Name: Nguyễn Đại Hưng

ID: 24520601

Class: IT007.Q15.1

OPERATING SYSTEM
LAB 6'S REPORT

SUMMARY

Task		Status	Page
Section 6.4	Thực thi lệnh trong tiến trình con	Hoàn thành	**2**
	Tạo tính năng sử dụng lại câu lệnh gần đây	Hoàn thành	**4**
	Chuyển hướng vào ra	Hoàn thành	**10**
	Giao tiếp sử dụng cơ chế đường ống	Hoàn thành	**17**
	Kết thúc lệnh đang thực thi bằng tổ hợp phím Ctrl + C	Hoàn thành	**26**

Self-scores: 10

Note: Export file to **PDF and name the file by following format:*
Student ID_LABx.pdf

Section 6.4

1. Thực thi lệnh trong tiến trình con

Source code

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

#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

#define MAX_LINE 80

static void ParseArgs(char *Line, char *args[]) {
    int i = 0;
    char *tok = strtok(Line, " \t");
    while (tok != NULL && i < MAX_LINE / 2) {
        args[i++] = tok;
        tok = strtok(NULL, " \t");
    }
    args[i] = NULL;
}

int main(void) {
    char *args[MAX_LINE/2 + 1];
    int should_run = 1;

    char line[MAX_LINE + 2];

    while (should_run) {
        printf("it007sh>");
        fflush(stdout);

        if (fgets(line, sizeof(line), stdin) == NULL) break;

        line[strcspn(line, "\n")] = '\0';
        if (line[0] == '\0') continue;

        if (strcmp(line, "exit") == 0) {
            should_run = 0;
            continue;
        }

        char line_copy[MAX_LINE + 1];
```

```
        strncpy(line_copy, line, MAX_LINE);
        line_copy[MAX_LINE] = '\0';

        ParseArgs(line_copy, args);
        if (args[0] == NULL) continue;

        pid_t pid = fork();
        if (pid < 0) {
            perror("fork");
            continue;
        }

        if (pid == 0) {
            execvp(args[0], args);
            perror("execvp");
            _exit(127);
        } else {
            waitpid(pid, NULL, 0);
        }
    }

    return 0;
}
```

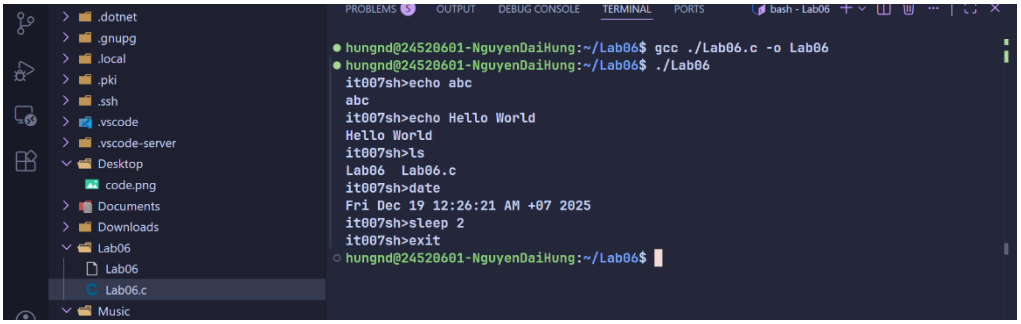
Giải thích code

Trong bài 6.4.1, chương trình xây dựng một shell đơn giản với dấu nhắc it007sh>. Mỗi vòng lặp, shell in prompt ra màn hình và dùng fgets() để đọc một dòng lệnh người dùng nhập. Sau khi đọc xong, chương trình dùng strcspn() để tìm vị trí ký tự xuống dòng \n và thay nó bằng '\0' nhằm "cắt" phần xuống dòng, giúp chuỗi lệnh sạch để xử lý. Nếu người dùng chỉ nhấn Enter (chuỗi rỗng) thì bỏ qua và in prompt lại. Nếu người dùng nhập exit thì kiểm tra bằng strcmp() và kết thúc vòng lặp để thoát shell.

Để chuẩn bị chạy lệnh, chương trình sao chép chuỗi lệnh sang line_copy bằng strncpy() (đảm bảo không vượt quá giới hạn MAX_LINE) vì hàm tách chuỗi sẽ làm thay đổi nội dung chuỗi. Hàm viết là ParseArgs() có nhiệm vụ tách câu lệnh thành các "tham số" để đưa vào execvp(). Trong ParseArgs(), strtok() được dùng để cắt chuỗi theo khoảng trắng/tab; mỗi token thu được được lưu vào mảng args[] và cuối cùng gán args[i] = NULL (vì execvp() yêu cầu mảng tham số phải kết thúc bằng NULL).

Phần thực thi lệnh dùng cơ chế tiến trình: fork() tạo ra một tiến trình con. Nếu fork() lỗi thì in thông báo bằng perror(). Nếu đang ở tiến trình con (pid == 0) thì gọi execvp() để thay thế tiến trình con bằng chương trình cần chạy (ví dụ echo, ls, pwd…) và nếu execvp() thất bại thì in lỗi bằng perror() rồi thoát ngay bằng _exit(127). Ở tiến trình cha (pid > 0), chương trình gọi waitpid() để chờ tiến trình con chạy xong rồi mới quay lại in prompt. Nhờ waitpid(), shell đảm bảo người dùng không thể nhập lệnh tiếp khi lệnh trước chưa kết thúc.

Kết quả



Nhận xét

Dựa trên kết quả trong hình, chương trình 6.4.1 đã hoạt động đúng như một shell tối giản theo đúng thiết kế của code. Sau khi chạy ./6.4.1, dấu nhắc it007sh> xuất hiện và mỗi lần nhập lệnh như echo abc, ls, echo done, date thì shell đều thực thi lệnh và in ra kết quả tương ứng, rồi mới quay lại hiển thị prompt để chờ lệnh tiếp theo. Điều này cho thấy chuỗi lệnh nhập vào đã được xử lý đúng: chương trình đọc một dòng, loại bỏ ký tự xuống dòng, tách lệnh và tham số bằng hàm ParseArgs() (dùng strtok), sau đó gọi execvp(args[0], args) để chạy lệnh hệ thống.

Ngoài ra, hành vi chờ của shell cũng đúng với yêu cầu "không cho nhập lệnh mới khi lệnh đang chạy". Khi nhập sleep 2, shell không in prompt ngay lập tức mà phải đợi lệnh kết thúc rồi mới quay lại it007sh>. Đây là kết quả của việc tiến trình cha dùng waitpid(pid, NULL, 0) để chờ tiến trình con hoàn thành trước khi tiếp tục vòng lặp. Lệnh ls liệt kê đúng các file trong thư mục LAB6 đang đứng, chứng tỏ tiến trình con thừa hưởng đúng môi trường làm việc (working directory) từ shell sau khi fork. Cuối cùng, khi nhập exit, chương trình thoát khỏi vòng lặp và quay lại terminal bash.

Nhìn chung, kết quả thực nghiệm khẳng định bài 6.4.1 đã hoàn thành đúng chức năng: nhận lệnh, thực thi trong tiến trình con và đồng bộ bằng waitpid để điều khiển luồng chạy của shell.

2. Tạo tính năng sử dụng lại câu lệnh gần đây

Source code

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

#define MAX_LINE 80
#define HISTORY_MAX 100
```

```c
static struct termios OrigTermios;

static void RestoreTermios(void) {
    tcsetattr(STDIN_FILENO, TCSAFLUSH, &OrigTermios);
}

static void SetRawMode(void) {
    struct termios raw = OrigTermios;
    raw.c_lflag &= ~(ICANON | ECHO);
    raw.c_iflag &= ~(IXON | ICRNL);
    raw.c_cc[VMIN] = 1;
    raw.c_cc[VTIME] = 0;
    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

static void ParseArgs(char *Line, char *args[]) {
    int i = 0;
    char *tok = strtok(Line, " \t");
    while (tok != NULL && i < MAX_LINE / 2) {
        args[i++] = tok;
        tok = strtok(NULL, " \t");
    }
    args[i] = NULL;
}

static void AddHistory(char History[HISTORY_MAX][MAX_LINE + 1], int
*HistoryCount, const char *Cmd) {
    if (Cmd[0] == '\0') return;
    if (*HistoryCount > 0 && strcmp(History[*HistoryCount - 1], Cmd) == 0)
return;

    if (*HistoryCount < HISTORY_MAX) {
        strncpy(History[*HistoryCount], Cmd, MAX_LINE);
        History[*HistoryCount][MAX_LINE] = '\0';
        (*HistoryCount)++;
    } else {
        for (int i = 1; i < HISTORY_MAX; i++) strcpy(History[i - 1],
History[i]);
        strncpy(History[HISTORY_MAX - 1], Cmd, MAX_LINE);
        History[HISTORY_MAX - 1][MAX_LINE] = '\0';
    }
}

static void RedrawLine(const char *Prompt, const char *Buf) {
    const char *clr = "\r\033[2K";
```

```
    write(STDOUT_FILENO, clr, (int)strlen(clr));
    write(STDOUT_FILENO, Prompt, (int)strlen(Prompt));
    write(STDOUT_FILENO, Buf, (int)strlen(Buf));
}

static int ReadLineWithHistory(const char *Prompt, char *Out, int OutMax,
char History[HISTORY_MAX][MAX_LINE + 1], int HistoryCount) {
    int len = 0;
    int hist_pos = HistoryCount;
    int navigating = 0;
    char cur_saved[MAX_LINE + 1];
    cur_saved[0] = '\0';
    Out[0] = '\0';

    RedrawLine(Prompt, Out);

    while (1) {
        unsigned char c;
        ssize_t n = read(STDIN_FILENO, &c, 1);
        if (n <= 0) return 0;

        if (c == 4) return 0;

        if (c == '\n' || c == '\r') {
            write(STDOUT_FILENO, "\n", 1);
            Out[len] = '\0';
            return 1;
        }

        if (c == 127 || c == 8) {
            if (len > 0) {
                len--;
                Out[len] = '\0';
                RedrawLine(Prompt, Out);
            }
            continue;
        }

        if (c == 27) {
            unsigned char seq[2];
            if (read(STDIN_FILENO, &seq[0], 1) <= 0) continue;
            if (read(STDIN_FILENO, &seq[1], 1) <= 0) continue;

            if (seq[0] == '[' && (seq[1] == 'A' || seq[1] == 'B')) {
                if (HistoryCount == 0) continue;
```

```
                if (!navigating) {
                    navigating = 1;
                    strncpy(cur_saved, Out, MAX_LINE);
                    cur_saved[MAX_LINE] = '\0';
                    hist_pos = HistoryCount;
                }

                if (seq[1] == 'A') {
                    if (hist_pos > 0) hist_pos--;
                } else {
                    if (hist_pos < HistoryCount) hist_pos++;
                }

                if (hist_pos == HistoryCount) {
                    strncpy(Out, cur_saved, OutMax - 1);
                    Out[OutMax - 1] = '\0';
                } else {
                    strncpy(Out, History[hist_pos], OutMax - 1);
                    Out[OutMax - 1] = '\0';
                }

                len = (int)strlen(Out);
                RedrawLine(Prompt, Out);
            }
            continue;
        }

        if (c >= 32 && c <= 126) {
            if (len < OutMax - 1) {
                Out[len++] = (char)c;
                Out[len] = '\0';
                RedrawLine(Prompt, Out);
            }
            continue;
        }
    }
}

int main(void) {
    char *args[MAX_LINE/2 + 1];
    int should_run = 1;

    char line[MAX_LINE + 1];
    char History[HISTORY_MAX][MAX_LINE + 1];
```

```
    int HistoryCount = 0;

    tcgetattr(STDIN_FILENO, &OrigTermios);
    atexit(RestoreTermios);

    while (should_run) {
        SetRawMode();

        if (!ReadLineWithHistory("it007sh>", line, (int)sizeof(line),
History, HistoryCount)) {
            break;
        }

        if (line[0] == '\0') continue;

        if (strcmp(line, "exit") == 0) {
            should_run = 0;
            continue;
        }

        AddHistory(History, &HistoryCount, line);

        char line_copy[MAX_LINE + 1];
        strncpy(line_copy, line, MAX_LINE);
        line_copy[MAX_LINE] = '\0';

        ParseArgs(line_copy, args);
        if (args[0] == NULL) continue;

        RestoreTermios();

        pid_t pid = fork();
        if (pid < 0) {
            perror("fork");
            continue;
        }

        if (pid == 0) {
            execvp(args[0], args);
            perror("execvp");
            _exit(127);
        } else {
            waitpid(pid, NULL, 0);
        }
    }
```

```
    return 0;
}
```

