

Họ và tên: Nguyễn Tiến Đạt – Vũ Nguyễn Duy Linh – Đặng Quang Khánh Linh –
Nguyễn Đỗ Đức Minh
Mã số sinh viên: 22520226 – 22520780 – 22520756 - 22520972
Lớp: IT007.O14.1

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	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Chụp hình minh chứng (chạy ít nhất 3 lệnh)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Giải thích code, kết quả	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Tư chấm điểm: 9.5

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

Link code: <https://github.com/duylinh1510/LAB6-OS/blob/main/vinashell.c>

6.4. BÀI TẬP THỰC HÀNH

1. Câu 1: Thực thi command trong tiến trình con

Trình bày giải thuật & giải thích code:

(Trong code đã ghi chú thích ở từng dòng, nên ở các câu hỏi sẽ chỉ ra vị trí và nêu cách thực hiện ý tưởng của code)

- Để hiển thị dấu nhắc ví dụ 'IT007> ', ta tạo 1 vòng lặp với điều kiện lặp là biến `should_run` (biến cờ hiệu, vòng lặp này sẽ luôn lặp cho tới khi biến cờ hiệu này mang giá trị bằng 0):

```
120     int should_run = 1; // Biến cờ để xác định chương trình có tiếp tục chạy hay không
121     while (should_run)
122     {
123         printf("it007sh> "); // Dấu nhắc
124         fflush(stdout);      // Xóa bộ đệm
```

- Để người dùng nhập lệnh và thực thi lệnh đã nhập, ta lấy dữ liệu input từ người dùng. Dùng hàm `strcspn` tìm vị trí đầu tiên của '\n' trong chuỗi và thay bằng '\0' để kết thúc chuỗi tại đó nhằm loại bỏ ký tự xuống dòng:

```
126     char command[MAX_LINE];
127     fgets(command, MAX_LINE, stdin); // Nhập câu lệnh
128     command[strcspn(command, "\n")] = '\0'; // strcspn tìm vị trí đầu tiên có '\n'
```

- Bỏ qua câu lệnh rỗng nếu giá trị trả về của `strlen(command)` bằng 0. Kiểm tra câu lệnh nhập vào có phải là 'exit' hay không, nếu phải thì thực hiện tắt shell bằng cách gán giá trị của biến `should_run = 0` → thoát khỏi vòng lặp `while(should_run)`:

```
130     // Bỏ qua nếu câu lệnh rỗng
131     if (strlen(command) == 0)
132         continue;
133
134     // Kiểm tra lệnh đặc biệt của người dùng nhập vào như "exit"
135     if (strcmp(command, "exit") == 0)
136     {
137         should_run = 0;
138         continue;
139     }
```

- Tiếp đến là kiểm tra xem lệnh người dùng nhập vào có phải là lệnh 'HF' hay không và tiến hành xử lý nếu phải (sẽ trình bày chi tiết ở câu 2):

```
141 // Kiểm tra lệnh đặc biệt HF
142 char *hfPtr = strstr(command, "HF"); // Trả về con trỏ đến vị trí đầu tiên
143 int hfUsed = hfPtr == NULL ? 0 : 1; // Biến cờ để xác định người dùng đã nh
144 if (hfUsed)
145 > { ...
151 }
152
153 // Lưu lịch sử lại câu lệnh vừa nhập (nếu không phải lệnh HF)
154 if (!hfUsed)
155 > { ...
171 }
172
```

- Bước tiếp theo và cũng là ý chính của yêu cầu 1, chương trình gọi hàm `fork()` để tạo ra thêm 1 tiến trình con để thực thi các câu lệnh:

```
173 // Đẻ ra tiến trình con để giao nhiệm vụ thực thi lệnh cho nó
174 pid = fork();
175 if (pid == -1)
176 {
177     perror("fork() failed. Try again.");
178     continue;
179 }
```

* **Chú thích:** biến *pid* trong ảnh trên là một biến toàn cục được định nghĩa ở đầu file .C. Mục đích sử dụng của nó là phân biệt giữa các tiến trình với nhau (sẽ trình bày chi tiết về biến *pid* ở **câu 5**). Nếu *pid* = -1 thì hàm `fork()` đã gặp phải lỗi là ta gọi *continue* để vòng lặp `while(should_run)` chạy lại từ đầu.

- Sau khi `fork`, nếu là tiến trình con, ta thực hiện một loạt các thao tác sau:

```
181 // Tiến trình con
182 if (pid == 0)
183 {
```

- + Từ câu lệnh người dùng nhập vào, tự động phát hiện nếu người dùng có ý đồ muốn sử dụng cơ chế pipeline nối tiếp nhiều câu lệnh với nhau (sẽ giải thích ở **câu 4**) và nếu có thì tách câu lệnh đó ra thành nhiều câu lệnh nhỏ phân tách bằng ký tự '|' rồi thêm chúng vào một danh sách lệnh. Nếu không thì vẫn thêm câu lệnh người dùng nhập vào danh sách lệnh trên:

```
187 // Khúc này là để tách chuỗi command thành nhiều sub-command bằng dấu '|'
188 char *token = strtok(command, "|");
189 while (token != NULL && cmd_count < MAX_COMMAND)
190 > { ...
206 }
207
```

+ Tiếp đó, thực thi từng câu lệnh trong danh sách lệnh đã thu thập được:

```
208 // Khúc này là để chạy từng sub-command. Nếu có nhiều hơn 1 sub-command thì cơ chế pipe sẽ có hiệu lực
209 int prevfd = STDIN_FILENO;
210 for (int i = 0; i < cmd_count; i++)
211 > { ...
243 }
244 exit(EXIT_SUCCESS);
245 }
```

** **Chú thích:** Vì vòng lặp for() ở ảnh trên liên quan đến cơ chế pipeline của **câu 4** nên ở đây chỉ giải thích sơ lược như sau: điểm mấu chốt để có thể thực thi các câu lệnh trong danh sách lệnh là gọi hàm exec_cmd() được định nghĩa ở bên ngoài hàm main():*

```
32 void exec_cmd(char *command)
33 {
34     argsCount = 0;
35     // Xử lý chuỗi nhập vào
36     char *token = strtok(command, " "); // Tách chuỗi ký tự đầu tiên ra khi có dấu cách và lưu vào
37     while (token != NULL)
38     {
39         args[argsCount++] = token; // args[i] bằng một chuỗi ký tự ngăn bởi dấu cách hay nói cách
40         token = strtok(NULL, " "); // Tiếp tục tách chuỗi đến khi token là ký tự rỗng khi bị cắt
41     }
42     args[argsCount] = NULL; // Từ cuối cùng sẽ là kết thúc chuỗi từ
43
44     // Nếu lệnh là chuyển hướng đầu vào ra
45     int input_fd, output_fd;
46     int redirect_input = 0, redirect_output = 0;
47
48     for (int j = 0; j < argsCount; j++)
49     {
50         // Dấu hiệu chuyển hướng đầu ra của một tệp
51         if (strcmp(args[j], ">") == 0)
52         {
53             args[j] = NULL; // Đánh dấu vị trí chuyển hướng là NULL
54             redirect_output = 1; // Cờ chuyển hướng đầu ra
55
56             if (args[j + 1] == NULL)
57             { // Nơi đầu ra được chuyển hướng vào là NULL nghĩa là không có file output.
58                 printf("Missing output file\n");
59                 exit(EXIT_FAILURE); // Báo lỗi xong thoát
60             }
61         }
62     }
```

```
32 void exec_cmd(char *command)
33 {
34     argsCount = 0;
35
36     // Xử lý chuỗi nhập vào
37     char *token = strtok(command, " "); // Tách chuỗi ký tự đầu tiên ra khi có dấu cách
38     while (token != NULL)
39     {
40         args[argsCount++] = token; // args[i] bằng một chuỗi ký tự ngăn bởi dấu cách
41         token = strtok(NULL, " "); // Tiếp tục tách chuỗi đến khi token là ký tự rỗng
42     }
43     args[argsCount] = NULL; // Từ cuối cùng sẽ là kết thúc chuỗi từ
44 }
```

