字符串（如 ls, -l），存入临时数组 argv
2. 压入字符串内容：遍历 argv，将每个字符串的内容（包括 \0）压入栈中，并记录每个字符串在栈中的地址
3. 栈对齐：计算当前栈指针的位置，压入适当数量的填充字节（padding），确保后续压入指针后的 %esp 是 16 字节对齐的
4. 压入指针数组：
   - 压入 NULL 指针（argv[argc]）
   - 逆序压入指向各参数字符串的指针（argv[i]）
   - 压入 argv 数组的首地址（char **）
5. 压入元数据：压入 argc（参数个数）和虚假的返回地址（0）

```
/* Populates the user stack with command-line arguments.
   This function parses FILE_NAME, pushes the argument strings onto the stack,
   aligns the stack pointer, and then pushes the pointers (argv) and metadata (argc). */
void push_arguments(const char *file_name, void **esp) {
    char *argv[128]; // Array to hold pointers to argument strings
    int argc = 0;
    char *token, *save_ptr;

    /* Parse the command line into individual arguments. */
    for (token = strtok_r((char *)file_name, " ", &save_ptr); token != NULL; token = strtok_r(NUL
L, " ", &save_ptr)) {
        argv[argc++] = token;
    }
    /* Used to store the stack address of each string after pushing it */
    uint32_t argv_addrs[128];
    /* Step 1: Push the argument strings (the actual data) onto the stack. */
    for (int i = argc - 1; i >= 0; i--) {
```

```
        size_t len = strlen(argv[i]) + 1;
        *esp = (void *)((char *)*esp - len);
        memcpy(*esp, argv[i], len);
        argv_addrs[i] = (uint32_t)*esp;
    }
    /* Step 2: Calculate alignment. */
    int ptr_size = (argc + 4) * 4;
    uintptr_t current_esp = (uintptr_t)*esp;
    int padding = 0;

    /* Calculate necessary padding to ensure 16-byte stack alignment. */
    while ((current_esp - padding - ptr_size) % 16 != 12) {
        padding++;
    }

    /* Push padding bytes (word alignment). */
    *esp = (void *)((char *)*esp - padding);
    memset(*esp, 0, padding);
    /* Push argv[argc] (Null Sentinel). Required by C standard. */
    *esp = (void *)((uint32_t *)*esp - 1);
    *((uint32_t *)*esp) = 0;
    /* Push argv pointers (argv[0]...argv[argc-1]) in reverse order. */
    for (int i = argc - 1; i >= 0; i--) {
        *esp = (void *)((uint32_t *)*esp - 1);
        *((uint32_t *)*esp) = argv_addrs[i];
    }
    /* Push pointer to argv (the address of argv[0] we just pushed). */
    uint32_t argv_ptr = (uint32_t)*esp;
    *esp = (void *)((uint32_t *)*esp - 1);
    *((uint32_t *)*esp) = argv_ptr;
    /* Push argc. */
    *esp = (void *)((uint32_t *)*esp - 1);
    *((int *)*esp) = argc;
    /* Push fake return address (0). */
    *esp = (void *)((uint32_t *)*esp - 1);
    *((uint32_t *)*esp) = 0;
}
```

4.2.2 内存访问安全

为了防止用户程序通过传递非法指针（如指向内核空间或未映射区域）导致内核崩溃，在以下函数中实现了严格的内存检查机制。

- check_ptr_and_size(vaddr, size)：检查指针是否非空、是否属于用户虚拟地址空间（is_user_vaddr）、以及是否已映射物理内存（pagedir_get_page）
- check_string(str)：专门用于检查以 null 结尾的字符串，逐字节验证直到字符串结束

```
/* Validates that a memory region [vaddr, vaddr + size) is safe for user access. */
static void check_ptr_and_size(const void* vaddr, unsigned size) {
  /* Check for NULL pointer. */
  if (vaddr == NULL) sys_exit(-1);

  /* Check the start address. */
  if (!is_user_vaddr(vaddr) || pagedir_get_page(thread_current()->pcb->pagedir, vaddr) == NULL)
 {
    sys_exit(-1);
  }
  /* Check the end address. */
  if (size > 0) {
```

```
    void* end_addr = (void*)((uint8_t*)vaddr + size - 1);
    if (!is_user_vaddr(end_addr) || pagedir_get_page(thread_current()->pcb->pagedir, end_addr) ==
= NULL) {
      sys_exit(-1);
    }
  }
}
/* Validates a C-style string (null-terminated). */
static void check_string(const char* str) {
  /* Check the first character. */
  check_ptr_and_size(str, 1);
  /* Iterate through the string, validating every subsequent character
     to ensure the string does not run off into invalid memory. */
  while (*str != '\0') {
    str++;
    check_ptr_and_size(str, 1);
  }
}
/* Helper to extract arguments from the stack.
  f->esp points to the system call number.
  Arguments follow immediately after (esp+4, esp+8, etc.). */
static void get_args(struct intr_frame *f, int *args, int n) {
  int *ptr;
  for (int i = 0; i < n; i++) {
    /* Calculate the address of the i-th argument.
       (int *)f->esp casts the void pointer to int pointer.
       +1 skips the syscall number itself.
       +i moves to the i-th argument. */
    ptr = (int *)f->esp + i + 1;
    /* Validate that the stack pointer itself is in valid memory
       before we try to dereference it. */
    check_ptr_and_size((void *)ptr, sizeof(int));
    /* Dereference to get the value. */
    args[i] = *ptr;
  }
}
```