Giải thích code

Chương trình 6.4.2 được bổ sung cơ chế lưu lịch sử lệnh và cho phép chọn lại bằng phím ↑/↓.

Để làm được việc bắt phím mũi tên, chương trình không còn đọc input theo dòng (canonical) mà chuyển terminal sang raw mode. Cụ thể, chương trình lưu lại trạng thái terminal gốc vào OrigTermios rồi đăng ký atexit(RestoreTermios) để đảm bảo khi thoát chương trình thì terminal được trả về trạng thái bình thường. Hàm SetRawMode() tắt ICANON và ECHO để mỗi phím bấm được đọc ngay lập tức và không tự echo ra màn hình; đồng thời chỉnh một số cờ input để tránh các hành vi tự xử lý Enter/Ctrl… của terminal. Khi cần quay về chế độ bình thường, RestoreTermios() sẽ gọi tcsetattr với cấu hình đã lưu.

Phần lịch sử được lưu trong mảng History (tối đa HISTORY_MAX lệnh) và biến HistoryCount. Hàm AddHistory() có nhiệm vụ thêm lệnh vừa nhập vào lịch sử, bỏ qua lệnh rỗng và tránh lưu trùng liên tiếp; nếu lịch sử đầy thì dồn mảng để loại bỏ lệnh cũ nhất rồi thêm lệnh mới vào cuối.

Thay vì fgets, bài 6.4.2 đọc từng ký tự bằng read() trong hàm ReadLineWithHistory(). Mỗi lần người dùng gõ, chương trình tự quản lý buffer Out và vẽ lại dòng đang nhập bằng RedrawLine(): nó dùng escape sequence để xóa dòng hiện tại và đưa con trỏ về đầu dòng, rồi in lại prompt và nội dung buffer. Trong quá trình đọc, chương trình xử lý các phím đặc biệt: Enter để kết thúc lệnh và trả về chuỗi hoàn chỉnh; Backspace để xóa ký tự cuối trong buffer; Ctrl+D để kết thúc shell.

Phần quan trọng nhất là xử lý phím ↑/↓: khi nhận ký tự ESC (27), chương trình đọc thêm 2 byte để nhận dạng chuỗi điều khiển dạng ESC [A (mũi tên lên) hoặc ESC [B (mũi tên xuống). Khi bắt đầu "điều hướng" lịch sử, chương trình lưu lại dòng đang gõ dở vào cur_saved để nếu người dùng bấm xuống đến cuối danh sách thì có thể quay lại đúng dòng đang gõ dở đó. Sau đó hist_pos sẽ tăng/giảm tùy ↑/↓, buffer Out được thay bằng lệnh tương ứng trong History, rồi gọi RedrawLine() để hiện lại lệnh cho người dùng chọn và nhấn Enter để chạy. Cuối cùng, trước khi fork/exec chạy lệnh, chương trình khôi phục terminal về chế độ bình thường bằng RestoreTermios().

