Báo cáo thực hành môn Hệ điều hành - Giảng viên: Trần Hoàng Lộc.

Họ và tên: Nguyễn Minh Nguyễn
Mã số sinh viên: 23521046
Lớp: IT007.P14

HỆ ĐIỀU HÀNH
BÁO CÁO LAB 6

CHECKLIST

6.4. BÀI TẬP THỰC HÀNH

	Câu 1	Câu 2	Câu 3	Câu 4	Câu 5
Trình bày giải thuật	☒	☒	☒	☒	☒
Chụp hình minh chứng (chạy ít nhất 3 lệnh)	☒	☒	☒	☒	☒
Giải thích code, kết quả	☒	☒	☒	☒	☒

Tự chấm điểm: 10

**Lưu ý: Xuất báo cáo theo định dạng PDF, đặt tên theo cú pháp:*
<Tên nhóm>_LAB6.pdf

6.4. BÀI TẬP THỰC HÀNH

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

#define MAX_CMD_LEN 1024
#define MAX_ARG_LEN 100
#define MAX_CMDS 10
#define HISTORY_SIZE 10 // Kích thước lịch sử lệnh

char *history[HISTORY_SIZE]; // Mảng để lưu trữ lịch sử lệnh
int history_count = 0; // Đếm số lệnh trong lịch sử

void add_to_history(const char *cmd) {
    if (history_count < HISTORY_SIZE) {
        history[history_count++] = strdup(cmd); // Lưu lệnh vào lịch sử
    } else {
        free(history[0]); // Giải phóng lệnh cũ nhất
        for (int i = 1; i < HISTORY_SIZE; i++) {
            history[i - 1] = history[i]; // Di chuyển lệnh lên
        }
        history[HISTORY_SIZE - 1] = strdup(cmd); // Thêm lệnh mới vào cuối
    }
}

void print_history() {
    for (int i = 0; i < history_count; i++) {
        printf("%d: %s\n", i + 1, history[i]);
    }
}

void execute_command(char *cmd) {
    char *commands[MAX_CMDS];
    char *input_file = NULL;
    char *output_file = NULL;
```

```c
    int background = 0;

    // Phân tích cú pháp lệnh để tách các lệnh theo dấu '|'
    char *token = strtok(cmd, "|");
    int cmd_count = 0;

    while (token != NULL) {
        commands[cmd_count++] = token;
        token = strtok(NULL, "|");
    }
    commands[cmd_count] = NULL; // Kết thúc danh sách các lệnh

    int pipefd[2 * (cmd_count - 1)]; // Mảng để lưu file descriptors của các pipe

    // Tạo các pipe
    for (int i = 0; i < cmd_count - 1; i++) {
        if (pipe(pipefd + i * 2) < 0) {
            perror("pipe failed");
            exit(EXIT_FAILURE);
        }
    }

    for (int i = 0; i < cmd_count; i++) {
        char *args[MAX_ARG_LEN];
        token = strtok(commands[i], " \n");
        int j = 0;
        input_file = NULL;  // Reset input and output files for each command
        output_file = NULL;

        // Phân tích cú pháp để tìm tệp đầu vào và đầu ra
        while (token != NULL) {
            if (strcmp(token, "&") == 0) {
                background = 1; // Chạy lệnh ở chế độ nền
            } else if (strcmp(token, "<") == 0) {
                input_file = strtok(NULL, " \n"); // Lưu tên tệp đầu vào
            } else if (strcmp(token, ">") == 0) {
                output_file = strtok(NULL, " \n"); // Lưu tên tệp đầu ra
```

```
            } else {
                args[j++] = token; // Lưu các tham số lệnh
            }
            token = strtok(NULL, " \n");
        }
        args[j] = NULL; // Kết thúc danh sách tham số

        pid_t pid = fork();
        if (pid == 0) { // Tiến trình con
            // Kết nối đầu vào và đầu ra cho các lệnh trong pipeline
            if (i > 0) {
                dup2(pipefd[(i - 1) * 2], STDIN_FILENO); // Đầu vào từ pipe trước
            }
            if (i < cmd_count - 1) {
                dup2(pipefd[i * 2 + 1], STDOUT_FILENO); // Đầu ra vào pipe sau
            }