Tham số của hàm `exec_cmd()` sẽ là câu lệnh chúng ta nhập vào shell (nguyên vẹn) hoặc câu lệnh nhỏ được tách ra bởi ký tự '|'. Biến `argsCount` chính là để đếm số từ có trong câu lệnh đó, **các từ ngăn nhau bởi dấu cách**. Biến toàn cục `args[]` đã được tạo bên trên với mục đích là mỗi một từ trong câu lệnh sẽ được lưu vào với dạng một phần tử của của mảng này.

```
17 char *args[MAX_LINE / 2 + 1]; // Mảng để lưu trữ lệnh và các tham số
```

Trong quá trình tách các từ bằng dấu cách, nếu phát hiện ký tự chuyển hướng '<', '>', hàm `exec_cmd` sẽ có cơ chế xử lý riêng (giải thích ở **câu 3**).

```
49 for (int j = 0; j < argsCount; j++)
50 {
51     // Dấu hiệu chuyển hướng đầu ra của một tệp
52     if (strcmp(args[j], ">") == 0)
53     {
```

Sau khi đã tách câu lệnh thành các từ và lưu từng từ vào chuỗi `args[]`, thì lúc này phần tử `args[0]` chính là câu lệnh ta đã nhập với **tham số** là `args[1]` đến `args[n]` → Ta dùng hàm `execvp(args[0], args)` để thực hiện lệnh `args[0]` với mảng đối số cho lệnh là `args`. Ta kiểm tra giá trị trả về của `execvp`, nếu là `-1` thì lệnh không tồn tại trong hệ thống → tiến trình con được tạo ở hàm `main` in ra "Command does not..." → tiến trình con kết thúc với mã `EXIT_FAILURE` (mã báo hiệu có lỗi xảy ra)

```
97 // Chạy và kiểm tra lỗi
98 if (execvp(args[0], args) == -1)
99 {
100     printf("Command does not exist.\n");
101     exit(EXIT_FAILURE);
102 }
103 }
```

**

- Nếu là tiến trình cha, dùng lệnh `waitpid(-1, NULL, 0)` để đợi cho tiến trình con thực thi xong:

```
247 // Tiến trình cha
248 else
249 {
250     waitpid(-1, NULL, 0); // Chờ tiến trình con thực thi xong
251 }
252 }
```

****Chú thích:** `waitpid(-1, NULL, 0)`: Theo quy định của hàm `waitpid()`, tham số '-1' là đợi tất cả tiến trình khác hoàn thành.*

**

Khi con hoàn thành, cha cũng đã ở cuối vòng lặp `while(should_run)` nên tiếp tục chạy lại vòng lặp từ đầu, và quy trình đợi người dùng nhập lệnh, xử lý lệnh đặc biệt, xử lý pipeline, xử lý chuyển hướng input/output tiếp diễn cho đến khi người dùng nhập lệnh 'exit'.

Hình minh chứng & giải thích kết quả:

```
● nguyentiendat-22520226@Nobara_PC:~$ ./LAB6
it007sh> ls
LAB6 LAB6.c cbonsai
it007sh> touch TEST
it007sh> ls
LAB6 LAB6.c TEST cbonsai
it007sh> rm -r TEST
it007sh> ls
LAB6 LAB6.c cbonsai
it007sh>
it007sh> exit
○ Goodbye!nguyentiendat-22520226@Nobara_PC:~$
```

GIẢI THÍCH:

- Câu lệnh đầu tiên nhập vào là lệnh **ls**, tự động phát hiện nếu người dùng có ý đồ muốn sử dụng cơ chế pipeline nối tiếp nhiều câu lệnh với nhau và nếu có thì tách câu lệnh đó ra thành nhiều câu lệnh nhỏ phân tách bằng ký tự '|' rồi thêm chúng vào một danh sách lệnh. Nếu không thì vẫn thêm câu lệnh người dùng nhập vào danh sách lệnh, tiếp đó ta thực thi các câu lệnh trong danh sách lệnh. Ở đây lệnh **ls** sẽ được thêm vào danh sách lệnh. Vì không có các câu lệnh con khác nào ngoài lệnh **ls** thì hệ điều hành sẽ thực thi câu lệnh **ls** (tức là liệt kê các file ở thư mục hiện hành).
- Tương tự như trên với lệnh **touch TEST** thì tách câu lệnh đó ra thành nhiều câu lệnh nhỏ rồi thêm chúng vào danh sách lệnh. Ta thực thi các câu lệnh trong danh sách lệnh bằng cách gọi hàm **exec_cmd** trong vòng lặp với hàm **exec_cmd** đã được định nghĩa, tham số của hàm **exec_cmd()** sẽ là câu lệnh chúng ta nhập vào shell (nguyên vẹn) hoặc câu lệnh nhỏ được tách ra bởi ký tự '|'. Biến **argsCount** chính là để đếm số từ có trong câu lệnh đó, **các từ ngăn nhau bởi dấu cách**. Biến toàn cục **args[]** đã được tạo bên trên với mục đích là mỗi một từ trong câu lệnh sẽ được lưu vào với dạng một phần tử của của mảng này. Sau khi đã tách câu lệnh thành các từ và lưu từng từ vào chuỗi **args[]**, thì lúc này phần tử **args[0]** chính là câu lệnh **touch** ta đã nhập với **tham số** là **args[1]** đến **args[n]** → Ta dùng hàm **execvp(args[0], args)** để thực hiện lệnh **touch** với mảng đối số cho lệnh là **TEST**. Thì kết quả trả về cho ta là lệnh **touch TEST** giống như shell nguyên bản.
- **rm -r TEST**, tương tự như trên thì **args[0]** là lệnh **rm** với các đối số truyền vào là **-r** và **TEST**. Thì kết quả trả về cho ta là lệnh **rm -r TEST** giống như shell nguyên bản.

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

Trình bày giải thuật & giải thích code:

Ở đầu chương trình, ta tạo mảng **HF[HISTORY_SIZE]** để lưu lịch sử các câu lệnh và biến **historyCount** để đếm các câu lệnh nếu chưa đầy lịch sử (lịch sử chứa tối đa 5 câu lệnh gần nhất):

```
11 #define HISTORY_SIZE 5
12 #define MAX_LINE 80 // Số ký tự tối đa trên 1 dòng
13 #define MAX_COMMAND 4 // Số command tối đa được sử dụng trong pipe, ví dụ: 'ls | grep a | wc -l' là 3 command
14 #define DEFAULT_PID 1 // id mặc định để gán cho process chính khi hàm fork() chưa được gọi
15
16 char *HF[HISTORY_SIZE]; // Mảng để lưu lịch sử các câu lệnh
17 char *args[MAX_LINE / 2 + 1]; // Mảng để lưu trữ lệnh và các tham số của lệnh
18 int historyCount = 0;
19 int argsCount;
```

Ở trong hàm main, cứ mỗi khi nhập xong 1 câu lệnh

```
char command[MAX_LINE];
fgets(command, MAX_LINE, stdin); // Nhập câu lệnh
command[strcspn(command, "\n")] = '\0'; // strcspn tìm vị trí đầu tiên có '\n' trong chuỗi;
```

Nếu câu lệnh không phải là 'exit', ta sẽ tiến hành kiểm tra xem người dùng có nhập lệnh 'HF' hay không, cách kiểm tra như sau:

```
// Kiểm tra lệnh đặc biệt HF
char *hfPtr = strstr(command, "HF"); // Trả về con trỏ đến vị trí đầu tiên của chuỗi "HF" trong chuỗi lớn "command"
int hfUsed = hfPtr == NULL ? 0 : 1; // Biến cờ để xác định người dùng đã nhập lệnh HF
if (hfUsed)
{
```

- Dùng hàm **strstr(command, "HF")**: hàm sẽ dò tìm trong command (câu lệnh người dùng nhập) trả về con trỏ tới vị trí đầu tiên của chuỗi "HF" trong câu lệnh đó. Nếu không tìm thấy, hàm trả về NULL. Giá trị trả về của strstr() được lưu vào biến con trỏ **char* hfPtr**.
- **Nếu tìm thấy chuỗi "HF" trong command:**
 - + Tạo biến **hfCount** với giá trị khởi tạo là 1 (nếu hfCount là 1 thì thực hiện lại câu lệnh gần nhất, nếu là 2 thì thực hiện câu lệnh gần thứ 2, tương tự với 3, 4, 5):

```
144         if (hfUsed)
145         {
146             int hfCount = 1;
```