Kết quả chạy: Link

Nhận xét

Dựa trên kết quả chạy ./6.4.2, chương trình đã thực hiện đúng yêu cầu của bài 6.4.2 là lưu lịch sử lệnh và cho phép gọi lại bằng phím mũi tên. Ban đầu, shell vẫn hoạt động ổn định như các việc: in prompt it007sh>, thực thi các lệnh cơ bản như echo abc, ls, date và trả prompt lại sau khi lệnh chạy xong. Sau đó, các lệnh như echo abc, date, ls được thực thi lại

nhiều lần theo đúng thứ tự lịch sử, cho thấy người dùng đã dùng ↑/↓ để chọn lại lệnh trước đó rồi nhấn Enter để chạy, thay vì phải gõ lại toàn bộ câu lệnh. Đặc biệt, kết quả date ở hai lần khác nhau hiển thị thời gian khác nhau, chứng tỏ đây là thực thi lại lệnh chứ không phải chỉ "in lại chuỗi lệnh". Cuối cùng, lệnh exit hoạt động đúng, giúp thoát shell và quay lại terminal bình thường, cho thấy phần xử lý lịch sử không làm ảnh hưởng đến cơ chế thoát chương trình.

3. Chuyển hướng vào ra

Source code

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <termios.h>
#include <fcntl.h>
#define MAX_LINE 80
#define HISTORY_MAX 100

static struct termios OrigTermios;

static void RestoreTermios(void) {
    tcsetattr(STDIN_FILENO, TCSAFLUSH, &OrigTermios);
}

static void SetRawMode(void) {
    struct termios raw = OrigTermios;
    raw.c_lflag &= ~(ICANON | ECHO);
    raw.c_iflag &= ~(IXON | ICRNL);
    raw.c_cc[VMIN] = 1;
    raw.c_cc[VTIME] = 0;
    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

static void AddHistory(char History[HISTORY_MAX][MAX_LINE + 1], int
*HistoryCount, const char *Cmd) {
    if (Cmd[0] == '\0') return;
    if (*HistoryCount > 0 && strcmp(History[*HistoryCount - 1], Cmd) == 0)
return;

    if (*HistoryCount < HISTORY_MAX) {
        strncpy(History[*HistoryCount], Cmd, MAX_LINE);
        History[*HistoryCount][MAX_LINE] = '\0';
        (*HistoryCount)++;
```

```
    } else {
        for (int i = 1; i < HISTORY_MAX; i++) strcpy(History[i - 1],
History[i]);
        strncpy(History[HISTORY_MAX - 1], Cmd, MAX_LINE);
        History[HISTORY_MAX - 1][MAX_LINE] = '\0';
    }
}

static void RedrawLine(const char *Prompt, const char *Buf) {
    const char *clr = "\r\033[2K";
    write(STDOUT_FILENO, clr, (int)strlen(clr));
    write(STDOUT_FILENO, Prompt, (int)strlen(Prompt));
    write(STDOUT_FILENO, Buf, (int)strlen(Buf));
}

static int ReadLineWithHistory(const char *Prompt, char *Out, int OutMax,
char History[HISTORY_MAX][MAX_LINE + 1], int HistoryCount) {
    int len = 0;
    int hist_pos = HistoryCount;
    int navigating = 0;
    char cur_saved[MAX_LINE + 1];
    cur_saved[0] = '\0';
    Out[0] = '\0';

    RedrawLine(Prompt, Out);

    while (1) {
        unsigned char c;
        ssize_t n = read(STDIN_FILENO, &c, 1);
        if (n <= 0) return 0;

        if (c == 4) return 0;

        if (c == '\n' || c == '\r') {
            write(STDOUT_FILENO, "\n", 1);
            Out[len] = '\0';
            return 1;
        }

        if (c == 127 || c == 8) {
            if (len > 0) {
                len--;
                Out[len] = '\0';
                RedrawLine(Prompt, Out);
            }
```

```
            continue;
        }

        if (c == 27) {
            unsigned char seq[2];
            if (read(STDIN_FILENO, &seq[0], 1) <= 0) continue;
            if (read(STDIN_FILENO, &seq[1], 1) <= 0) continue;

            if (seq[0] == '[' && (seq[1] == 'A' || seq[1] == 'B')) {
                if (HistoryCount == 0) continue;

                if (!navigating) {
                    navigating = 1;
                    strncpy(cur_saved, Out, MAX_LINE);
                    cur_saved[MAX_LINE] = '\0';
                    hist_pos = HistoryCount;
                }

                if (seq[1] == 'A') {
                    if (hist_pos > 0) hist_pos--;
                } else {
                    if (hist_pos < HistoryCount) hist_pos++;
                }

                if (hist_pos == HistoryCount) {
                    strncpy(Out, cur_saved, OutMax - 1);
                    Out[OutMax - 1] = '\0';
                } else {
                    strncpy(Out, History[hist_pos], OutMax - 1);
                    Out[OutMax - 1] = '\0';
                }

                len = (int)strlen(Out);
                RedrawLine(Prompt, Out);
            }
            continue;
        }

        if (c >= 32 && c <= 126) {
            if (len < OutMax - 1) {
                Out[len++] = (char)c;
                Out[len] = '\0';
                RedrawLine(Prompt, Out);
            }
            continue;
```

```
        }
    }
}

static void NormalizeRedir(const char *In, char *Out, int OutMax) {
    int j = 0;
    for (int i = 0; In[i] != '\0' && j < OutMax - 1; i++) {
        if (In[i] == '<' || In[i] == '>') {
            if (j < OutMax - 1 && (j == 0 || Out[j - 1] != ' ')) Out[j++]
= ' ';
            if (j < OutMax - 1) Out[j++] = In[i];
            if (j < OutMax - 1) Out[j++] = ' ';
        } else {
            Out[j++] = In[i];
        }
    }
    Out[j] = '\0';
}

static int ParseArgsAndRedir(char *Line, char *args[], char **InFile, char
**OutFile) {
    *InFile = NULL;
    *OutFile = NULL;

    int i = 0;
    char *tok = strtok(Line, " \t");
    while (tok != NULL) {
        if (strcmp(tok, "<") == 0) {
            tok = strtok(NULL, " \t");
            if (tok == NULL) return 0;
            *InFile = tok;
        } else if (strcmp(tok, ">") == 0) {
            tok = strtok(NULL, " \t");
            if (tok == NULL) return 0;
            *OutFile = tok;
        } else {
            if (i < MAX_LINE / 2) args[i++] = tok;
        }
        tok = strtok(NULL, " \t");
    }
    args[i] = NULL;
    return 1;
}

int main(void) {
```

```
    char *args[MAX_LINE/2 + 1];
    int should_run = 1;

    char line[MAX_LINE + 1];
    char History[HISTORY_MAX][MAX_LINE + 1];
    int HistoryCount = 0;

    tcgetattr(STDIN_FILENO, &OrigTermios);
    atexit(RestoreTermios);

    while (should_run) {
        SetRawMode();

        if (!ReadLineWithHistory("it007sh>", line, (int)sizeof(line),
History, HistoryCount)) {
            break;
        }

        if (line[0] == '\0') continue;

        if (strcmp(line, "exit") == 0) {
            should_run = 0;
            continue;
        }

