**TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI**



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

-------------------------------------------



**BÁO CÁO BÀI TẬP LỚN**

**MÔN HỆ ĐIỀU HÀNH**

**Đề tài: LẬP TRÌNH MỘT SHELL ĐƠN GIẢN**

**TRONG HỆ ĐIỀU HÀNH LINUX**

**Giảng viên: Phạm Đăng Hải**

**Thành viên nhóm:**

|  |  |
| --- | --- |
| Vũ Đức Hiếu | 20183916 |
| Kiều Minh Hướng | 20183928 |

Hà Nội, 2020

# **LỜI CẢM ƠN**

Đối với sinh viên năm 2 khoa Công nghệ thông tin, những môn học cốt lõi ngành như Nguyên lý hệ điều hành đóng một vai trò cực kì quan trọng trong việc tạo cho chúng em nền tảng tốt để có thể theo đuổi chuyên ngành mình mong ước. Trong quá trình học tập môn học này, chúng em đã nhận được sự chỉ bảo tận tình của thầy Phạm Đăng Hải. Điều đó khiến chúng em cảm thấy vô cùng vui mừng.

Sau khi thầy ra đề bài lập trình một shell đơn giản, chúng em cũng đã cố gắng hết sức để hoàn thành project này. Tuy nhiên, do kiến thức cũng như kinh nghiệm thực tế, tư duy còn nhiều hạn chế, giới hạn trong phạm vi kiến thức của sinh viên nên không thể tránh khỏi những thiếu sót. Rất mong nhận được sự quan tâm và đóng góp ý kiến của thầy để project này được hoàn thiện hơn.

Chúng em xin chân thành cảm ơn!

# **MỤC LỤC**

[**LỜI CẢM ƠN** 1](#_Toc43674469)

[**MỤC LỤC** 2](#_Toc43674470)

[**LỜI MỞ ĐẦU** 3](#_Toc43674471)

[**CHƯƠNG I: VÒNG LẶP CỦA SHELL** 5](#_Toc43674472)

[1.1. Phân tích vòng lặp shell 5](#_Toc43674473)

[1.2. Đọc một lệnh từ stdin 5](#_Toc43674474)

[1.3. Phân tích lệnh 7](#_Toc43674475)

[1.4. Thực hiện lệnh 9](#_Toc43674476)

[1.5. Xây dựng shell 11](#_Toc43674477)

[**CHƯƠNG II: CÁC DÒNG LỆNH TRONG SHELL** 16](#_Toc43674478)

[2.1. Lệnh “cd” 16](#_Toc43674479)

[2.2. Lệnh “date” 16](#_Toc43674480)

[2.3. Lệnh “send” và “receive” 18](#_Toc43674481)

[2.4. Lệnh “interrupt” 21](#_Toc43674482)

[2.5. Lệnh “environment” 23](#_Toc43674483)

[2.6. Lệnh “list” 24](#_Toc43674484)

[2.7. Lệnh “help” và “exit” 25](#_Toc43674485)

[2.8. Chạy chương trình 25](#_Toc43674486)

[**KẾT LUẬN** 26](#_Toc43674487)

[**TÀI LIỆU THAM KHẢO** 27](#_Toc43674488)

[**PHỤ LỤC: Một số hàm mới sử dụng và thư viện** 28](#_Toc43674489)

# **LỜI MỞ ĐẦU**

**1. Lý do chọn đề tài**

Đối với hầu hết tất cả mọi người, việc chuyển từ sử dụng hệ điều hành Windows sang hệ điều hành Linux sẽ gặp rất nhiều bỡ ngỡ. Chúng ta sẽ đặt ra những câu hỏi như: Tại sao lại sử dụng giao diện dòng lệnh, không sử dụng giao diện đồ họa; tại sao không chia thành các ổ đĩa như Windows; tại sao không có Microsoft Office; shell là gì, kernel là gì?... Tuy nhiên, sau một thời gian sử dụng, ta sẽ thấy rất nhiều những ưu điểm của Linux so với Windows nói riêng và so với một số hệ điều hành khác nói chung, ví dụ như Linux là mã nguồn mở miễn phí, hỗ trợ tốt cho lập trình viên – quản trị mạng, linh hoạt, độ an toàn với khả năng bảo mật, hạn chế sự tấn công của các mã độc và virus cao, hoạt động tốt trên những máy tính cấu hình yếu, …

Từ những ngày đầu tiên Linux xuất hiện, nhiều người đã quan tâm đến cơ chế quản lý tiến trình trên Linux. Các bản phân phối Linux cũng giống như các hệ điều hành hiện đại ngày nay, hoạt động theo cơ chế đa nhiệm, tức là trong cùng 1 thời điểm có thể có nhiều chương trình cùng (có vẻ) thực thi tại 1 thời điểm. Tất nhiên thực tế điều này không bao giờ xảy ra, các chương trình đã được phân chia thời gian hoạt động và hệ điều hành điều phối hoạt động tốt đến mức ta không nhận ra được các chương trình thực tế đang chạy tuần tự mà nghĩ rằng nó đang chạy song song.

Nhận thấy việc tìm hiểu sâu hơn về quản lý tiến trình trong Linux là cực kì cần thiết, nhóm chúng em đã quyết định lập trình một shell đơn giản bằng ngôn ngữ C, với mục đích nghiên cứu các hàm và cơ chế quản lý tiến trình trong hệ điều hành Linux. Hi vọng việc làm này sẽ được nhận sự đánh giá tích cực của thầy Phạm Đăng Hải cũng như sự quan tâm của các bạn cùng học phần.