4.2.3 系统调用处理框架

系统调用通过中断 0x30 触发。syscall_handler 函数作为总入口，从栈指针 esp 处读取系统调用号，并分发到具体的处理逻辑。

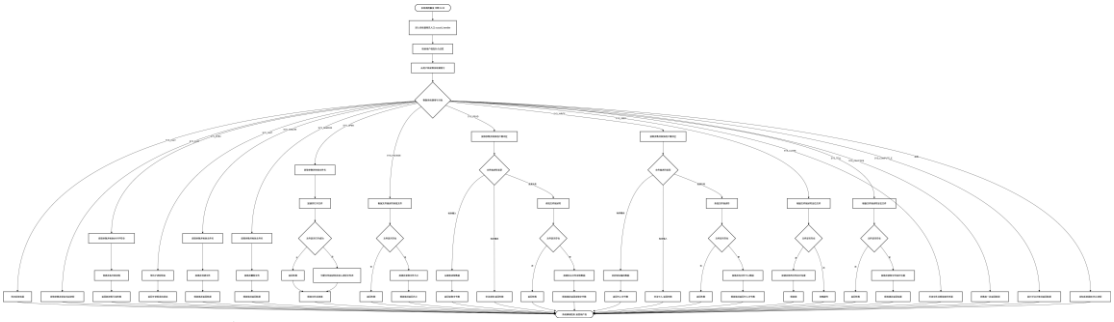
```
/* Main handler for system calls. */
static void syscall_handler(struct intr_frame* f) {
  /* Validate the stack pointer. */
  check_ptr_and_size(f->esp, sizeof(int));
  /* Read the system call number from the stack. */
  int syscall_num = *(int*)(f->esp);
  /* Buffer to hold arguments popped from the stack. */
  int args[3];
  struct thread* cur = thread_current();
  switch (syscall_num) {
    case SYS_HALT:
      /* Shut down the system completely. */
      shutdown_power_off();
      break;
    case SYS_EXIT:
      /* Exit the current process with a status code. */
      get_args(f, args, 1);
      sys_exit(args[0]);
```

```
    break;
  case SYS_EXEC:
    /* Execute a new process. */
    get_args(f, args, 1);
    const char* cmd_line = (const char*)args[0];
    /* Validate the command line string before reading it. */
    check_string(cmd_line);
    /* f->eax holds the return value (new process ID or -1). */
    f->eax = process_execute(cmd_line);
    break;
  case SYS_WAIT:
  /* Wait for a child process to exit. */
    get_args(f, args, 1);
    f->eax = process_wait((pid_t)args[0]);
    break;
  case SYS_CREATE:
    /* Create a new file. */
    get_args(f, args, 2);
    check_string((const char*)args[0]);
    /* File system operations must be synchronized. */
    lock_acquire(&filesys_lock);
    f->eax = filesys_create((const char*)args[0], (unsigned)args[1]);
    lock_release(&filesys_lock);
    break;
  case SYS_REMOVE:
    /* Delete a file. */
    get_args(f, args, 1);
    check_string((const char*)args[0]);
    lock_acquire(&filesys_lock);
    f->eax = filesys_remove((const char*)args[0]);
    lock_release(&filesys_lock);
    break;
  case SYS_OPEN:
    /* Open a file. */
    get_args(f, args, 1);
    const char* name_open = (const char*)args[0];
    check_string(name_open);
    lock_acquire(&filesys_lock);
    struct file* file = filesys_open(name_open);

    if (file == NULL) {
      f->eax = -1;
    } else {
      /* If open successful, create a file descriptor (FD) wrapper. */
      struct file_desc* fd_struct = malloc(sizeof(struct file_desc));
      if (fd_struct == NULL) {
        file_close(file);
        f->eax = -1;
      } else {
        fd_struct->file = file;
        /* Assign a unique FD. */
        fd_struct->fd = cur->next_fd++;
        list_push_back(&cur->file_list, &fd_struct->elem);
        f->eax = fd_struct->fd;
      }
    }
    lock_release(&filesys_lock);
```

```
    break;
  case SYS_FILESIZE:
  /* Return the size of an open file. */
    get_args(f, args, 1);
    struct file_desc* fd_s = get_file_desc(args[0]);

    if (fd_s == NULL) {
       f->eax = -1;
    } else {
       lock_acquire(&filesys_lock);
       f->eax = file_length(fd_s->file);
       lock_release(&filesys_lock);
    }
    break;
  case SYS_READ:
  /* Read from a file or stdin. */