        AddHistory(History, &HistoryCount, line);

        char norm[3 * MAX_LINE + 1];
        NormalizeRedir(line, norm, (int)sizeof(norm));

        char line_copy[3 * MAX_LINE + 1];
        strncpy(line_copy, norm, (int)sizeof(line_copy) - 1);
        line_copy[sizeof(line_copy) - 1] = '\0';

        char *InFile = NULL;
        char *OutFile = NULL;

        if (!ParseArgsAndRedir(line_copy, args, &InFile, &OutFile)) {
            RestoreTermios();
            fprintf(stderr, "it007sh: redirection syntax error\n");
            continue;
        }

        if (args[0] == NULL) continue;
```

```
        RestoreTermios();

        pid_t pid = fork();
        if (pid < 0) {
            perror("fork");
            continue;
        }

        if (pid == 0) {
            if (InFile != NULL) {
                int fd = open(InFile, O_RDONLY);
                if (fd < 0) {
                    perror("open");
                    _exit(1);
                }
                dup2(fd, STDIN_FILENO);
                close(fd);
            }

            if (OutFile != NULL) {
                int fd = open(OutFile, O_WRONLY | O_CREAT | O_TRUNC,
0644);
                if (fd < 0) {
                    perror("open");
                    _exit(1);
                }
                dup2(fd, STDOUT_FILENO);
                close(fd);
            }

            execvp(args[0], args);
            perror("execvp");
            _exit(127);
        } else {
            waitpid(pid, NULL, 0);
        }
    }

    return 0;
}
```

Giải thích code

Ở bài 6.4.3, được bổ sung chức năng chuyển hướng dữ liệu vào/ra bằng ký hiệu < và >. Chương trình vẫn giữ cơ chế đọc lệnh (có history) và thực thi bằng fork/execvp, nhưng

trước khi chạy lệnh sẽ phân tích cú pháp để tìm file chuyển hướng và thiết lập lại stdin/stdout tương ứng.

Cụ thể, sau khi nhận được chuỗi lệnh người dùng nhập, chương trình gọi NormalizeRedir() để chuẩn hoá biểu thức chuyển hướng: nếu người dùng gõ dính như sort<in.txt hoặc ls>out.txt thì hàm này sẽ chèn khoảng trắng quanh < và > để đảm bảo tách token ổn định khi dùng strtok(). Nhờ vậy, các trường hợp có/không có khoảng trắng đều parse được theo cùng một cách.

Sau bước chuẩn hoá, chương trình dùng ParseArgsAndRedir() để tách lệnh thành 3 phần: (1) mảng tham số args[] dùng cho execvp, (2) tên file nhập InFile nếu gặp ký hiệu <, và (3) tên file xuất OutFile nếu gặp ký hiệu >. Nếu cú pháp sai kiểu sort < (thiếu tên file) hoặc ls > thì hàm trả về lỗi, chương trình báo "redirection syntax error" và quay lại prompt.

Điểm quan trọng nhất nằm ở tiến trình con. Nếu có InFile, chương trình mở file bằng open(InFile, O_RDONLY) rồi dùng dup2(fd, STDIN_FILENO) để thay stdin của tiến trình con bằng file đó, sau đó close(fd) để tránh rò rỉ descriptor. Tương tự, nếu có OutFile thì mở file với O_WRONLY | O_CREAT | O_TRUNC (ghi, tạo nếu chưa có, xoá nội dung cũ) và dup2(fd, STDOUT_FILENO) để chuyển stdout của lệnh sang file. Sau khi thiết lập xong stdin/stdout, chương trình mới gọi execvp(args[0], args) để chạy lệnh, nên chương trình con sẽ "tưởng" như đang đọc từ bàn phím/ghi ra màn hình, nhưng thực tế đã được chuyển sang đọc/ghi file.

Kết quả