**2. Tổng quan về đề tài**

Linux là một hệ điều hành mã nguồn mở, vì vậy tài liệu về hệ điều hành này cũng vô cùng phong phú và đa dạng. Một trong số những quyển sách về Nguyên lý Hệ điều hành nói chung nổi tiếng nhất đó là Modern Operating System của giáo sư Tanenbaum. Đây là quyển sách gối đầu giường cho bất kỳ ai muốn trở thành một chuyên gia trong lĩnh vực Hệ điều hành. Quyển sách xây dựng cực kì chi tiết những khía cạnh trong hệ điều hành với lối diễn đạt dễ hiểu và hấp dẫn.

Phát triển ứng dụng trên nền tảng Linux đòi hỏi người lập trình phải biết rõ về cơ chế quản lý tiến trình cũng như quản lý bộ nhớ trên Linux, vì thế những tài liệu về lập trình shell trên Linux là cực kì đa dạng. Lướt qua một vòng trên github – một dịch vụ lưu trữ các source code hàng đầu – ta có thể tìm ra vô vàn những project tạo shell trong Linux của hàng ngàn người trên thế giới. Tuy nhiên nếu chỉ đọc ở đó, ta không thể nào hiểu được hoạt động của các hàm cũng như bản chất của cơ chế quản lý tiến trình trong Linux.

Tài liệu tiếng Việt về lập trình shell còn khá hiếm, rất ít tác giả nghiên cứu và công bố công trình nghiên cứu của mình thành sách. Trong số các tài liệu ít ỏi đó có cuốn sách “Lập trình C/C++ trên Linux” của tác giả Nguyễn Trí Thành có ghi khá đầy đủ, chi tiết về những hàm quản lý bằng ngôn ngữ C trên Linux.

**3. Mục đích nghiên cứu**

Chúng em nghiên cứu đề tài này nhằm củng cố lại kiến thức mình đã học được trong môn hệ điều hành, đồng thời trau dồi khả năng, tư duy lập trình cũng như tìm hiểu thêm về một số hàm xử lý trong ngôn ngữ C – ngôn ngữ được coi là mạnh mẽ và có thể can thiệp sâu vào phần cứng và hệ điều hành.

**4. Phạm vi nghiên cứu**

Đề tài này tập trung nghiên cứu các hàm tạo tiến trình, quản lý file, hàng đợi thông điệp, một số hàm xử lý thời gian, lệnh ngắt và biến môi trường.

**5. Đóng góp của đề tài**

Với việc thực hiện đề tài này, chúng em hi vọng rằng sẽ đóng góp được một chút kiến thức của mình vào kho tài liệu về việc nghiên cứu các hàm quản lý tiến trình trên Linux, đồng thời nhận được sự phản hồi tích cực của thầy Phạm Đăng Hải.

**6. Kết cấu bài báo cáo**

Ngoài lời cảm ơn, lời mở đầu, mục lục, kết luận và tài liệu tham khảo, phần chính báo cáo được chia làm 2 phần:

Chương 1: Vòng lặp của shell

Chương 2: Các dòng lệnh trong shell

# **CHƯƠNG I: VÒNG LẶP CỦA SHELL**

## ***1.1. Phân tích vòng lặp shell***

Một shell thực hiện 3 nhiệm vụ cơ bản sau:

- Khởi tạo: shell được khởi tạo bằng việc chúng ta phiên dịch chương trình tạo shell.

- Phiên dịch: shell đọc các lệnh từ standard input (có thể là lệnh nhập của người dùng hoặc tệp) và thực thi chúng.

- Kết thúc: Sau khi các lệnh của nó được thực thi, shell sẽ nhận lệnh ngắt hoặc tắt nó, giải phóng mọi bộ nhớ và chấm dứt.

Các bước này rất chung chung để ta áp dụng trong nhiều trường hợp, tuy nhiên chúng em dùng nó làm cơ sở để tạo shell của mình. Shell của chúng em khá đơn giản, không có khả năng đọc tệp. Chúng em chỉ gọi hàm lặp và sau đó chấm dứt. Nhưng thực tế thời gian tồn tại của chương trình không đơn thuần chỉ là vòng lặp. Điều này chúng em sẽ giải thích ở phần sau.

Chúng em sẽ xây dựng một shell sử dụng Foreground Process.

## ***1.2. Đọc một lệnh từ stdin***

Đọc một lệnh nghe có vẻ đơn giản, tuy nhiên trong C đó là một điều khá phức tạp. Chúng ta không thể biết rằng người dùng sẽ nhập vào bao nhiêu kí tự vào shell. Chúng ta không thể phân bổ một vùng bộ nhớ và hi vọng rằng họ sẽ nhập số lượng kí tự không vượt qua nó. Thay vào đó, ta nên bắt đầu bằng một vùng bộ nhớ cố định, và khi người dùng nhập lệnh vượt qua nó, ta sẽ phân thêm bộ nhớ. Đây là một chiến lược phổ biến trong C. Từ đó ta xây dựng được hàm shell\_read\_line() để đọc một dòng:

#define SHELL\_RL\_BUFSIZE 1024 // Phan bo vung nho

char \*shell\_read\_line(void)

{

int bufsize = SHELL\_RL\_BUFSIZE;

int position = 0;

char \*buffer = malloc(sizeof(char) \* bufsize);

int c;

if (!buffer) {

fprintf(stderr, "\aError: allocation error\n");

exit(EXIT\_FAILURE);

