操作系统 Lab 01

姓名: 雍崔扬
学号: 21307140051

Task 0

0.1 调用约定

x86-64 Linux 调用约定 (System V ABI) 如下:

- **Caller-saved/Callee-saved 寄存器:**
  (调用者保存): 调用者在调用函数时必须保存这些寄存器的值，因为被调用者可能会修改它们.
  `rax`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `r10`, `r11`
  (被调用者保存): 被调用者需要在函数返回之前恢复这些寄存器的值，因为调用者期望这些寄存器的值在调用之后保持不变.
  `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, `r15`
- **普通函数调用的参数传递:**
  前 6 个整型或指针类型的参数按顺序通过以下寄存器传递:
  `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
  剩下的参数通过栈传递.

  函数返回值通过 `rax` 寄存器传递.
  如果返回多个值，可以通过 `rdx` 传递第二个返回值.
- **系统调用的参数传递:**
  系统调用号通过 `rax` 寄存器传递.
  前 6 个整型或指针类型的参数按顺序通过以下寄存器传递:
  `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`
  剩下的参数通过栈传递.

  系统调用返回值通过 `rax` 寄存器传递.

  系统调用可能会改变寄存器 `rcx`, `r11` 的值.
  因此如果调用者需要用到这两个寄存器中的值，则需要在系统调用前保存好它们.

0.2 Solution

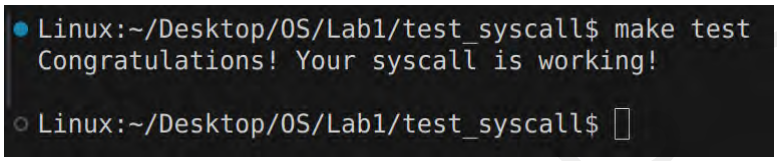
任务: 实现 `libsyscall/_syscall.S` 中的 `_syscall_` 函数.
思考: 是否需要保存 `rcx` 和 `r11` 寄存器的值?
回答: 无需保存，因为 `_syscall_` 函数的 `caller` 已经保存了寄存器 `rcx` 和 `r11` 的值.

```
.global _syscall_
.intel_syntax noprefix
_syscall_:

    # 将系统调用号 (num, 位于 rdi) 放入 rax
    mov rax, rdi

    # 将前 6 个参数放入寄存器
    mov rdi, rsi          # arg1 -> rdi
```

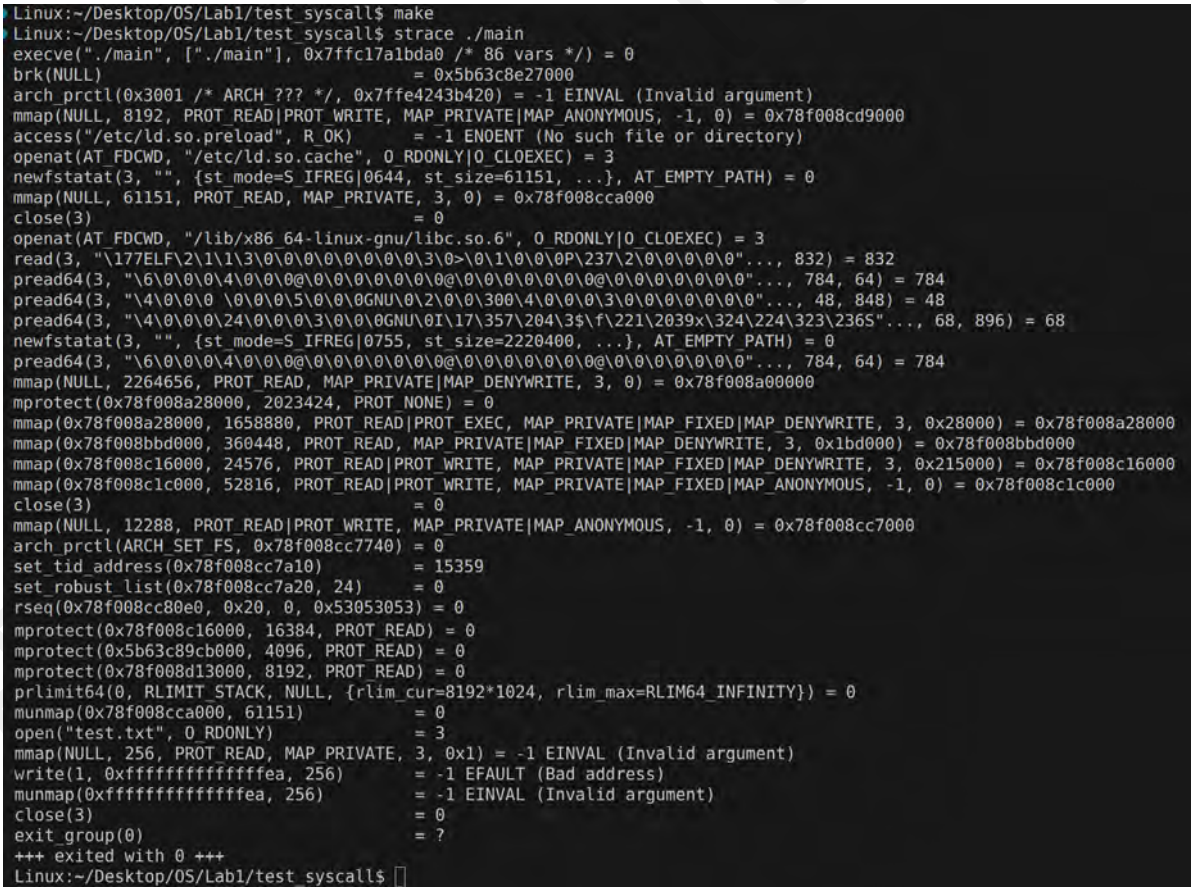
```
    mov rsi, rdx            # arg2 -> rsi
    mov rdx, rcx            # arg3 -> rdx
    mov r10, r8             # arg4 -> r10
    mov r8, r9              # arg5 -> r8

    # 将第 6 个参数从栈中加载到 r9
    mov r9, [rsp + 8]       # arg6 位于栈上

    # 调用 syscall 指令
    syscall

    # 返回结果 (结果已经在 rax 中)
    ret
```