- + Tiếp tục dùng lệnh strstr() để dò tìm từ "HF" trong các ký tự còn lại của câu lệnh (các ký tự chưa dò trước đó vì hàm strstr() dừng lại khi nó tìm được bất kỳ từ "HF" nào trong chuỗi) và cứ lặp lại cho đến khi không còn từ "HF" nào hoặc hfCount đã bằng với historyCount (số phần tử hiện đang có trong mảng lưu lịch sử HF):

```
while ((hfPtr = strstr(hfPtr + strlen("HF"), "HF")) != NULL && hfCount < historyCount) // Tiếp tục
    hfCount++;
```

****Chú thích:** 'hfPtr + strlen("HF")' có ý nghĩa là ta lấy con trỏ đang trỏ tới từ "HF" được tìm thấy trước đó và dịch chuyển nó tới 2 đơn vị (độ dài của từ "HF"). Như vậy, hàm strstr() sẽ bắt đầu tìm kiếm từ "HF" mới ở phần đoạn của **command** mà chưa được tìm kiếm trước đó.*

- + Bước cuối cùng là thay đổi giá trị của **command** hiện tại bằng câu lệnh gần nhất thứ n trong lịch sử (n = hfCount), sau đó in ra dòng chữ "it007sh>

command” để báo hiệu cho người dùng để nắm bắt được lệnh nào trong lịch sử sắp sửa được thực thi lại

```
149 strcpy(command, HF[historyCount - hfCount]); // Lấy câu lệnh từ
150 printf("it007sh> %s\n", command);
151 }
```

- Nếu không tìm thấy chuỗi “HF” trong command, thì command mà người dùng nhập vào sẽ được lưu vào lịch sử:

```
// Lưu lịch sử lại câu lệnh vừa nhập (nếu không phải lệnh HF)
else
{
    if (historyCount < HISTORY_SIZE) // Nếu lịch sử còn chỗ trống
    {
        strcpy(HF[historyCount], command);
        historyCount++;
    }
    else // Nếu lịch sử đã đầy
    {
        // Xóa lệnh đầu tiên và dịch chuyển các lệnh cũ lên trên
        for (int i = 0; i < HISTORY_SIZE - 1; i++)
        {
            strcpy(HF[i], HF[i + 1]);
        }
        strcpy(HF[HISTORY_SIZE - 1], command); // Lưu lệnh mới nhất vào cuối
    }
}
```

Chú thích:* Nếu lịch sử đã đầy, ta tiến hành xóa câu lệnh cũ nhất trong lịch sử. Vì mảng **HF được tổ chức theo kiểu FIFO → **HF[0]** là lệnh cũ nhất → Thực hiện phép dời mảng $HF[i] = HF[i + 1]$ để ghi đè lên **HF[0]** → Thêm **command** mới vào **HF[HISTORY_SIZE - 1]**.

⇒ Đó là toàn bộ quá trình xử lý History Feature của shell. **Tóm tắt** lại các bước xử lý:

- Trong **command** của người dùng nhập, tìm và đếm số lần chữ “HF” xuất hiện
- Nếu command xuất hiện n lần ($n > 0, n < 5$): Thay thế command bằng câu lệnh gần thứ n trong lịch sử. Nếu $n = 0$, giữ nguyên **command** và lưu vào lịch sử (nếu lịch sử đã đầy thì xóa câu lệnh cũ nhất rồi lưu).
- Đưa **command** cho các phần tiếp theo của chương trình xử lý.

Hình minh chứng:

```
./vinashell
it007sh> echo hello1
hello1
it007sh> echo hello2
hello2
it007sh> echo hello3
hello3
it007sh> echo hello4
hello4
it007sh> echo hello5
hello5
it007sh> HF
it007sh> echo hello5
hello5
it007sh> HF HF
it007sh> echo hello4
hello4
it007sh> HF HF HF
it007sh> echo hello3
hello3
it007sh> HF HF HF HF
it007sh> echo hello2
hello2
it007sh> HF HF HF HF HF
it007sh> echo hello1
hello1
it007sh> █
```

Nhập n lần HF để quay lại câu lệnh gần thứ n

Giải thích kết quả:

Ở ví dụ trong ảnh trên, ta thực thi một loạt các câu lệnh “echo hello1”, “echo hello2”, ... , “echo hello5”. Sau khi thực thi xong, lúc này trong mảng HF đang chứa 5 câu lệnh theo thứ tự từ cũ nhất đến mới nhất:

HF = [“echo hello1”, “echo hello2”, “echo hello3”, “echo hello4”, “echo hello5”]

Có thể thấy khi ta gõ lệnh “HF”, chương trình duyệt lệnh mà ta vừa gõ và tìm được 1 từ “HF” → thực hiện lại lệnh gần nhất trong lịch sử:

```
it007sh> HF
it007sh> echo hello5
hello5
```

...

Tương tự, khi ta gõ lệnh “HF HF HF HF HF”, chương trình tìm thấy 5 từ “HF” và in ra câu lệnh gần thứ 5 trong lịch sử, cũng chính là câu lệnh cũ nhất còn lưu lại vì kích thước tối đa của lịch sử là 5:

```
it007sh> HF HF HF HF HF
it007sh> echo hello1
hello1
```

*** Bonus:**

Ngoài ra, vì chương trình đã được cài đặt để đếm số chữ “HF” không vượt quá kích thước tối đa của mảng HF[]. Nên dù ta có cố tình nhập 6 chữ “HF”, chương trình cũng chỉ dừng lại ở chữ “HF” thứ 5:

```
it007sh> HF HF HF HF HF HF
it007sh> echo hello1
hello1
it007sh> █
```

Cuối cùng, để kiểm tra tính năng xóa lệnh cũ nhất khỏi lịch sử khi mảng HF[] đã đầy, biết rằng lúc này mảng HF[] đang có 5 phần tử. Nếu ta nhập câu lệnh “echo hello6” vào thì chắc chắn câu lệnh cũ nhất (“echo hello1”) sẽ bị xóa ra khỏi mảng HF[] để chừa chỗ cho lệnh “echo hello6”:

```
it007sh> echo hello6
hello6
it007sh> █
```

Lúc này, mảng HF[] trở thành:

HF = [“echo hello2”, “echo hello3”, “echo hello4”, “echo hello5”, “echo hello6”]

Để kiểm chứng xem mảng HF[] có giống như vậy hay không, ta sẽ chạy một loạt các lệnh HF nối chuỗi nhau như ở phần trước để in ra các câu lệnh hiện có trong lịch sử theo thứ tự từ mới nhất đến cũ nhất:

```
it007sh> HF
it007sh> echo hello6
hello6
it007sh> HF HF
it007sh> echo hello5
hello5
it007sh> HF HF HF
it007sh> echo hello4
hello4
it007sh> HF HF HF HF
it007sh> echo hello3
hello3
it007sh> HF HF HF HF HF
it007sh> echo hello2
hello2
it007sh> █
```

**

3. Câu 3: Chuyển hướng vào ra

Trình bày giải thuật, giải thích code:

Sau khi đã cắt câu lệnh thành các từ như ở câu 1 (được thực hiện ở hàm `exec_cmd`).

```
32 void exec_cmd(char *command)
33 {
34     argsCount = 0;
35
36     // Xử lý chuỗi nhập vào
37     char *token = strtok(command, " "); // Tách chuỗi ký tự đầu tiên ra
38     while (token != NULL)
39     {
40         args[argsCount++] = token; // args[i] bằng một chuỗi ký tự ngắn
41         token = strtok(NULL, " "); // Tiếp tục tách chuỗi đến khi token
42     }
43     args[argsCount] = NULL; // Từ cuối cùng sẽ là kết thúc chuỗi từ
44 }
```

Trước khi thực hiện `execvp`, ta kiểm tra trong chuỗi `args[]` có từ nào bằng với toán tử chuyển hướng hay không ('>' và '<'). Nếu có `args[j]` nào đó là '>' (tức là chuyển hướng đầu ra), ta sẽ đánh dấu vị trí chuyển hướng là `NULL`. Lúc này cờ chuyển hướng đầu ra sẽ là 1. Nếu `args[j+1]` là `NULL` thì câu lệnh này không có file đầu ra vậy nên ta sẽ báo lỗi và thoát. Nếu có tên file đầu ra, chương trình sẽ mở file đó và biến nó thành đầu ra chuẩn mới thay cho đầu ra chuẩn mặc định bằng lệnh `dup2`.

```
69     dup2(output_fd, STDOUT_FILENO); // Overwrite file STDOUT
```