}

while (1) {

// Doc ki tu

c = getchar();

// Neu enter thi la xong

if (c == '\n') {

printf("\a");

buffer[position] = '\0';

return buffer;

} else {

buffer[position] = c;

}

position++;

// Neu vuot qua bo dem thi phan bo lai va bao loi

if (position >= bufsize) {

bufsize += SHELL\_RL\_BUFSIZE;

buffer = realloc(buffer, bufsize);

if (!buffer) {

fprintf(stderr, "\aError: allocation error\n");

exit(EXIT\_FAILURE);

}

}

}

}

Ngoài các thư viện, dòng đầu tiên ta định nghĩa vùng nhớ tối đa là 1024 kí tự. Thân hàm nằm trong vòng lặp while(1), phần trên chỉ để kiểm tra xem có thể tạo ra một con trỏ trỏ tới vùng nhớ hay không, nếu không thì báo lỗi và thoái với câu lệnh exit(EXIT\_FAILURE). Trong vòng lặp, ta đọc một kí tự, và nếu kí tự đó là phím enter, ta sẽ gán cho buffer[position] giá trị NULL để báo kết thúc xâu và trả về xâu này, nếu không, ta sẽ đặt kí tự vào mảng và tiếp tục duyệt.

Sau đó ta kiểm tra nếu người dùng nhập vào một lệnh lớn hơn 1024 kí tự, kích cỡ của bộ nhớ sẽ tăng thêm 1024 và tiếp tục đọc dữ liệu. Kết quả cuối cùng sẽ trả về câu lệnh người dùng gõ vào.

## ***1.3. Phân tích lệnh***

Qua hàm shell\_read\_line() ta đã đọc được một dòng đầu vào. Ta sẽ cần đơn giản hóa việc shell thực hiện lệnh. Chúng ta sẽ không có những đối số đi kèm (-a -b -c… trong Linux) hay gạch | để thể hiện có nhiều câu lệnh khác nhau. Thay vào đó, chúng ta sẽ chỉ sử dụng khoảng trắng để tách các câu lệnh khỏi nhau. Ví dụ lệnh “send receive” sẽ không thực hiện lệnh như vậy mà thực tế sẽ thực hiện lệnh send trước lệnh receive.

#define SHELL\_TOK\_BUFSIZE 64

#define SHELL\_TOK\_DELIM " \t\r\n\a"

char \*\*shell\_split\_line(char \*line)

{

int bufsize = SHELL\_TOK\_BUFSIZE, position = 0;

char \*\*tokens = malloc(bufsize \* sizeof(char\*));

char \*token;

if (!tokens) {

fprintf(stderr, "\aError: allocation error\n");

exit(EXIT\_FAILURE);

}

token = strtok(line, SHELL\_TOK\_DELIM);

while (token != NULL) {

tokens[position] = token;

position++;

if (position >= bufsize) {

bufsize += SHELL\_TOK\_BUFSIZE;

tokens = realloc(tokens, bufsize \* sizeof(char\*));

if (!tokens) {

fprintf(stderr, "\aError: allocation error\n");

exit(EXIT\_FAILURE);

}

}

token = strtok(NULL, SHELL\_TOK\_DELIM);

}

tokens[position] = NULL;

return tokens;

}

Thoạt nhìn chúng ta có thể thấy nó tương tự chiến lược phân vùng bộ nhớ trong hàm shell\_read\_line(), tuy nhiên có điểm khác biệt đó là ta sử dụng một mảng các con trỏ kết thúc NULL thay vì mảng các kí tự kết thúc NULL. Hàm char \*strtok(char \*str, const char \*delim) chia chuỗi str thành một dãy các token được phân biệt riêng rẽ bởi dấu tách delim. Trong hàm shell\_split\_line() trên, ta sẽ tách chuỗi mà người dùng nhập vào thành các lệnh đơn nếu người dùng sử dụng phím Space, Enter, Tab.

Khi hàm bắt đầu, ta tách chuỗi thành các token, lưu trong mảng với con trỏ tokens trỏ tới một mảng char. Sau đó ta phân bổ lại vùng nhớ nếu bufsize đã đầy. Như vậy, một khi hàm thực hiện, ta sẽ nhận được một loạt các câu lệnh sẵn sàng để thực thi.

Nhưng câu hỏi đặt ra là chúng ta làm thế nào để thực hiện được nó?

## ***1.4. Thực hiện lệnh***

Trước khi bắt đầu viết các câu lệnh để thực hiện lệnh trong shell, ta cần phải biết cách các tiến trình làm việc trong Linux. Có 2 cách để khởi tạo một tiến trình trên Linux:

- Cách 1: Bằng việc sử dụng init. Init là tiến trình mặc định của Linux, nó bắt đầu chạy khi ta khởi động máy tính. Khi máy tính khởi động, nhân của nó được tải và bắt đầu tiến trình init. Quá trình này chạy trong toàn bộ thời gian máy tính được bật và nó quản lý tải lên phần còn lại của các tiến trình mà ta cần với lời gọi fork(). Khi chức năng này được gọi, hệ điều hành tạo ra một bản sao của tiến trình cha, gọi là tiến trình con. Lời gọi fork() trả về giá trị int là PID của tiến trình con, nếu bằng 0 có nghĩa là tiến trình không có tiến trình con nào. Về bản chất, điều này có nghĩa là cách duy nhất để tạo tiến trình là tạo ra bản sao của nó.