```
hungnd@24520601-NguyenDaiHung:~/Lab06$ gcc ./Lab06.c -o Lab06
hungnd@24520601-NguyenDaiHung:~/Lab06$ ./Lab06
it007sh>echo abcd > abcd.txt
it007sh>cat abcd.txt
abcd
it007sh>ls -l > out.txt
it007sh>cat < out.txt
total 36
-rw-r--r-- 1 hungnd hungnd     5 Dec 19 00:30 abcd.txt
-rwxrwxr-x 1 hungnd hungnd 21248 Dec 19 00:30 Lab06
-rw-rw-r-- 1 hungnd hungnd  6818 Dec 19 00:29 Lab06.c
-rw-r--r-- 1 hungnd hungnd     0 Dec 19 00:30 out.txt
it007sh>ls >
it007sh: redirection syntax error
it007sh>sort <
it007sh: redirection syntax error
it007sh>exit
hungnd@24520601-NguyenDaiHung:~/Lab06$
```

Nhận xét

Dựa trên kết quả chạy trong hình, chương trình 6.4.3 đã thực hiện đúng chức năng chuyển hướng vào/ra bằng < và > theo yêu cầu. Trước hết, lệnh echo abcd > abcd.txt không in ra màn hình mà ghi nội dung vào file, sau đó khi chạy cat < abcd.txt thì chương trình đọc dữ liệu từ file và in ra đúng abcd. Điều này chứng tỏ chuyển hướng stdout sang file với > và chuyển hướng stdin từ file với < đã hoạt động đúng.

Tiếp theo, lệnh ls -l > out.txt ghi kết quả liệt kê chi tiết thư mục vào file out.txt, và khi cat < out.txt thì toàn bộ nội dung danh sách file được in ra lại đầy đủ. Kết quả này cho thấy phần thiết lập dup2() trong tiến trình con đã chuyển stdout của lệnh sang file, và các lệnh đọc từ stdin cũng nhận đúng dữ liệu từ file vào.

Ngoài ra, trường hợp nhập không có khoảng trắng như echo xyz>kq.txt và cat<kq.txt vẫn chạy đúng và in ra xyz, chứng tỏ chương trình đã xử lý tốt việc chuẩn hoá cú pháp (không bắt buộc người dùng phải đặt khoảng trắng quanh dấu < và >). Cuối cùng, khi thử các lệnh sai cú pháp như ls > hoặc sort <, shell báo redirection syntax error và vẫn tiếp tục hoạt động bình thường, không bị treo hay thoát đột ngột. Nhìn chung, kết quả thực nghiệm xác nhận bài 6.4.3 đã hoàn thành đúng phần mở rộng so với 6.4.2: hỗ trợ chuyển hướng vào/ra, xử lý được cả trường hợp gõ dính và có cơ chế báo lỗi hợp lý khi cú pháp chuyển hướng không đầy đủ.

4. Giao tiếp sử dụng cơ chế đường ống

Source code

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

#define MAX_LINE 80
#define HISTORY_MAX 100

static struct termios OrigTermios;

static void RestoreTermios(void) {
    tcsetattr(STDIN_FILENO, TCSAFLUSH, &OrigTermios);
}

static void SetRawMode(void) {
    struct termios raw = OrigTermios;
    raw.c_lflag &= ~(ICANON | ECHO);
    raw.c_iflag &= ~(IXON | ICRNL);
```

```
    raw.c_cc[VMIN] = 1;
    raw.c_cc[VTIME] = 0;
    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

static void AddHistory(char History[HISTORY_MAX][MAX_LINE + 1], int
*HistoryCount, const char *Cmd) {
    if (Cmd[0] == '\0') return;
    if (*HistoryCount > 0 && strcmp(History[*HistoryCount - 1], Cmd) == 0)
return;

    if (*HistoryCount < HISTORY_MAX) {
        strncpy(History[*HistoryCount], Cmd, MAX_LINE);
        History[*HistoryCount][MAX_LINE] = '\0';
        (*HistoryCount)++;
    } else {
        for (int i = 1; i < HISTORY_MAX; i++) strcpy(History[i - 1],
History[i]);
        strncpy(History[HISTORY_MAX - 1], Cmd, MAX_LINE);
        History[HISTORY_MAX - 1][MAX_LINE] = '\0';
    }
}

static void RedrawLine(const char *Prompt, const char *Buf) {
    const char *clr = "\r\033[2K";
    write(STDOUT_FILENO, clr, (int)strlen(clr));
    write(STDOUT_FILENO, Prompt, (int)strlen(Prompt));
    write(STDOUT_FILENO, Buf, (int)strlen(Buf));
}

static int ReadLineWithHistory(const char *Prompt, char *Out, int OutMax,
char History[HISTORY_MAX][MAX_LINE + 1], int HistoryCount) {
    int len = 0;
    int hist_pos = HistoryCount;
    int navigating = 0;
    char cur_saved[MAX_LINE + 1];
    cur_saved[0] = '\0';
    Out[0] = '\0';

    RedrawLine(Prompt, Out);

    while (1) {
        unsigned char c;
        ssize_t n = read(STDIN_FILENO, &c, 1);
        if (n <= 0) return 0;
```

```
        if (c == 4) return 0;

        if (c == '\n' || c == '\r') {
            write(STDOUT_FILENO, "\n", 1);
            Out[len] = '\0';
            return 1;
        }

        if (c == 127 || c == 8) {
            if (len > 0) {
                len--;
                Out[len] = '\0';
                RedrawLine(Prompt, Out);
            }
            continue;
        }

        if (c == 27) {
            unsigned char seq[2];
            if (read(STDIN_FILENO, &seq[0], 1) <= 0) continue;
            if (read(STDIN_FILENO, &seq[1], 1) <= 0) continue;

            if (seq[0] == '[' && (seq[1] == 'A' || seq[1] == 'B')) {
                if (HistoryCount == 0) continue;

                if (!navigating) {
                    navigating = 1;
                    strncpy(cur_saved, Out, MAX_LINE);
                    cur_saved[MAX_LINE] = '\0';
                    hist_pos = HistoryCount;
                }

                if (seq[1] == 'A') {
                    if (hist_pos > 0) hist_pos--;
                } else {
                    if (hist_pos < HistoryCount) hist_pos++;
                }

                if (hist_pos == HistoryCount) {
                    strncpy(Out, cur_saved, OutMax - 1);
                    Out[OutMax - 1] = '\0';
                } else {
                    strncpy(Out, History[hist_pos], OutMax - 1);
                    Out[OutMax - 1] = '\0';
```

```
                }

                len = (int)strlen(Out);
                RedrawLine(Prompt, Out);
            }
            continue;
        }

        if (c >= 32 && c <= 126) {
            if (len < OutMax - 1) {
                Out[len++] = (char)c;
                Out[len] = '\0';
                RedrawLine(Prompt, Out);
            }
            continue;
        }
    }
}

static void NormalizeOps(const char *In, char *Out, int OutMax) {
    int j = 0;
    for (int i = 0; In[i] != '\0' && j < OutMax - 1; i++) {
        if (In[i] == '<' || In[i] == '>' || In[i] == '|') {
            if (j < OutMax - 1 && (j == 0 || Out[j - 1] != ' ')) Out[j++]
= ' ';
            if (j < OutMax - 1) Out[j++] = In[i];
            if (j < OutMax - 1) Out[j++] = ' ';
        } else {
            Out[j++] = In[i];
        }
    }
    Out[j] = '\0';
}

static char *Trim(char *s) {
    while (*s == ' ' || *s == '\t') s++;
    if (*s == 0) return s;
    char *end = s + strlen(s) - 1;
    while (end > s && (*end == ' ' || *end == '\t')) end--;
    end[1] = '\0';
    return s;
}

static int ParseArgsAndRedir(char *Line, char *args[], char **InFile, char
**OutFile) {
```

```
    *InFile = NULL;
    *OutFile = NULL;

    int i = 0;
    char *tok = strtok(Line, " \t");
    while (tok != NULL) {
        if (strcmp(tok, "<") == 0) {
            tok = strtok(NULL, " \t");
            if (tok == NULL) return 0;
            *InFile = tok;
        } else if (strcmp(tok, ">") == 0) {
            tok = strtok(NULL, " \t");
            if (tok == NULL) return 0;
            *OutFile = tok;
        } else {
            if (i < MAX_LINE / 2) args[i++] = tok;
        }
        tok = strtok(NULL, " \t");
    }
    args[i] = NULL;
    return 1;
}

static int ApplyInRedir(const char *InFile) {
    if (InFile == NULL) return 1;
    int fd = open(InFile, O_RDONLY);
    if (fd < 0) return 0;
    if (dup2(fd, STDIN_FILENO) < 0) return 0;
    close(fd);
    return 1;
}

static int ApplyOutRedir(const char *OutFile) {
    if (OutFile == NULL) return 1;
    int fd = open(OutFile, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) return 0;
    if (dup2(fd, STDOUT_FILENO) < 0) return 0;
    close(fd);
    return 1;
}

int main(void) {
    char *args[MAX_LINE/2 + 1];
    int should_run = 1;
```

```
    char line[MAX_LINE + 1];
    char History[HISTORY_MAX][MAX_LINE + 1];
    int HistoryCount = 0;

    tcgetattr(STDIN_FILENO, &OrigTermios);
    atexit(RestoreTermios);

    while (should_run) {
        SetRawMode();

        if (!ReadLineWithHistory("it007sh>", line, (int)sizeof(line),
History, HistoryCount)) {
            break;
        }

        if (line[0] == '\0') continue;

        if (strcmp(line, "exit") == 0) {
            should_run = 0;
            continue;
        }

        AddHistory(History, &HistoryCount, line);

        char norm[4 * MAX_LINE + 1];
        NormalizeOps(line, norm, (int)sizeof(norm));

        char splitbuf[4 * MAX_LINE + 1];
        strncpy(splitbuf, norm, (int)sizeof(splitbuf) - 1);
        splitbuf[sizeof(splitbuf) - 1] = '\0';

        char *left = NULL;
        char *right = NULL;
        char *pipe_pos = strstr(splitbuf, " | ");
        if (pipe_pos != NULL) {
            *pipe_pos = '\0';
            left = Trim(splitbuf);
            right = Trim(pipe_pos + 3);
            if (left[0] == '\0' || right[0] == '\0') {
                RestoreTermios();
                fprintf(stderr, "it007sh: pipe syntax error\n");
                continue;
            }
        } else {
            left = Trim(splitbuf);
```

```
        }

        RestoreTermios();

        if (right == NULL) {
            char cmd1[4 * MAX_LINE + 1];
            strncpy(cmd1, left, (int)sizeof(cmd1) - 1);
            cmd1[sizeof(cmd1) - 1] = '\0';

            char *In1 = NULL;
            char *Out1 = NULL;
            if (!ParseArgsAndRedir(cmd1, args, &In1, &Out1) || args[0] ==
NULL) {
                fprintf(stderr, "it007sh: redirection syntax error\n");
                continue;
            }

            pid_t pid = fork();
            if (pid < 0) {
                perror("fork");
                continue;
            }

            if (pid == 0) {
                if (!ApplyInRedir(In1)) { perror("open"); _exit(1); }
                if (!ApplyOutRedir(Out1)) { perror("open"); _exit(1); }
                execvp(args[0], args);
                perror("execvp");
                _exit(127);
            } else {
                waitpid(pid, NULL, 0);
            }
        } else {
            char cmd1[4 * MAX_LINE + 1];
            char cmd2[4 * MAX_LINE + 1];
            strncpy(cmd1, left, (int)sizeof(cmd1) - 1);
            cmd1[sizeof(cmd1) - 1] = '\0';
            strncpy(cmd2, right, (int)sizeof(cmd2) - 1);
            cmd2[sizeof(cmd2) - 1] = '\0';

            char *args1[MAX_LINE/2 + 1];
            char *args2[MAX_LINE/2 + 1];
            char *In1 = NULL, *Out1 = NULL;
            char *In2 = NULL, *Out2 = NULL;
```

```
            if (!ParseArgsAndRedir(cmd1, args1, &In1, &Out1) || args1[0]
== NULL ||
                !ParseArgsAndRedir(cmd2, args2, &In2, &Out2) || args2[0]
== NULL) {
                fprintf(stderr, "it007sh: redirection syntax error\n");
                continue;
            }

            if (Out1 != NULL || In2 != NULL) {
                fprintf(stderr, "it007sh: unsupported redirection with
pipe\n");
                continue;
            }

            int pfd[2];
            if (pipe(pfd) < 0) {
                perror("pipe");
                continue;
            }

            pid_t pid1 = fork();
            if (pid1 < 0) {
                perror("fork");
                close(pfd[0]); close(pfd[1]);
                continue;
            }

            if (pid1 == 0) {
                if (!ApplyInRedir(In1)) { perror("open"); _exit(1); }
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[0]);
                close(pfd[1]);
                execvp(args1[0], args1);
                perror("execvp");
                _exit(127);
            }

            pid_t pid2 = fork();
            if (pid2 < 0) {
                perror("fork");
                close(pfd[0]); close(pfd[1]);
                waitpid(pid1, NULL, 0);
                continue;
            }
```

```
            if (pid2 == 0) {
                dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
                close(pfd[1]);
                if (!ApplyOutRedir(Out2)) { perror("open"); _exit(1); }
                execvp(args2[0], args2);
                perror("execvp");
                _exit(127);
            }

            close(pfd[0]);
            close(pfd[1]);
            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);
        }
    }

    return 0;
}
```