Trương tự với `args[j]` là '<' (tức là chuyển hướng đầu vào).

```
51 // Dấu hiệu chuyển hướng đầu ra của một tệp
52 if (strcmp(args[j], ">") == 0)
53 {
54     args[j] = NULL; // Đánh dấu vị trí chuyển hướng là NULL
55     redirect_output = 1; // Cờ chuyển hướng đầu ra
56
57     if (args[j + 1] == NULL)
58     { // Nơi đầu ra được chuyển hướng vào là NULL nghĩa là không có file output.
59         printf("Missing output file.\n");
60         exit(EXIT_FAILURE); // Báo lỗi xong thoát
61     }
62     // Tạo file args[i] nếu chưa có; file này sẽ thế chỗ đầu ra cho luồng chuẩn STDOUT
63     output_fd = open(args[j + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644); // output_fd lưu
64     if (output_fd < 0)
65     {
66         perror("System fail to execute.");
67         exit(EXIT_FAILURE); // Không mở được file nên báo lỗi rồi thoát
68     }
69     dup2(output_fd, STDOUT_FILENO); // Overwrite file STDOUT hiện tại bằng file output_fd
70     close(output_fd);
71 }
72
```

Xử lý chuyển hướng đầu ra của một tệp

```
73 // Dấu hiệu chuyển hướng đầu vào của một tệp
74 else if (strcmp(args[j], "<") == 0)
75 {
76     args[j] = NULL; // Đánh dấu vị trí chuyển hướng là NULL
77     redirect_input = 1; // Cờ chuyển hướng đầu vào
78
79     if (args[j + 1] == NULL)
80     { // Nơi đầu vào được chuyển hướng vào là NULL nghĩa là không có file input.
81         printf("Missing input file.\n");
82         exit(EXIT_FAILURE); // Báo lỗi xong thoát
83     }
84
85     // Tạo file args[i] nếu chưa có; file này sẽ thế chỗ đầu vào cho file STDIN của terminal
86     input_fd = open(args[j + 1], O_RDONLY); // input_fd lưu trữ giá trị file descriptor nên phía trên
87     if (input_fd < 0)
88     {
89         perror("System fail to execute.");
90         exit(EXIT_FAILURE); // Không mở được file nên báo lỗi rồi thoát
91     }
92     dup2(input_fd, STDIN_FILENO); // Overwrite file STDIN hiện tại bằng file input_fd vừa tạo, luồng đ
93     close(input_fd);
94 }
```

Xử lý chuyển hướng đầu vào của một tệp

Chụp hình minh chứng:

```
./vinashell
~/thuchanh/LAB6-OS  P main *1 ?9 10:56:38
> ./vinashell
it007sh> cat test.txt
banana
orange
kiwi
hedieuhanh
apple
merry christmas
it007sh> sort < test.txt
apple
banana
hedieuhanh
kiwi
merry christmas
orange
it007sh> wc -l < test.txt
6
it007sh> echo "Hello world IT007, merry christmas & happy new year" > hello.txt
it007sh> cat hello.txt
"Hello world IT007, merry christmas & happy new year"
it007sh>
```

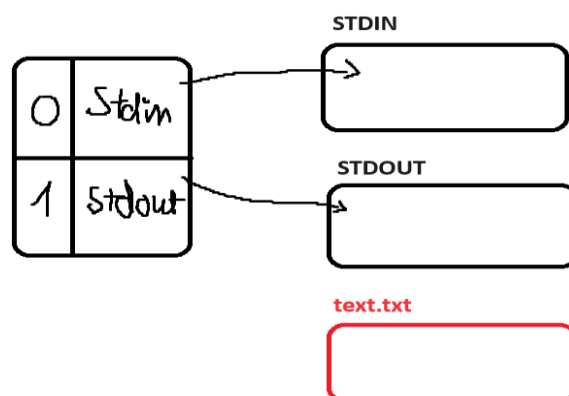
GIẢI THÍCH: (chỉ minh họa câu lệnh đầu, các câu lệnh sau chạy tương tự giải thuật đã trình bày trên)

B1: cắt lệnh thành các từ, ta có $\text{args}[0] = \text{"sort"}, \text{args}[1] = \text{"<"}, \text{args}[2] = \text{"test.txt"}$

B2: kiểm tra thấy $\text{args}[1]$ là toán tử chuyển hướng đầu vào "<"

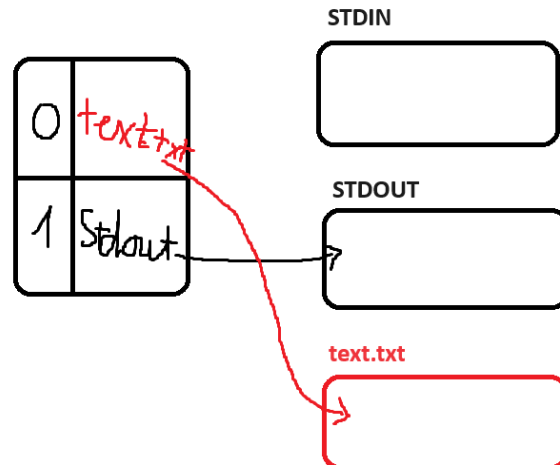
B3: kiểm tra thấy $\text{args}[1+1] = \text{args}[2] = \text{"text.txt"}$ khác rỗng nên có file đầu vào là text.txt

B4: kiểm tra trong hệ thống đã tồn tại file tên $\text{args}[2] = \text{"text.txt"}$ hay chưa, nếu chưa sẽ tạo file đó và mở file. Do mở file thành công nên không báo lỗi.



B5: tạo biến để lưu file descriptor của file vừa tạo là text.txt

B6: đổi file descriptor mặc định ở đầu đọc dữ liệu thành descriptor của file text.txt



B7: đóng file

Vậy là sau khi thực hiện lệnh, đầu vào lúc này của chương trình sẽ là file text.txt. Do vậy khi thực hiện lệnh sort, sort sẽ nhận đầu vào là dữ liệu sau:

```
it007sh> cat text.txt
banana
orange
kiwi
hedieuhanh
apple
merry christmas
```

Do đó khi sort xong, ta thấy các dữ liệu của file text.txt đã được sắp xếp.

```
it007sh> sort < text.txt
apple
banana
hedieuhanh
kiwi
merry christmas
orange
```

4. Câu 4: Giao tiếp sử dụng cơ chế đường ống

Trình bày giải thuật:

Ở yêu cầu số 1, ta đã giải thích sơ lược về việc tiến trình con ở hàm main, dùng tiến trình con này để thực hiện **command** người dùng nhập; tiến trình cha chờ tiến trình con hoàn thành xong, sẽ thực hiện vòng lặp để xuất hiện dấu nháy. Cứ mỗi lần nhập một câu lệnh là ta sẽ đều tạo tiến trình con, đều xử lý pipeline câu lệnh đó dù nó có sử dụng cơ chế pipeline hay không. Với mỗi **sub-command** trong pipeline như vậy, chúng đều được quyền chuyển hướng đầu vào/ra theo ý muốn. Vì thế, ta phải đưa các từng **sub-command** này vào hàm **exec_cmd()** để xử lý chuyển hướng vào/ra riêng biệt cũng như thực thi chúng.

Bắt đầu giải thích từ hàm main, ta tạo một tiến trình con để xử lý lệnh nhập vào. Tiến trình này sẽ tách lệnh nhập từ input thành các đoạn lệnh nhỏ hơn với '|' là dấu ngăn cách và lưu vào chuỗi cmd_list. Nếu câu lệnh không có dấu '|' thì chuỗi cmd_list sẽ chỉ có 1 phần tử.