            // Xử lý tệp đầu vào
            if (input_file) {
                int fd = open(input_file, O_RDONLY);
                if (fd < 0) {
                    perror("open input file");
                    exit(EXIT_FAILURE);
                }
                dup2(fd, STDIN_FILENO); // Chuyển hướng đầu vào
                close(fd);
            }

            // Xử lý tệp đầu ra
            if (output_file) {
                int fd = open(output_file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
                if (fd < 0) {
                    perror("open output file");
                    exit(EXIT_FAILURE);
                }
                dup2(fd, STDOUT_FILENO); // Chuyển hướng đầu ra
                close(fd);
            }
```

```
            // Đóng tất cả các pipe descriptors
            for (int k = 0; k < 2 * (cmd_count - 1); k++) {
                close(pipefd[k]);
            }

            execvp(args[0], args); // Thực thi lệnh
            perror("execvp failed");
            exit(EXIT_FAILURE);
        } else if (pid < 0) {
            perror("fork failed");
        }
    }

    // Đóng tất cả các pipe descriptors trong tiến trình cha
    for (int i = 0; i < 2 * (cmd_count - 1); i++) {
        close(pipefd[i]);
    }

    // Đợi tiến trình con kết thúc nếu không chạy ở chế độ nền
    if (!background) {
        for (int i = 0; i < cmd_count; i++) {
            wait(NULL); // Đợi các tiến trình con
        }
    }
}

void sigint_handler(int sig) {
    printf("\nReceived SIGINT. Use 'exit' to quit the shell.\n");
}

int main() {
    char cmd[MAX_CMD_LEN];

    // Đăng ký hàm xử lý tín hiệu SIGINT
    signal(SIGINT, sigint_handler);

    while (1) {
        printf("it007sh> ");
        if (!fgets(cmd, sizeof(cmd), stdin)) {
```

```
            break; // Thoát nếu có lỗi khi đọc
        }

        // Nếu lệnh là "exit", thoát khỏi chương trình
        if (strncmp(cmd, "exit", 4) == 0) {
            break;
        }

        // Nếu lệnh là "HF", in ra lịch sử lệnh
        if (strncmp(cmd, "HF", 2) == 0) {
            print_history();
            continue; // Trở lại vòng lặp
        }

        // Thêm lệnh vào lịch sử
        add_to_history(cmd);
        execute_command(cmd);
    }

    // Giải phóng bộ nhớ cho lịch sử lệnh
    for (int i = 0; i < history_count; i++) {
        free(history[i]);
    }

    return 0;
}
```

Đoạn code của cả 5 câu

Câu 1:

Khi người dùng nhập lệnh như it007sh> cat abc.txt, chương trình sẽ tạo một tiến trình con để thực thi lệnh này. Giải thuật thực hiện bao gồm việc sử dụng hàm fork() để tạo tiến trình con, sau đó sử dụng execvp() để thay thế tiến trình con bằng chương trình thực thi lệnh cat. Trong khi tiến trình con đang hoạt động, shell sẽ không cho phép người dùng nhập lệnh mới cho đến khi lệnh cat hoàn tất. Sau khi lệnh thực thi xong, dấu nhắc it007sh> sẽ được hiển thị lại, cho phép người dùng nhập lệnh tiếp theo.

Trong đoạn mã trên, phần thực thi lệnh được thực hiện trong hàm execute_command. Khi người dùng nhập lệnh như cat abc.txt, hàm này được gọi để xử lý lệnh.