- Cách 2: Khi ta tạo tiến trình, điều ta muốn làm là một tiến trình mới chứ không phải là bản sao của tiến trình hiện tại. Vì vậy hệ điều hành cung cấp một lời gọi khác là lời gọi exec(). Nó tạo tiến trình mới thay thế tiến trình đang chạy. Điều này có nghĩa là khi ta gọi exec(), hệ điều hành sẽ ngừng tiến trình hiện tại, tải lên tiến trình mới và khởi động chương trình mới vào vị trí đó. Một tiến trình thì không thể được trả về khi gọi exec() trừ khi nó gặp lỗi.

Với 2 cách gọi hệ thống này, ta có thể xây dựng được các khối cho hầu hết chương trình bằng việc gọi fork() để sao chép tiến trình đang thực hiện và gọi exec() để thay thế tiến trình sao chép đó thành tiến trình mới. Tiến trình cha có thể tiếp tục thực hiện công việc dang dở hoặc dùng lệnh wait() để chờ tiến trình con kết thúc.

Sau đây là hàm shell\_launch để thực hiện lệnh đã được nhập từ người dùng:

int shell\_launch(char \*\*args){

pid\_t pid, wpid;

int status;

pid = fork();

if (pid == 0) {

// Tien trinh con

if (execvp(args[0], args) == -1) {

perror("\aError");

}

exit(EXIT\_FAILURE);

} else if (pid < 0) {

// Khong tao duoc tien trinh con

perror("\aError");

} else {

// Tien trinh cha

do {

wpid = waitpid(pid, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

return 1;

}

Đầu tiên, hàm lấy những đối số - chính là các lệnh mà người dùng nhập vào, sau đó tạo tiến trình con và lưu giá trị trả về. Sau khi lời gọi fork() trả về, ta sẽ có 2 tiến trình chạy song song. Tiến trình con sẽ kết thúc khi pid = 0 (tiến trình cha được trả về giá trị 0 – có nghĩa không còn tiến trình con của nó nữa).

Trong tiến trình con, ta chạy lệnh mà người dùng đưa ra. Ta sử dụng lệnh execvp(const char \*file, char \*const argv[]), một lệnh trong họ lệnh exec\_(), thay thế tiến trình hiện tại bằng chương trình khác được truyền qua tham số file. Như vậy nếu ở đoạn mã tiến trình con ta gọi hàm này thì hàm sẽ thực hiện lệnh khác, cụ thể là lệnh người dùng nhập vào.

Nếu lệnh execpv trả về -1 tức là đã có lỗi, vì thế ta dùng lệnh perror để in ra lỗi giúp người dùng biết lỗi từ đâu, sau đó thoát ra và tiếp tục sử dụng shell.

Điều kiện số 2, nếu lệnh fork() trả về giá trị âm thì không tạo được tiến trình con, và tiếp tục báo lỗi cho người dùng rồi tiếp tục shell.

Nếu tạo được tiến trình con thành công, điều ta phải làm là đưa ra câu lệnh waitpid(pid\_t pid, int \*status, int options). Câu lệnh này được dùng để yêu cầu tiến trình đợi một tiến trình khác với PID bằng pid. Nếu biến status khác NULL thì kết quả trả về của tiến trình sẽ được lưu ở biến được tham chiếu bởi con trỏ status. Tuy nhiên hệ điều hành chỉ dùng 8 bit thấp của số nguyên kiểu pid\_t để chứa kết quả trả về của một tiến trình, phần còn lại sẽ dùng với ý nghĩa khác. Do đó để thông dịch ý nghĩa của nó, ta có thể sử dụng một số macro:

- WIFEXITED(stat\_val): trả lại giá trị khác 0 nếu tiến trình đó thoát một cách bình thường.

- WEXITSTATUS(stat\_val): nếu macro WIFEXITED ở trên trả lại giá trị khác 0 thì macro này sẽ cho giá trị trả lại của tiến trình.

- WIFSIGNALED(stat\_val): nếu khác 0 thì có nghĩa là tiến trình bị ngắt bởi một tín hiệu không được quản lý.

- WTERMSIG(stat\_val): nếu macro WIFSIGNALED trả lại giá trị khác 0 thì macro này trả lại số hiệu của tín hiệu làm cho tiến trình dừng.

- WIFSTOPPED(stat\_val): trả lại giá trị khác 0 nếu tiến trình con đã dừng

- WSTOPSIG(stat\_val): nếu macro WIFSTOPPED trả lại giá trị khác 0 thì macro này trả lại số tín hiệu.

Như vậy tiến trình cha sẽ chờ tiến trình con cho đến khi nào các tiến trình con thoát hoặc bị ngắt.

\*/ Đọc lệnh, phân tích lệnh, thực hiện lệnh sẽ được đưa vào Header File “loop.h”.

## ***1.5. Xây dựng shell***

Hầu hết các lệnh mà shell thực thi là các chương trình, tuy nhiên một số thì lại không phải. Ví dụ lệnh chdir() sẽ thay đổi thư mục thực thi hiện tại. Vấn đề ở chỗ thư mục hiện tại là thuộc tính của một tiến trình, nếu ta thay đổi thư mục bằng một hàm chức năng, nó chỉ thay đổi thư mục đó rồi chấm dứt. Thư mục hiện tại của tiến trình cha sẽ không đổi. Vì thế, câu lệnh này phải thực hiện trực tiếp vào trong shell để khi gọi tiến trình con, nó sẽ kế thừa từ chính tiến trình cha đó. Như vậy, chúng ta phải thêm một số lệnh vào chính shell, cụ thể là 2 lệnh cd và exit.