Ví dụ: 'ls | wc -l' khi được nhập vào sẽ có cmd_list[0] là 'ls' và cmd_list[1] là 'wc -l'. Nếu câu lệnh nhập vào chỉ có 'ls' thì cmd_list chỉ có 1 phần tử là cmd_list[0] = 'ls'.

```
172 // Tiến trình con
173 if (pid == 0)
174 {
175     char cmd_list[MAX_COMMAND][MAX_LINE]; // Mảng để chứa các câu lệnh phân tách bằng dấu '|'
176     int cmd_count = 0;
177
178     // Khúc này là để tách chuỗi command thành nhiều sub-command bằng dấu '|'
179     char *token = strtok(command, "|");
180     while (token != NULL && cmd_count < MAX_COMMAND)
181     {
182         // Remove leading and trailing spaces from the token
183         while (*token == ' ' || *token == '\t')
184         {
185             token++;
186         }
187
188         int length = strlen(token);
189         while (length > 0 && (token[length - 1] == ' ' || token[length - 1] == '\t'))
190         {
191             token[length - 1] = '\0';
192             length--;
193         }
194
195         strcpy(cmd_list[cmd_count++], token); // lưu subcommand vào câu lệnh cmd_list
196         token = strtok(NULL, "|");           // tìm subcommand tiếp theo
197     }
```

Khi tách các câu lệnh ngăn bởi dấu '|' (sub-command) ta cũng phải bỏ đi khoảng trắng ở phía trước các sub-command đó.

Sau khi đã tách thành các sub-command, ta gán cho biến `prevfd` bằng với đầu vào mặc định. Điều này là để sub-command đầu tiên sẽ có đầu vào mặc định. Khi thực hiện xong tiến trình, các `prevfd` trừ sub-command đầu tiên đều sẽ bị đổi đầu STDIN mặc định thành đầu `pipefd[0]`. Các sub-command trừ sub-command cuối cùng sẽ bị đổi đầu STDOUT mặc định thành đầu `pipefd[1]`.

200

```
int prevfd = STDIN_FILENO;
```

Ta tạo vòng lặp với mỗi sub-command có được.

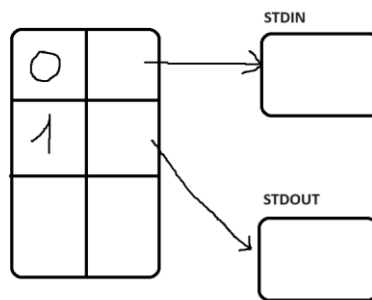
```
201     for (int i = 0; i < cmd_count; i++)
202     {
203         int pipefd[2];
204         if (pipe(pipefd) == -1)
205         {
206             // Xử lý lỗi khi không thể tạo đường ống
207             perror("Pipe fail. Try again");
208             continue;
209         }
210
211         pid_t cpid = fork(); // Đẻ ra tiến trình con để chạy từng cmd trong cmd_list[]
212         if (cpid == -1)
213         {
214             perror("fork() failed in pipeline. Try again.");
215             exit(EXIT_FAILURE);
216         }
```

Trong mỗi sub-command ta sẽ phải chỉnh 2 thành phần đó là stdin và stdout của sub-command đó. Tạo một ống `pipefd[2]`, đầu đọc là `pipefd[0]`, đầu viết là `pipefd[1]`. Báo lỗi nếu không tạo được ống.

Ta tạo một tiến trình con để thực hiện điều chỉnh `pipefd[0]` và `pipefd[1]`, tiến trình cha sẽ dùng để đổi giá trị `prevfd`, cho sub-command sau dùng. Như vậy sub-command đầu tiên sẽ có STDIN mặc định, sub-command cuối cùng sẽ có STDOUT mặc định.

```
218     else if (cpid == 0)
219     {
220         close(pipefd[0]);
221         dup2(prevfd, STDIN_FILENO);
222         if (i < cmd_count - 1)
223             dup2(pipefd[1], STDOUT_FILENO); // Chuyển hướng đầu ra của tất cả cmd trừ cmd cuối
224         exec_cmd(cmd_list[i]);
225         close(pipefd[1]);
226         exit(EXIT_SUCCESS); // Tiến trình con hỉ sinh sau khi chạy xong cmd
227     }
```

Nếu chỉ có một sub-command, lệnh `dup2(prefd, STDIN_FILENO)` sẽ không làm thay đổi đầu vào của sub-command này; do `prefd` đầu tiên đặt ngoài vòng lặp có giá trị là `STDIN_FILENO`. Lệnh `dup2(pipefd[1], STDOUT_FILENO)` cũng sẽ không được thực hiện. Điều này khiến cho sub-command duy nhất được nhập này có đầu vào và đầu ra như hình bên dưới.



Với đầu vào và ra như trên, đây trở thành một câu lệnh bình thường mà không có ống dẫn, được thực hiện hàm `exec_cmd` bình thường (trong `exec_cmd` có hỗ trợ chuyển hướng đọc ở câu 2).

Khi có nhiều hơn một sub-command, điều này cũng có nghĩa trong câu lệnh nhập vào có ống dẫn.

```
218         else if (cpid == 0)
219         {
220             close(pipefd[0]);
221             dup2(preffd, STDIN_FILENO);
222             if (i < cmd_count - 1)
223             {
224                 dup2(pipefd[1], STDOUT_FILENO); // Chuyển hướng đầu ra của tất cả cmd trừ cmd cuối
225             }
226             exec_cmd(cmd_list[i]);
227             close(pipefd[1]);
228             exit(EXIT_SUCCESS); // Tiến trình con hi sinh sau khi chạy xong cmd
```

Lúc này sub-command đầu tiên vẫn sẽ có đầu vào mặc định. Tuy nhiên đầu ra sẽ được chuyển vào ống viết `pipefd[1]`. Tiến trình cha của sub-command đầu tiên sẽ đổi `preffd` thành đầu ống đọc `pipefd[0]`.

```
229     else
230     {
231         close(pipefd[1]);
232         close(preffd);
233         prefd = pipefd[0];
234         waitpid(-1, NULL, 0); // Chờ tiến trình con hi sinh
235     }
```

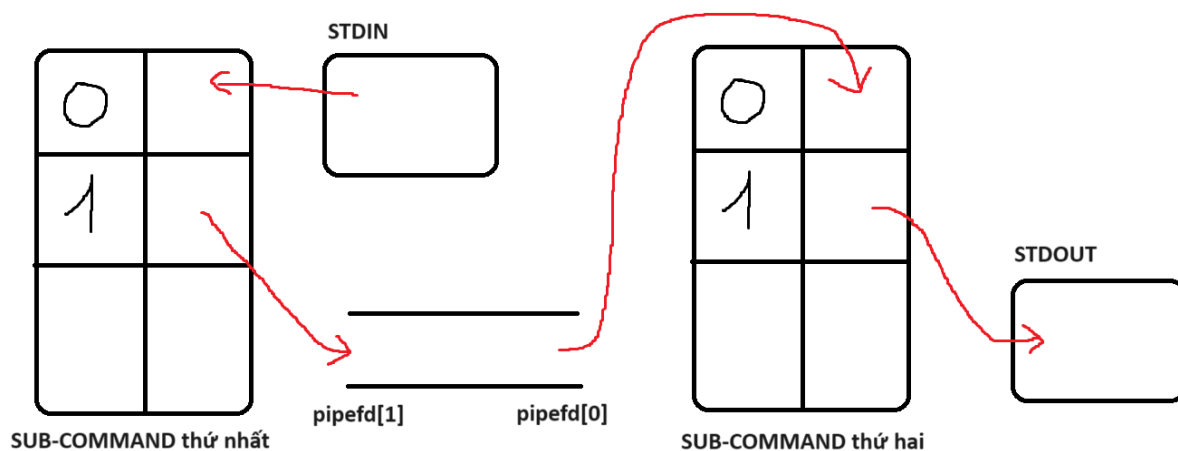
Vòng lặp for làm cho tiến trình của sub-command thứ 2 được chạy.

```
201     for (int i = 0; i < cmd_count; i++)
202     {
203         int pipefd[2];
204         if (pipe(pipefd) == -1)
205         {
206             // Xử lý lỗi khi không thể tạo đường ống
207             perror("Pipe fail. Try again");
208             continue;
209         }
210
211         pid_t cpid = fork(); // Đẻ ra tiến trình con để chạy từng cmd trong cmd_list[]
212         if (cpid == -1)
213         {
214             perror("fork() failed in pipeline. Try again.");
215             exit(EXIT_FAILURE);
216         }
```