```
    // Phân tích cú pháp để tìm tệp đầu vào và đầu ra
    while (token != NULL) {
        if (strcmp(token, "&") == 0) {
            background = 1; // Chạy lệnh ở chế độ nền
        } else if (strcmp(token, "<") == 0) {
            input_file = strtok(NULL, " \n"); // Lưu tên tệp đầu vào
        } else if (strcmp(token, ">") == 0) {
            output_file = strtok(NULL, " \n"); // Lưu tên tệp đầu ra
        } else {
            args[j++] = token; // Lưu các tham số lệnh
        }
        token = strtok(NULL, " \n");
    }
    args[j] = NULL; // Kết thúc danh sách tham số
```

Tách lệnh: Đầu tiên, lệnh được tách thành các phần tử nhỏ hơn bằng cách sử dụng strtok, nhằm phân chia các lệnh (nếu có nhiều lệnh được phân cách bằng ký tự |).

```
pid_t pid = fork();
if (pid == 0) { // Tiến trình con
    // Kết nối đầu vào và đầu ra cho các lệnh trong pipeline
    if (i > 0) {
        dup2(pipefd[(i - 1) * 2], STDIN_FILENO); // Đầu vào từ pipe trước
    }
    if (i < cmd_count - 1) {
        dup2(pipefd[i * 2 + 1], STDOUT_FILENO); // Đầu ra vào pipe sau
    }

    // Xử lý tệp đầu vào
    if (input_file) {
```

Tạo tiến trình con: Vòng lặp for sẽ lặp qua từng lệnh tách được. Trong mỗi vòng lặp, fork() được gọi để tạo một tiến trình con. Nếu fork() trả về 0, có nghĩa là chúng ta đang trong tiến trình con.

```
        execvp(args[0], args); // Thực thi lệnh
        perror("execvp failed");
        exit(EXIT_FAILURE);
    } else if (pid < 0) {
        perror("fork failed");
    }
```

Thực thi lệnh: Trong tiến trình con, execvp(args[0], args) được gọi để thay thế tiến trình con bằng chương trình thực thi lệnh cat (hoặc bất kỳ lệnh nào khác). Nếu lệnh thực thi thành công, tiến trình con sẽ không trở lại shell.

Chờ lệnh hoàn tất: Trong tiến trình cha, sau khi gọi execute_command, shell sẽ đợi cho tiến trình con hoàn tất bằng cách sử dụng wait(NULL) (nếu không chạy ở chế độ nền), đảm bảo rằng người dùng không thể nhập lệnh mới cho đến khi lệnh trước đó hoàn tất.

Kết quả:

```
┌──(kali㉿kali)-[~/OS_lab6]
└─$ ./lab
it007sh> ls
lab  lab.c
it007sh> touch text.txt
it007sh> ls
lab  lab.c  text.txt
it007sh> echo "Hello" > text.txt
it007sh> cat text.txt
"Hello"
it007sh>
```

Câu 2

Giải thuật:

l**ưu lệnh vào lịch sử**: Mỗi khi người dùng nhập lệnh, lệnh đó sẽ được thêm vào mảng lịch sử bằng cách gọi hàm add_to_history.

Hiển thị lịch sử: Khi người dùng nhập HF, hàm print_history sẽ được gọi để in ra tất cả các lệnh đã lưu trong mảng lịch sử.

Giải thích code:

```c
char *history[HISTORY_SIZE]; // Mảng để lưu trữ lịch sử lệnh
int history_count = 0; // Đếm số lệnh trong lịch sử

void add_to_history(const char *cmd) {
    if (history_count < HISTORY_SIZE) {
        history[history_count++] = strdup(cmd); // Lưu lệnh vào lịch sử
    } else {
        free(history[0]); // Giải phóng lệnh cũ nhất
        for (int i = 1; i < HISTORY_SIZE; i++) {
            history[i - 1] = history[i]; // Di chuyển lệnh lên
        }
        history[HISTORY_SIZE - 1] = strdup(cmd); // Thêm lệnh mới vào cuối
    }
}

