ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN 01 MÔN HỆ ĐIỀU HÀNH

Đề tài:

SIMPLE SHELL

Người thực hiện:

Nguyễn Đình Hoàng Phúc - 18120143

Vũ Gia Tuệ - 18120151

Trương Trọng Lộc - 18120197

Giảng viên hướng dẫn:

Thầy Lê Giang Thanh

Thành phố Hồ Chí Minh – Tháng 11, 2020

MỤC LỤC

1. MỤC TIÊU ĐỒ ÁN:

Khi thực hiện đồ án này, nhóm chúng em đề ra những mục tiêu sau đây:

- Tìm hiểu để có kiến thức về Unix Shell.

- Tìm hiểu các quá trình xử lý tiến trình cha, tiến trình con, Điều hướng Input cũng như Output, cách thức giao tiếp giữa hai lệnh thông qua Unix Pipe.

- Thực hiện được một ứng dụng Simple Shell đơn giản có thể thực hiện được các câu lệnh hệ thống bằng cách áp dụng các kiến thức tìm hiểu và sử dụng các lời gọi hệ thống (system call).

- Hoàn thành đồ án được giao với mức độ hoàn thành 100%, đồng thời tìm hiểu sâu hơn để có thể phát triển đồ án thêm, không chỉ dừng lại ở việc hoàn thành các nhiệm vụ trong đồ án đã đề ra.

2. NGÔN NGỮ VÀ MÔI TRƯỜNG LẬP TRÌNH:

2.1. Về ngôn ngữ:

Đồ án được viết hoàn toàn bằng ngôn ngữ C và được biên dịch bằng *gcc*.

2.2. Môi trường lập trình:

- Nhóm sử dụng môi trường Ubuntu (version 16.04 trở lên) để thực hiện biên dịch và chạy chương trình thông qua Terminal.

- Các công cụ / IDE hỗ trợ trong quá trình viết chương trình: Visual Studio Code, Visual Studio 2019, VIM, CLion.

3. THÀNH VIÊN VÀ PHÂN CHIA CÔNG VIỆC

STT	MSSV	Họ và tên	Công việc
1	18120143	Nguyễn Đình Hoàng Phúc	Xử lý phần tách command thành các arguments, xử lý phần redirect và ghép các modules code lại với nhau.

2	18120151	Vũ Gia Tuệ	Xử lý song song hai tiến trình, xử lý các lệnh internal, và external đơn.
3	18120197	Trương Trọng Lộc	Xử lý pipe, viết cấu trúc báo cáo, kiểm tra chương trình.

4. TỔNG QUAN VỀ KIẾN THỨC:

4.1. Shell:

Shell là một chương trình đặc biệt cung cấp giao diện để người dùng có thể sử dụng dịch vụ của nhân hệ điều hành (kernel) nhằm thực hiện các lệnh (command - thực chất cũng là các chương trình) hệ thống.

4.2. Lệnh nội bộ (internal command) và lệnh bên ngoài (external command):

- Internal Command là các lệnh *built-in* tức có sẵn trong shell, khi dùng lệnh hệ thống không phát sinh bất kỳ tiến trình nào. Trong đồ án Simple Shell này, nhóm chỉ thực hiện minh họa 1 lệnh Internal, đó là lệnh cd.

- External Command là các lệnh không có sẵn trong shell, khi dùng lệnh hệ thống sẽ khởi tạo một process để chạy lệnh đó.

- Khi người dùng nhập lệnh và nhấn Enter, tiến trình cha đọc những gì mà người dùng nhập vào, sau đó tạo ra một tiến trình con để thực thi lệnh đó. Tuy nhiên, nếu sử dụng kỹ thuật này, tiến trình cha sẽ phải đợi tiến trình con thực hiện xong rồi mới tiếp tục. Do đó, một kỹ thuật khác cho phép tiến trình con chạy dưới nền, nghĩa là tiến trình cha không phải đợi tiến trình con thực hiện xong mới tiếp tục. Để thực hiện kỹ thuật này, người dùng cần thêm ký tự '&' vào sau câu lệnh.

4.3. Điều hướng Output:

- Dấu hiệu để nhận diện điều hướng Output thông qua ký tự '>'.

- Bên trái dấu '>', ta có thể hiểu đây là dữ liệu nguồn, bên phải dấu '>' là nơi cần điều hướng đến.

- Đối với xử lý điều hướng, ta cần xác định file descriptor. Đây đơn giản chỉ là một số nguyên dùng để định danh file. Trong trường hợp điều hướng Output, việc định danh file descriptor được thực hiện thông qua hàm *create()*, dễ hiểu bởi vì ở trường hợp này ta cần tạo ra "cái mới" để có thể ghi dữ liệu vào.

4.4. Điều hướng Input:

- Dấu hiệu để nhận diện điều hướng Input thông qua ký tự '<'.

- Bên trái dấu '<', ta có thể hiểu đây là lệnh được điều hướng đến với dữ liệu được nhận từ lệnh bên phải dấu '<'.

- Tương tự như điều hướng Output, ta cần xác định file descriptor. Với điều hướng Input, việc định danh file descriptor được thực hiện thông qua hàm *open()*, dễ hiểu bởi vì ở trường hợp này ta cần mở file để có thể nhận input nhằm điều hướng đến lệnh cần thực hiện.

4.5. Giao tiếp thông qua Pipe:

- Dấu hiệu để nhận diện giao tiếp qua Pipe nhờ vào ký tự '|'. Khi đó, câu lệnh người dùng nhập vào sẽ được phân tách làm 2 phần tương ứng với 2 lệnh giao tiếp với nhau, trong đó Pipe đóng vai trò như một "đường ống" nối 2 tiến trình lại với nhau.

- Lệnh bên trái ký tự '|' sau khi thực hiện cho ra Output, Output này sẽ trở thành Input để thực hiện lệnh sau ký tự '|' để ra được Output cuối cùng.

- Để thực hiện việc giao tiếp giữa 2 tiến trình, ta cần một mảng số nguyên biểu thị 2 file descriptor. *fd[0]* được hiểu đơn giản là tính năng đọc, còn *fd[1]* được hiểu là tính năng ghi.

- Trong quá trình thực hiện, cần lưu ý đến việc đóng file descriptor cách hợp lý để tránh xảy ra các xung đột. Cụ thể, đóng những file descriptor không cần dùng, vì mỗi tiến trình có 2 file descriptor nhưng chỉ sử dụng 1.

4.6. Lịch sử lệnh:

- Khi người dùng thực hiện gõ câu lệnh "!!", tính năng lịch sử lệnh được thực hiện.

- Tính năng này nhằm đưa ra câu lệnh gần nhất cho người dùng (kể cả những câu lệnh thực hiện được lẫn nhưng câu lệnh lỗi do sai cú pháp,…).

- Nếu người dùng chưa nhập câu lệnh nào mà sử dụng tính năng xem lịch sử lệnh thì thông báo "Không có lệnh nào trong lịch sử".

4.7. Cách thức vận hành của Shell:

Qua tìm hiểu, nhóm đã rút ra được các bước xử lý lệnh cơ bản của một Shell - cụ thể hơn là của bash (Bourn-Again Shell):

4.7.1. Nhập dữ liệu:

- Shell dùng thư viện GNU readline để đọc tương tác (interactive) input của người dùng. Ngoài ra shell còn có thể dùng stdio hoặc là buffered input package của bash để làm việc này.

- Dù có là cách nào đi chăng nữa thì output của bước này cũng sẽ là một buffer gồm các kí tự, ngừng đọc khi thấy dấu xuống hàng - newline (\n).

4.7.2. Xử lý dấu nháy (quotes):

- Bash tìm vị trí của các dấu nháy, dãy kí tự nằm giữa một cặp dấu nháy sẽ được đánh dấu là ESCAPED, nghĩa là chúng sẽ chỉ được coi như những kí tự thông thường mà không có ý nghĩa đặc biệt gì cả. Chẳng hạn như các dấu <, >, |, … ở trên, chúng sẽ mất tính năng điều hướng input/output, piping nếu nằm giữa một cặp dấu nháy. Cụ thể hơn:

- Dấu nháy đơn - single quote (‘): Tất cả những kí tự không dấu nháy đơn sẽ trở thành ESCAPED.
- Dấu nháy kép - double quote(“): Tất cả những kí tự không nằm trong tập {“, $, `, \ } sẽ trở thành ESCAPED.

- Điều này giải thích một số trường hợp mà người nhập đã bấm Enter newline nhưng Shell vẫn đợi input tiếp của người dùng. Đó là vì dấu newline đó đã được đánh dấu là ESCAPED, nên không còn tác dụng báo cho shell là đã terminate input. Chỉ có dấu newline không bị đánh dấu - UNESCAPED - thì mới được tính.

4.7.3. Tách thành các lệnh (command):

Bash tách input thành một hoặc nhiều các command sử dụng dấu chấm phẩy (;). Lưu ý là chỉ những dấu chấm phẩy UNESCAPED (ở trên) mới được dùng để tách command.

4.7.4. Tách các toán tử đặc biệt:

Với mỗi command được tách ra ở bước trên, shell tìm những toán tử (tất nhiên là UNESCAPED) như là dấu ngoặc kép {..}, dấu < .. , > .. , ..|.., << .., >> .. v.v. Các toán tử này được xử lý theo một thứ tự nhất định nào đó.

Những toán tử điều hướng thì bị xóa khỏi command, số khác thì được tính toán và kết quả tính toán của chúng được thay vào vị trí thích hợp trong command.

Qua bài tập này, nhóm đã biết được rằng, toán tử & (subshell) thực hiện trước toán tử | (piping). Và 2 toán tử điều hướng > và < thực hiện sau hai toán tử này.

4.7.5. Mở rộng - Expansions:

- Shell có nhiều toán tử mở rộng phức tạp, được xử lý như một hệ thống pipeline. Dưới đây chỉ là danh sách một số toán tử chứ không phải thứ tự thực hiện của chúng.

- Mở rộng Parameter (Parameter expansions): Được ký hiệu là $<tên parameter>. Khi gặp dấu này shell thay nó bằng giá trị tương ứng với parameter.

- Tách từ (Word splitting): Lệnh được tách thành các tên lệnh và các arguments, dựa trên dấu cách - whitespace.

- Pathname Expansions, còn gọi là Globbing.

- Tilde expansion: mở rộng dấu '~'.

- Arithmetic Expansions.

- Variable Expansions...

4.7.6. Xử lý lệnh:

Sau tất cả các bước xử lý, mở rộng ở trên, các câu lệnh bây giờ đã sẵn sàng để được shell/bash đọc hiểu và xử lý:

- Câu lệnh đơn (Simple Commands) : bao gồm các lệnh nội bộ và lệnh bên ngoài (4.2).

- Điều khiển công việc (Job Control).