Lúc này, `dup2(prevfd, STDIN_FILENO)` sẽ làm cho sub-command thứ hai đổi file đầu vào mặc định thành ống `pipefd[0]`; do `prevfd` đã được đổi nhờ tiến trình cha của sub-command thứ nhất. Còn câu lệnh `dup2(pipefd[1], STDOUT_FILENO)` sẽ không được thực hiện do đã là sub-command cuối cùng.

```
218     else if (cpid == 0)
219     {
220         close(pipefd[0]);
221         dup2(prevfd, STDIN_FILENO);
222         if (i < cmd_count - 1)
223             dup2(pipefd[1], STDOUT_FILENO); // Chuyển hướng đầu ra của tất cả cmd trừ cmd cuối
224         exec_cmd(cmd_list[i]);
225         close(pipefd[1]);
226         exit(EXIT_SUCCESS); // Tiến trình con hi sinh sau khi chạy xong cmd
227     }
```

Kết quả sẽ như hình minh họa bên dưới:



Chụp hình minh chứng & giải thích kết quả:

```
~/thuchanh/LAB6-OS main *1 ?5
> ./vinashell
it007sh> sudo apt install cowsay
[sudo] password for ducmint:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
cowsay is already the newest version (3.03+dfsg2-8).
The following packages were automatically installed and are no longer required:
  python-pkg-resources python-setuptools
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 195 not upgraded.
it007sh>
```

Tải chương trình 'cowsay'

```
~/thuchanh/LAB6-OS main *1 ?3 11:10:09
> ./vinashell
it007sh> ls -l
total 40
-rw-r--r-- 1 ducmint ducmint 54 Dec 20 10:58 hello.txt
-rw-r--r-- 1 ducmint ducmint 52 Dec 20 10:53 test.txt
-rwxr-xr-x 1 ducmint ducmint 17256 Dec 20 11:05 vinashell
-rw-r--r-- 1 ducmint ducmint 9918 Dec 20 11:04 vinashell.c
it007sh> ls -l | grep a
total 40
-rwxr-xr-x 1 ducmint ducmint 17256 Dec 20 11:05 vinashell
-rw-r--r-- 1 ducmint ducmint 9918 Dec 20 11:04 vinashell.c
it007sh> ls -l | grep a
total 40
-rwxr-xr-x 1 ducmint ducmint 17256 Dec 20 11:05 vinashell
-rw-r--r-- 1 ducmint ducmint 9918 Dec 20 11:04 vinashell.c
it007sh> cat hello.txt
"Hello world IT007, merry christmas & happy new year"
it007sh> cat hello.txt | cowsay

/ "Hello world IT007, merry christmas & \
\ happy new year" /
  ^__^
  (oo)\_______
  (__)\       )\/\
      ||----w |
      ||     ||

it007sh> pwd | ls | grep h
hello.txt
vinashell
vinashell.c
```

Chạy các câu lệnh

***Bonus:**

```
it007sh> echo Ultimate testcase:
Ultimate testcase:
it007sh> ps | grep a | sort | cowsay

/ 103 pts/0 00:00:00 gitstatusd-linu \
| 7213 pts/0 00:00:00 vinashell      |
\ 8622 pts/0 00:00:00 vinashell      /

      ^ ^
      (oo)\_____
      (__) \       )\/\
            ||----w |
            ||     ||

it007sh> ps | grep a | cowsay | sort

      ^ ^
      (oo)\_____
      (__) \       )\/\
            ||----w |
            ||     ||

/ 103 pts/0 00:00:00 gitstatusd-linu \
\ 8627 pts/0 00:00:00 vinashell      /
|
|
| 7213 pts/0 00:00:00 vinashell      |
it007sh>
```

Chạy các câu lệnh

GIẢI THÍCH:

- Lấy ví dụ với câu lệnh **cat hello.txt | cow say**, theo như giải thích thuật toán ở phía trên, ta có:

1. Chương trình sẽ tách input và chia tách dấu "|" thành các command và tham số:
 - Command 1: "**cat**"
 - Tham số của command 1: "**hello.txt**"
 - Command 2: "**cowsay**"
2. Chương trình sẽ tạo một tiến trình con để thực hiện command "**cat hello.txt**" và chuyển output của command này thành input cho command "**cowsay**".
3. Tiến trình con sẽ thực hiện command "**cat hello.txt**" để đọc nội dung của file "**hello.txt**".
4. Output của command "**cat hello.txt**" sẽ được chuyển vào command "**cowsay**".

5. Command "**cowsay**" sẽ nhận input từ output của command trước đó và thực hiện công việc của nó. Output của ta sẽ là kết quả của lệnh **cowsay** với input là output của **cat hello.txt**.

- Lấy ví dụ với câu lệnh **pwd | ls | grep h**, theo như giải thích thuật toán ở phía trên, ta có:

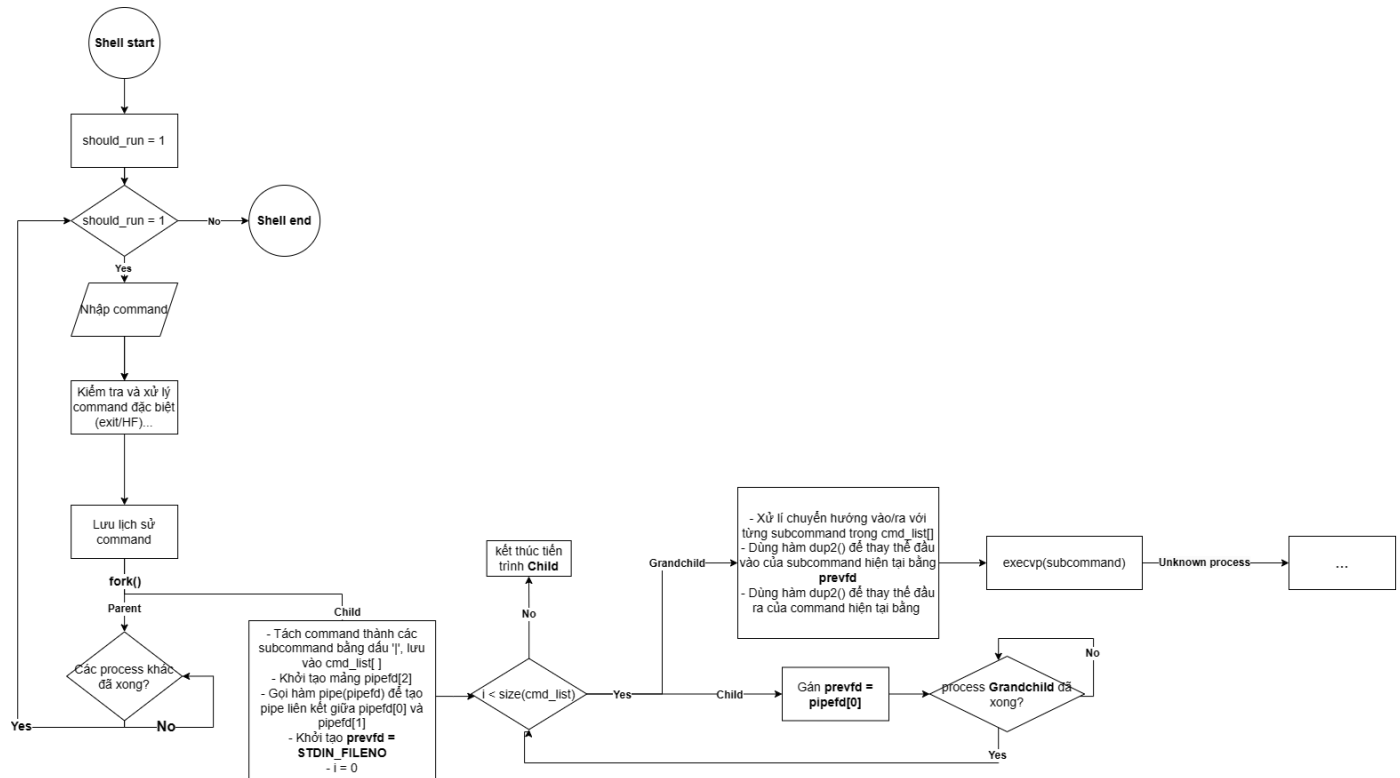
1. Chương trình sẽ tách input và chia tách dấu "|" thành các command và tham số:
 - Command 1: "**pwd**"
 - Command 2: "**ls**"
 - Command 3: "**grep**"
 - Tham số của command 3: "**h**"
2. Chương trình sẽ tạo một tiến trình con để thực hiện command "**pwd**" và chuyển output của command này thành input cho command "**ls**".
3. Tiến trình con sẽ thực hiện command "**pwd**" để xem thư mục hiện hành.
4. Output của command "**pwd**" sẽ được chuyển vào command "**ls**".
5. Command "**ls**" sẽ nhận input từ output của command trước đó và thực hiện công việc của nó. Nó sẽ thực hiện câu lệnh "**ls**" với input truyền vào là output của câu lệnh "**pwd**". Output lúc này sẽ được chuyển tới câu lệnh **grep**.
6. Tiếp tới câu lệnh "**grep**" với input là **h** và output của bước 5, ta sẽ có kết quả của toàn bộ câu lệnh là "xuất ra các dòng/các file có ký tự **h**" của thư mục hiện hành"