    get_args(f, args, 3);
    int fd_read = args[0];
    void* buffer_read = (void*)args[1];
    unsigned size_read = (unsigned)args[2];
    /* Security Check: Verify the buffer is valid user memory. */
    check_ptr_and_size(buffer_read, size_read);

    if (fd_read == 0) {
      /* FD 0 is STDIN (Keyboard input). */
      uint8_t* buf = (uint8_t*)buffer_read;
      for (unsigned i = 0; i < size_read; i++) {
        buf[i] = input_getc();
      }
      f->eax = size_read;
    } else if (fd_read == 1) {
      /* FD 1 is STDOUT. Reading from STDOUT is invalid. */
      f->eax = -1;
    } else {
      /* Read from a regular file. */
      struct file_desc* fd_struct = get_file_desc(fd_read);
      if (fd_struct == NULL) {
        f->eax = -1;
      } else {
        lock_acquire(&filesys_lock);
        f->eax = file_read(fd_struct->file, buffer_read, size_read);
        lock_release(&filesys_lock);
      }
    }
    break;
  case SYS_WRITE:
    /* Write to a file or stdout. */
    get_args(f, args, 3);
    int fd_write = args[0];
    const void* buffer_write = (const void*)args[1];
    unsigned size_write = (unsigned)args[2];
    check_ptr_and_size(buffer_write, size_write);
    if (fd_write == 1) {
      /* FD 1 is STDOUT (Console output). */
      putbuf(buffer_write, size_write);
      f->eax = size_write;
    } else if (fd_write == 0) {
```

```
      /* FD 0 is STDIN. Writing to STDIN is invalid. */
      f->eax = -1;
    } else {
      /* Write to a regular file. */
      struct file_desc* fd_struct = get_file_desc(fd_write);
      if (fd_struct == NULL) {
        f->eax = -1;
      } else {
        lock_acquire(&filesys_lock);
        f->eax = file_write(fd_struct->file, buffer_write, size_write);
        lock_release(&filesys_lock);
      }
    }
    break;
  case SYS_SEEK:
    /* Change position in a file. */
    get_args(f, args, 2);
    struct file_desc* fd_struct_seek = get_file_desc(args[0]);
    if (fd_struct_seek != NULL) {
        lock_acquire(&filesys_lock);
        file_seek(fd_struct_seek->file, (unsigned)args[1]);
        lock_release(&filesys_lock);
    }
    break;
  case SYS_TELL:
    /* Report current position in a file. */
    get_args(f, args, 1);
    struct file_desc* fd_struct_tell = get_file_desc(args[0]);
    if (fd_struct_tell == NULL) {
        f->eax = -1;
    } else {
        lock_acquire(&filesys_lock);
        f->eax = file_tell(fd_struct_tell->file);
        lock_release(&filesys_lock);
    }
    break;
  case SYS_CLOSE:
    /* Close a file descriptor. */
    get_args(f, args, 1);
    close_file_by_fd(args[0]);
    break;
  case SYS_PRACTICE:
    /* Simple test syscall: returns arg + 1. */
    get_args(f, args, 1);
    f->eax = args[0] + 1;
    break;

  /* Handler for floating point computation. */
  case SYS_COMPUTE_E:
    get_args(f, args, 1);
    f->eax = sys_compute_e(args[0]);
    break;
  default:
    printf("Unknown syscall: %d\n", syscall_num);
    sys_exit(-1);
 }
}
```