Ngoài những lệnh #include và #define, chúng ta phải cần 3 phần:

- Phần 1: chứa các khai báo về các hàm chức năng:

int shell\_cd(char \*\*args);

int shell\_exit(char \*\*args);

- Phần 2: một mảng các tên lệnh xây dựng sẵn, theo sau là một mảng các hàm tương ứng (phải đúng thứ tự nếu không lệnh này sẽ chèn vào lệnh kia). Bằng việc xây dựng các mảng này, ta có thể thêm vào các lệnh mới một cách đơn giản.

char \*build\_str[] = {

"cd",

"exit"

};

int (\*functions[]) (char \*\*) = {

&shell\_cd,

&shell\_exit

};

int shell\_num\_builds() {

return sizeof(build\_str) / sizeof(char \*);

}

\*build\_str[] là một mảng con trỏ hàm (một mảng các chuỗi kí tự) chứa các câu lệnh mà người dùng có thể nhập vào. Thực tế, nó chứa những chuỗi kí tự, sau đó khi người dùng nhập một chuỗi, shell sẽ so sánh chuỗi đó với mảng này và thực hiện lệnh tương ứng, nếu không có, shell sẽ báo lỗi.

Hàm số 2 là một mảng các hàm mà ta sẽ xây dựng cho shell, nó trả về một giá trị int, điều này được sử dụng để sau này với các câu lệnh ta sẽ return 1, còn lệnh exit sẽ return 0 để thoát khỏi shell.

Hàm cuối là một hàm để trả về số lệnh đã được tạo.

Sau đó xây dựng các hàm chi tiết:

int shell\_cd(char \*\*args){

// Sẽ trình bày ở phần sau

return 1;

}

int shell\_exit(char \*\*args){

return 0;

}

Sau đó, ta phải kết hợp tất cả các hàm lại, thành một hàm thực thi:

int shell\_execute(char \*\*args){

int i;

if (args[0] == NULL) {

// Mot cau lenh trong

return 1;

}

for (i = 0; i < shell\_num\_builds(); i++) {

if (strcmp(args[0], build\_str[i]) == 0) {

return (\*functions[i])(args);

}

}

return shell\_launch(args);

}

Đầu tiên kiểm tra nếu người dùng chỉ Enter mà không nhập gì thì đơn giản ta chỉ cần bỏ qua và đợi nhận một câu lệnh mới.

Nếu người dùng nhập vào một câu lệnh, ta sẽ duyệt tuyến tính mảng lệnh đã xây dựng ở trên để tìm ra câu lệnh và thực hiện.

Sau đó là hàm loop:

void shell\_loop(void){

char \*line;

char \*\*args;

int status;

do {

printf(">");

line = shell\_read\_line();

args = shell\_split\_line(line);

status = shell\_execute(args);

free(line);

free(args);

} while (status);

}

Hàm loop hoàn toàn rõ ràng và dễ hiểu. Trong vòng lặp, ta in một dấu “>”, gọi một hàm để đọc một dòng và tách dòng thành các đối số rồi thực hiện. Cuối cùng ta giải phóng dòng và đối số thực hiện trước đó. Ta sử dụng biến status để xác định thời điểm thoát (lệnh exit trả về 0).

Cuối cùng là hàm main:

int main(int argc, char \*\*argv){

shell\_loop();

return EXIT\_SUCCESS;

}

Trong hàm main ta gọi vòng lặp và trả về giá trị kết thúc thành công.

# **CHƯƠNG II: CÁC DÒNG LỆNH TRONG SHELL**

## ***2.1. Lệnh “cd”***

Lệnh “cd” dùng để chuyển đổi thư mục mà tiến trình đang xử lí. Trong chương trình, khi muốn chuyển thư mục làm việc, ta dùng hàm chdir(const char \*path) với tham số path chứa đường dẫn đến thư mục cần chuyển. Khi thành công, chdir() trả về giá trị 0, ngược lại giá trị -1.

int shell\_cd(char \*\*args){

if (args[1] == NULL) {

fprintf(stderr, "\aError: expected argument to \"cd\"\n");

} else {

if (chdir(args[1]) != 0) {

perror("\aError");

}

}

return 1;

}

Đầu tiên ta kiểm tra sau lệnh “cd” nếu không có đối số truyền vào thì sẽ báo lỗi không có đối số, nếu có đối số ta kiểm tra nếu đối số truyền vào không tồn tại trong thư mục đó thì sẽ báo lỗi, nếu không có lỗi thì lệnh chdir() thực hiện thành công. Trong Linux, thư mục cha sẽ có giá trị “..” còn thư mục hiện tại sẽ có giá trị “.”. Để kiểm tra xem đã chuyển được thư mục chưa, ta có thể dùng lệnh list để kiểm tra các file trong thư mục hiện tại là rõ (sẽ trình bày ở phần sau).