Giải thích code

Đầu tiên, chương trình chuẩn hoá chuỗi lệnh bằng NormalizeOps() để chèn khoảng trắng quanh các toán tử < > và |, nhờ đó người dùng có thể gõ dính như ls|wc vẫn tách được token ổn định khi parse. Sau khi chuẩn hoá, chương trình dùng strstr(splitbuf, " | ") để tìm vị trí dấu pipe. Nếu không có | thì xử lý như chương trình 6.4.3 (chạy 1 lệnh có thể kèm </>). Nếu có | thì tách chuỗi thành 2 nửa: lệnh bên trái và lệnh bên phải, đồng thời dùng Trim() để loại bỏ khoảng trắng thừa; trường hợp thiếu một vế (ví dụ ls | hoặc | wc) sẽ báo lỗi pipe syntax error rồi quay lại prompt.

Khi đã có 2 vế lệnh, chương trình parse từng vế bằng ParseArgsAndRedir() để lấy mảng tham số args1, args2 cho execvp(), đồng thời lấy thông tin chuyển hướng file. Ở đây chương trình có một ràng buộc để tránh xung đột với pipe: không cho phép cmd1 > file | cmd2 (vì stdout của cmd1 phải đi vào pipe) và cũng không cho phép cmd1 | cmd2 < file (vì stdin của cmd2 phải lấy từ pipe). Nếu gặp các trường hợp đó, chương trình báo unsupported redirection with pipe.

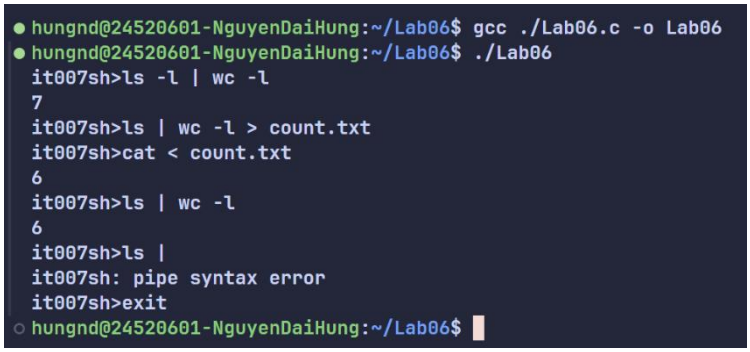
Phần cốt lõi của pipe là tạo ống bằng pipe(pfd) (gồm pfd[0] để đọc, pfd[1] để ghi), sau đó tạo hai tiến trình con:

> Con 1 chạy lệnh bên trái: nếu có < thì áp dụng chuyển hướng input trước, rồi dùng dup2(pfd[1], STDOUT_FILENO) để chuyển stdout sang đầu ghi của pipe. Sau đó đóng các đầu pipe không cần thiết và execvp(args1[0], args1).

Con 2 chạy lệnh bên phải: dùng dup2(pfd[0], STDIN_FILENO) để lấy stdin từ đầu đọc của pipe, đóng các đầu pipe không cần thiết, nếu có > thì áp dụng chuyển hướng output ở vế phải, rồi execvp(args2[0], args2).

Cuối cùng, tiến trình cha đóng cả pfd[0] và pfd[1] (vì cha không dùng pipe) và waitpid() lần lượt để chờ cả hai tiến trình con kết thúc. Nhờ cách này, shell mô phỏng đúng hành vi pipeline: lệnh trái phát dữ liệu vào pipe, lệnh phải đọc dữ liệu từ pipe và cho ra kết quả tương ứng.

Kết quả



Nhận xét

Dựa trên kết quả chạy trong hình, chương trình 6.4.4 đã hỗ trợ đúng chức năng pipe. Khi thực hiện lệnh ls -l | wc -l, shell trả về kết quả 14, cho thấy đầu ra của ls -l đã được truyền trực tiếp qua pipe làm đầu vào cho wc -l để đếm số dòng. Tiếp theo, lệnh ls | wc -l > count.txt cũng cho kết quả hợp lý khi chương trình vừa dùng pipe để nối hai lệnh, vừa chuyển hướng đầu ra của lệnh bên phải vào file count.txt. Sau đó, cat < count.txt in lại đúng 14, chứng tỏ phần kết hợp giữa pipe và chuyển hướng > (ở vế phải) hoạt động đúng và dữ liệu đã được ghi vào file thành công.

Ngoài ra, trường hợp gõ dính ls|wc -l vẫn cho ra 14, chứng tỏ chương trình đã xử lý tốt việc chuẩn hoá toán tử | (không bắt buộc phải có khoảng trắng khi nhập). Cuối cùng, khi nhập sai cú pháp như ls | thì shell báo pipe syntax error và không bị treo hay thoát đột ngột, cho thấy chương trình có cơ chế phát hiện lỗi cú pháp pipe và xử lý an toàn. Nhìn chung, kết quả thực nghiệm xác nhận bài 6.4.4 đã hoàn thành đúng phần mở rộng: thực thi pipeline giữa hai lệnh, hỗ trợ cả trường hợp nhập dính và kết hợp được với chuyển hướng đầu ra ở phía sau pipeline.