使用 `test_syscall` 文件夹的测试程序的测试结果:

```
Linux:~/Desktop/OS/Lab1/test_syscall$ make test
Congratulations! Your syscall is working!
```



使用 `strace` 的测试结果:



下面是 `strace` 输出结果的逐行拆解 (我们关心的系统调用步骤为 ⑭～⑱):

- ① 编译、链接和运行:

```
Linux:~/Desktop/OS/Lab1/test_syscall$ make
Linux:~/Desktop/OS/Lab1/test_syscall$ strace ./main
```

- ② `execve` 系统调用执行可执行文件 `./main`，返回值 0 代表程序开始运行.
  `0x7ffc7740d400` 是指向环境变量的地址

```
execve("./main", ["./main"], 0x7ffc7740d400 /* 86 vars */) = 0
```

- ③ `brk` 设置程序数据段的末端 (堆的起始位置)
  传递 `NULL` 表示查询当前位置，返回 `0x58a35990a000` 即当前程序堆的起始位置.

```
brk(NULL) = 0x58a35990a000
```

- ④ `arch_prctl` 是用于设置或获取线程特定的 CPU 架构状态。
  由于传递的参数无效 (`0x3001`)，返回了 `EINVAL` 错误 (无效参数)

```
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffce7cb87d0) = -1 EINVAL (Invalid
argument)
```

- ⑤ `mmap` 系统调用分配内存，大小为 8192 字节，具有读写权限 (`PROT_READ|PROT_WRITE`)
  `MAP_PRIVATE|MAP_ANONYMOUS` 表示这是匿名内存映射，不关联文件，返回地址
  `0x742b4f568000`

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x742b4f568000
```

- ⑥ 尝试访问 `/etc/ld.so.preload`，该文件不存在，返回 `ENOENT` 错误

```
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
```

- ⑦ `openat` 系统调用以只读模式 (`O_RDONLY|O_CLOEXEC`) 打开动态链接库缓存文件
  `/etc/ld.so.cache`，返回文件描述符 `3`

```
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
```

- ⑧ `newfstatat` 查询文件状态，文件大小为 61151 字节，模式为常规文件 `0644`

```
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=61151, ...}, AT_EMPTY_PATH)
= 0
```

- ⑨ 将 `/etc/ld.so.cache` 文件映射到内存，大小为 61151 字节，权限为只读，返回映射地址
  `0x742b4f559000`

```
mmap(NULL, 61151, PROT_READ, MAP_PRIVATE, 3, 0) = 0x742b4f559000
```

- ⑩ 关闭文件描述符 `3`

```
close(3) = 0
```

- ⑪ 以只读模式打开标准C库 `libc.so.6`

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

- ⑫ 读取 ELF 文件头，用于加载动态库

```
read(3,
"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) =
832
```

- ⑬ 偏移读取库的其他部分数据，用于链接和加载动态库

```
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
```

- ⑭ 打开文件 `test.txt` 并返回文件描述符 `3`
  这与 `test_syscall/main.c` 中的系统调用 `int fd = _open("test.txt", 0, 0);` 相吻合

```
open("test.txt", O_RDONLY) = 3
```

- ⑮ 将文件 `test.txt` 内容映射到内存，大小为 256 字节，返回映射地址 `0x742b4f5a1000`
  这与 `test_syscall/main.c` 中的 `char* buf = (char*)_mmap(0, BUFSIZE, PROT_READ, MAP_PRIVATE, fd, 0);` 相吻合
  其中`BUFSIZE = 0x100` 正是 256 字节.