流程图如下：



4.2.4 进程控制系统调用

1. Exec (执行新进程)： 在 syscall_exec 中调用 process_execute。为了保证父进程能得知子进程是否加载成功，使用了同步机制：
   - 父进程调用 process_execute 创建子进程，并在 exec_aux.load_done 信号量上阻塞 (sema_down)
   - 子进程在 start_process 中加载可执行文件。加载完成后，设置 exec_aux.success 标志，并唤醒父进程 (sema_up)
   - 父进程被唤醒后，检查成功标志，返回子进程 PID 或 -1

```
/* Starts a new thread running a user program loaded from
   FILENAME. The new thread is created, and this function
   waits for the child process to successfully load the executable.
   Returns the new process's process id (pid), or TID_ERROR if the
   thread cannot be created or if loading the executable fails. */
pid_t process_execute(const char* file_name) {
  char* fn_copy;
  tid_t tid;
  /* Parse the thread name (executable name) from the command line. */
  char *thread_name = palloc_get_page(0);
  if (thread_name == NULL) return TID_ERROR;
  strlcpy(thread_name, file_name, PGSIZE);
  char *save_ptr;
  /* Extract the first token (the program name) to use as the thread name. */
  char *executable_name = strtok_r(thread_name, " ", &save_ptr);
  /* Make a copy of FILE_NAME.
     Otherwise there's a race between the caller and load(). */
  fn_copy = palloc_get_page(0);
  if (fn_copy == NULL) {
    palloc_free_page(thread_name);
    return TID_ERROR;
  }
  strlcpy(fn_copy, file_name, PGSIZE);
  /* Prepare synchronization structure to wait for the child process to load. */
  struct exec_info exec_aux;
  exec_aux.file_name = fn_copy;
  sema_init(&exec_aux.load_done, 0);
  exec_aux.success = false;
  /* Create the thread using executable_name as the thread name,
     passing exec_aux as auxiliary arguments. */
  tid = thread_create(executable_name, PRI_DEFAULT, start_process, &exec_aux);

  /* Free the temporary buffer used for parsing the thread name. */
  palloc_free_page(thread_name);
  if (tid == TID_ERROR) {
    /* If thread creation failed, we must free the copy of the filename here. */
    palloc_free_page(fn_copy);
    return TID_ERROR;
```

```
  }
  /* Wait for the child process to complete loading. */
  sema_down(&exec_aux.load_done);
  /* If loading failed, return TID_ERROR. */
  if (!exec_aux.success) {
      return TID_ERROR;
  }
  return tid;
}
/* A thread function that loads a user process and starts it
   running. */
static void start_process(void* aux) {
  struct exec_info *info = (struct exec_info *)aux;
  char* file_name = info->file_name;
  struct thread* t = thread_current();
  struct intr_frame if_;
  bool success, pcb_success;
  /* Extract the actual executable name from the command line for use in load(). */
  char *cmd_copy = palloc_get_page(0);
  if (cmd_copy == NULL) {
      info->success = false;
      sema_up(&info->load_done);
      thread_exit();
  }
  strlcpy(cmd_copy, file_name, PGSIZE);
  char *save_ptr;
  char *real_name = strtok_r(cmd_copy, " ", &save_ptr);
  /* Allocate process control block */
  struct process* new_pcb = malloc(sizeof(struct process));
  success = pcb_success = new_pcb != NULL;
  /* Initialize process control block */
  if (success) {
    // Ensure that timer_interrupt() -> schedule() -> process_activate()
    // does not try to activate our uninitialized pagedir
    new_pcb->pagedir = NULL;
    t->pcb = new_pcb;
    // Continue initializing the PCB as normal
    t->pcb->main_thread = t;
    strlcpy(t->pcb->process_name, t->name, sizeof t->name);
  }
  /* Initialize interrupt frame and load executable. */
  if (success) {
    memset(&if_, 0, sizeof if_);
    if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
    if_.cs = SEL_UCSEG;
    if_.eflags = FLAG_IF | FLAG_MBS;
    success = load(real_name, &if_.eip, &if_.esp);
  }
  /* Free the temporary buffer used for parsing the executable name. */
  palloc_free_page(cmd_copy);
  /* Handle failure with succesful PCB malloc. Must free the PCB */
  if (!success && pcb_success) {
    // Avoid race where PCB is freed before t->pcb is set to NULL
    // If this happens, then an unfortuantely timed timer interrupt
    // can try to activate the pagedir, but it is now freed memory
    struct process* pcb_to_free = t->pcb;
    t->pcb = NULL;
```

```
    free(pcb_to_free);
  }
  /* If loading was successful, set up the stack with arguments. */
  if (success) {
     push_arguments(file_name, &if_.esp);
  }
  /* Notify the parent process of the load status (success or failure). */
  info->success = success;
  sema_up(&info->load_done);
  /* Clean up. Free the file_name copy passed from process_execute. */
  palloc_free_page(file_name);
  if (!success) {
    thread_exit();
  }
  /* Start the user process by simulating a return from an
     interrupt, implemented by intr_exit (in
     threads/intr-stubs.S).  Because intr_exit takes all of its
     arguments on the stack in the form of a `struct intr_frame',
     we just point the stack pointer (%esp) to our stack frame
     and jump to it. */
  asm volatile("movl %0, %%esp; jmp intr_exit" : : "g"(&if_) : "memory");
  NOT_REACHED();
}
```