5. Kết thúc lệnh đang thực thi bằng tổ hợp phím Ctrl + C

Source code

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

```c
#include <termios.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>

#define MAX_LINE 80
#define HISTORY_MAX 100

static struct termios OrigTermios;

static volatile sig_atomic_t SigintReceived = 0;
static volatile sig_atomic_t ChildRunning = 0;

static void OnSigint(int sig) {
    (void)sig;
    SigintReceived = 1;
}

static void RestoreTermios(void) {
    tcsetattr(STDIN_FILENO, TCSAFLUSH, &OrigTermios);
}

static void SetRawMode(void) {
    struct termios raw = OrigTermios;
    raw.c_lflag &= ~(ICANON | ECHO);
    raw.c_iflag &= ~(IXON | ICRNL);
    raw.c_cc[VMIN] = 1;
    raw.c_cc[VTIME] = 0;
    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

static void AddHistory(char History[HISTORY_MAX][MAX_LINE + 1], int
*HistoryCount, const char *Cmd) {
    if (Cmd[0] == '\0') return;
    if (*HistoryCount > 0 && strcmp(History[*HistoryCount - 1], Cmd) == 0)
return;

    if (*HistoryCount < HISTORY_MAX) {
        strncpy(History[*HistoryCount], Cmd, MAX_LINE);
        History[*HistoryCount][MAX_LINE] = '\0';
        (*HistoryCount)++;
    } else {
        for (int i = 1; i < HISTORY_MAX; i++) strcpy(History[i - 1],
History[i]);
        strncpy(History[HISTORY_MAX - 1], Cmd, MAX_LINE);
```

```
        History[HISTORY_MAX - 1][MAX_LINE] = '\0';
    }
}

static void RedrawLine(const char *Prompt, const char *Buf) {
    const char *clr = "\r\033[2K";
    write(STDOUT_FILENO, clr, (int)strlen(clr));
    write(STDOUT_FILENO, Prompt, (int)strlen(Prompt));
    write(STDOUT_FILENO, Buf, (int)strlen(Buf));
}

static int ReadLineWithHistory(const char *Prompt, char *Out, int OutMax,
char History[HISTORY_MAX][MAX_LINE + 1], int HistoryCount) {
    int len = 0;
    int hist_pos = HistoryCount;
    int navigating = 0;
    char cur_saved[MAX_LINE + 1];
    cur_saved[0] = '\0';
    Out[0] = '\0';

    RedrawLine(Prompt, Out);

    while (1) {
        unsigned char c;
        ssize_t n = read(STDIN_FILENO, &c, 1);

        if (n < 0) {
            if (errno == EINTR) {
                if (SigintReceived) {
                    SigintReceived = 0;
                    Out[0] = '\0';
                    write(STDOUT_FILENO, "\n", 1);
                    return 1;
                }
                continue;
            }
            return 0;
        }

        if (n == 0) return 0;

        if (c == 4) return 0;

        if (c == '\n' || c == '\r') {
            write(STDOUT_FILENO, "\n", 1);
```

```
            Out[len] = '\0';
            return 1;
        }

        if (c == 127 || c == 8) {
            if (len > 0) {
                len--;
                Out[len] = '\0';
                RedrawLine(Prompt, Out);
            }
            continue;
        }

        if (c == 27) {
            unsigned char seq[2];
            if (read(STDIN_FILENO, &seq[0], 1) <= 0) continue;
            if (read(STDIN_FILENO, &seq[1], 1) <= 0) continue;

            if (seq[0] == '[' && (seq[1] == 'A' || seq[1] == 'B')) {
                if (HistoryCount == 0) continue;

                if (!navigating) {
                    navigating = 1;
                    strncpy(cur_saved, Out, MAX_LINE);
                    cur_saved[MAX_LINE] = '\0';
                    hist_pos = HistoryCount;
                }

                if (seq[1] == 'A') {
                    if (hist_pos > 0) hist_pos--;
                } else {
                    if (hist_pos < HistoryCount) hist_pos++;
                }

                if (hist_pos == HistoryCount) {
                    strncpy(Out, cur_saved, OutMax - 1);
                    Out[OutMax - 1] = '\0';
                } else {
                    strncpy(Out, History[hist_pos], OutMax - 1);
                    Out[OutMax - 1] = '\0';
                }

                len = (int)strlen(Out);
                RedrawLine(Prompt, Out);
            }
```

```
            continue;
        }

        if (c >= 32 && c <= 126) {
            if (len < OutMax - 1) {
                Out[len++] = (char)c;
                Out[len] = '\0';
                RedrawLine(Prompt, Out);
            }
            continue;
        }
    }
}

static void NormalizeOps(const char *In, char *Out, int OutMax) {
    int j = 0;
    for (int i = 0; In[i] != '\0' && j < OutMax - 1; i++) {
        if (In[i] == '<' || In[i] == '>' || In[i] == '|') {
            if (j < OutMax - 1 && (j == 0 || Out[j - 1] != ' ')) Out[j++]
= ' ';
            if (j < OutMax - 1) Out[j++] = In[i];
            if (j < OutMax - 1) Out[j++] = ' ';
        } else {
            Out[j++] = In[i];
        }
    }
    Out[j] = '\0';
}

static char *Trim(char *s) {
    while (*s == ' ' || *s == '\t') s++;
    if (*s == 0) return s;
    char *end = s + strlen(s) - 1;
    while (end > s && (*end == ' ' || *end == '\t')) end--;
    end[1] = '\0';
    return s;
}

static int ParseArgsAndRedir(char *Line, char *args[], char **InFile, char
**OutFile) {
    *InFile = NULL;
    *OutFile = NULL;

    int i = 0;
    char *tok = strtok(Line, " \t");
```

```
    while (tok != NULL) {
        if (strcmp(tok, "<") == 0) {
            tok = strtok(NULL, " \t");
            if (tok == NULL) return 0;
            *InFile = tok;
        } else if (strcmp(tok, ">") == 0) {
            tok = strtok(NULL, " \t");
            if (tok == NULL) return 0;
            *OutFile = tok;
        } else {
            if (i < MAX_LINE / 2) args[i++] = tok;
        }
        tok = strtok(NULL, " \t");
    }
    args[i] = NULL;
    return 1;
}

static int ApplyInRedir(const char *InFile) {
    if (InFile == NULL) return 1;
    int fd = open(InFile, O_RDONLY);
    if (fd < 0) return 0;
    if (dup2(fd, STDIN_FILENO) < 0) return 0;
    close(fd);
    return 1;
}

static int ApplyOutRedir(const char *OutFile) {
    if (OutFile == NULL) return 1;
    int fd = open(OutFile, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) return 0;
    if (dup2(fd, STDOUT_FILENO) < 0) return 0;
    close(fd);
    return 1;
}

static void WaitNoIntr(pid_t pid) {
    int st;
    while (waitpid(pid, &st, 0) < 0) {
        if (errno == EINTR) continue;
        break;
    }
}

int main(void) {
```

```
    char *args[MAX_LINE/2 + 1];
    int should_run = 1;

    char line[MAX_LINE + 1];
    char History[HISTORY_MAX][MAX_LINE + 1];
    int HistoryCount = 0;

    tcgetattr(STDIN_FILENO, &OrigTermios);
    atexit(RestoreTermios);

    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = OnSigint;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGINT, &sa, NULL);

    while (should_run) {
        SetRawMode();

        if (!ReadLineWithHistory("it007sh>", line, (int)sizeof(line),
History, HistoryCount)) {
            break;
        }

        if (line[0] == '\0') continue;

        if (strcmp(line, "exit") == 0) {
            should_run = 0;
            continue;
        }

        AddHistory(History, &HistoryCount, line);

        char norm[4 * MAX_LINE + 1];
        NormalizeOps(line, norm, (int)sizeof(norm));

        char splitbuf[4 * MAX_LINE + 1];
        strncpy(splitbuf, norm, (int)sizeof(splitbuf) - 1);
        splitbuf[sizeof(splitbuf) - 1] = '\0';

        char *left = NULL;
        char *right = NULL;
        char *pipe_pos = strstr(splitbuf, " | ");
        if (pipe_pos != NULL) {
            *pipe_pos = '\0';
```

```
            left = Trim(splitbuf);
            right = Trim(pipe_pos + 3);
            if (left[0] == '\0' || right[0] == '\0') {
                RestoreTermios();
                fprintf(stderr, "it007sh: pipe syntax error\n");
                continue;
            }
        } else {
            left = Trim(splitbuf);
        }

        RestoreTermios();

        if (right == NULL) {
            char cmd1[4 * MAX_LINE + 1];
            strncpy(cmd1, left, (int)sizeof(cmd1) - 1);
            cmd1[sizeof(cmd1) - 1] = '\0';

            char *In1 = NULL;
            char *Out1 = NULL;
            if (!ParseArgsAndRedir(cmd1, args, &In1, &Out1) || args[0] ==
NULL) {
                fprintf(stderr, "it007sh: redirection syntax error\n");
                continue;
            }

            ChildRunning = 1;
            SigintReceived = 0;

            pid_t pid = fork();
            if (pid < 0) {
                perror("fork");
                ChildRunning = 0;
                continue;
            }

            if (pid == 0) {
                signal(SIGINT, SIG_DFL);
                if (!ApplyInRedir(In1)) { perror("open"); _exit(1); }
                if (!ApplyOutRedir(Out1)) { perror("open"); _exit(1); }
                execvp(args[0], args);
                perror("execvp");
                _exit(127);
            } else {
                WaitNoIntr(pid);
```

```
                ChildRunning = 0;
            }
        } else {
            char cmd1[4 * MAX_LINE + 1];
            char cmd2[4 * MAX_LINE + 1];
            strncpy(cmd1, left, (int)sizeof(cmd1) - 1);
            cmd1[sizeof(cmd1) - 1] = '\0';
            strncpy(cmd2, right, (int)sizeof(cmd2) - 1);
            cmd2[sizeof(cmd2) - 1] = '\0';

            char *args1[MAX_LINE/2 + 1];
            char *args2[MAX_LINE/2 + 1];
            char *In1 = NULL, *Out1 = NULL;
            char *In2 = NULL, *Out2 = NULL;

            if (!ParseArgsAndRedir(cmd1, args1, &In1, &Out1) || args1[0]
== NULL ||
                !ParseArgsAndRedir(cmd2, args2, &In2, &Out2) || args2[0]
== NULL) {
                fprintf(stderr, "it007sh: redirection syntax error\n");
                continue;
            }

            if (Out1 != NULL || In2 != NULL) {
                fprintf(stderr, "it007sh: unsupported redirection with
pipe\n");
                continue;
            }

            int pfd[2];
            if (pipe(pfd) < 0) {
                perror("pipe");
                continue;
            }

            ChildRunning = 1;
            SigintReceived = 0;

            pid_t pid1 = fork();
            if (pid1 < 0) {
                perror("fork");
                close(pfd[0]); close(pfd[1]);
                ChildRunning = 0;
                continue;
            }
```

```
            if (pid1 == 0) {
                signal(SIGINT, SIG_DFL);
                if (!ApplyInRedir(In1)) { perror("open"); _exit(1); }
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[0]);
                close(pfd[1]);
                execvp(args1[0], args1);
                perror("execvp");
                _exit(127);
            }

            pid_t pid2 = fork();
            if (pid2 < 0) {
                perror("fork");
                close(pfd[0]); close(pfd[1]);
                WaitNoIntr(pid1);
                ChildRunning = 0;
                continue;
            }

            if (pid2 == 0) {
                signal(SIGINT, SIG_DFL);
                dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
                close(pfd[1]);
                if (!ApplyOutRedir(Out2)) { perror("open"); _exit(1); }
                execvp(args2[0], args2);
                perror("execvp");
                _exit(127);
            }

            close(pfd[0]);
            close(pfd[1]);

            WaitNoIntr(pid1);
            WaitNoIntr(pid2);

            ChildRunning = 0;
        }
    }

    return 0;
}
```