```
mmap(NULL, 256, PROT_READ, MAP_PRIVATE, 3, 0) = 0x742b4f5a1000
```

- ⑯ 将 `test.txt` 的内容写入标准输出 (文件描述符 `1`)，成功写入 256 字节
  这与 `test_syscall/main.c` 中的系统调用 `_write(1, buf, BUFSIZE);` 相吻合

```
write(1, "Congratulations! Your syscall is"..., 256) = 256
```

- ⑰ 解除内存映射并释放映射的 256 字节
  这与 `test_syscall/main.c` 中的系统调用 `_munmap(buf, BUFSIZE);` 相吻合
  其中`BUFSIZE = 0x100` 正是 256 字节.

```
munmap(0x742b4f5a1000, 256) = 0
```

- ⑱ 关闭文件描述符 `3`
  这与 `test_syscall/main.c` 中的系统调用 `_close(fd);` 相吻合，其中 `fd` 正是 `3`

```
close(3) = 0
```

- ⑲ 程序调用 `exit_group` 退出，返回值 `0` 表示程序正常退出

```
exit_group(0) = ?
+++ exited with 0 +++
```

Task 1

本任务使用 `open`、`getdents64` 和 `write` 这三个系统调用来实现一个简单的 `ls`
首先使用 `man <syscall>` 来查看系统调用的详细使用方法:

> 在 `man` 页面中也有简单的搜索功能，你可以使用 `/` 来搜索关键字，使用 `n/N` 来查看下一个/上一个搜索结果.
> 如果你使用 `man open` 发现打开了 `xdg-open` 的文档，
> 那么可以在终端中输入 `man 2 open` 来指定查看作为系统调用的 `open` 的文档.

1.1 open

思考: 查阅 `open` 系统调用的文档，回答以下问题:

- ① `open` 系统调用的三个参数分别是什么?
- ② 如何指定 `open` 使用只读模式打开文件? 只写和读写模式呢?
- ③ `O_DIRECTORY` 是什么? 在我们实现 `ls` 的过程中，需要用到这个标志吗?

回答:
`open` 系统调用的原型为:

```
int open(const char *pathname, int flags, mode_t mode);
```

- `pathname`: 字符串类型参数，指定要打开的文件的路径，可以是绝对路径或相对路径.
- `flags`: 整数类型参数，指定打开文件的访问方式等属性.
  其中三种访问方式 (只读、只写和读写) 的标志位分别为: `O_RDONLY`、`O_WRONLY` 和 `O_RDWR`
- `mode`: 可选的第三个参数，仅当创建新文件 (存在 `O_CREAT` 标志位) 时使用
  用来设置所有者、组和其他用户的读写执行权限.
- 成功时返回一个非负的文件描述符; 失败时返回 `-1`，并设置 `errno` 表示错误类型.

关于标志位 `O_DIRECTORY`:
它用于确保打开的文件是一个目录.
如果 `pathname` 指向的不是一个目录，则 `open` 系统调用将失败并返回错误 `ENOTDIR`
在实现 `ls` 的任务中，我们的主要目标是列出目录中的文件和子目录，使用 `O_DIRECTORY` 可以避免误将文件当作目录处理.

1.2 getdents64

`getdents64` 系统调用用于读取目录文件中的目录项，并将它们存储到用户提供的缓冲区中.
其调用原型为:

```
int getdents64(unsigned int fd, struct linux_dirent64 *dirp, unsigned int
count);
```

- `fd`: 待读取的目录的文件描述符 (文件描述符必须是通过 `open` 系统调用打开的目录)
- `dirp`: 指向用户提供的缓冲区，系统调用会将读取到的目录项信息存储到该缓冲区中.
  该缓冲区的内容是多个连续的 `linux_dirent64` 结构体:

```
struct linux_dirent64 {
    ino64_t        d_ino;    // 文件索引节点号
    off64_t        d_off;    // 到下一个目录项的偏移量
    unsigned short d_reclen; // 此目录项的长度
    unsigned char  d_type;   // 文件类型
    char           d_name[]; // 文件名
};
```

这个结构体的大小是可变的，由 `d_reclen` 指定，因此 `d_name` 没有声明大小.

- `count`：缓冲区的大小，以字节为单位，表示最多可以读取多少字节的目录项.
- 成功时，返回读取到的字节数，即填充到缓冲区中的目录项总字节数;
  如果返回值为 0，则表示已到达目录的末尾，没有更多的目录项可读.
  如果发生错误，则返回 `-1`，并设置 `errno` 来指示错误的原因.

1.3 write

`write` 系统调用用于将数据从用户空间的缓冲区写入到文件描述符所指向的文件或设备.
其调用原型为:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- `fd`：文件描述符，指明将数据写入到哪个文件或设备，通常使用系统调用 `open` 来获取.
  标准文件描述符:
  - `0`:标准输入(`stdin`)
  - `1`:标准输出(`stdout`)
  - `2`:标准错误(`stderr`)
- `buf`:指向用户空间的缓冲区，该缓冲区包含要写入的数据。这个缓冲区的大小应至少为 `count` 字节.
- `count`：要写入的数据字节数，表示从缓冲区中写入多少字节到文件或设备。
- 成功时，`write` 返回写入的字节数，通常它应该与 `count` 相等.
  如果写入的字节数比 `count` 少，通常意味着磁盘已满或设备的写缓冲区已满.
  失败时，返回 `-1`，并设置 `errno` 来指示错误的原因.

1.4 Solution

任务: 基于框架提供的系统调用包装函数，完成 `ls/main.c` 中的 TODO 内容.

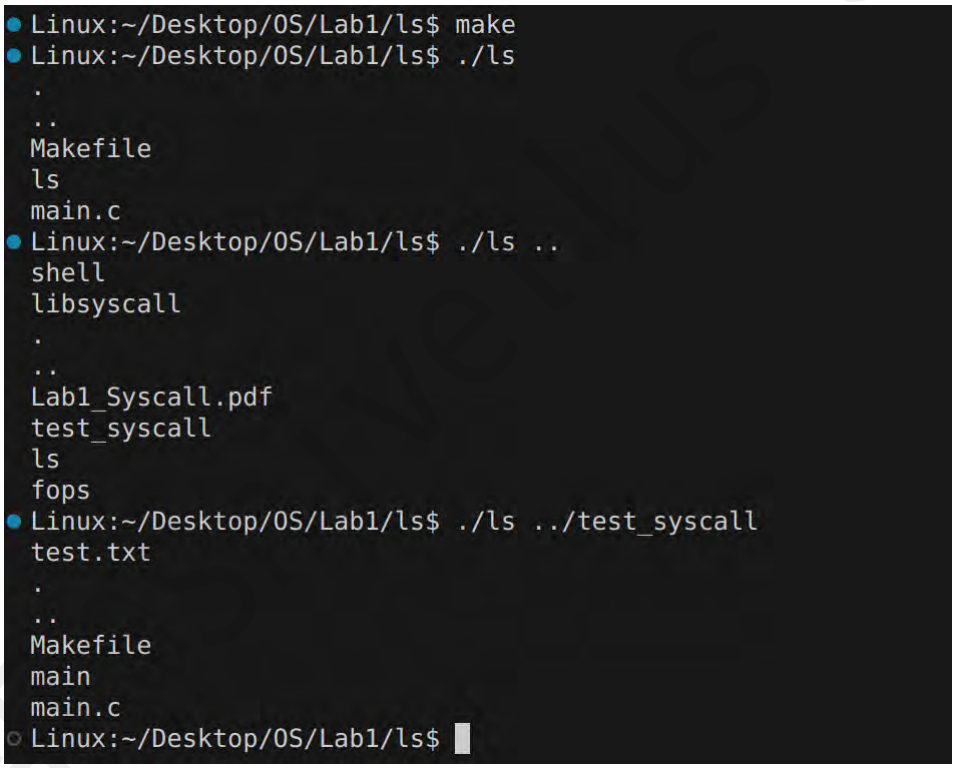
```
#define _GNU_SOURCE
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include "_syscall.h"

#define BUF_SIZE 1024

int main(int argc, char *argv[]) {
    int fd;
```

```
    int nread;
    char buf[BUF_SIZE];

    char *directory = ".";
    if (argc > 1) {
        directory = argv[1];
    }

    /* TODO: Use open syscall to open the directory. */
    fd = _open(directory, O_RDONLY | O_DIRECTORY, 0);

    if (fd == -1) {
        perror("open failed");
        return 1;
    }

    for (;;) {
        /* TODO: Use getdents64 syscall to read directory entries. */
        nread = _getdents64(fd, (struct linux_dirent64 *)buf, BUF_SIZE);

        // Return value -1 means error.
        if (nread == -1) {
            perror("_getdents64 failed");
            close(fd);
            return 1;
        }

        // Return value 0 means end of directory.
        if (nread == 0) break;

        /* TODO: Make a loop to iterate over the directory entries. */
        int entry_len = 0;
        for (int pos = 0; pos < nread; pos += entry_len)
        {
            /* TODO: get the current directory entry. */
            struct linux_dirent64 *d = (struct linux_dirent64 *)(buf + pos);
            entry_len = d->d_reclen;

            /* TODO: Use write syscall to print the file name to stdout (fd: 1).
*/
            _write(1, d->d_name, strlen(d->d_name));

            /* TODO: Don't forget to print a newline. */
            _write(1, "\n", 1);
        }
    }

    close(fd);
    return 0;
}
```

运行结果:

```
Linux:~/Desktop/OS/Lab1/ls$ make
Linux:~/Desktop/OS/Lab1/ls$ ./ls
.
..
```

```
Makefile
ls
main.c
Linux:~/Desktop/OS/Lab1/ls$ ./ls ..
shell
libsyscall
.
..
Lab1_Syscall.pdf
test_syscall
ls
fops
Linux:~/Desktop/OS/Lab1/ls$ ./ls ../test_syscall
test.txt
.
..
Makefile
main
main.c
Linux:~/Desktop/OS/Lab1/ls$
```



Task 2

本任务使用系统调用 `fork` 和 `execve` 以及库函数 `waitpid` 实现一个只支持绝对地址执行程序的简单 shell.
其中 `fork` 用于创建一个子进程，该子进程会复制父进程的内存空间.
我们可以在子进程中调用 `execve` 来执行一个新的程序.
`execve` 会将子进程的内存空间替换为新程序的内存空间，然后开始执行新程序.
`waitpid` 用于等待子进程执行完毕.

实验文件已经提供了参数解析代码来进行参数解析，只需要实现子进程执行过程即可.
注意在 `fork` 之后 父子进程有着不同的任务分工: 子进程执行新的程序、父进程等待子进程完成.
从 Task 2 开始，可以直接使用标准库中提供的系统调用相关函数.

2.1 Solution

任务: 完成 `shell/shell.c` 中的 `TODO` 内容:

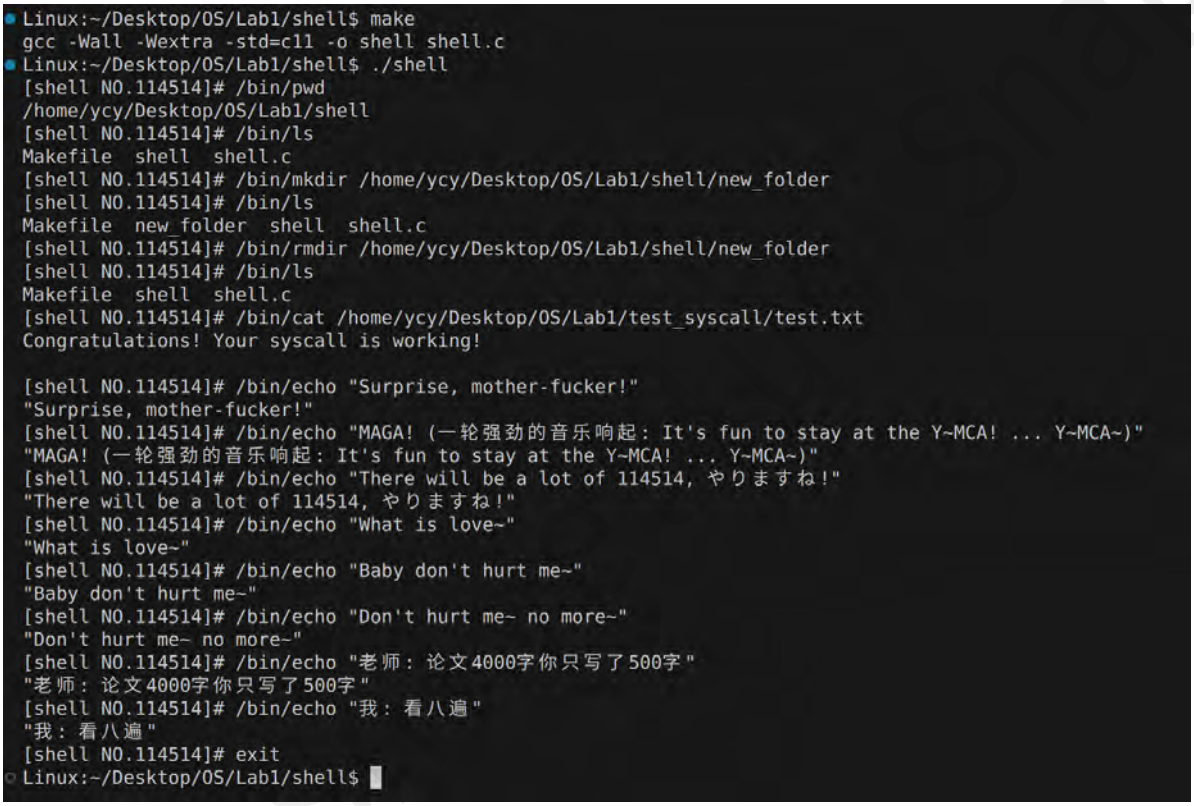
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAX_ARGS 64

#define PROMPT "[shell NO.114514]# " // やりますね

void parse_args(char *input, char **args)
{
    int i = 0;

    // Use space and newline as delimiters.
    args[i] = strtok(input, " \n");

    while (args[i] != NULL && i < MAX_ARGS - 1) {
        i++;
        args[i] = strtok(NULL, " \n");
    }
    args[i] = NULL;
}

int execute(char **args)
{
    /**
     *  TODO: Use fork, execve, and waitpid to execute the command.
     *  Hint: args[0] is the command to execute.
     */

    // Creat a child process
    pid_t pid = fork();

    if (pid < 0)
    {
        perror("fork failed");
        return -1;
    }

    if (pid == 0) // Child proces
    {
        if (execve(args[0], args, NULL) < 0)
        {
            perror("execve failed");
            exit(1); // execution failed, exit from child process
        }
    }
    else // Parent process
    {
        int status;
        waitpid(pid, &status, 0);
```

```
        return WEXITSTATUS(status); // return the exit status of child process
    }

    return 0;
}

int main() {
    char input[1024];
    char *args[MAX_ARGS];

    while (1) {
        printf(PROMPT);
        fflush(stdout);

        if (fgets(input, sizeof(input), stdin) == NULL) {
            break;
        }

        parse_args(input, args);

        if (args[0] == NULL) {
            continue;
        }

        if (strcmp(args[0], "exit") == 0) {
            break;
        } else {
            execute(args);
        }
    }

    return 0;
}
```