## ***2.2. Lệnh “date”***

Lệnh “date” sẽ trả về giá trị ngày tháng năm và thời gian giờ phút giây. Lệnh được xây dựng trong Header File “getdate.h”. Lệnh xây dựng trong hàm main() tương tự như xây dựng lệnh cd và exit. Trong “getdate.h”, hàm được xây dựng như sau:

#include<time.h>

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<string.h>

void getdate(){

struct tm \*tm\_ptr;

time\_t the\_time;

char name\_of\_weekDay[10];

(void) time(&the\_time);

tm\_ptr = gmtime(&the\_time);

switch(tm\_ptr-> tm\_wday){

case 0: {strcpy(name\_of\_weekDay, "Sunday"); break;}

case 1: {strcpy(name\_of\_weekDay, "Monday"); break;}

case 2: {strcpy(name\_of\_weekDay, "Tuesday"); break;}

case 3: {strcpy(name\_of\_weekDay, "Wednesday"); break;}

case 4: {strcpy(name\_of\_weekDay, "Thursday"); break;}

case 5: {strcpy(name\_of\_weekDay, "Friday"); break;}

case 6: {strcpy(name\_of\_weekDay, "Saturday"); break;}

}

printf("Date: %s %02d/%02d/%d\n", name\_of\_weekDay, tm\_ptr->tm\_mday, tm\_ptr->tm\_mon+1, tm\_ptr -> tm\_year +1900);

printf("Time: %02d:%02d:%02d\n",tm\_ptr->tm\_hour, tm\_ptr->tm\_min, tm\_ptr->tm\_sec);

}

Trong thư viện “time.h”, kiểu thời gian được định nghĩa trong Linux là time\_t, là một biến kiểu nguyên. Trong “time.h” có lệnh time() để lấy thời gian tính theo giây tính từ 0h00 ngày 1/1/1970 – ngày ra đời của Unix. Để thời gian dễ hiểu hơn đối với con người, ta có hàm gmtime(time\_val) với time\_val là giá trị thời gian. Hàm sẽ trả về biến cấu trúc tm chứa các giá trị trực quan hơn: tm\_hour, tm\_mday,…

Ta có một hàm switch để lấy tên thứ trong ngày (0->6 tương ứng Sunday -> Saturday) và hàm printf để in ra giá trị.

## ***2.3. Lệnh “send” và “receive”***

Các lệnh “send” và “receive” mô tả quá trình truyền thông liên tiến trình theo cơ chế hàng đợi thông điệp (Message queue). Có nhiều cách truyền thông liên tiến trình như dùng đường ống, hàng đợi thông điệp, bộ nhớ chia sẻ, socket,… Dùng hàng đợi thông điệp cho ta khả năng gắn kiểu cho dữ liệu được truyền giữa 2 tiến trình và bài toán đồng bộ cũng không quá phức tạp. Tuy nhiên khi xây dựng 2 hàm này, nếu ta không thực hiện lệnh “send” trước mà đã thực hiện “receive” thì shell sẽ bị lỗi do không có hàng đợi thông điệp nào khi trong “receive” ta xây dựng hàm để xóa hàng đợi sau khi đọc.

- Lệnh “send” được đưa vào trong Header File “send.h” với chi tiết như sau:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<errno.h>

#include<unistd.h>

#include<sys/msg.h>

#define MAX\_TEXT 512

void send(){

struct my\_msg\_st{

long int my\_msg\_type;

char some\_text[MAX\_TEXT];

};

int running =1;

struct my\_msg\_st some\_data;

int msgid;

char buffer[BUFSIZ];

msgid = msgget ((key\_t) 1234, 0666 | IPC\_CREAT);

if (msgid == -1) {

fprintf(stderr, "message get failed with error: %d\n", errno);

exit(EXIT\_FAILURE);

}

printf("Type some text. Enter \"end\" to finish\n");

while(running){

printf("Enter text here: ");

fgets(buffer, BUFSIZ,stdin);

some\_data.my\_msg\_type =1;

strcpy(some\_data.some\_text, buffer);

if (msgsnd(msgid, (void \*)&some\_data, MAX\_TEXT, 0) ==-1) {

fprintf(stderr, "message send failed\n");

exit(EXIT\_FAILURE);

}

if (strncmp(buffer, "end", 3) ==0) running =0;

}

}

Hàm msgget(key\_t key, int msgflg) dùng để tạo một hàng đợi thông điệp cũng như truy cập hàng đợi. Tham số thứ nhất key là một giá trị khóa duy nhất trong hệ thống để tránh việc nhiều tiến trình cùng truy cập vào hàng đợi của hệ thống ứng dụng khác. Tham số thứ 2 msgflg là giá trị cờ truy nhập và có thể tổ hợp OR với giá trị macro đặc biệt là IPC\_CREAT để tạo ra một hàng đợi. Một macro khác là IPC\_PRIVATE ít dùng hơn vì nếu giao tiếp trong nội bộ tiến trình thì có nhiều cách tốt hơn là hàng đợi thông điệp. Nếu hàng đợi thông điệp đã có thì giá rtị IPC\_CREAT được bỏ qua (tức ta dùng lệnh “send” sau khi “send” một lần từ trước thì vẫn chỉ có 1 hàng đợi mà thôi). Nếu thành công, msgget sẽ trả về một giá trị là số hiệu/định dạng của hàng đợi, nếu không sẽ trả về giá trị -1.

Trong hình, ta tạo ra 1 cấu trúc ten là some\_data với 2 giá trị là kiểu dữ liệu và độ dài xâu kí tự cần gửi. Sau khi thiết lập biến trạng thái running =1, ta tạo ra một buffer để chứa thông điệp. Dùng biến msgid để kiểm tra xem có tạo được hàng đợi không, nếu không được thì báo lỗi, ngược lại yêu cầu người dùng nhập các thông điệp cần gửi. Hàm msgsnd(int msgid, const void \*msg\_ptr,size\_t msg\_sz,int msgflg) cho phép chèn thêm một dòng thông điệp vào trong hàng đợi có định danh msgid, được trỏ bởi con trỏ msg\_ptr, với kích thước thông điệp là msg\_sz được tính bằng công thức msg\_sz = sizeof(my\_message)-sizeof(long int). Tham số msgflg điều khiển cơ chế truyền thông khi hàng đợi đầy. Nếu nó là IPC\_NOWAIT thì hàm msgsnd() sẽ thoát luôn và trả lại giá trị -1 lỗi. Nếu giá trị là 0 thì hàm sẽ đợi hàng đợi có chỗ trống để gửi thông điệp. Cuối cùng là lệnh để kiểm tra nếu người dùng nhập “end” thì kết thúc gửi thông điệp.