2. Wait (等待子进程)： process_wait 实现了复杂的父子进程同步：
   - 遍历当前线程的 child_list 查找对应 tid 的 child_entry
   - 如果未找到或已等待过，直接返回 -1
   - 如果找到，调用 sema_down(&child->wait_sema) 阻塞，直到子进程退出
   - 子进程退出时，会将退出码写入 child_entry 并 sema_up
   - 父进程被唤醒，读取退出码，释放 child_entry 内存，返回状态

```
/* Waits for process with PID child_pid to die and returns its exit status.
   If it was terminated by the kernel (i.e. killed due to an
   exception), returns -1.  If child_pid is invalid or if it was not a
   child of the calling process, or if process_wait() has already
   been successfully called for the given PID, returns -1
   immediately, without waiting.
   This function will be implemented in problem 2-2.  For now, it
   does nothing. */
int process_wait(pid_t child_pid) {
  /* Retrieve the current thread (the parent process). */
  struct thread *cur = thread_current();
  /* Iterator for traversing the child list. */
  struct list_elem *e;
  /* Pointer to store the found child entry, initialized to NULL. */
  struct child_entry *child = NULL;
  /* Iterate through the current thread's list of children to find the
     one matching child_pid. Only direct children are valid. */
  for (e = list_begin(&cur->child_list); e != list_end(&cur->child_list); e = list_next(e)) {
     struct child_entry *entry = list_entry(e, struct child_entry, elem);
     if (entry->tid == child_pid) {
        child = entry;
        break;
     }
  }
  /* If the child was not found (not a direct child) or if we have
     already waited on this child, return -1. */
  if (child == NULL || child->wait) {
     return -1;
```

```
  }
  /* Mark the child as "waited on" to prevent double waiting. */
  child->wait = true;
  /* Wait for the child process to exit.
     This blocks the current thread until the child signals 'wait_sema'. */
  sema_down(&child->wait_sema);
  /* Retrieve the exit status stored by the child. */
  int status = child->exit_status;
  /* Remove the child entry from the list and free the resource.
     This is necessary to prevent memory leaks now that we have the status. */
  list_remove(&child->elem);
  free(child);
  return status;
}
```

3. Exit (进程退出)： process_exit 负责清理资源：
   - 关闭所有打开的文件和当前可执行文件
   - 如果父进程存在，更新父进程持有的 child_entry（设置 exit_status 和 is_exit），并 sema_up 唤醒父进程
   - 释放所有孤儿进程的 child_entry
   - 销毁页表并释放 PCB

```
/* Free the current process's resources. */
void process_exit(void) {
  /* Get the current running thread. */
  struct thread* cur = thread_current();
  uint32_t* pd;

  /* Close the executable file of the current process. */
  if (cur->executable != NULL) {
     lock_acquire(&filesys_lock);
     file_close(cur->executable);
     lock_release(&filesys_lock);
     cur->executable = NULL;
  }
  /* Close all files opened by this process. */
  while (!list_empty(&cur->file_list)) {
     struct list_elem *e = list_pop_front(&cur->file_list);
     struct file_desc *fd = list_entry(e, struct file_desc, elem);
     lock_acquire(&filesys_lock);
     file_close(fd->file);
     lock_release(&filesys_lock);
     free(fd);
  }
  /* Update the child_entry information in the Parent process.
     If this process has a parent waiting for it (or tracking it),
     record the exit status and signal the parent to wake up. */
  if (cur->my_child_entry != NULL) {
     cur->my_child_entry->exit_status = cur->exit_code;
     cur->my_child_entry->is_exit = true;
     /* Wake up the parent process if it is waiting in process_wait(). */
     sema_up(&cur->my_child_entry->wait_sema);
  }
  /* Free all child process entries owned by this process.
     These are the "children" that THIS process created. Since we are dying,
     we can no longer wait for them, so we free the metadata (orphaning them). */
  while (!list_empty(&cur->child_list)) {
     struct list_elem *e = list_pop_front(&cur->child_list);
     struct child_entry *child = list_entry(e, struct child_entry, elem);
```

```
    free(child);
  }
  /* Sanity check: If there is no PCB (e.g., a kernel thread), just exit. */
  if (cur->pcb == NULL) {
    thread_exit();
    NOT_REACHED();
  }
  /* Destroy the page directory. */
  pd = cur->pcb->pagedir;
  if (pd != NULL) {
    cur->pcb->pagedir = NULL;
    pagedir_activate(NULL);
    pagedir_destroy(pd);
  }
  /* Free the Process Control Block (PCB). */
  struct process* pcb_to_free = cur->pcb;
  cur->pcb = NULL;
  free(pcb_to_free);
  /* Final thread destruction. */
  thread_exit();
}
```