运行结果:

```
Linux:~/Desktop/OS/Lab1/shell$ make
gcc -Wall -Wextra -std=c11 -o shell shell.c
Linux:~/Desktop/OS/Lab1/shell$ ./shell
[shell NO.114514]# /bin/pwd
/home/ycy/Desktop/OS/Lab1/shell
[shell NO.114514]# /bin/ls
Makefile  shell  shell.c
[shell NO.114514]# /bin/mkdir /home/ycy/Desktop/OS/Lab1/shell/new_folder
[shell NO.114514]# /bin/ls
Makefile  new_folder  shell  shell.c
[shell NO.114514]# /bin/rmdir /home/ycy/Desktop/OS/Lab1/shell/new_folder
[shell NO.114514]# /bin/ls
Makefile  shell  shell.c
[shell NO.114514]# /bin/cat /home/ycy/Desktop/OS/Lab1/test_syscall/test.txt
Congratulations! Your syscall is working!

[shell NO.114514]# /bin/echo "Surprise, mother-fucker!"
"Surprise, mother-fucker!"
[shell NO.114514]# /bin/echo "MAGA! (一轮强劲的音乐响起: It's fun to stay at the
Y~MCA! ... Y~MCA~)"
"MAGA! (一轮强劲的音乐响起: It's fun to stay at the Y~MCA! ... Y~MCA~)"
```

```
[shell NO.114514]# /bin/echo "There will be a lot of 114514, やりますね!"
"There will be a lot of 114514, やりますね!"
[shell NO.114514]# /bin/echo "What is love~"
"What is love~"
[shell NO.114514]# /bin/echo "Baby don't hurt me~"
"Baby don't hurt me~"
[shell NO.114514]# /bin/echo "Don't hurt me~ no more~"
"Don't hurt me~ no more~"
[shell NO.114514]# /bin/echo "老师: 论文4000字你只写了500字"
"老师: 论文4000字你只写了500字"
[shell NO.114514]# /bin/echo "我: 看八遍"
"我: 看八遍"
[shell NO.114514]# exit
Linux:~/Desktop/OS/Lab1/shell$
```



2.2 思考

回答以下问题:

问题 ①:
除了 `execve` 以外，还有 `execl` 、 `execv` 、 `execle` 、 `execvp` 、 `execvpe` 、 `execlp` 等函数
请使用 `man` 命令查看这些函数的区别，并简要归纳这几个函数和 `execve` 的联系和区别.
(不需要说明每个函数功能，简单说明和 `execve` 的联系区别即可)

回答 ①:
在 Linux 系统中， `execve` 是最基础的执行函数.
所有其他的 `exec` 系列函数 (`execl` , `execv` , `execle` , `execvp` , `execvpe` , `execlp`) 本质上都是 `execve` 的封装.
它们都用于替换当前进程的映像并调用 `execve` 系统调用来执行新的程序.

- **参数传递形式**:
  - `execve` : 需要提供程序路径、参数数组 `argv[]` 以及环境变量数组 `envp[]`

- `execl` / `execlp`: 使用可变参数列表来传递程序路径和参数，而不是数组形式
- `execv` / `execvp`: 参数通过数组 `argv[]` 传递
- `execle` / `execvpe`: 允许显式指定环境变量 `envp[]`

- **路径解析**:
  - `execve`: 需要给定程序的**绝对路径**或**相对路径**，不会搜索环境变量 `PATH`
  - `execlp`, `execvp`, `execvpe`: 会通过 `PATH` 环境变量来搜索可执行文件，无需指定绝对路径
- **环境变量处理**:
  - `execve`, `execle`, `execvpe`: 允许显式指定 `envp[]` 环境变量
  - `execl`, `execv`, `execlp`, `execvp`: 继承当前进程的环境变量，不能显式指定 `envp[]`

问题 ②:
什么是**内建命令**? 为什么我们自己实现的 shell 不支持 `cd`?

回答 ②:

- **内建命令**是指那些在 shell 内部实现的命令 (例如 `cd` 和 `exit`)，而不是独立的可执行文件.
  当我们在 shell 中输入这些命令时，shell 会直接处理它们，而不需要调用外部程序.
- 我们自己实现的 shell 中没有实现 "cd" 的内建指令，只实现了 "exit".
  毕竟我们目前专注于 shell 如何调用外部程序，而不涉及 shell 内部状态的管理.
  支持内建命令需要管理更多的状态信息，比如当前目录、环境变量等.
  这对于一个简单的 shell 实现而言，是不值得的.