- Lệnh “receive” được đưa vào trong Header File “receive.h” với chi tiết như sau:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<errno.h>

#include<unistd.h>

#include<sys/msg.h>

void receive(){

struct my\_msg\_st{

long int my\_msg\_type;

char some\_text[BUFSIZ];

};

int running = 1;

int msgid;

struct my\_msg\_st some\_data;

long int msg\_to\_receive = 0;

msgid = msgget((key\_t) 1234, 0666 | IPC\_CREAT);

if (msgid ==-1){

fprintf(stderr, "message get failed with error: %d\n", errno);

exit(EXIT\_FAILURE);

}

while(running){

if(msgrcv(msgid, (void \*)&some\_data, BUFSIZ, msg\_to\_receive, 0) == -1){

fprintf(stderr, "message receive failed with error: %d\n", errno);

exit(EXIT\_FAILURE);

}

printf("You wrote: %s", some\_data.some\_text);

if (strncmp(some\_data.some\_text, "end", 3)==0) running =0;

}

if (msgctl(msgid, IPC\_RMID,0)== -1) {

fprintf(stderr, "msgctl(IPC\_RMID) failed \n");

exit(EXIT\_FAILURE);

}

}

Hàm nhận thông điệp được xây dựng hoàn toàn tương tự, có lệnh msgrcv xây dựng tương tự msgsnd. Ngoài ra có hàm msgctl(int msgid, int msgflg, struct \*msgqid\_ds) để đóng hàng đợi hoặc xóa hàng đợi thông điệp khi kết thúc với msgid là chỉ số của hàng đợi, một cờ điều khiển, trong ví dụ này là cờ IPC\_RMID đó là xóa hàng đợi và tham số thứ 3 là con trỏ trỏ đến cấu trúc msgqid\_ds, ta để là 0.

## ***2.4. Lệnh “interrupt”***

Vì ta đang xây dựng một shell chạy Foreground process nên ta cần một lệnh để mô phỏng lệnh ngắt, khi nhấn Ctrl+C thì sẽ dừng tiến trình đang chạy. Lệnh “interrupt” xây dựng trong Header File “interrupt.h”:

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <signal.h>

void interrupt(){

int sig =0;

void sighandler(int signum)

{

printf("\nCatch signal %d, break!\n", signum);

sig =1;

}

signal(SIGINT, sighandler);

while(1) {

printf("Type \"Ctrl+C\" to end task\n");

sleep(1);

if(sig==1) break;

}

}

Tín hiệu là một cách thức điều khiển tiến trình, được dùng để báo cho tiến trình về một sự kiện nào đó. Nhiều người so sánh tín hiệu giống với ngắt của phần cứng. Khi có lệnh ngắt thì toàn bộ vi xử lí sẽ thực hiện lệnh ngắt trước, tương tự với tiến trình cũng vậy. Trong ví dụ này, ta có lệnh signal(int signum, function())với giá trị signum là tín hiệu cần bắt, trong ví dụ là tín hiệu SIGINT là khi ta nhấn tổ hợp Ctrl+C, tham số thứ 2 là con trỏ trỏ tới một hàm, tức là khi ta ấn Ctrl+C thì tiến trình sẽ thực thi hàm. Ta có một vòng lặp while, kết thúc khi giá trị kiểm tra sig =1. Giá trị này chỉ được gán bằng 1 khi thực hiện hàm sighandler với tham số signum là số hiệu của tham số, muốn vậy ta phải gửi tín hiệu SIGINT có số hiệu là 2 đến bằng việc ấn Ctrl+C. Tóm lại, khi ấn Ctrl+C thì hàm sẽ kết thúc.

## ***2.5. Lệnh “environment”***

Biến môi trường là các đại lượng có các giá trị cụ thể. Một số biến môi trường được cung cấp các giá trị đặt trước của hệ thống và các biến khác được đặt trực tiếp bởi người dùng, tại dòng lệnh hoặc trong khi khởi động các tập lệnh khác.

Biến môi trường là một chuỗi ký tự chứa thông tin được sử dụng bởi một hoặc nhiều ứng dụng. Linux định nghĩa rất nhiều biến môi trường dùng cho các mục đích khác nhau. Biến môi trường dùng kiểm soát và cung cấp thông tin để các chương trình và script của shell hoạt động, dùng cấu hình môi trường làm việc của người dùng.