CÁC CÂU LỆNH CÒN LẠI CÁCH THỨC CHẠY TƯƠNG TỰ

5. Câu 5: Kết thúc lệnh đang thực thi

Trình bày giải thuật & giải thích code:

Để khái quát lại cách chạy của chương trình shell từ câu 1 → 4, ta có lưu đồ sau:



Lưu đồ giải thuật tổng quan câu 1 - 4

Nhìn vào lưu đồ trên, ta có thể nhận xét được rằng, khi người dùng gửi tín hiệu SIGINT bằng cách nhấn Ctrl + C, cơ chế xử lý mặc định của hệ điều hành sẽ kết thúc tất cả các tiến trình **Parent, Child, Grandchild/Unknown process**. Trong khi đó, tiến trình mà ta mong muốn kết thúc chỉ có **Unknown process**.

⇒ Ý tưởng giải thuật cho yêu cầu 5:

May mắn thay, ta có thể thay đổi cơ chế xử lý tín hiệu SIGINT của từng process bằng cách gọi hàm **signal(sig, func)** với **func** là những thao tác mà ta muốn tiến trình đó thực hiện khi nhận được **sig**. Lệnh **signal()** này sẽ được đặt ở đầu hàm **main()**, chính vì vậy nó xuất hiện trước toàn bộ các lệnh **fork()**, nên tất cả các tiến trình mới được sinh ra đều đã gọi hàm **signal()**:

```
105 int main()
106 {
107     // Listener để xử lý khi người dùng nhấn Ctrl + C
108     signal(SIGINT, sigint_handler);
109 }
```


Hàm signal với tham số truyền vào là SIGINT (nhận tín hiệu Ctrl + C) và hàm `sigint_handler`

```
22 // Hàm để kill tiến trình con
23 void sigint_handler()
24 {
25     if (pid == 0)
26     {
27         // Nếu là tiến trình con, in ra dòng chữ này
28         printf("Ctrl + C pressed, quitting program...\n");
29     }
30 }
31
```

Hàm `sigint_handler`

Chi tiết hơn, ở giai đoạn tiền thực thi các sub-command (lúc người dùng chưa input command), chương trình chỉ có 1 tiến trình hiện hữu là Parent. Khi Parent nhận được SIGINT, nó không phản hồi.

Sau đó, ở giai đoạn thực thi các sub-command (sau khi người dùng đã input command), các tiến trình hiện hữu là Parent, Child, GrandChild. Khi nhận được SIGINT:

- Tiến trình Parent: không phản hồi
- Tiến trình Child: in ra dòng chữ “Ctrl + C pressed...” (với ý nghĩa thông báo cho người dùng).
- Tiến trình Grandchild: Grandchild nhanh chóng gọi `execvp()`. Lúc này, code của nó sẽ bị thay thế thành code của sub-command và nó trở thành Unknown process. Dựa vào cơ chế mặc định của hệ điều hành thì chương trình Unknown process này bị kết thúc.

Bên cạnh đó, để hàm `sigint_handler` đã giới thiệu phía trên phân biệt giữa các tiến trình với nhau, ta dùng một biến toàn cục `pid_t pid` như sau:

```
14 #define DEFAULT_PID 1 // id mặc định để gán cho process chính khi hàm fork() chưa được gọi
20 pid_t pid = DEFAULT_PID;

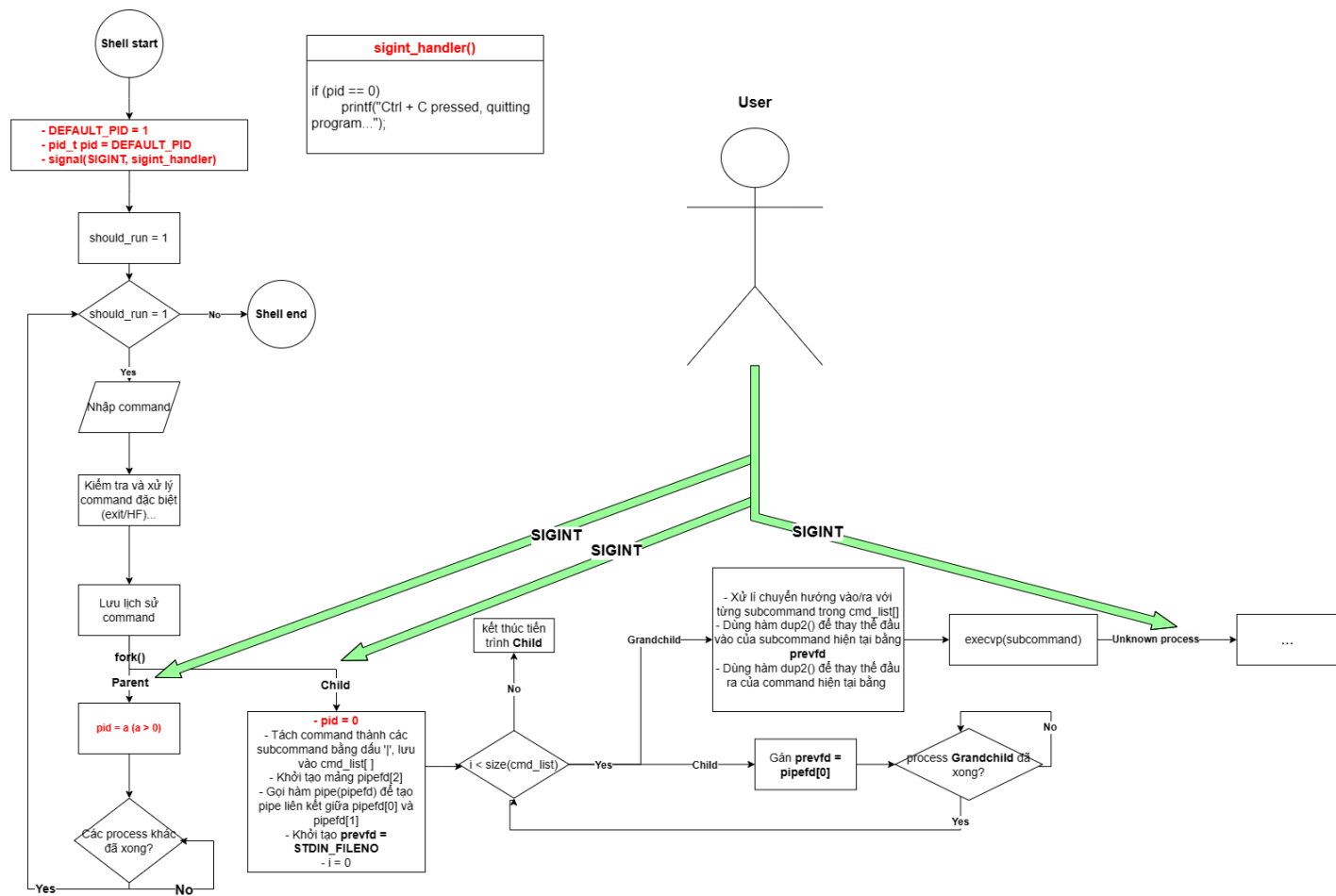
168 pid = fork();
```

Như vậy, `pid` của mỗi tiến trình trong quá trình chạy sẽ là:

- Ở giai đoạn tiền thực thi các sub-command:

- + Tiến trình Parent: (lý do gắn với DEFAULT_PID là vì **pid** có giá trị mặc định là 0, điều này sẽ làm cho Parent in ra 'Ctrl + C pressed...' không mong muốn).
- Ở giai đoạn thực thi các sub-command:
 - + Tiến trình Parent: một số lớn hơn 0
 - + Tiến trình Child: 0
 - + Tiến trình Grandchild: 0 (vì Grandchild sinh ra từ Child nên copy lại biến **pid** của Child) nhưng nhanh chóng biến thành Unknown Process.

