OS labwork report

一、Reverse

1.Idea

This labwork primarily focuses on reversing and outputting text. There are three input formats corresponding to three functionalities. The goal is to create a C program, reverse.c, on a Linux system to achieve the required functionality.

The reverse.c program consists of three main functions: input, reverse, and output. The input function processes the input text, including handling exceptions and addressing three scenarios: empty input file, non-empty input file, and exception detection. The reverse function reverses the input text, returning the reversed text. The output function manages text output, including scenarios for empty output path (printing to the terminal) and non-empty output path (printing to an output file). Within the main function, char* type input and output paths and FILE* type file streams are defined. Values are passed based on the number of input format parameters, and certain exceptional cases are restricted.

2.Some problems

2.1 Lack of familiarity with dynamic memory allocation and pointer usage in code.

2.2 Issues encountered with the getline function while reading text, as the function includes the newline character when reading each line.

2.3 Different file names for input and output may be hard-linked in the system, necessitating comparison for in-code processing of exceptions.

3.Code

```
//reverse.c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif


void output(char *output_path,FILE * output_file,char** text, int line_cnt) {
    if (output_path == NULL){
        for (int i = 0; i < line_cnt; i++)
            printf("%s", text[i]);
    }
    else {
        for (int i = 0; i < line_cnt; i++) {
            fprintf(output_file,"%s", text[i]); //可能出现一些格式问题 因为getline函
数是把换行符也存了
```

```
        }
    }
}

char** reverse(char** text, int line_cnt) {
    char** res = (char**) malloc((sizeof(char*) * line_cnt));
    if (!res) {
        fprintf(stderr, "malloc failed ");
        exit(1);
    }
    for (int i = 0; i < line_cnt; i++) {
        res[i] = text[line_cnt - 1 - i];
    }
    return res;
}

char** input(FILE* input_file, size_t* line_cnt) {
    char** text = (char**)malloc(sizeof(char*) *100 );
    if (!text) {
        fprintf(stderr, "malloc failed ");
        exit(1);
    }
    if (input_file == NULL) {
        size_t line_length = 0;
        *line_cnt = 0;
        while(getline(&text[*line_cnt],&line_length,stdin)!= -1){
            printf("%s", text[*line_cnt]);
            (*line_cnt)++;
        }
    }
    if (input_file != NULL) {
        size_t line_length = 0;
        *line_cnt = 0;
        while (getline(&text[*line_cnt], &line_length, input_file) != -1) {
            (*line_cnt)++;
        }
    }

    return text;
}
int main(int argc, char*argv[]) {
    char* input_path = NULL;
    char* output_path = NULL;
    FILE* input_file=stdin;
    FILE* output_file=stdout;  //测试中注意有stdout需要写入一个特定的文件

    if (argc == 2) {
        input_path = argv[1];
    }
    if (argc == 3) {
        input_path = argv[1];
        output_path = argv[2];
    }
    if (argc > 3) {
        fprintf(stderr, "usage: reverse <input> <output>\n");
        exit(1);
    }
```

```
    if (input_path != NULL) {
        input_file = fopen(input_path,"r");
        if (input_file == NULL) {
            fprintf(stderr, "reverse: cannot open file '%s'\n", input_path);
            exit(1);
        }
    }
    if (output_path != NULL) {  // open output file
        output_file = fopen(output_path,"w");
        if (output_file == NULL) {
            fprintf(stderr, "reverse: cannot open file '%s'\n", output_path);
            exit(1);
        }
    }
    if (input_path!=NULL&&output_path!=NULL) {  //测试中出现输入和输出文件是硬连接的
需要比较incode
        if (strcmp(input_path, output_path) == 0) {
            fprintf(stderr, "reverse: input and output file must differ\n");
            exit(1);
        }
        else {
            struct stat input_stat, output_stat;
            stat(input_path, &input_stat);
            stat(output_path, &output_stat);

            if (input_stat.st_ino == output_stat.st_ino) {
                fprintf(stderr, "reverse: input and output file must differ\n");
                exit(1);
            }
        }
    }

    size_t cnt = 0;

    char** text = input(input_file, &cnt);
    int cnt_p=(int)cnt;
    text = reverse(text, cnt_p);
    output(output_path, output_file, text, cnt_p);
    return 0;
}
```

二、Syscall

The main files involved in addsystem calls include:

File	Function
`syscall.h`	Defines system call numbers.
`usys.S`	Provides the interface for transitioning between user space and kernel space.
`syscall.c`	Forwards user-initiated system calls to the kernel.
`user.h`	Defines the method for passing parameters in system calls.
`sysfile.c`	Implements system calls.

syscall.h

This file defines index numbers for system calls. Therefore, the system call index "getreadcount" needs to be added.

```
#define SYS_getreadcount 22
```

usys.S

This file provides the interface for transitioning between user space and kernel space, so the required system call interface needs to be added.

```
SYSCALL(getreadcount)
```

syscall.c

The file needs to add corresponding code in a column of system calls.

```
extern int sys_getreadcount(void)
```

In `static int (*syscalls[])(void)`, add the following:

```
[SYS_getreadcount] sys_getreadcount
```

user.h

This file defines the method for passing parameters in system calls.

```
int getreadcount(void);
```

sysfile.c

This file is used to implement system call functionality.

```
int readcount=0;
int sys_getreadcount(void){
    return readcount;
}
```

Firstly, define a global variable `readcount` as 0; the `getreadcount` system call returns the value of this variable. Since each call to the `read` system call will increment `readcount`, the `read` system call fuunction also needs to implement `readcount++`.