void print_history() {
    for (int i = 0; i < history_count; i++) {
        printf("%d: %s\n", i + 1, history[i]);
    }
}
```

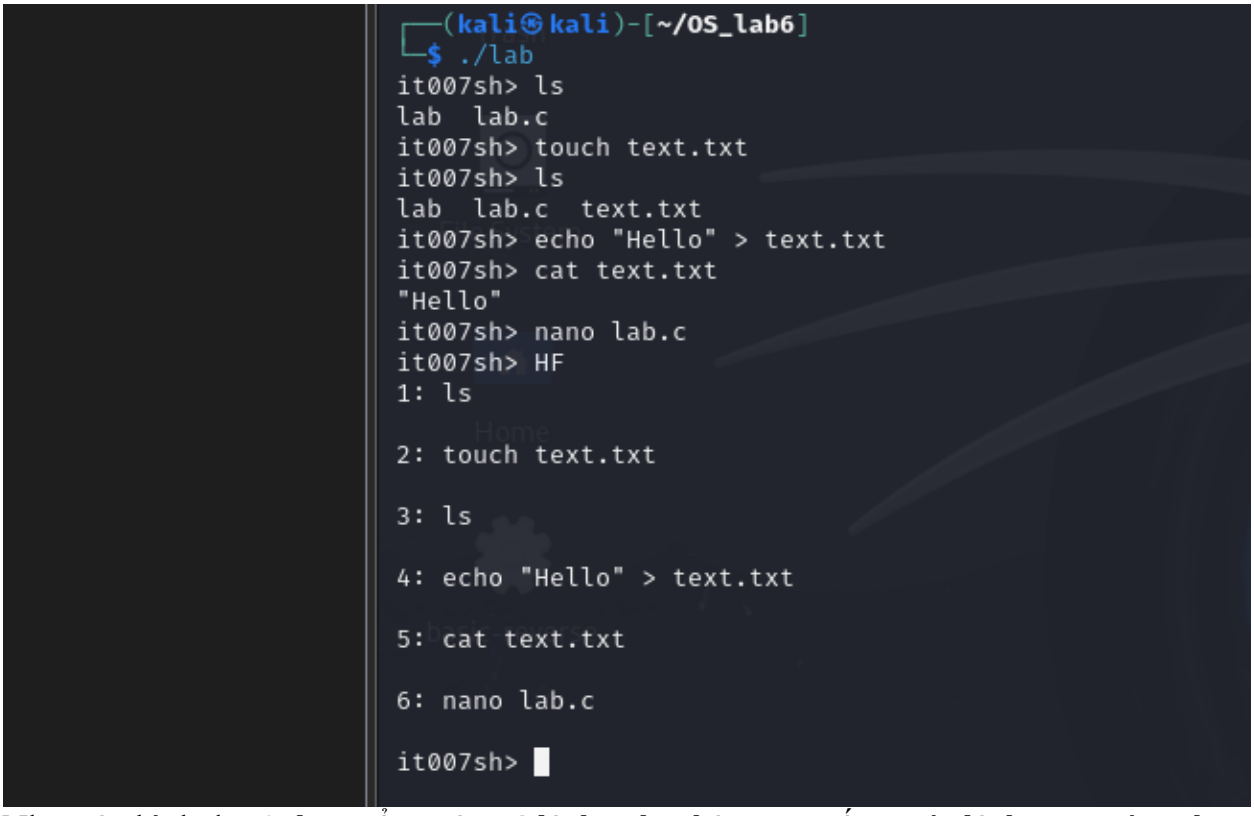
Mảng history được sử dụng để lưu trữ các lệnh đã thực thi, với kích thước được xác định bởi HISTORY_SIZE.

Hàm add_to_history kiểm tra xem mảng lịch sử đã đầy chưa. Nếu chưa, lệnh mới sẽ được thêm vào. Nếu đã đầy, lệnh cũ nhất sẽ được giải phóng và các lệnh khác sẽ được di chuyển lên để tạo chỗ cho lệnh mới.

Hàm print_history duyệt qua mảng lịch sử và in ra từng lệnh cùng với chỉ số tương ứng, giúp người dùng dễ dàng nhận diện các lệnh đã thực thi.

Khi người dùng nhập HF, chương trình kiểm tra và gọi print_history, sau đó quay lại vòng lặp để chờ lệnh tiếp theo mà không thực thi thêm lệnh nào.

Kết quả:



Như trên hình đã sử dụng tổng cộng 6 lệnh. Khi nhập HF xuất ra các lệnh tương ứng theo thứ tự từ cũ nhất đến mới nhất.

Câu 3:

Giải thuật:

Phân tích cú pháp lệnh: Khi người dùng nhập lệnh, chương trình sẽ phân tích xem có các ký tự > hoặc < trong lệnh hay không.

Xử lý chuyển hướng đầu ra: Nếu tìm thấy ký tự >, chương trình sẽ mở tệp đích và chuyển hướng đầu ra của lệnh đến tệp đó.

Xử lý chuyển hướng đầu vào: Nếu tìm thấy ký tự <, chương trình sẽ mở tệp nguồn và chuyển hướng đầu vào của lệnh từ tệp đó.

Giải thích code:

```
// Phân tích cú pháp để tìm tệp đầu vào và đầu ra
while (token != NULL) {
    if (strcmp(token, "&") == 0) {
        background = 1; // Chạy lệnh ở chế độ nền
    } else if (strcmp(token, "<") == 0) {
        input_file = strtok(NULL, " \n"); // Lưu tên tệp đầu vào
    } else if (strcmp(token, ">") == 0) {
        output_file = strtok(NULL, " \n"); // Lưu tên tệp đầu ra
    } else {
        args[j++] = token; // Lưu các tham số lệnh
    }
    token = strtok(NULL, " \n");
}
args[j] = NULL; // Kết thúc danh sách tham số
```

Trong vòng lặp này, chương trình sẽ kiểm tra từng phần của lệnh. Nếu gặp ký tự <, nó sẽ lưu tên tệp vào biến input_file. Nếu gặp ký tự >, nó sẽ lưu tên tệp vào biến output_file.

```
// Xử lý tệp đầu vào
if (input_file) {
    int fd = open(input_file, O_RDONLY);
    if (fd < 0) {
        perror("open input file");
        exit(EXIT_FAILURE);
    }
    dup2(fd, STDIN_FILENO); // Chuyển hướng đầu vào
    close(fd);
}
```

Nếu có tệp đầu vào được chỉ định, chương trình sẽ mở tệp đó với quyền đọc. Nếu thành công, nó sẽ sử dụng dup2() để chuyển hướng đầu vào của lệnh từ tệp.

```
// Xử lý tệp đầu ra
if (output_file) {
    int fd = open(output_file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open output file");
        exit(EXIT_FAILURE);
    }
    dup2(fd, STDOUT_FILENO); // Chuyển hướng đầu ra
    close(fd);
}
```

Tương tự, nếu có tệp đầu ra được chỉ định, chương trình sẽ mở tệp đó với quyền ghi (tạo mới nếu chưa tồn tại, và cắt bỏ nội dung nếu đã tồn tại). Sau đó, nó sẽ chuyển hướng đầu ra của lệnh đến tệp.

Kết quả:

```
it007sh> ls
lab  lab.c  text.txt
it007sh> touch ls.txt
it007sh> ls > ls.txt
it007sh> cat < ls.txt
lab
lab.c
ls.txt
text.txt
it007sh>
```

Như trên hình output của lệnh ls đã được ghi vào file ls.txt.

Câu 4:

Giải thuật:

Phân tích cú pháp lệnh: Khi người dùng nhập lệnh có chứa ký tự |, chương trình sẽ tách lệnh thành các thành phần trước và sau ký tự này.

Tạo các tiến trình con: Mỗi lệnh sẽ được thực thi trong một tiến trình con riêng biệt.

Thiết lập đường ống: Sử dụng hàm pipe() để tạo một đường ống, cho phép kết nối đầu ra của lệnh đầu tiên với đầu vào của lệnh thứ hai.
Giải thích code:

```
// Phân tích cú pháp lệnh để tách các lệnh theo dấu '|'
char *token = strtok(cmd, "|");
int cmd_count = 0;

while (token != NULL) {
    commands[cmd_count++] = token;
    token = strtok(NULL, "|");
}
commands[cmd_count] = NULL; // Kết thúc danh sách các lệnh
```

Đoạn mã này tách lệnh người dùng nhập vào thành các lệnh riêng biệt dựa trên ký tự |, lưu chúng vào mảng commands.