- Câu lệnh phức (Compound Commands): Ta có thể đặt các câu lệnh đơn trong các keyword như if, while, for, ... để điều khiển luồng thực hiện của shell, tương tự như một ngôn ngữ lập trình thực thụ. Và đúng như thế, có một cái tên đặt cho loại ngôn ngữ lập trình đặc biệt này, đó là 'Shell programming language'.

5. MÔ TẢ CÁCH THỨC VẬN HÀNH CHƯƠNG TRÌNH:

Chương trình của nhóm chủ yếu dựa trên các bước hoạt động của Shell/ bash đã được trình bày ở phần 4.7, có sự đơn giản hóa nhất định.

- **Bước 1:** Khi bắt đầu chạy Shell, chương trình chờ người dùng nhập vào câu lệnh thông qua vòng lặp. Sau đó, đọc dữ liệu mà người dùng nhập vào từ chuỗi string. Vòng lặp được kết thúc, nghĩa là chương trình được thoát khi người dùng nhập "exit".

- **Bước 2:** Kiểm tra người dùng có nhập lệnh "!!" để yêu cầu xem lịch sử lệnh gần nhất hay không. Nếu có, xuất ra thông báo / kết quả phù hợp. Nếu không, chuyển xuống bước 3.

- **Bước 3:** Ta thực hiện parse arguments từ chuỗi string nhận được ở bước 1, dấu hiệu để phân tách giữa các argument dựa vào khoảng cách. Ngoài việc parse arguments, ta cũng thực hiện kiểm tra câu lệnh người dùng có chứa '&' hay không và lưu lại trạng thái có / không thông qua biến mode.

- **Bước 4:** Ưu tiên thực thi xử lý song song (nếu biến mode = 1), khi thực hiện xử lý song song, kiểm tra xử lý Pipe (lưu ý, trong xử lý Pipe có gọi hàm xử lý Redirection để thực thi các lệnh).

- Nếu có chứa '|', thực hiện xử lý Pipe bên trong xử lý song song.
- Nếu không chứa '|', thực hiện gọi lệnh xử lý Redirection.
    - Kiểm tra có chứa '>' hoặc '<' hay không, nếu có thực thi xử lý Redirection.
    - Nếu không, chuyển sang hàm xử lý thông thường. Nếu đó là lệnh built-in, gọi đến hàm xử lý lệnh Internal thực thi. Nếu không, thực thi bình thường thông qua hàm xử lý lệnh External.

6. MÔ TẢ CÁC TÍNH NĂNG, CÁCH CÀI ĐẶT XỬ LÝ CÁC LỆNH:

6.1. Phân tích cú pháp lệnh:

6.1.1. Đọc một vào một chuỗi chứa dòng lệnh:

Hàm char * readLine() thực hiện các câu lệnh nhằm mục đích sau:

- Xin cấp phát vùng nhớ cho chuỗi cần nhập.

- Đọc lần lượt từng ký tự cho đến khi gặp ký tự ‘EOF’ hoặc ‘\n’.

- Trong lúc đọc, xử lý trường hợp nhiều dấu cách liền kề nhau (chỉ giữ nguyên lại một dấu cách). Đối với các dấu cách ở đầu và cuối chuỗi, ta không đọc vào.

- Trong lúc đọc, kiểm tra xem lúc nào vùng nhớ cho chuỗi đầy thì xin cấp phát thêm một lượng BUFFER_SIZE bytes.

- Trong lúc đọc, nếu độ dài chuỗi > MAX_LENGTH thì thông báo lệnh quá dài và thoát chương trình.

- Trong quá trình xin cấp phát luôn kiểm tra xem quá trình cấp phát có thành công hay không.

=> Trả về chuỗi thể hiện dòng lệnh vừa được nhập vào sau khi xóa các dấu cách dư thừa.

6.1.2. Chia chuỗi đã cho thành mảng các arguments:

Hàm char ** parseArgs(char * str, int * mode) thực hiện các chức năng sau:

- Khai báo char ** args, mảng để lưu kết quả.

- Duyệt qua từng ký tự trong xâu str, sau khi gặp dấu cách, tăng kích thước của args, và tạo vùng nhớ mới để chép các ký tự của một argument trong xâu str sang phần tử cuối cùng của args.

- Sau khi duyệt xong thêm argument cuối vào args.

- Kiểm tra nếu argument cuối là “&”, gán mode = 1 và chỉnh argument cuối thành 0. Ngược lại mode = 0 và thêm vào args một phần tử 0 ở cuối để báo kết thúc mảng arguments.

- Trong quá trình xin cấp phát luôn kiểm tra xem quá trình cấp phát có thành công hay không.

6.2. Các lệnh built-in và lệnh thực thi cơ bản (internal and external command):

- Chi tiết trong hàm: *void processSimpleCommand(char **args)*. Hàm này có nhiệm vụ nhận diện lệnh đang gọi là lệnh i*nternal* hay *external* và có cách xử lý phù hợp:

- Trước tiên gọi hàm *int executeInternalCommand()*. Hàm này bao gồm việc nhận diện lệnh có phải là lệnh built-in của Shell hay không và nếu phải thì xử lý lệnh đó. Để minh họa, hàm mới chỉ xử lý một lệnh internal duy nhất là ***cd***. Để thay đổi working directory của một tiến trình cần một lệnh đặc biệt là *chdir*(<đường dẫn mới>) thuộc thư viện <*unistd.h*>. Ta xét argument (duy nhất) của lệnh:

- Nếu không có arg nào, cd sẽ chuyển tới *$HOME*. Ta tìm địa chỉ của *HOME* bằng lệnh *getenv("HOME")*, và *chdir()* đến địa chỉ đó.
- Nếu *arg* là *'-'*: lệnh *'cd -'* đưa người dùng về working directory gần đây nhất. Shell thật sự sẽ dùng biến environment là *$OLDPWD* để biết được địa chỉ cần đó, vì thế để mô phỏng thì chương trình Shell của ta sẽ khai báo một string toàn cục là *OLDPWD[]* để lưu thông tin này, và *chdir()* đến địa chỉ của *OLDPWD[]* (trước đó lưu *OLDPWD[]* là working directory hiện tại thông qua mảng tạm là *temp[]*).
- Nếu không phải 2 trường hợp trên ta chỉ cần *chdir()* đến nội dung của argument.
- Hàm trả về -1 nếu lệnh truyền vào không phải lệnh built-in, 0 nếu thực hiện lệnh built-in không thành công, và 1 nếu thực hiện lệnh built-in thành công.

- Nếu hàm executeInternalCommand() trả về -1 nghĩa là lệnh được nhập là lệnh *external*. Ta gọi hàm *void executeExternalCommand():*

- Hàm sẽ *fork()* một tiến trình mới và gọi system call *execvp(args)* để thực hiện lệnh được truyền vào thay cho chúng ta.
- Tiến trình cha sẽ đợi tiến trình con thực hiện xong bằng lệnh *wait()*.

6.3. Thực thi dưới nền (&):

- Cụ thể trong hàm *void processParallel(char **args, int mode)*. Biến *mode = 1* cho biết hàm *pargeArg()* đã phát hiện ra toán tử &, và ta cần xử lý nó.

- Toán tử & có ý nghĩa là tạo ra một subshell mới và chạy câu lệnh đã nhập trong subshell này. Để mô phỏng thì chương trình Shell sẽ:

- Tạo ra một tiến trình con thông qua lệnh *fork()*. Khác với xử lý lệnh external thì Shell chính không đợi Subshell, nên tiến trình cha lúc này sẽ không đợi tiến trình con (bằng *wait()*).

- Dù có tạo subshell hay không, ta tiếp tục xử lý những toán tử khác của lệnh, bằng cách truyền args (lúc này chắc chắn không còn toán tử &) vào hàm *processPipe()*.

6.4. Điều hướng Output và điều hướng Input:

- Chi tiết trong hàm: *void processRedirectCommand(char** args)*

- Kiểm tra xem mảng các arguments của lệnh có chứa kí hiệu redirection hay không, nếu có ta tiến hành xử lí như sau:

6.4.1. Điều hướng Output:

- Tạo ra file và luồng xuất mới bằng lệnh creat.

- Lưu lại stdout hiện thời.

- Điều hướng luồng xuất sang file.

- Sau khi thực thi các lệnh, điều hướng output lại sang stdout đã lưu trước đó.

- Trong lúc thực hiện các bước trên luôn kiểm tra xem việc tạo luồng mới hay đóng file có thành công hay không.

6.4.2. Điều hướng Input:

- Thực hiện tương tự như điều hướng Output, tuy nhiên ta sẽ lưu lại trạng thái của stdin để điều hướng lại lúc sau.

- Ghi chú: thao tác điều hướng nhóm không thực hiện tạo một process con rồi điều hướng trên process con đó mà chỉ thay đổi luồng nhập xuất trực tiếp trên process chính để đảm bảo những lệnh internal có thể thực hiện đúng.

6.5. Giao tiếp giữa hai lệnh thông qua Pipe:

- Chi tiết chức năng trong hàm: *void processPipe(char** args)*

- Kiểm tra xem mảng các arguments của lệnh có chứa kí hiệu pipe (|) hay không thông qua hàm tìm vị trí của argument chứa ký tự '|' *positionPipe(char** args)*. Hàm này trả về số nguyên biểu thị vị trí của argument chứ ký tự '|', nếu không có trả về -1.Ta tiến hành xử lí tiếp theo như sau:

- Nếu không chứa ký hiệu Pipe, ta thực hiện gọi hàm *processRedirectCommand(char** args)* đã được mô tả ở mục 6.5.
- Nếu kết quả trả về của hàm là 0, nghĩa là người dùng nhập ký tự '|' đầu tiên trong câu lệnh, ta xuất ra thông báo lỗi Syntax và return khỏi hàm.
- Nếu kết quả trả về của hàm lớn hơn 0, ta cần tạo Pipe thông qua *pipe()* system call. Việc thực hiện này thông qua hàm int pipe trong thư viện *#include <unistd.h>* với tham số là mảng số nguyên 2 phần tử đại diện cho 2 file descriptors: đọc và ghi.
    - Sau đó, tiến hành parse để lấy các arguments trước và sau ký hiệu '|' thông qua hàm *parseFirstArgsPipe(char** args, int position)* để lấy phần đầu của câu lệnh Pipe và *parseSecondArgsPipe(char** args, int position)* để lấy phần sau của câu lệnh Pipe.
    - Sau khi tách các arguments, thực hiện *fork()* để tạo ra tiến trình mới tương ứng với lệnh trước ký hiệu '|'. Sau đó, tiến hành duplicate *fds[1]* vào *standard output* để phù hợp với điều kiện output của lệnh 1 sẽ trở thành input của lệnh 2.
    - Gọi hàm processRedirectCommand(char** args) để xử lý lệnh với tham số là *char** argsFirst.*
    - Thực hiện tương tự để thực hiện cho lệnh 2 (*fork()* rồi đến duplicate *fds[0]* vào *standard input* và gọi hàm *processRedirectCommand(char** args)* để xử lý lệnh với tham số là *char** argsSecond*).
    - Trong lúc thực hiện các bước trên luôn kiểm tra xem việc tạo tiến trình mới hay đóng file có thành công hay không.