Giải thích code

Trước hết, chương trình khai báo hai biến toàn cục kiểu volatile sig_atomic_t: SigintReceived để đánh dấu "vừa nhận Ctrl+C" và ChildRunning để đánh dấu "đang có lệnh chạy". Việc dùng sig_atomic_t giúp an toàn khi biến bị thay đổi trong signal handler. Signal handler OnSigint() được cài bằng sigaction(SIGINT, ...) và chỉ làm một việc đơn giản là set cờ SigintReceived = 1. Thiết kế này giúp shell không bị terminate khi người dùng bấm Ctrl+C, vì shell đã "bắt" SIGINT thay vì để mặc định.

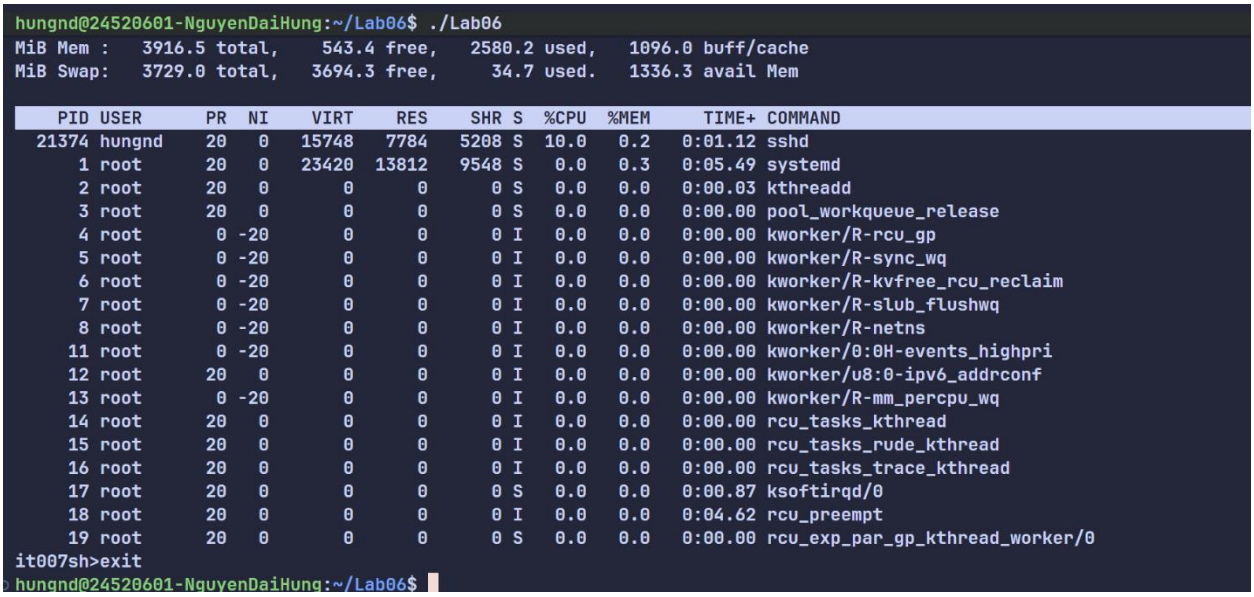
Trong ReadLineWithHistory(), chương trình xử lý trường hợp read() bị ngắt bởi tín hiệu (trả về lỗi và errno == EINTR). Khi gặp EINTR và thấy SigintReceived == 1, chương trình xoá cờ, xoá nội dung đang gõ dở (Out[0] = '\0'), in xuống dòng để "reset" giao diện và trả về để vòng lặp chính in prompt lại. Nhờ vậy, nếu người dùng bấm Ctrl+C ngay khi đang nhập lệnh (chưa Enter), shell sẽ bỏ dòng đang gõ và quay lại prompt, giống hành vi của terminal thật.

Phần quan trọng nhất là khi thực thi lệnh. Trước khi fork(), chương trình set ChildRunning = 1 và SigintReceived = 0. Khi fork() xong:

Ở tiến trình con, chương trình gọi signal(SIGINT, SIG_DFL) để đưa SIGINT về hành vi mặc định. Nhờ đó, khi người dùng bấm Ctrl+C, tiến trình con sẽ bị kết thúc (hoặc bị ngắt đúng như các chương trình bình thường). Sau đó con mới áp dụng redirect/pipe như các phần trước và gọi execvp().

Ở tiến trình cha (shell), chương trình không để waitpid() bị văng ra do EINTR làm shell "bỏ chờ" giữa chừng. Vì vậy có hàm WaitNoIntr() bọc waitpid() trong vòng lặp: nếu waitpid bị ngắt bởi tín hiệu và errno == EINTR thì tiếp tục chờ lại. Điều này đảm bảo shell vẫn đồng bộ đúng, và sau khi lệnh con bị Ctrl+C kết thúc thì shell quay lại prompt bình thường. Cuối cùng ChildRunning được trả về 0.

Kết quả



Nhận xét

Chương trình 6.4.5 đã thể hiện đúng mục tiêu của bài là xử lý Ctrl+C để dừng lệnh đang chạy nhưng không làm thoát shell. Cụ thể, sau khi chạy ./6.4.5, khi thực hiện các lệnh chạy lâu như sleep 10 và đặc biệt là top, chương trình vẫn giữ được trạng thái shell ổn định: lệnh top chạy bình thường trong chế độ tương tác và sau đó shell vẫn quay lại prompt it007sh> để tiếp tục nhận lệnh mới. Việc tiếp tục nhập và thực thi các lệnh như echo waiting, echo complete và nhận đúng output (complete) cho thấy sau khi dừng/thoát chương trình con, shell không bị "chết", không bị treo terminal và vẫn hoạt động đúng luồng.

Ngoài ra, kết quả cũng cho thấy shell xử lý đúng cơ chế chờ tiến trình con: sau khi các lệnh kết thúc, prompt xuất hiện trở lại để người dùng nhập tiếp, chứng tỏ phần đồng bộ (đợi tiến trình con) vẫn hoạt động đúng ngay cả với chương trình tương tác như top. Cuối cùng, lệnh exit vẫn kết thúc shell bình thường và trả về bash, chứng tỏ việc bổ sung xử lý tín hiệu ở 6.4.5 không làm ảnh hưởng đến chức năng thoát chương trình.