4.2.5 文件操作系统调用

所有文件操作均通过全局锁 filesys_lock 进行保护，以确保线程安全。

- 文件描述符管理：sys_open 成功打开文件后，会创建一个 struct file_desc，分配唯一的 fd（从 2 开始递增），并将其添加到当前线程的 file_list 中。get_file_desc(fd) 辅助函数用于根据 fd 查找对应的文件对象
- 读写操作：SYS_READ 和 SYS_WRITE 需特殊处理 fd 0 (STDIN) 和 fd 1 (STDOUT)。对于普通文件，则调用文件系统接口 file_read / file_write
- ROX (Run-on-eXecutable)：在 load 函数中，调用 file_deny_write(file) 锁定当前运行的可执行文件，防止其在运行期间被修改

```
/* Loads an ELF executable from FILE_NAME into the current thread.
   Stores the executable's entry point into *EIP
   and its initial stack pointer into *ESP.
   Returns true if successful, false otherwise. */
bool load(const char* file_name, void (**eip)(void), void** esp) {
  struct thread* t = thread_current();
  struct Elf32_Ehdr ehdr;
  struct file* file = NULL;
  off_t file_ofs;
  bool success = false;
  int i;
  /* Allocate and activate page directory. */
  t->pcb->pagedir = pagedir_create();
  if (t->pcb->pagedir == NULL)
    goto done;
  process_activate();
  /* Open the executable file. */
  lock_acquire(&filesys_lock);
  file = filesys_open(file_name);
  if (file == NULL) {
    printf("load: %s: open failed\n", file_name);
    goto done;
  }
  /* Open executable file. */
  file = filesys_open(file_name);
  if (file == NULL) {
```

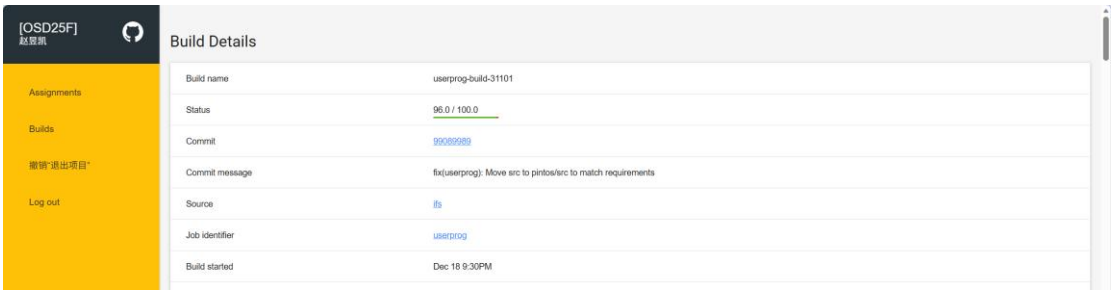
```
  printf("load: %s: open failed\n", file_name);
  goto done;
 }
 /* Deny write access to the executable. */
 file_deny_write(file);
 t->executable = file;
 /* Read and verify executable header. */
 if (file_read(file, &ehdr, sizeof ehdr) != sizeof ehdr ||
    memcmp(ehdr.e_ident, "\177ELF\1\1\1", 7) || ehdr.e_type != 2 || ehdr.e_machine != 3 ||
    ehdr.e_version != 1 || ehdr.e_phentsize != sizeof(struct Elf32_Phdr) || ehdr.e_phnum > 1024) {
  printf("load: %s: error loading executable\n", file_name);
  goto done;
 }
 /* Read program headers. */
 file_ofs = ehdr.e_phoff;
 for (i = 0; i < ehdr.e_phnum; i++) {
  struct Elf32_Phdr phdr;
  if (file_ofs < 0 || file_ofs > file_length(file))
    goto done;
  file_seek(file, file_ofs);
  if (file_read(file, &phdr, sizeof phdr) != sizeof phdr)
    goto done;
  file_ofs += sizeof phdr;
  switch (phdr.p_type) {
    case PT_NULL:
    case PT_NOTE:
    case PT_PHDR:
    case PT_STACK:
    default:
      /* Ignore this segment. */
      break;
    case PT_DYNAMIC:
    case PT_INTERP:
    case PT_SHLIB:
      goto done;
    case PT_LOAD:
      if (validate_segment(&phdr, file)) {
        bool writable = (phdr.p_flags & PF_W) != 0;
        uint32_t file_page = phdr.p_offset & ~PGMASK;
        uint32_t mem_page = phdr.p_vaddr & ~PGMASK;
        uint32_t page_offset = phdr.p_vaddr & PGMASK;
        uint32_t read_bytes, zero_bytes;
        if (phdr.p_filesz > 0) {
          /* Normal segment.
                Read initial part from disk and zero the rest. */
          read_bytes = page_offset + phdr.p_filesz;
          zero_bytes = (ROUND_UP(page_offset + phdr.p_memsz, PGSIZE) - read_bytes);
        } else {
          /* Entirely zero.
                Don't read anything from disk. */
          read_bytes = 0;
          zero_bytes = ROUND_UP(page_offset + phdr.p_memsz, PGSIZE);
        }
        if (!load_segment(file, file_page, (void*)mem_page, read_bytes, zero_bytes, writable))
          goto done;
      } else
        goto done;
```

```
      break;
    }
  }
  /* Set up stack. */
  if (!setup_stack(esp))
    goto done;
  /* Start address. */
  *eip = (void (*)(void))ehdr.e_entry;
  success = true;
done:
  if (!success && file != NULL) {
     file_close(file);
  }
  /* Release the global file system lock. */
  lock_release(&filesys_lock);
  return success;
}
```