```
// Tạo các pipe
for (int i = 0; i < cmd_count - 1; i++) {
    if (pipe(pipefd + i * 2) < 0) {
        perror("pipe failed");
        exit(EXIT_FAILURE);
    }
}
```

Ở đây, chương trình tạo ra các đường ống cần thiết cho số lượng lệnh. Mỗi đường ống cần hai file descriptors (một cho đầu vào và một cho đầu ra).

```
if (pid == 0) { // Tiến trình con
    // Kết nối đầu vào và đầu ra cho các lệnh trong pipeline
    if (i > 0) {
        dup2(pipefd[(i - 1) * 2], STDIN_FILENO); // Đầu vào từ pipe trước
    }
    if (i < cmd_count - 1) {
        dup2(pipefd[i * 2 + 1], STDOUT_FILENO); // Đầu ra vào pipe sau
    }
```

Trong mỗi tiến trình con, nếu không phải là lệnh đầu tiên, chương trình sẽ chuyển hướng đầu vào từ đường ống trước đó. Nếu không phải là lệnh cuối cùng, chương trình sẽ chuyển hướng đầu ra vào đường ống kế tiếp.

```
// Đóng tất cả các pipe descriptors trong tiến trình cha
for (int i = 0; i < 2 * (cmd_count - 1); i++) {
    close(pipefd[i]);
}
```

Sau khi thiết lập kết nối, tất cả các file descriptors của đường ống sẽ được đóng trong tiến trình con để không gây rò rỉ tài nguyên.
Kết quả:

```
it007sh> nano text.txt
it007sh> cat text.txt | grep Hello
asdfHelloasdf
it007sh> ps aux | grep ps
root         488  0.0  0.0      0     0 ?        S    20:55   0:00 [psimon]
root        1008  0.0  0.4  13544  8192 ?        Ss   20:55   0:00 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
root        1032  0.0  0.0      0     0 ?        S    20:55   0:00 [psimon]
kali       46196  0.0  0.2  10872  4352 pts/0    R+   22:26   0:00 ps aux
kali       46197  0.0  0.1   6356  2176 pts/0    S+   22:26   0:00 grep ps
it007sh> ps aux | grep lab
kali       10772  0.0  0.0   2476  1536 pts/0    S+   21:13   0:00 ./lab
kali       46527  0.0  0.1   6356  2304 pts/0    S+   22:27   0:00 grep lab
it007sh>
```

Câu 5:

Giải thuật:

Đăng ký xử lý tín hiệu: Shell sẽ đăng ký một hàm xử lý tín hiệu cho tín hiệu SIGINT, cho phép chương trình xử lý khi người dùng nhấn Ctrl + C.

Gửi tín hiệu đến tiến trình: Khi tín hiệu SIGINT được nhận, hàm xử lý sẽ thực hiện việc dừng tiến trình hiện tại.

Giải thích code:

```
signal(SIGINT, sigint_handler);
```

Đoạn mã này đăng ký hàm sigint_handler để xử lý tín hiệu SIGINT. Khi người dùng nhấn Ctrl + C, tín hiệu này sẽ được gửi đến shell.

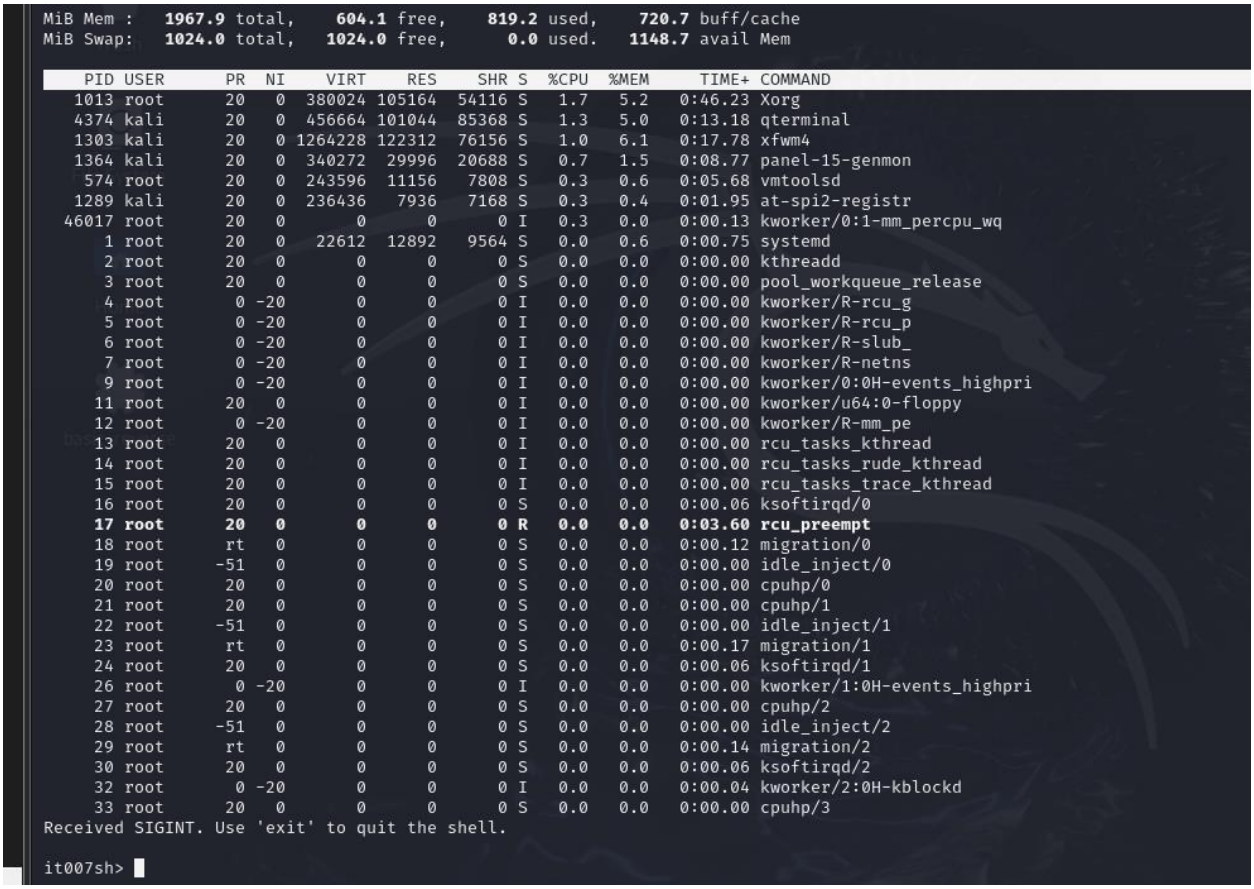
```
void sigint_handler(int sig) {
    printf("\nReceived SIGINT. Use 'exit' to quit the shell.\n");
}
```

Hàm này sẽ được gọi khi tín hiệu SIGINT được nhận. Nó in ra thông báo và cho phép người dùng biết rằng lệnh hiện tại đã bị dừng.

```
    // Đợi tiến trình con kết thúc nếu không chạy ở chế độ nền
    if (!background) {
        for (int i = 0; i < cmd_count; i++) {
            wait(NULL); // Đợi các tiến trình con
        }
    }
}
```

Nếu lệnh không chạy ở chế độ nền, shell sẽ chờ cho tiến trình con (lệnh đang chạy) kết thúc trước khi hiển thị lại dấu nhắc it007sh>.

Kết quả:



Lệnh top

Như trên hình khi lệnh top khi đang thực thi bị dừng bởi tín hiệu CTRL+C và quay trở lại giao diện người dùng