6.6. Lịch sử lệnh:

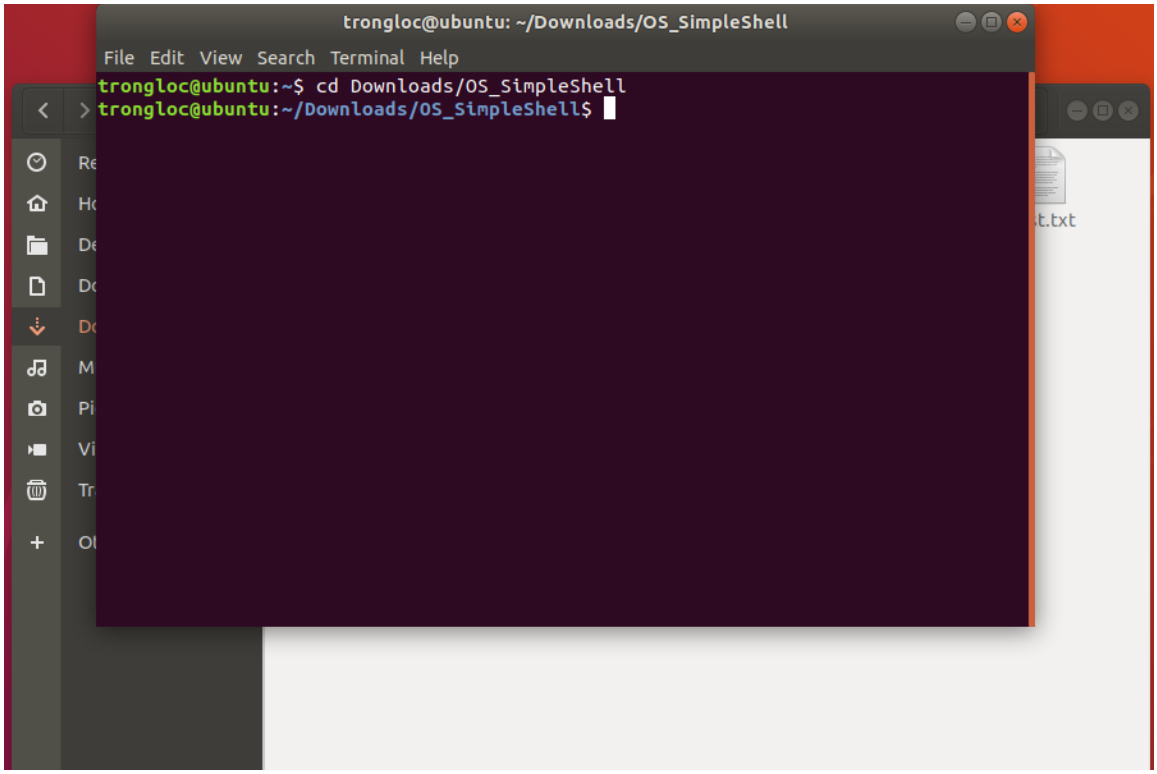
- Chi tiết chức năng trong hàm *void shLoop().*

- Để thực hiện lịch sử lệnh, ta sử dụng hai biến kiểu *char** là *command* và *last_command*. Trước khi đọc một lệnh, ta giải phóng vùng nhớ *cho last_command* và gán *last_command*

thành *command*. Tuy nhiên trong trường hợp hai biến này cùng trỏ đến một địa chỉ hoặc *last_command* mang giá trị không ta không thực hiện thao tác giải phóng vùng nhớ. Nếu đọc một lệnh mới mang giá trị "!!", ta gán command thành *last_command*. Nếu *last_command* có giá trị, *echo* lệnh đó và thực thi, ngược lại báo lỗi cho người dùng.

7. HƯỚNG DẪN SỬ DỤNG SIMPLE SHELL PLTsh:

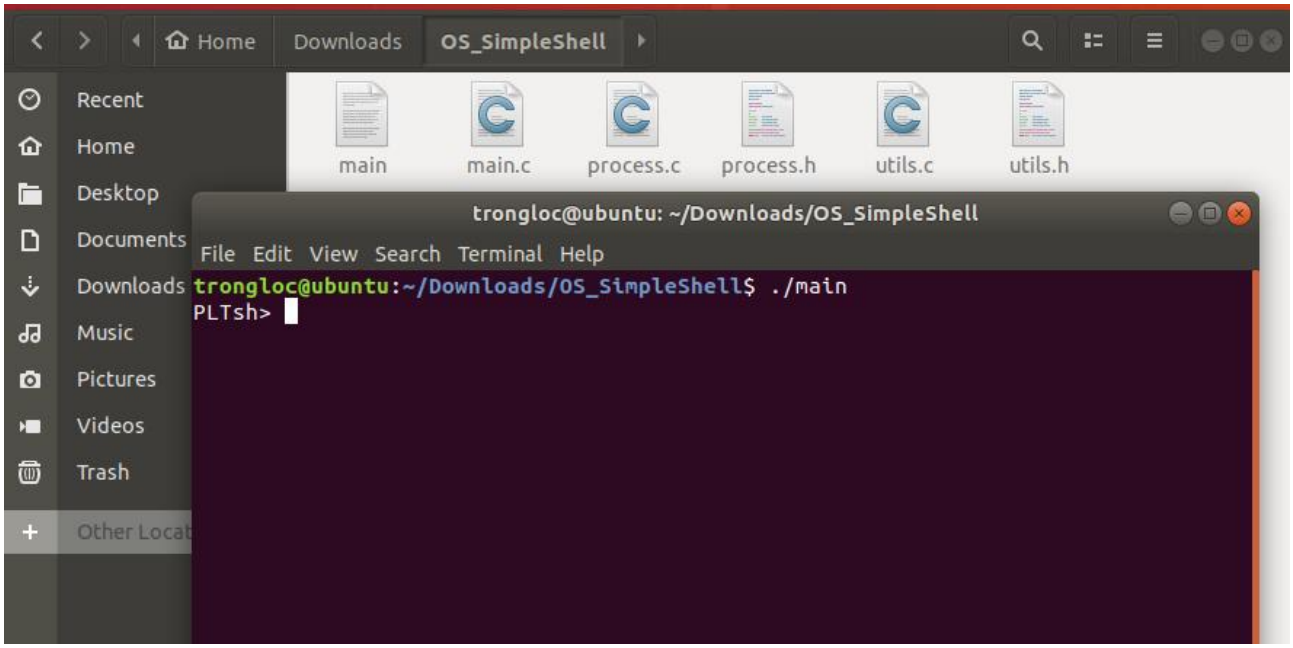
- Sử dụng *Terminal* trong môi trường *Ubuntu* như đã trình bày ở mục 2, tiến hành thay đổi đường dẫn đến folder *OS_SimpleShell*. Giả sử, ta lưu folder *OS_SimpleShell* trong thư mục *Downloads*, ta thực hiện:



- Khi đó, đường dẫn đã được thay đổi đến thư mục chứa source code đồ án, ta có 2 cách để vào *PLTsh*:

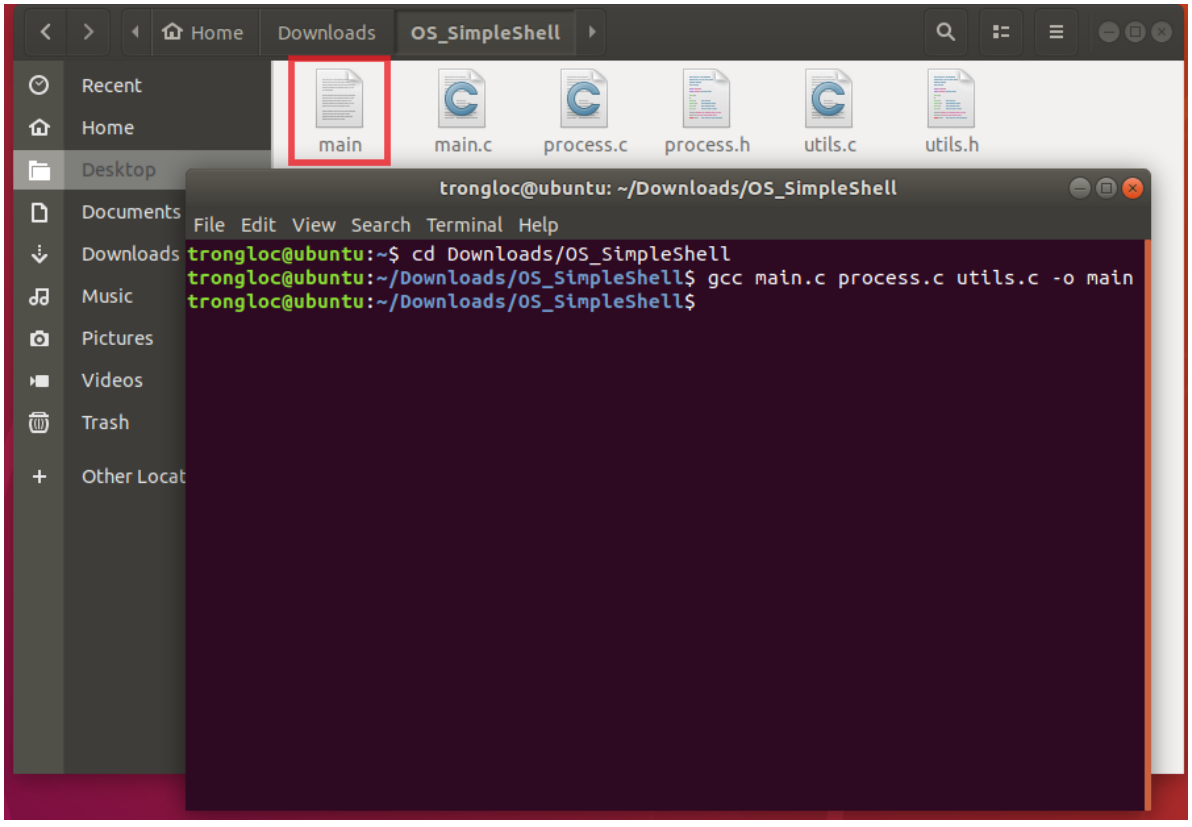
7.1. Chạy trực tiếp trên file mà nhóm đã biên dịch sẵn:

- Tại thư mục chứa source code đồ án, thực thi lệnh sau:

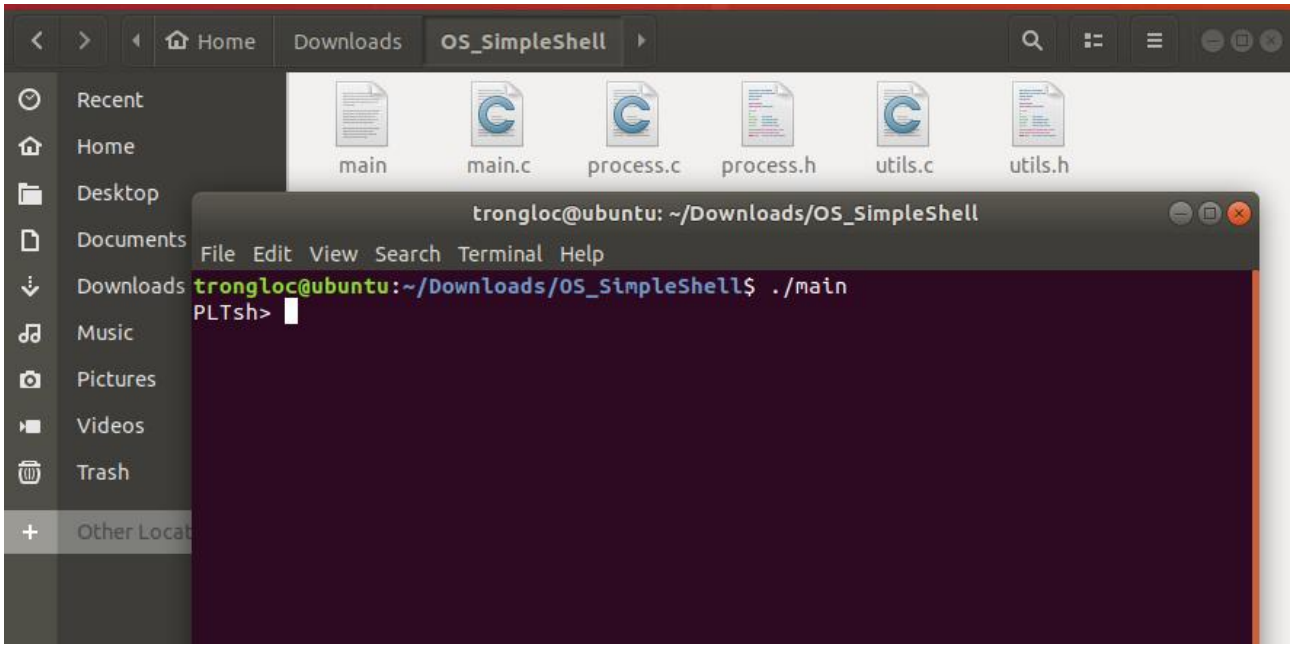


7.2. Tự biên dịch chương trình và chạy:

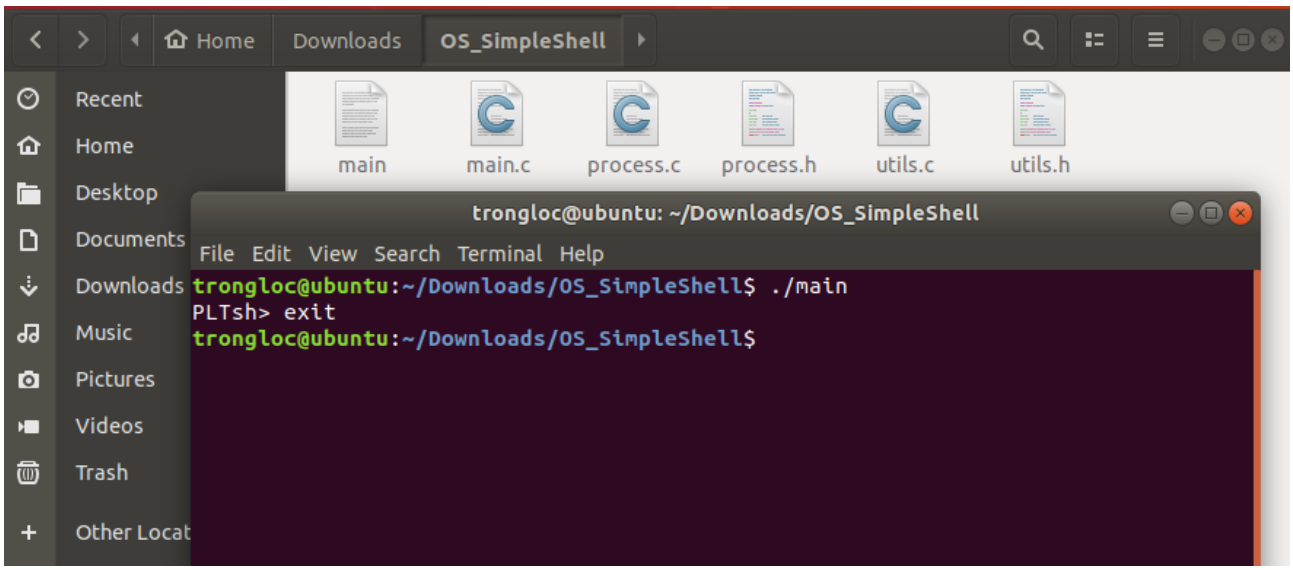
- Để biên dịch các file source được viết bằng C và đổi tên file output thành *main* như sau (lưu ý, ta cần phải cài *gcc* trước khi thực thi lệnh này):



- Sau khi thực hiện, ta thấy xuất hiện file *main* như hình trên, ta tiến hành chạy file này như sau:



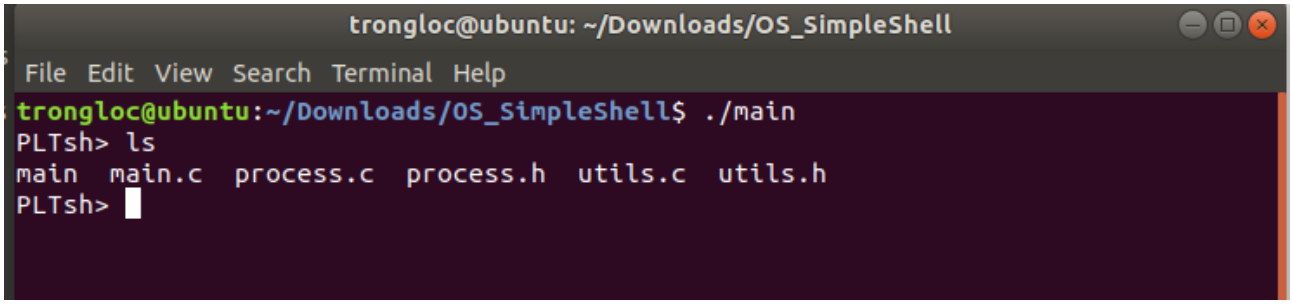
- Khi đó, ta thấy Shell PLTsh được khởi chạy, ta có thể nhập vào các lệnh để Shell này thực thi các lệnh đó. Khi không còn nhu cầu sử dụng Shell nữa, ta nhập *"exit"* để thoát như hình sau:



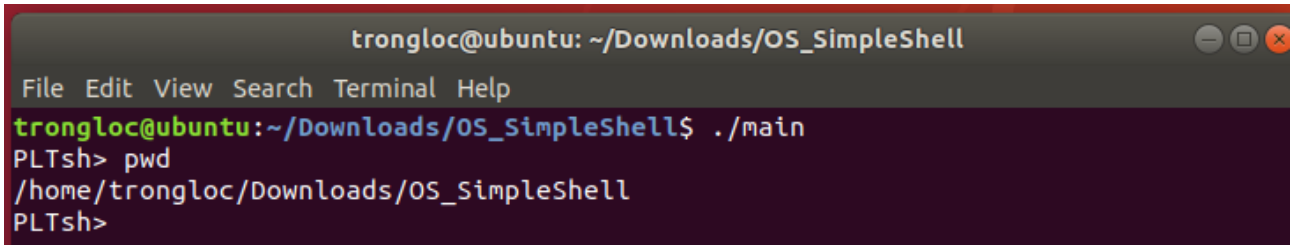
8. MINH HỌA ẢNH CHỤP KẾT QUẢ CÁC BỘ TEST:

8.1. Các lệnh thực thi cơ bản (external command):

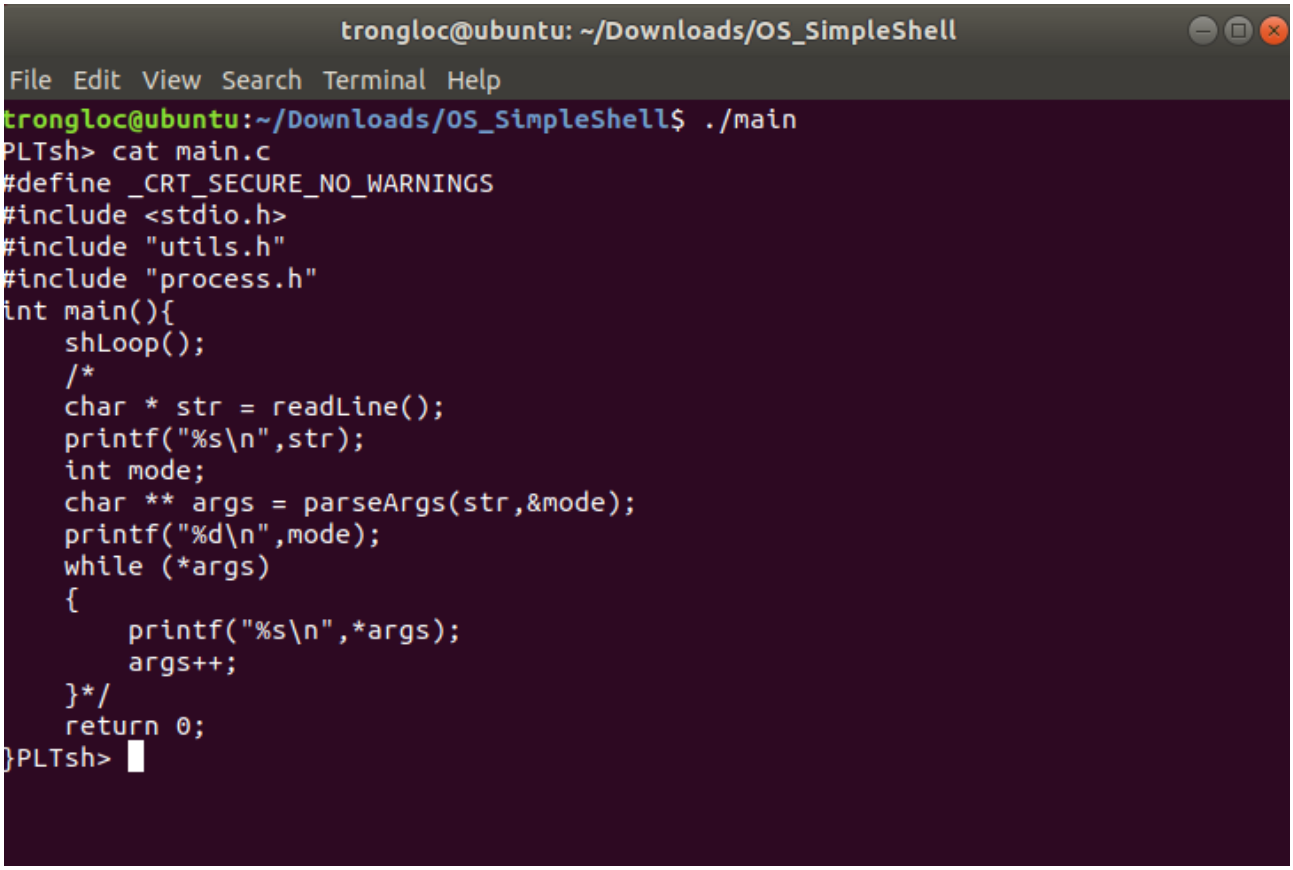
- Xem danh sách các file/thư mục có trong đường dẫn hiện tại:



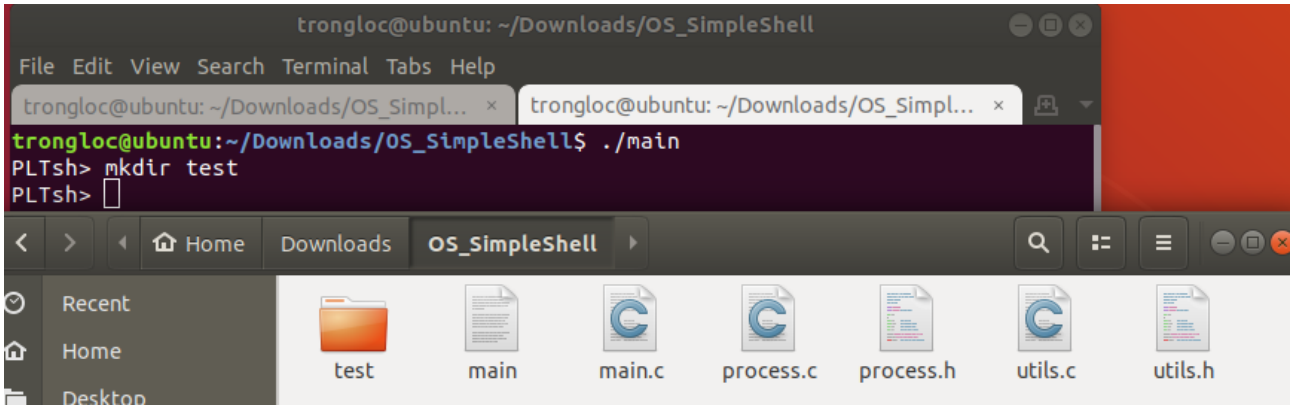
- Xem đường dẫn hiện tại:



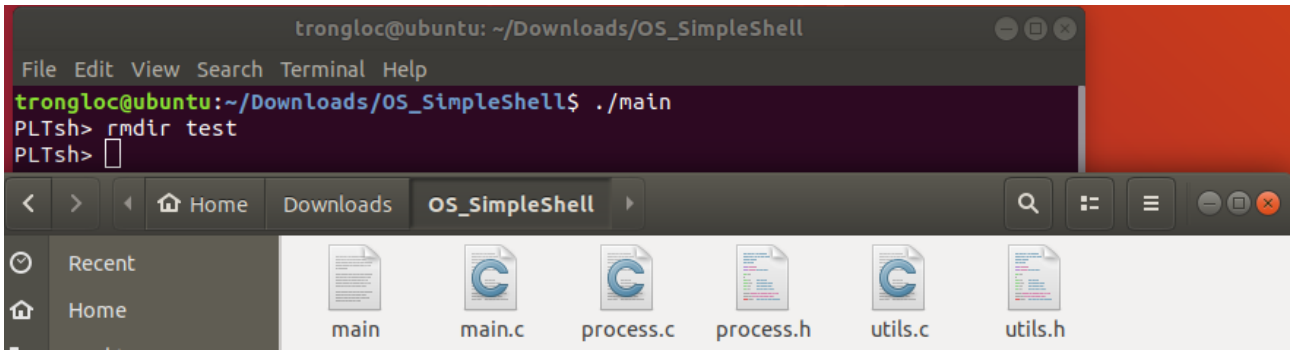
- Thực hiện lệnh *cat* để mở file và hiển thị nội dung:



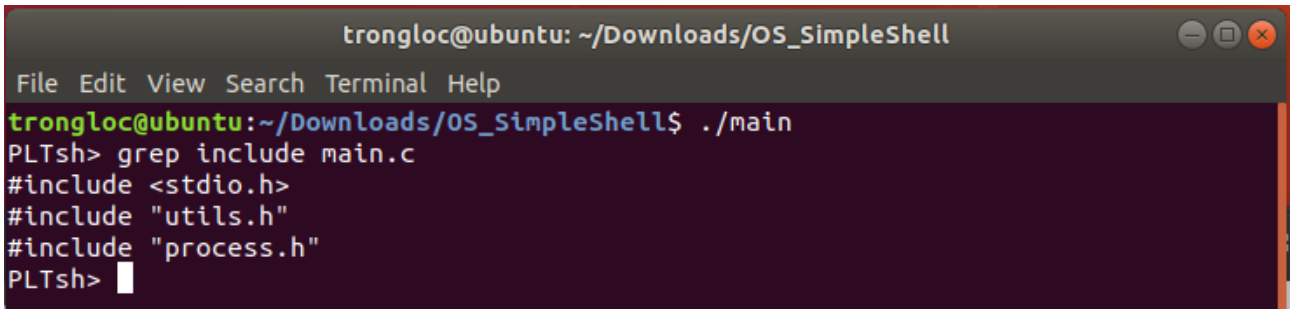
- Tạo thư mục mới:



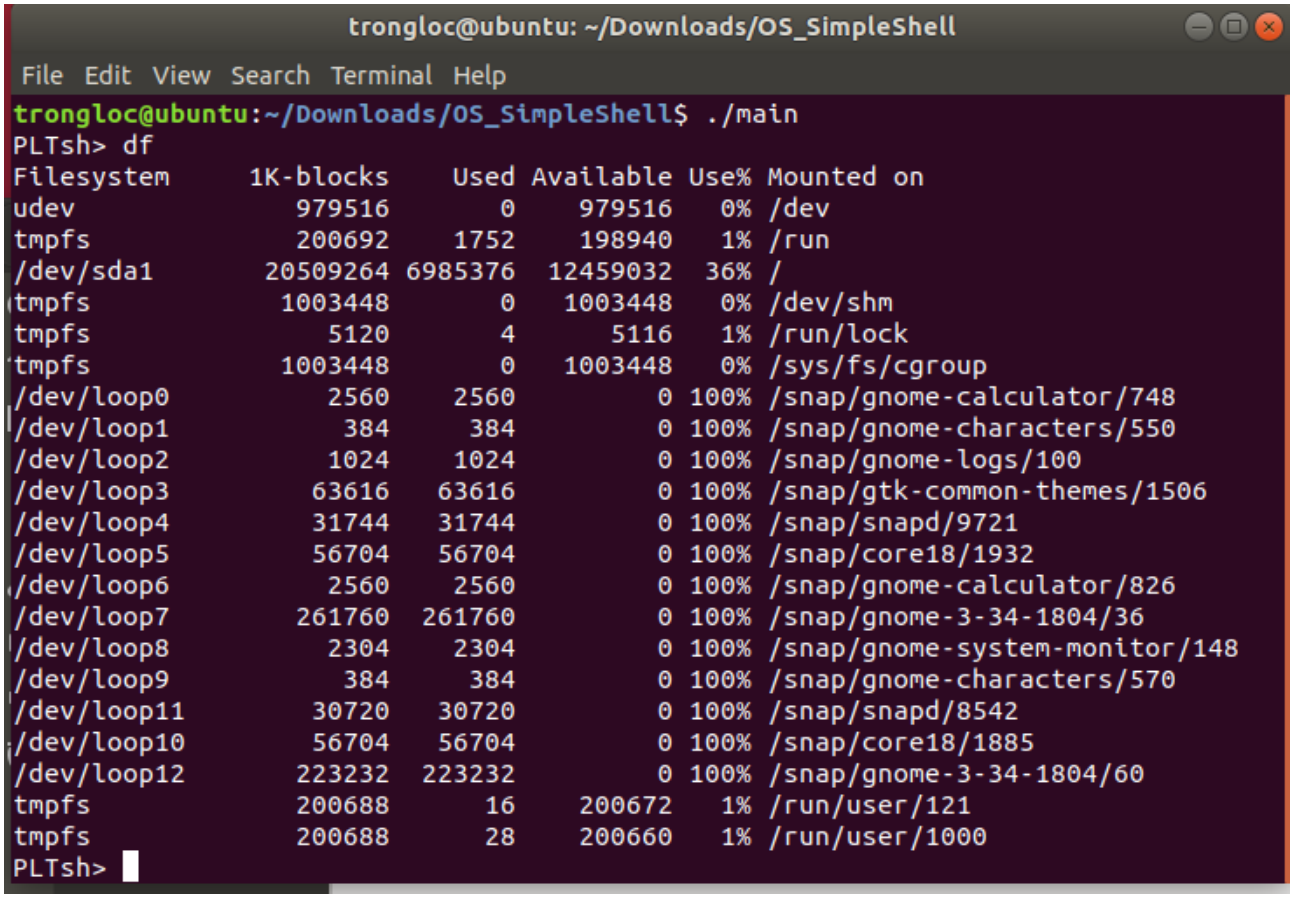
- Xóa thư mục vừa tạo:



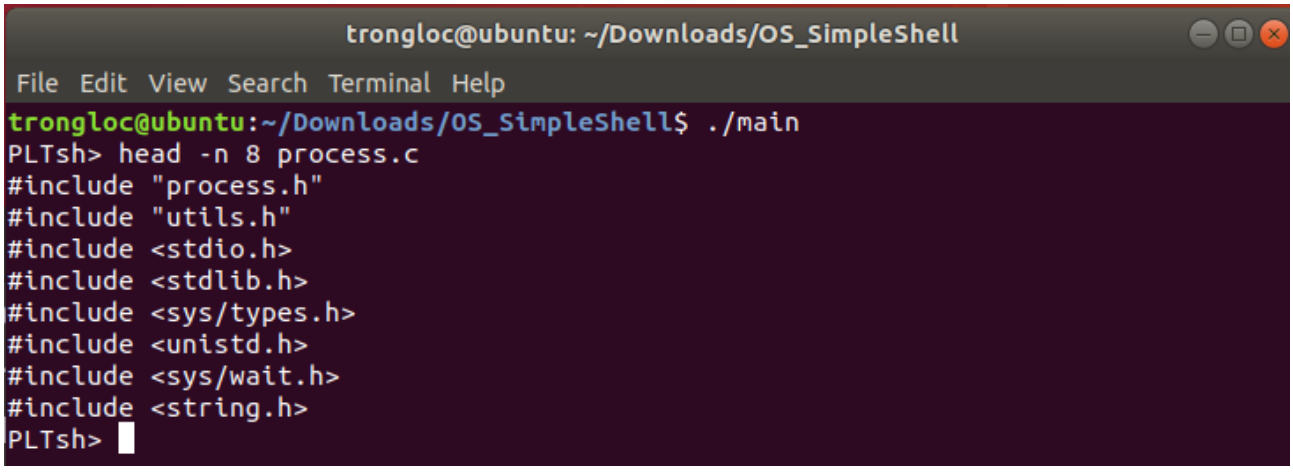
- Tìm kiếm thông qua từ khóa và xuất ra các dòng chứa từ khóa đó trong file:



- Đưa ra báo cáo về dung lượng lưu trữ được sử dụng trong hệ thống:

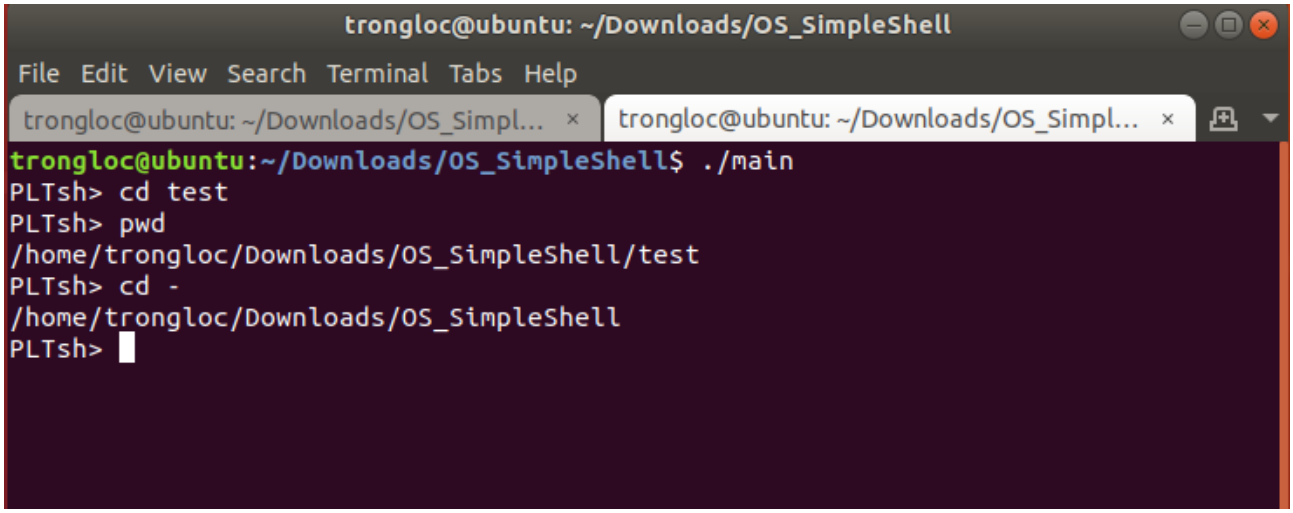


- Hiện 8 dòng đầu của file *process.c*:



8.2. Lệnh built-in (internal command):

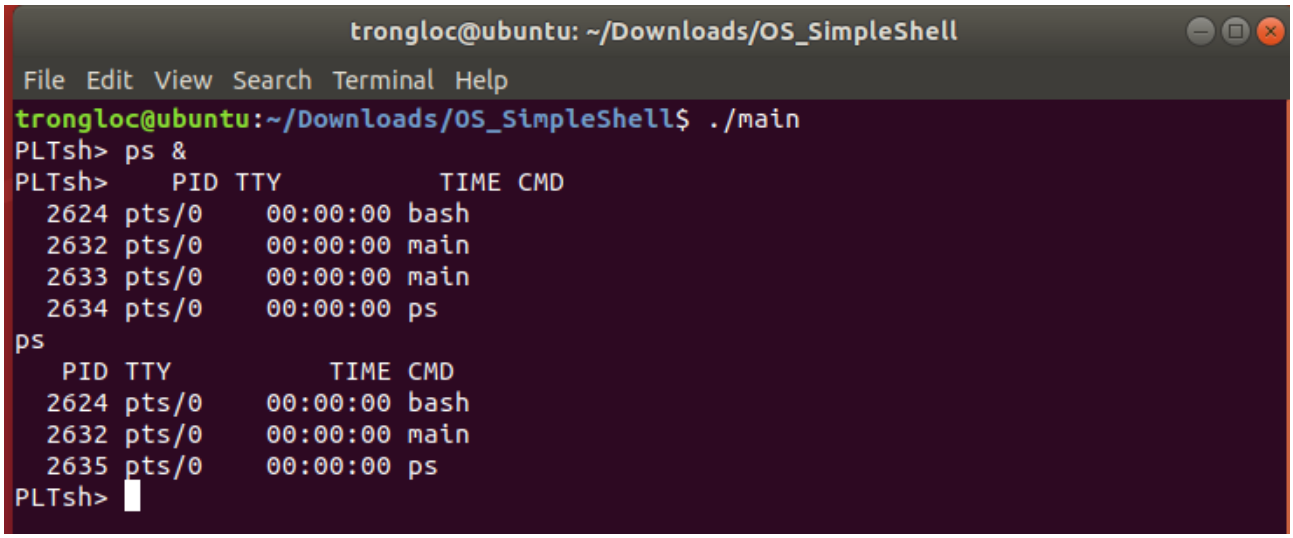
Giả sử trong thư mục có chứa thư mục con là *test*, ta thay đổi đường dẫn đến thư mục con này. Sau đó, kiểm tra đường dẫn hiện tại bằng lệnh *pwd* xem kết quả đúng chưa. Cuối cùng ta thực thi lệnh *cd* - để quay lại đường dẫn trước:



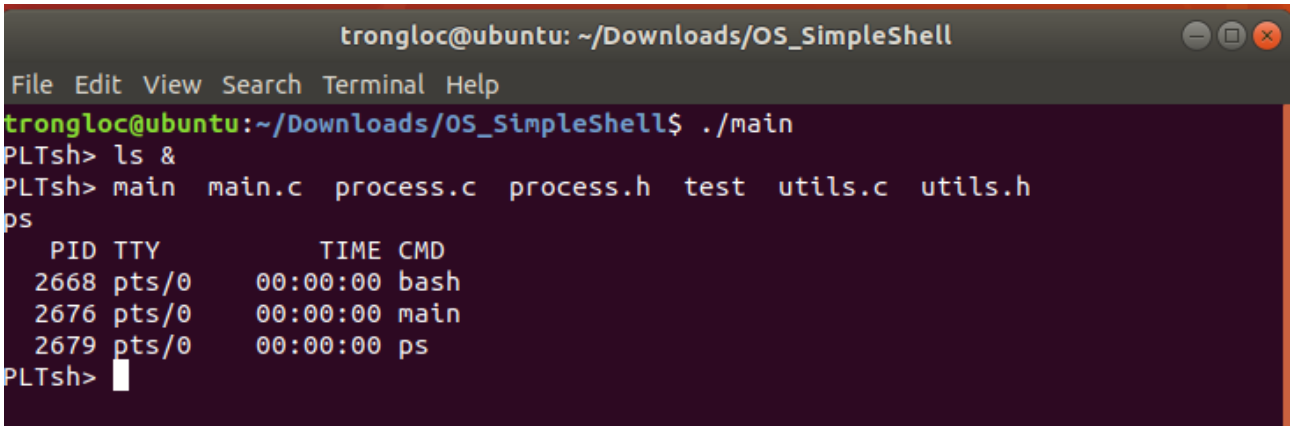
8.3. Thực thi dưới nền (&):

- Ta thực hiện lệnh *ls &*, khi đó tiến trình con thực hiện lệnh *ls* và trả kết quả, lúc này Shell vẫn hiện dấu nháy trên prompt chờ người dùng nhập lệnh để tiến trình cha xử lý.

- Tương tự như trên, ta có thể thay thế lệnh *ls &* thành *ps &* như sau:

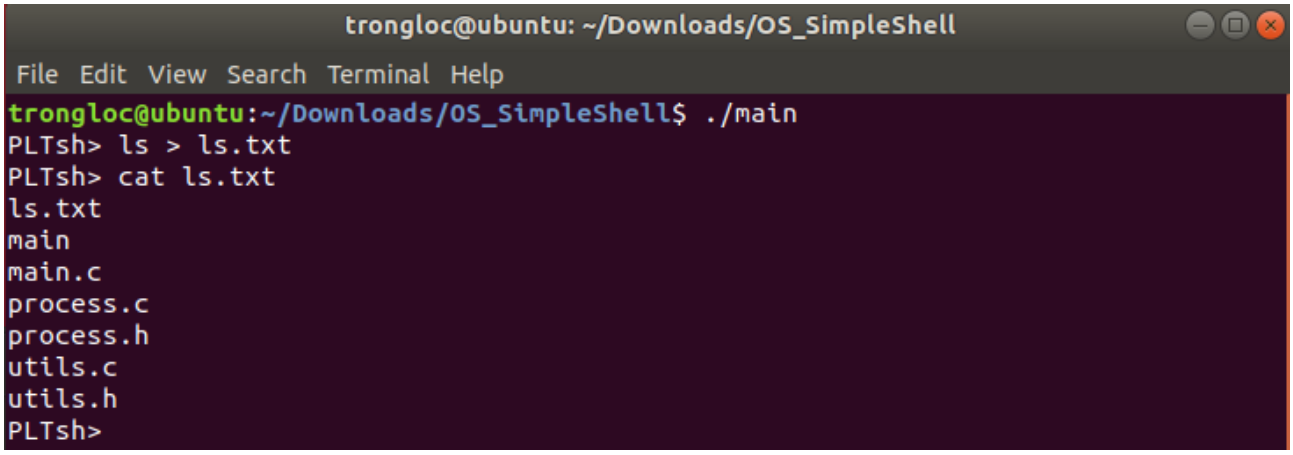


- Ngoài ra, ta có thể nhập lệnh *ls &* để cho tiến trình con xử lý lệnh *ls* và sau đó khi nhập lệnh cho tiến trình cha, ta có thể nhập lệnh *ps*:



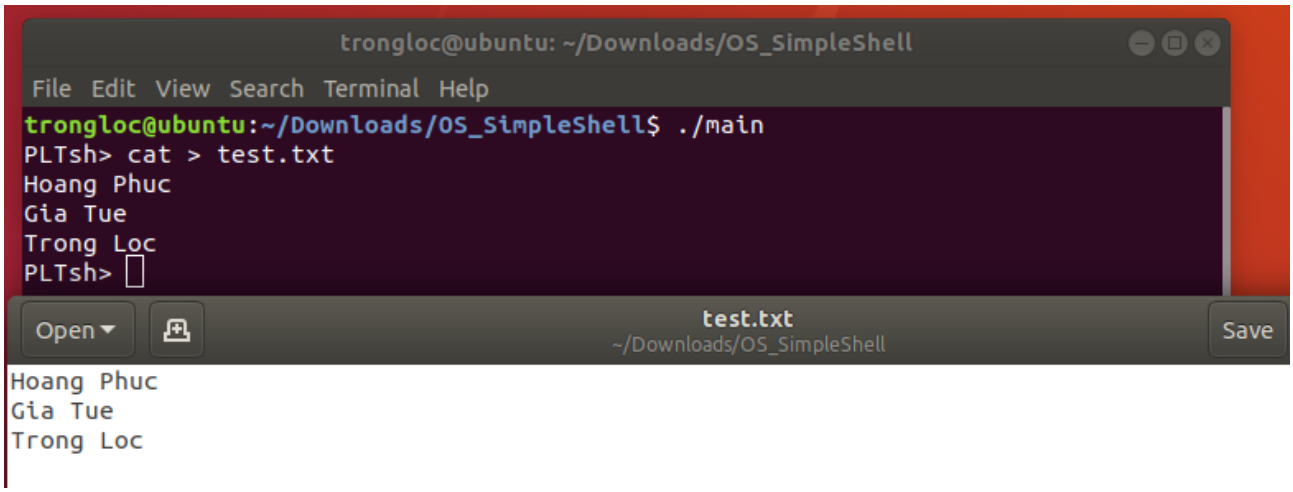
8.4. Điều hướng Output:

- Ta muốn lưu danh sách các file/thư mục con trong thư mục hiện tại vào file *list.txt*, ta thực hiện:



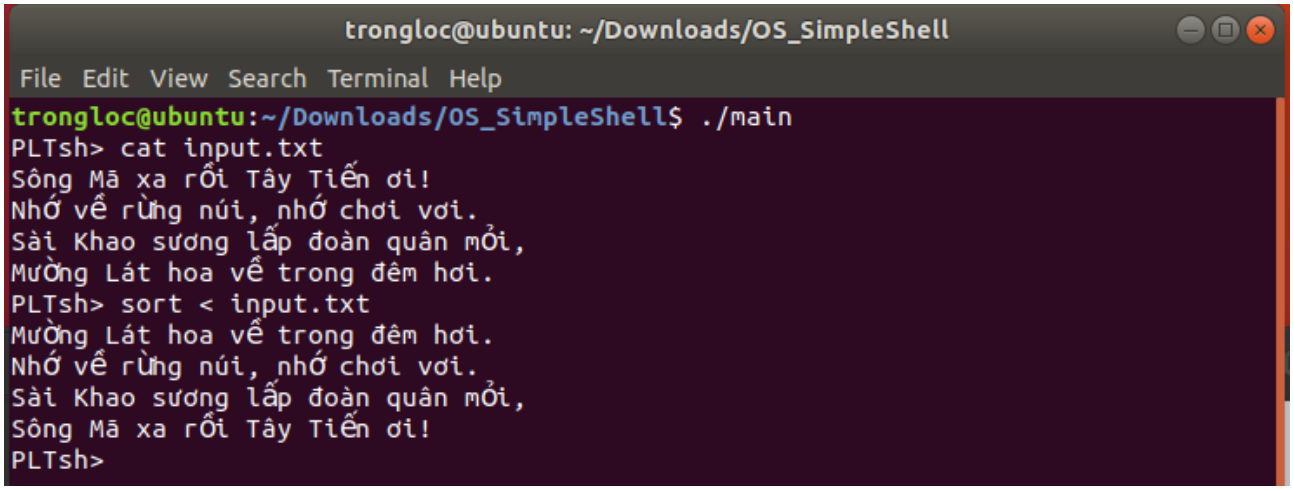
- Đọc nội dung file này và lưu vào file khác:

- Điều hướng đến file và viết nội dung vào file, kết thúc bằng cách nhấn Ctrl + D.

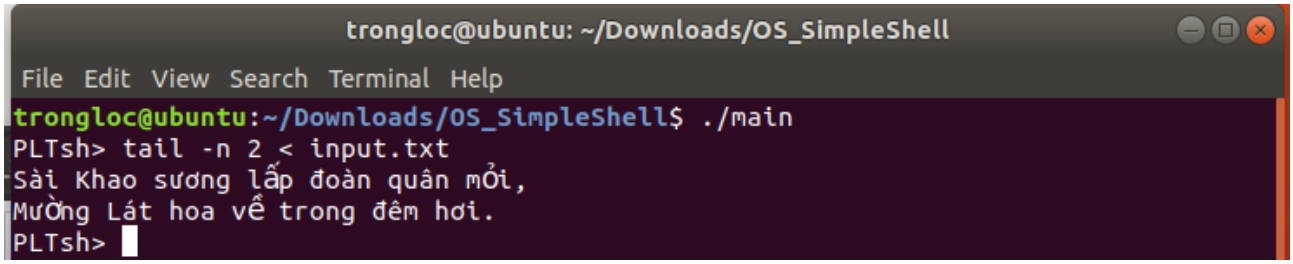


8.5. Điều hướng Input:

- Sắp xếp nội dung file:

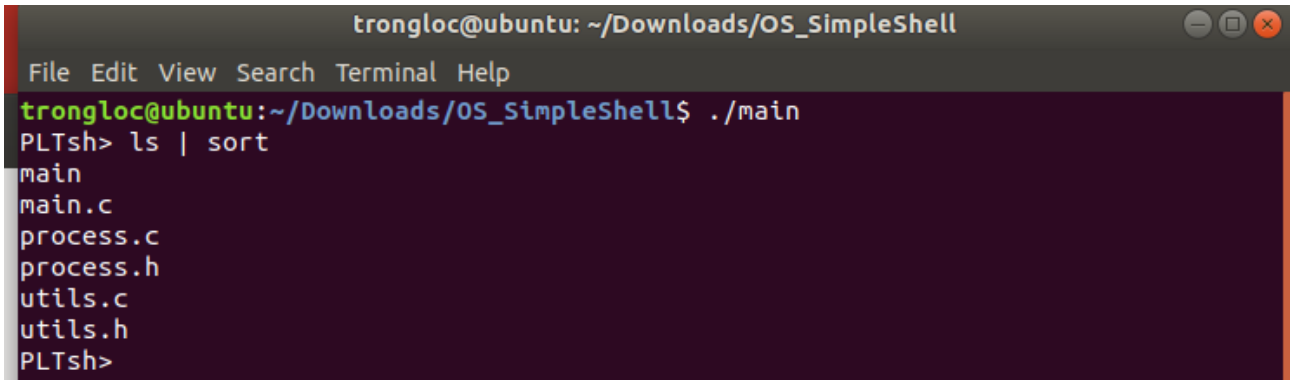


- Lấy 2 dòng cuối từ file *input.txt*:

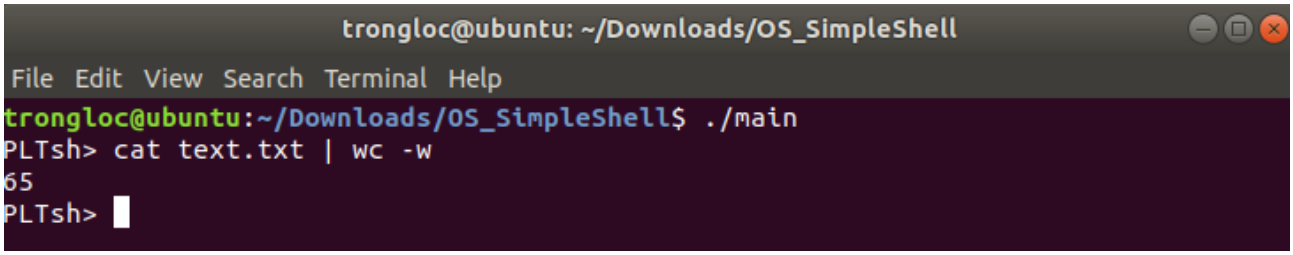


8.6. Giao tiếp giữa hai lệnh thông qua Pipe:

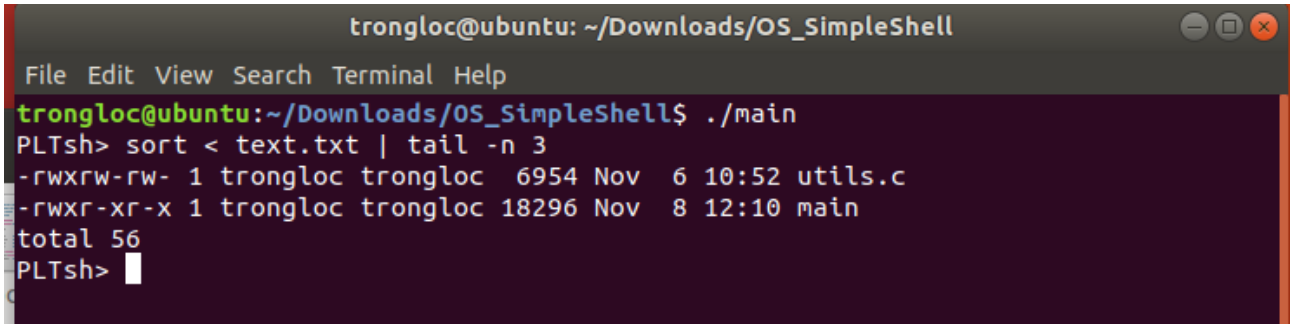
- Ta thực hiện giao tiếp giữa 2 lệnh *ls* và *sort:*