Task 3

本任务专注于比较标准 I/O (Standard I/O) 与内存映射文件的效率.

3.1 Background

标准 I/O 使用 `open`, `read`, `write`, `lseek`, `close` 等系统调用进行文件读写:

```
#include <stdio.h> // 标准输入输出库
#include <fcntl.h> // 文件控制库，包含 open 函数
#include <unistd.h> // POSIX 操作系统 API，包含 read、write、close 函数

int main()
{
    int file_desc;
    char buffer[100];
    ssize_t bytes_read, bytes_written;

    // 1. 打开文件
    // O_RDWR 表示以读写模式打开文件，O_CREAT 表示如果文件不存在则创建文件，O_TRUNC 表示如果文件存在则清空文件内容
    file_desc = open("example.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (file_desc < 0)
    {
        perror("open");
        return 1;
    }

    // 2. 写入数据到文件
    const char *data = "Hello, world!\n";
```

```
    bytes_written = write(file_desc, data, sizeof(data));
    if (bytes_written < 0)
    {
        perror("write");
        close(file_desc);
        return 1;
    }

    // 3. 将文件指针移到文件开头，准备读取文件内容
    lseek(file_desc, 0, SEEK_SET);

    // 4. 从文件读取数据
    bytes_read = read(file_desc, buffer, sizeof(buffer) - 1);
    if (bytes_read < 0)
    {
        perror("read");
        close(file_desc);
        return 1;
    }

    // 确保读取的内容以 null 字符结尾，以便正确打印字符串
    buffer[bytes_read] = '\0';
    // 打印读取的内容
    printf("Content: \n%s", buffer);

    // 5. 关闭文件
    if (close(file_desc) < 0)
    {
        perror("close"); // 关闭文件失败，输出错误信息
        return 1;
    }
    return 0;
}
```

但标准 I/O 可能存在性能问题 (特别是在 I/O 密集场景中).
一方面，系统调用的开销很高 (这是相对于一般的函数调用而言的)
另一方面，由于多数情况下，文件存储于硬盘上，而硬盘数据的读写时延极高，
尽管现代操作系统引入了多种手段试图减轻硬盘读写时延的影响，但其仍是重要的性能瓶颈.

一种解决方法便是将文件内容直接缓存到内存中，将文件的读写操作变为内存读写:
一方面，内存的读写不需要系统调用，避免了系统调用的开销
另一方面，内存的读写时延远远低于硬盘读写的时延

`mmap` 可用于 "将文件内容映射到内存中"，而 `munmap` 可用于解除映射 (`mmap` 的对偶操作).
我们利用 `mmap` 和 `munmap` 改写标准 I/O 的代码:

```
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main()
{
    const char *data = "Hello, world!\n";
    size_t data_len = strlen(data);
```

```
    // 1. 打开文件
    // O_RDWR 表示以读写模式打开文件，O_CREAT 表示如果文件不存在则创建文件，O_TRUNC 表示如果文件存在则清空文件内容
    int file_desc = open("example.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (file_desc < 0)
    {
        perror("open");
        return 1;
    }

    // 2. 调整文件大小到写入数据的长度
    if (ftruncate(file_desc, data_len) == -1)
    {
        perror("ftruncate");
        close(file_desc);
        return 1;
    }

    // 3. 将文件映射到内存
    char *mapped = mmap(NULL, data_len, PROT_READ | PROT_WRITE, MAP_SHARED,file_desc, 0);
    if (mapped == MAP_FAILED)
    {
        perror("mmap");
        close(file_desc);
        return 1;
    }

    // 4. 将数据写入映射的内存区域
    memcpy(mapped, data, data_len);

    // 5. 解除映射
    if (munmap(mapped, data_len) == -1)
    {
        perror("munmap");
    }

    // 6. 关闭文件
    close(file_desc);
    return 0;
}
```

代码解析:

- `mmap` 原型和调用语句如下:

  ```
  void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
  char *mapped = mmap(NULL, data_len, PROT_READ | PROT_WRITE, MAP_SHARED, file_desc, 0);
  ```

  - `addr`: 建议的映射起始地址 (可以为 `NULL`，让系统选择地址)
  - `length`: 映射区域的字节数
  - `prot`: 映射区域的保护标志，指定对映射内存的访问权限.
    常用值有: `PROT_READ` (可读), `PROT_WRITE` (可写), `PROT_EXEC` (可执行) 和 `PROT_NONE` (不可

访问)
    - `flags`: 映射选项，控制映射行为.
      常用值有: `MAP_SHARED` (共享映射, 修改会影响原文件) 和 `MAP_PRIVATE` (私有映射，修改不会影响原文件)
    - `fd`: 要映射的文件的文件描述符，通常通过 `open` 系统调用获取.
    - `offset`: 文件映射的偏移量 (必须是页面大小的倍数) 通常设置为0，代表从文件最前方开始对应.
- `mummap` 的原型和调用语句如下:

```
int munmap(void *addr, size_t length);
munmap(mapped, data_len)
```

    - `addr`: 要解除映射的内存区域的起始地址. 该地址必须是先前通过 `mmap` 映射的地址.
    - `length`: 解除映射的字节数，必须与 `mmap` 时请求的长度相同.