4.3 遇到的问题及解决方法

1. Wait 系统调用的同步死锁：
   - 父进程调用 wait 时，如果子进程已经退出，父进程会无限阻塞
   - 引入 struct child_entry 持久化存储子进程状态。即使子进程已销毁，其 child_entry 仍保留在父进程的列表中。process_wait 检查的是 child_entry 中的信号量，而非子进程本身是否存在
2. 多级指针的参数压栈错误：
   - 用户程序接收到的 argv 内容错乱
   - 在 push_arguments 中严格按照顺序：先压入字符串内容 -> 记录地址 -> 栈对齐 -> 压入地址指针。注意 argv 数组本身也是一个指针，需要二级间接引用
3. 内存泄漏问题：
   - 大量进程运行后系统内存耗尽
   - 在 process_exit 中完善了资源释放逻辑，确保关闭所有文件描述符，并正确释放 child_list 中所有未被等待的子进程条目

4.4 测试结果



5 总结与展望

5.1 总结

本次课程设计通过两个实验——简单 Shell 实现与 Pintos 用户程序，从用户态应用层到内核态底层，全面实践了操作系统的核心原理。

1. 实验二（简单 Shell 实现）总结：
   - 成功实现了一个具备交互能力的命令行解释器。通过 fork() 和 execv() 的配合，实现了外部程序的加载与执行；通过 waitpid() 和进程组管理，实现了前后台任务的调度与控制
   - 利用 dup2() 系统调用，成功实现了标准输入/输出的重定向功能（<, >），深入理解了 Linux 文件描述符表的工作机制

- 通过 signal() 和 tcsetpgrp()，实现了对 SIGINT (Ctrl-C) 等信号的精确控制，确保 Shell 自身不被意外终止，同时能够正确管理子进程的生命周期
- 实现了一个基于状态机的分词器，能够正确处理带引号的参数和特殊字符

2. 实验四（Pintos 用户程序）总结：
   - 在内核中实现了精细的参数传递逻辑。严格遵循 x86 调用约定，完成了参数字符串的压栈、指针数组构建以及栈指针的 16 字节对齐，确保了用户程序能正确接收 argc 和 argv
   - 搭建了通用的系统调用处理程序（Syscall Handler），实现了从 SYS_HALT 到 SYS_CLOSE 等一系列核心调用。特别是 exec 和 wait 的实现，解决了父子进程间的同步问题（通过信号量机制）和资源回收问题（通过 child_entry 结构）
   - 在内核层面对用户指针进行了严格的合法性检查，防止了空指针、越界访问及内核空间访问导致的系统崩溃
   - 通过全局锁机制解决了文件系统的并发竞争问题，并实现了文件描述符（fd）到内核文件对象的映射管理，以及 ROX（禁止写入正在运行的可执行文件）机制

5.2 不足

1. Shell 部分：
   - 当前 Shell 尚未支持管道操作（|），无法实现多命令级联处理；缺乏历史记录（History）和 Tab 键自动补全功能
   - 目前仅支持单行命令交互，不支持复杂的 Shell 脚本控制流
2. Pintos 部分：
   - 当前的文件系统操作依赖于一把粗粒度的全局锁 (filesys_lock)，，无法充分利用多核并发优势。未来应改进为更细粒度的 inode 锁或读写锁
   - wait 实现中使用了链表遍历查找子进程，当子进程数量巨大时，查找效率为 O(N)。在进程量大的系统中，可优化为哈希表结构
   - 目前尚未实现虚拟内存的换页（Swapping）机制，用户栈大小固定且无法动态增长，限制了大型程序的运行

5.3 展望

计划从以下几个角度对系统进行优化：

- 实现管道机制：在 Shell 中引入 pipe() 系统调用，支持多进程间的流式通信
- 优化内核同步：在 Pintos 中移除文件系统的全局锁，采用更高效的同步原语来实现细粒度的并发控制
- 完善虚拟内存：实现请求分页和内存映射文件，解除对物理内存大小的硬性依赖，提高系统的多道程序设计能力

6 工程能力与目标达成

6.1 专业知识与基础理论的深度理解

在理论层面，操作系统课程中抽象的概念在代码实践中得到了具象化验证。

- 进程管理与特权级切换：通过 Shell 实验中 fork() 与 execv() 的配合使用，我深刻理解了父子进程的创建、内存空间的复制以及用户态程序的加载过程 。而在 Pintos 实验中，通过实现 int 0x30 中断处理程序 syscall_handler，我从内核态视角重新审视了特权级切换，理解了系统调用是如何作为用户程序与内核之间的桥梁，通过寄存器（%eax, %esp）传递参数并返回结果的
- 并发与同步机制：在解决 Pintos 父子进程同步问题时，我深入应用了信号量和锁机制。理论课上关于“临界区”和“原子操作”的描述，在实现文件系统全局锁 filesys_lock 以及进程等待信号量 wait_sema 时转化为具体的代码逻辑，使我明白了如何防止竞争条件导致的内核崩溃