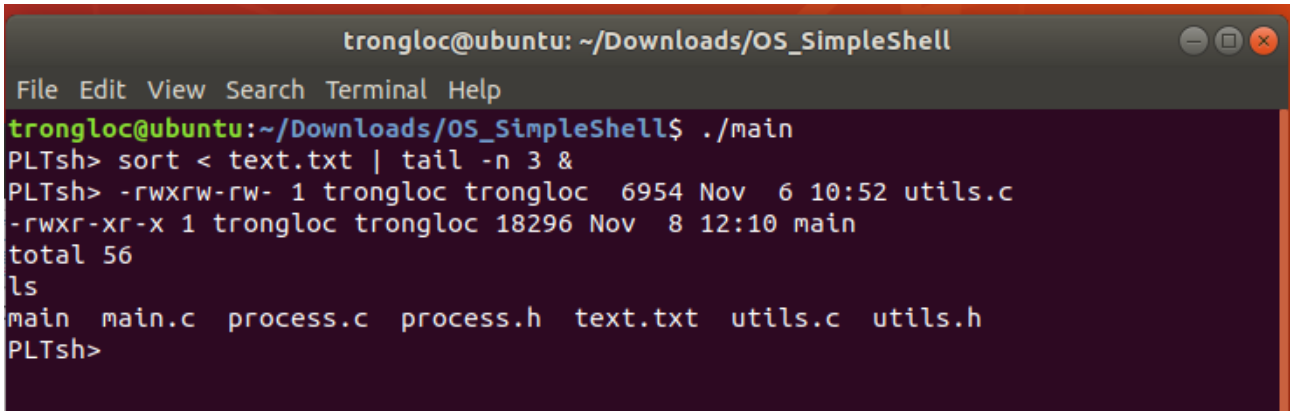
- Ta thực hiện giao tiếp giữa 2 lệnh *cat* và *wc -w* (Đếm số từ trong file):



- Ta cũng có thể kết hợp *redirection* và *pipe*:

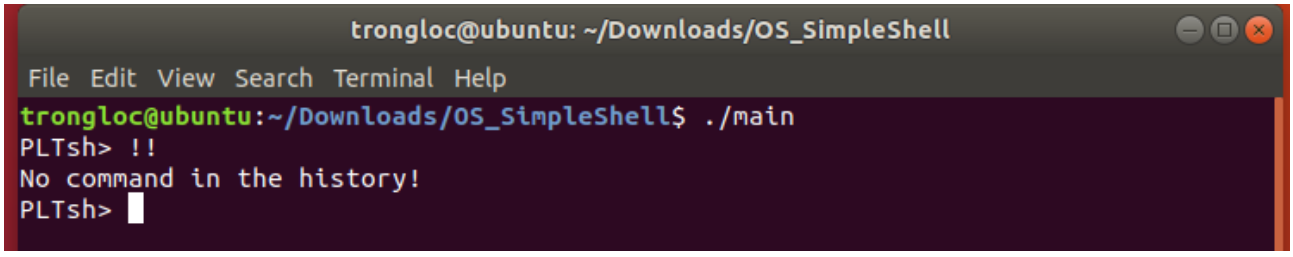


- Ta cũng có thể kết hợp *redirection*, *pipe*, và *chạy dưới nền*:

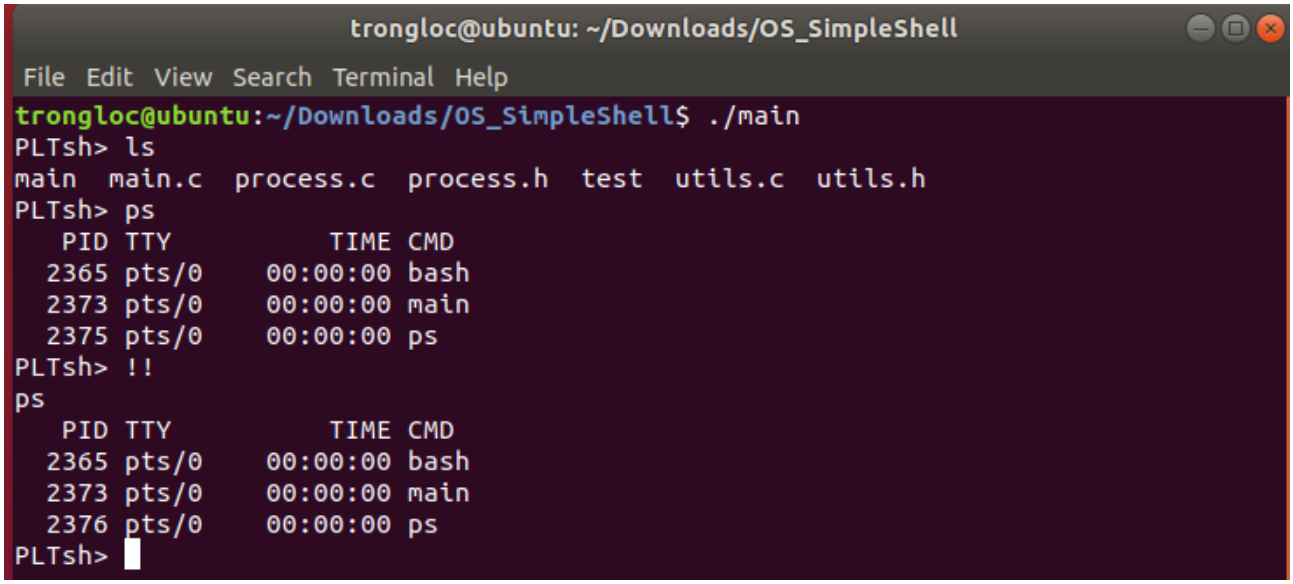


8.7. Lịch sử lệnh:

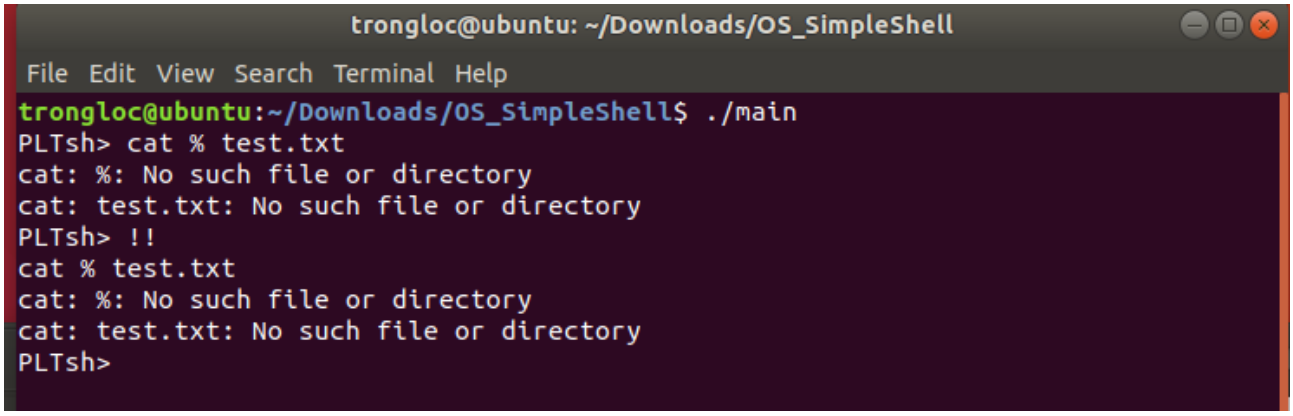
- Khi chưa có lệnh nào được thực hiện trước khi xem lịch sử, kết quả sẽ hiện "No command in the history!"



- Khi đã thực hiện nhiều lệnh, ta xem lịch sử, kết quả sẽ cho ra lệnh gần đây nhất:



- Với các lệnh sai cú pháp, lịch sử vẫn hiện các lệnh đó:



9. ĐÁNH GIÁ MỨC ĐỘ HOÀN THÀNH:

9.1. Về yêu cầu đồ án:

- Mức độ hoàn thành yêu cầu đồ án: **100%**

- Mức độ hoàn thành mục tiêu của nhóm: **100%**

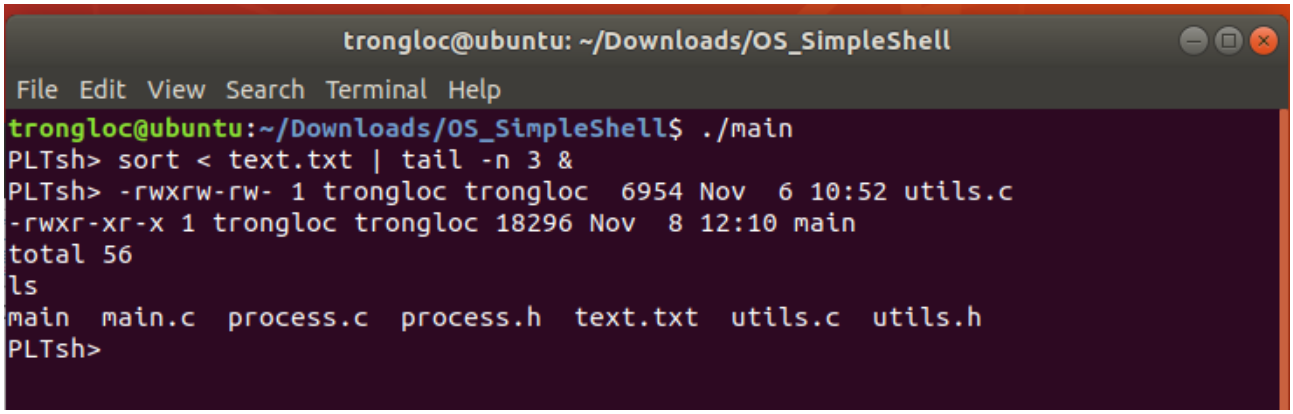
- Cụ thể:

Công việc	Mức độ hoàn thành
Phân tích cú pháp lệnh	100%
Các lệnh thực thi cơ bản	100%
Lệnh built-in	100%

Thực thi dưới nền (&)	100%
Điều hướng Output	100%
Điều hướng Input	100%
Giao tiếp giữa hai lệnh thông qua Pipe	100%
Lịch sử lệnh	100%

9.2. Điểm phát triển đồ án từ nhóm:

Không dừng lại ở yêu cầu đồ án, nhóm còn phát triển thêm tính năng mà Terminal trên Ubuntu thật cung cấp. Đó là việc kết hợp các nhóm tính năng chạy dưới nền, điều hướng input/output và pipe. Điểm phát triển này, được thể hiện rõ trong ví dụ đã được nêu ở mục 8.6, khi kết hợp cả 3 tính năng đã nêu:



TÀI LIỆU THAM KHẢO

[1] The Bash Parser, https://mywiki.wooledge.org/BashParser

[2] The Bourne-Again Shell, http://aosabook.org/en/bash.html

[3] https://stuff.lhunath.com/parser.png

[4] The Linux Information Project, http://www.linfo.org/

[5] The Linux Documentation Project, https://tldp.org/

[6] Linux fork() Introduction, https://youtu.be/9seb8hddeK4

[7] Linux Exec System Call, https://youtu.be/mj2VjcOXXs4