3.2 Solution

在这个任务中，我们设置了一个对于目标文件随机位置读/写字符 10^6 次的 workload，用于模拟密集随机 I/O 场景.
你将分别实现标准 I/O 和内存映射文件方案下的以下 4 种函数，框架代码将比较两种方案的效率:

- `<strategy>_file_init`: 初始化工作，包括打开文件、建立内存文件映射等.
- `<strategy>_file_read`: 从 offset 处读取一个字符到 result
- `<strategy>_file_write`: 向 offset 处写入一个字符 value
- `<strategy>_file_destroy`: 释放资源，包括关闭文件、取消内存文件映射等.

任务: 完成 `fops/std_file.c` (标准 I/O) 与 `fops/mmaped_file.c` (内存映射文件) 的 `TODO` 内容.
① `fops/std_file.c` (标准 I/O) 代码如下:

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <unistd.h>
#include "defs.h"

int std_file_init(const char *filename)
{
    return open(filename, O_RDWR);
}

int std_file_read(int fd, off_t offset, char *result)
{
    /**
     * TODO: Read `result` from the file at `offset`.
     */

    // 使用 lseek 移动文件指针到指定的 offset
    if (lseek(fd, offset, SEEK_SET) == -1)
    {
        perror("lseek failed");
        return -1;  // 移动失败
    }
```

```
    // 从文件中读取一个字节到 result
    ssize_t bytesRead = read(fd, result, 1);
    if (bytesRead == -1)
    {
        perror("read failed");
        return -1;  // 读取失败
    }

    return 0;  // 读取成功
}

int std_file_write(int fd, off_t offset, char value)
{
    /**
     * TODO: Write `value` to the file at `offset`.
     */

    // 使用 lseek 移动文件指针到指定的 offset
    if (lseek(fd, offset, SEEK_SET) == -1)
    {
        perror("lseek failed");
        return -1;  // 移动失败
    }

    // 向文件中写入一个字节 value
    ssize_t bytesWritten = write(fd, &value, 1);
    if (bytesWritten == -1)
    {
        perror("write failed");
        return -1;  // 写入失败
    }

    return 0;  // 写入成功
}

int std_file_destroy(int fd)
{
    return close(fd);
}
```

② `fops/mmaped_file.c` (内存映射文件) 代码如下:

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <unistd.h>
#include "defs.h"

char *mmaped_file_init(const char *filename)
{
    /**
     * TODO: Open the specified file with read-write access and memory-maps it
     * into the process's address space for efficient file I/O. If successful,
     * returns a pointer to the mapped memory region.
     */
```

```c
    // 打开文件，获取文件描述符
    int fd = open(filename, O_RDWR);
    if (fd == -1)
    {
        perror("open failed");
        return NULL;  // 打开失败
    }

    // 获取文件大小
    off_t filesize = lseek(fd, 0, SEEK_END);
    if (filesize == -1)
    {
        perror("lseek failed");
        close(fd);
        return NULL;  // 获取文件大小失败
    }

    // 将文件映射到内存
    char *mapped = mmap(NULL, filesize, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
0);
    if (mapped == MAP_FAILED)
    {
        perror("mmap failed");
        close(fd);
        return NULL;  // 映射失败
    }

    // 关闭文件描述符，映射后不再需要
    close(fd);
    return mapped;  // 返回映射的内存地址
}

int mmaped_file_read(char *file, off_t offset, char *result)
{
    /**
     * TODO: Read the character at the specified `offset` in the memory-mapped
     * file and stores it in `result`.
     */

    // 从映射的内存中读取字符
    *result = file[offset];
    return 0;  // 读取成功
}

int mmaped_file_write(char *file, off_t offset, char value)
{
    /**
     * TODO: Write `value` at the specified `offset` in the memory-mapped file.
     */

    // 向映射的内存中写入字符
    file[offset] = value;
    return 0;  // 写入成功
}

int mmaped_file_destroy(char *map)
{
```

```
    /**
     * TODO: Unmap the memory-mapped file.
     */

    // 获取文件大小以正确解除映射
    if (munmap(map, sizeof(map)) == -1) {
        perror("munmap failed");
        return -1;  // 解除映射失败
    }
    return 0;  // 解除映射成功
}
```

运行结果:

```
Linux:~/Desktop/OS/Lab1/fops$ make run
gcc -Wall -lrt -o fops main.c mmaped_file.c std_file.c
./fops
Standard I/O operations took: 45637.95 ms
mmap operations took: 114.12 ms
Linux:~/Desktop/OS/Lab1/fops$
```