Tóm lại, giải thuật câu 5 có thể được biểu diễn thông qua lưu đồ sau:



Lưu đồ giải thuật của câu 5

Chụp hình minh chứng & giải thích kết quả:

- Chạy lệnh ‘top’ để hiện thị một giao diện cho biết các tiến trình đang chạy trong hệ thống. Để thoát khỏi giao diện top và trở về shell, ta nhấn Ctrl + C (có thể thấy dòng chữ ‘Ctrl + C pressed, quitting program..’).

```
it007sh> top
top - 14:42:05 up 5:29, 0 users, load average: 0.04, 0.10, 0.06
Tasks: 37 total, 1 running, 36 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.0 sy, 0.0 ni, 99.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 3350.0 total, 2075.8 free, 759.3 used, 514.9 buff/cache
MiB Swap: 1024.0 total, 1024.0 free, 0.0 used. 2437.4 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
 292 ducmint   20   0   11.0g  198196  44320 S   0.3   5.8   4:24.89 node
   1 root      20   0    2456   1872   1756 S   0.0   0.1   0:00.05 init(Ubuntu-22.
   4 root      20   0    2472    196    196 S   0.0   0.0   0:00.08 init
 210 root      20   0    2472    116     0 S   0.0   0.0   0:00.01 SessionLeader
 211 root      20   0    2472    124     0 S   0.0   0.0   0:00.01 Relay(212)
 212 ducmint   20   0    2888   1004    904 S   0.0   0.0   0:00.03 sh
 213 ducmint   20   0    2888    972    880 S   0.0   0.0   0:00.00 sh
 218 ducmint   20   0    2888    956    864 S   0.0   0.0   0:00.00 sh
 222 ducmint   20   0   956520  94848  40616 S   0.0   2.8   0:44.04 node
 252 ducmint   20   0   849068  57688  37792 S   0.0   1.7   0:05.43 node
 328 ducmint   20   0   94036  43160  14916 S   0.0   1.3   1:05.48 cpptools
 564 ducmint   20   0   603416  54960  36844 S   0.0   1.6   0:02.31 node
1682 ducmint   20   0   4260864 14016   8920 S   0.0   0.4   0:03.11 cpptools-srv
1725 ducmint   20   0   4264748 24160  10268 S   0.0   0.7   0:12.70 cpptools-srv
6450 root      20   0    2464    116     0 S   0.0   0.0   0:00.10 SessionLeader
6451 root      20   0    2480    124     0 S   0.0   0.0   0:03.75 Relay(6452)
6452 ducmint   20   0   609956  64060  36456 S   0.0   1.9   0:14.29 node
6459 root      20   0    2464    116     0 S   0.0   0.0   0:00.00 SessionLeader
6460 root      20   0    2480    124     0 S   0.0   0.0   0:01.35 Relay(6461)
6461 ducmint   20   0   603488  55940  36492 S   0.0   1.6   0:06.10 node
9714 root      20   0    2472    116     0 S   0.0   0.0   0:00.00 SessionLeader
9715 root      20   0    2472    124     0 S   0.0   0.0   0:00.09 Relay(9716)
9716 ducmint   20   0   15612  11936   5312 S   0.0   0.3   0:01.51 zsh
9720 ducmint   20   0    9360   5332   2000 S   0.0   0.2   0:00.01 zsh

Ctrl + C pressed, quitting program...
it007sh>
```

Nhấn Ctrl + C khi chương trình ‘top’ đang chạy

- Tiếp đến ta thử với lệnh ‘cbonsai –live’. ‘cbonsai’ là chương trình tự động vẽ cây bonsai bằng các ký tự ngay trong terminal:



chương trình cbonsai

```
it007sh> sudo apt install cbonsai
[sudo] password for ducmint:
Reading package lists ... Done
Building dependency tree ... Done
Reading state information ... Done
cbonsai is already the newest version (1.2.1-1).
The following packages were automatically installed and are no longer required:
  python-pkg-resources python-setuptools
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 195 not upgraded.
it007sh> █
```

Tải chương trình cbonsai

```
it007sh> cbonsai —live  
Ctrl + C pressed, quitting program ...  
it007sh> █
```

Nhấn Ctrl + C để thoát khỏi cbonsai và trở lại shell

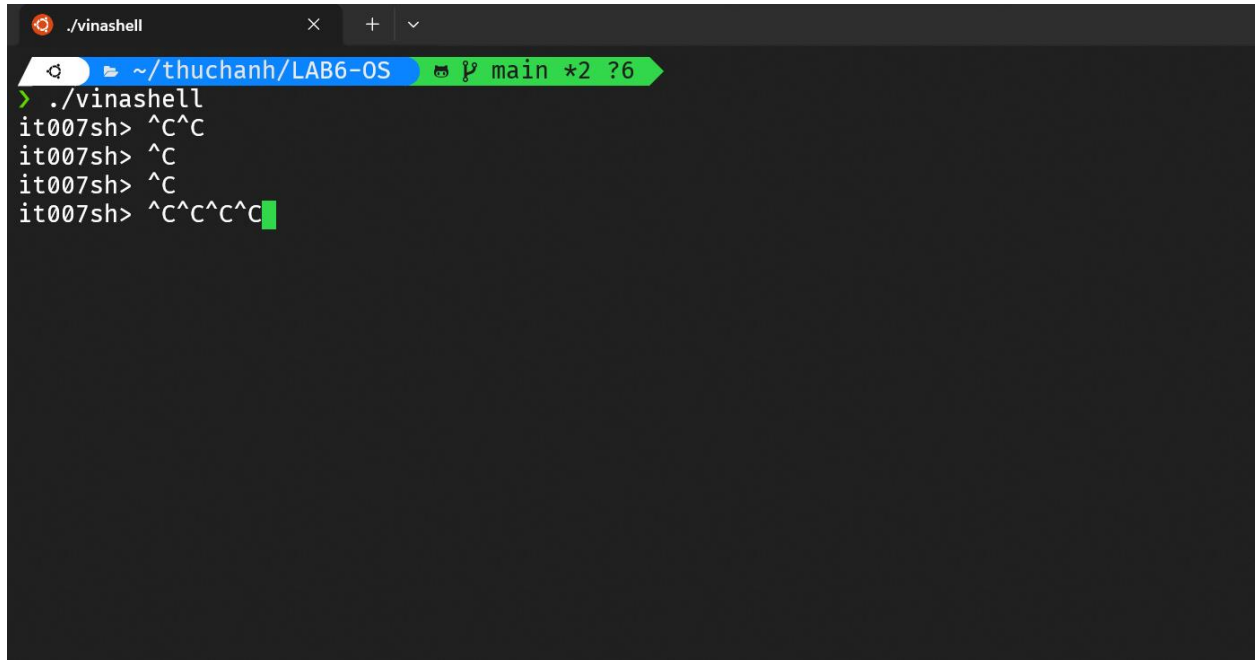
- Tương tự, thử với lệnh sort. Khi sort một file rất lớn, chương trình sort sẽ tốn nhiều thời gian và người dùng không thể thao tác với shell khi sort đang chạy. Ta có thể bấm tổ hợp phím Ctrl + C để ngắt ngang sort. Cùng lúc đó, tiến trình Parent sẽ không phản hồi, còn tiến trình Child sẽ in ra dòng chữ “Ctrl + C pressed, ...”.

```
it007sh> sort < very_large_file.txt  
^C Ctrl + C pressed, quitting program ...  
it007sh> █
```

Ngắt chương trình sort bằng tổ hợp Ctrl + C

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

- Ở giai đoạn tiền thực thi (khi chưa input command), chương trình chỉ có 1 tiến trình duy nhất là tiến trình Parent mà tiến trình này được cài đặt để không có bất kỳ phản hồi nào với tín hiệu SIGINT nên nếu ta bấm Ctrl + C thì sẽ không có chuyện gì xảy ra:



```
./vinashell
~/thuchanh/LAB6-OS main *2 ?6
> ./vinashell
it007sh> ^C^C
it007sh> ^C
it007sh> ^C
it007sh> ^C^C^C^C
```

Nhấn Ctrl + C khi chưa có lệnh nào được thực thi

Xin cảm ơn thầy đã đọc ٩(๑~ٓ~๑)ノ

HẾT