6.2 系统建模与运行机制推演

针对复杂工程问题，我学会了建立抽象模型来推演系统的运行状态。

- 有限状态机建模：在 Shell 的命令解析模块，针对带引号参数（如 "file name"）

的复杂输入，我建立了一个包含 MODE_NORMAL、MODE_SQ（单引号）、MODE_DQ（双引号）三种状态的有限状态机模型。通过推演字符流在不同状态间的跳转逻辑，成功解决了转义字符和空格处理的难题，验证了状态机模型在文本解析中的有效性

- 进程生命周期建模：在 Pintos 的 wait 系统调用设计中，我面临“父进程如何获取已退出子进程状态”的难题。为此，我构建了 struct child_entry 结构体模型，将“子进程的生命周期管理”与“子进程的执行实体”解耦。通过推演发现，即使子进程线程销毁，其驻留在父进程链表中的 child_entry 依然可以作为状态容器，从而完美模拟了 Linux 中父进程回收僵尸进程的机制

6.3 基本原理的应用与解决方案的多样性

在解决工程问题时，我尝试利用基本原理探索多种解决方案，并进行权衡分析。

- 参数传递的多样性处理：在实现参数压栈时，我利用 x86 栈帧结构原理，设计了严谨的内存对齐方案。面对参数分割问题，我利用 strtok_r 替代 strtok，虽然增加了代码复杂度，但通过分析线程安全性原理，避免了全局静态缓冲区可能导致的数据污染
- 同步机制的选择：在文件操作中，我选择了粗粒度的全局锁 filesys_lock 来快速实现线程安全；而在 exec 进程同步中，为了实现父进程阻塞等待子进程加载结果，我选用了二元信号量 load_done。这种根据场景选择不同同步原语的策略，体现了对操作系统同步原理的灵活运用

6.4 实验方案设计能力

本次课程设计锻炼了我的全流程方案设计能力。

- 环境构建与工具链：我设计了基于 Ubuntu 18.04 的开发环境方案，配置了 QEMU 模拟器和 GDB 调试器，并通过 .bashrc 环境变量配置实现了工具链的路径集成，确保了开发的高效性
- 防御性编程与验证设计：在内核开发中，为了防止用户程序非法指针导致 OS 崩溃，我设计了通用的内存验证方案 check_ptr_and_size 和 check_string。在执行任何系统调用前，先验证指针是否位于 PHYS_BASE 以下且已映射页表

6.5 工程表达与沟通交流

- 逻辑可视化：为了准确阐述 Shell 的交互逻辑和 Pintos 的系统调用流程，我采用了流程图的方式展示了控制流，将复杂的代码逻辑转化为直观的图形语言，降低了理解成本
- 规范化文档撰写：在撰写设计文档时，我遵循“需求分析—总体设计—具体实现—问题解决—测试验证”的工程文档范式。完整记录了在试验过程中遇到的问题及解决方法

7 个人感想

本次操作系统课程设计不仅是一次代码编写的训练，更是一次对计算机系统底层逻辑的探索。

1. 理论与实践的融合：在课堂学习中，“进程控制块 (PCB)”、“上下文切换 (Context Switch)”、“用户栈 (User Stack)”和“中断处理 (Interrupt Handling)”往往只是抽象的概念。但是，在 Pintos 的开发过程中，我必须亲手在 struct thread 中添加字段来维护进程状态，必须精确计算每一个字节的偏移量来构建用户栈，必须在汇编与 C 语言的边界处理中断帧。这种内核开发经历，让我对操作系统的运行机制有了具象化、透彻的理解
2. 对并发与同步的认识：实验过程中最难的是处理并发带来的竞争条件。在实现 exec 和 wait 系统调用时，父进程与子进程的执行顺序是不确定的。初期代码中，我曾因未正确使用信号量，导致父进程在子进程加载完成前就尝试获取 PID，引发了随机的内核崩溃。通过引入 semaphore 和 child_entry 结构进行同步，我体会到了并发编程中“时序”的重要性，以及资源管理者必须具备的严谨性

3. 调试能力的质的提升：内核级调试与应用层开发有很大的不同。在 Pintos 中，简单的 printf 并不总是可靠。我学会了使用 GDB 进行远程调试，查看寄存器（eax, esp）状态，通过 backtrace 分析内核崩溃栈，以及利用 ASSERT 宏进行防御性编程
4. 工程化思维的建立：面对庞大的代码库，我学会了先阅读文档和头文件，理解现有模块的接口设计，复用已有代码而非重复造轮子。这种阅读源码、遵循既有规范的开发模式，对于未来的软件工程实践大有帮助