Lệnh “environment” được xây dựng trong Header File “env.h”:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

extern char \*\*environ;

void env()

{

int count = 0;

printf("\n");

while(environ[count] != NULL)

{

printf("[%s] :: ", environ[count]);

count++;

}

char \*val = getenv("USER");

printf("\n\nCurrent value of environment variable USER is [%s]\n",val);

if(setenv("USER","Pham Dang Hai",1)){

printf("\n setenv() failed\n");

}

printf("\n Successfully Added a new value to existing environment variable USER\n");

val = getenv("USER");

printf("\nNew value of environment variable USER is [%s]\n",val);

}

Ta sử dụng một biến toàn cục environ để lấy giá trị biến môi trường trong Linux. Hàm char \* getenv (const char \* name) tìm kiếm biến môi trường được trỏ theo tên và trả về giá trị liên quan đến chuỗi. Trong ví dụ trên, tên biến môi trường được trỏ theo tên USER chính là tên của người sở hữu shell hiện tại. Sau đó ta sử dụng một hàm if để đặt giá trị biến môi trường mới cho biến được trỏ theo tên USER bằng hàm setenv(const char \*name, const char \*value, int overwrite) với tham số đầu tiên là USER, tham số thứ 2 là giá trị mới của biến môi trường (Trong trường hợp này ta lấy tên Thầy). Hàm trả về 0 nếu thành công và -1 nếu lỗi. Khi này nếu chạy lại lệnh “environment” ta sẽ thấy USER đã được thay đổi thành Pham Dang Hai.

## ***2.6. Lệnh “list”***

Lệnh “list” dùng để đưa ra danh sách những file đang hiện hữu ở trong tệp hiện tại, được xây dựng trong phần Header File “list.h”:

#include <dirent.h>

#include <stdio.h>

void list(){

DIR \*d;

struct dirent \*dir;

d = opendir(".");

if (d){

while ((dir = readdir(d)) != NULL){

printf("%s\n", dir->d\_name);

}

closedir(d);

}

}

Để duyệt nội dung của thư mục, ta có 2 cách. Cách thứ nhất là gọi lần lượt các file bằng các hàm opendir(), readdir(), closedir(), cách 2 là sử dụng scandir() (thuận tiện hơn). Ở đây ta dùng cách thứ nhất.

Đầu tiên một lệnh opendir(“.”) sẽ được thực thi, đó là mở ra thư mục hiện tại, nếu thành công sẽ trả về giá trị là một biến con trỏ trỏ đến đối tượng thư mục có cấu trúc DIR – đại diện cho các dòng thư mục là một chuỗi theo thứ tự của tất cả các file trong thư mục cụ thể. Sau khi thực hiện xong, con trỏ d sẽ trỏ đến từng file trong thư mục hiện tại. Hàm readdir(d) đọc một thư mục bằng cách trả về các mục liên tiếp trong một thư mục được trỏ đến bởi con trỏ d. Readdir() trả về một con trỏ tới cấu trúc dirent chứa thông tin về mục nhập thư mục tiếp theo. Readdir() di chuyển đến thư mục tiếp theo sau mỗi lần thực thi. Vì vậy khi không còn mục nào nữa tức câu lệnh while bị sai thì ta đã duyệt hết mục trong thư mục và thực hiện đóng thư mục bằng lệnh closedir().

## ***2.7. Lệnh “help” và “exit”***

Một lệnh “help” đơn giản là gồm những câu lệnh printf để đưa ra màn hình chức năng của từng lệnh sau đó return 1, còn lệnh exit thì return 0 để kết thúc shell.

## ***2.8. Chạy chương trình***

Để chạy chương trình, ta mở Terminal với thư mục hiện tại là nơi chứa các header file và hàm main.c. Tại đây ta thực hiện lệnh sau:

gcc main.c -o main -pthread

Do trong project có sử dụng thư viện pthread.h, ta cần thêm một tham số dòng lệnh -pthread để biên dịch. Sau khi biên dịch xong, ta gõ tiếp lệnh:

./main

để chạy chương trình

# **KẾT LUẬN**

Ngày nay, hệ điều hành Linux đang ngày một phát triển và được hàng trăm triệu người trên thế giới sử dụng. Tuy thực hiện trên giao diện dòng lệnh gây khó khăn cho người mới, nhưng khi đã quen sử dụng nó thì đây là một hệ điều hành tuyệt vời dành cho các lập trình viên và những người đam mê ngành Công nghệ thông tin.

Việc nghiên cứu về quản lí tiến trình cùng những khía cạnh khác trong Linux là một điều vô cùng lí thú và hấp dẫn đối với những người muốn tìm tòi.

Với quỹ thời gian của mình, chúng em cũng đã thành công trong việc xây dựng một shell đơn giản. Kết quả này thể hiện quá trình học tập cũng như nghiên cứu của chúng em, qua đó đã tự thu thập cho mình những kiến thức và kĩ năng cần thiết. Hi vọng rằng bài báo cáo cũng như mã nguồn chương trình có thể giúp cho người đọc phần nào nắm rõ về quản lí tiến trình trong hệ điều hành Linux và khơi gợi niềm đam mê về hệ điều hành Linux cũng như việc học lập trình.

# **TÀI LIỆU THAM KHẢO**

1. Nguyễn Trí Thành, Lập trình C/C++ trong Linux, NXB Giáo dục Việt Nam, 2010.

2. Andrew.S. Tanenbaum, Modern Operating System 5th edition, 2016.

3. Kay.A.Robbins, Steven Robbins, Unix System Programming Hardcover, 1995.

4. Một số hàm trên stackoverflow.

# **PHỤ LỤC**

**Một số hàm mới được sử dụng trong code và thư viện:**

|  |  |
| --- | --- |
| **Thư viện** | **Hàm** |
| sys/wait.h | wait(), waitpid() |
| unistd.h | fork(), exec(), pid\_t,… |
| stdlib.h | execvp(), getenv(), setenv(),… |
| string.h | strtok() |
| time.h | gmtime(), time(), time\_t |
| sys/msg.h | msgget(), msgsnd(), msgrcv(),… |
| signal.h | signal(), SIGINT,… |
| dirent.h | DIR, opendir(), readdir(), closedir(),… |