ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Tài liệu hướng dẫn thực hành

HỆ ĐIỀU HÀNH

Biên soạn: ThS Phan Đình Duy

ThS Nguyễn Thanh Thiện

KS Trần Đại Dương

ThS Trần Hoàng Lộc

KS Thân Thế Tùng

MỤC LỤC

NỘI QUY THỰC HÀNH

1. Sinh viên tham dự đầy đủ các buổi thực hành theo quy định của giảng viên hướng dẫn (GVHD) (6 buổi với lớp thực hành cách tuần hoặc 10 buổi với lớp thực hành liên tục).
2. Sinh viên phải chuẩn bị các nội dung trong phần "Sinh viên viên chuẩn bị" trước khi đến lớp. GVHD sẽ kiểm tra bài chuẩn bị của sinh viên trong 15 phút đầu của buổi học (nếu không có bài chuẩn bị thì sinh viên bị tính vắng buổi thực hành đó).
3. Sinh viên làm các bài tập ôn tập để được cộng điểm thực hành, bài tập ôn tập sẽ được GVHD kiểm tra khi sinh viên có yêu cầu trong buổi học liền sau bài thực hành đó. Điểm cộng tối đa không quá 2 điểm cho mỗi bài thực hành.

Bài 3. TIẾN TRÌNH

3.1 Mục tiêu

- Sinh viên làm quen với lập trình trên Hệ điều hành Ubuntu thông qua trình soạn thảo vim, trình biên dịch gcc và trình gỡ lỗi gdb
- Thực hành với tiến trình
- Hiện thực giao tiếp liên tiến trình với mô hình bộ nhớ chia sẻ
- Giới thiệu về cơ chế truyền thông: signal

3.2 Nội dung thực hành

- Sử dụng trình biên dịch gcc để biên dịch từ tệp tin mã nguồn sang tệp tin có thể thực thi
- Sử dụng trình gỡ lỗi gdb để gỡ lỗi chương trình
- Tạo, dừng tiến trình
- Giao tiếp liên tiến trình với mô hình bộ nhớ chia sẻ
- Truyền thông giữa các tiến trình bằng cơ chế signal

.

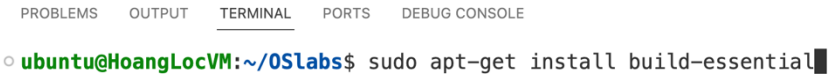
3.3 Sinh viên chuẩn bị

3.3.1 Cài đặt trình biên dịch gcc

Trình biên dịch (compiler) là phần mềm biên dịch từ mã nguồn (chương trình được viết bằng ngôn ngữ lập trình cấp cao) thành các chuỗi bit (chương trình có thể thực thi được trên máy tính). Trong bài thực hành này, gcc (GNU C Compiler) được sử dụng làm công cụ thực hành. Sinh viên có thể cài đặt gcc và các gói liên quan theo các bước sau:

- Đăng nhập hoặc kết nối SSH vào máy ảo, thực thi câu lệnh:

```
sudo apt-get update
sudo apt-get install build-essential
```



Hình 1. Dùng VSCode ssh vào máy ảo và cài đặt build-essential

Sau khi cài đặt, gcc sẽ thường được đặt tại /user/bin/gcc (sử dụng which gcc để biết chính xác vị trí gcc trong mỗi máy tính).

.

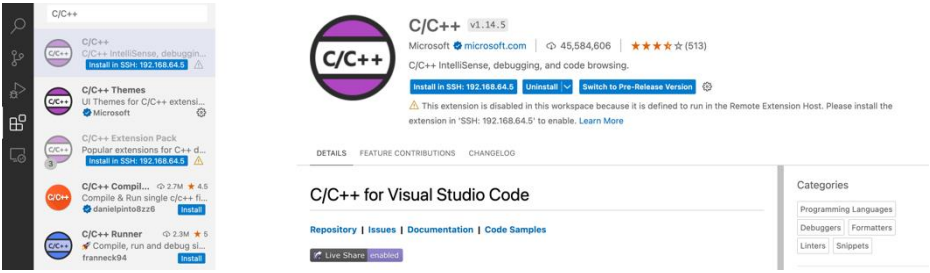
```
PROBLEMS  OUTPUT  TERMINAL  PORTS  DEBUG CONSOLE

ubuntu@HoangLocVM:~/OSlabs$ which gcc
/usr/bin/gcc
ubuntu@HoangLocVM:~/OSlabs$ gcc --version
gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

ubuntu@HoangLocVM:~/OSlabs$
```

Hình 2. Dùng lệnh which gcc để kiểm tra nơi cài đặt và lệnh gcc --version để chắc chắn gcc đã được cài đặt thành công

Để lập trình C một cách dễ dàng hơn, trên VSCode, ta có thể cài đặt extension C/C++ giúp hỗ trợ các thao tác biên dịch, debug và nhắc code. Trên VSCode, chọn tab **Extensions**, gõ tìm **C/C++**, sau đó bấm **Install in SSH:...** để cài đặt extension trên máy ảo.

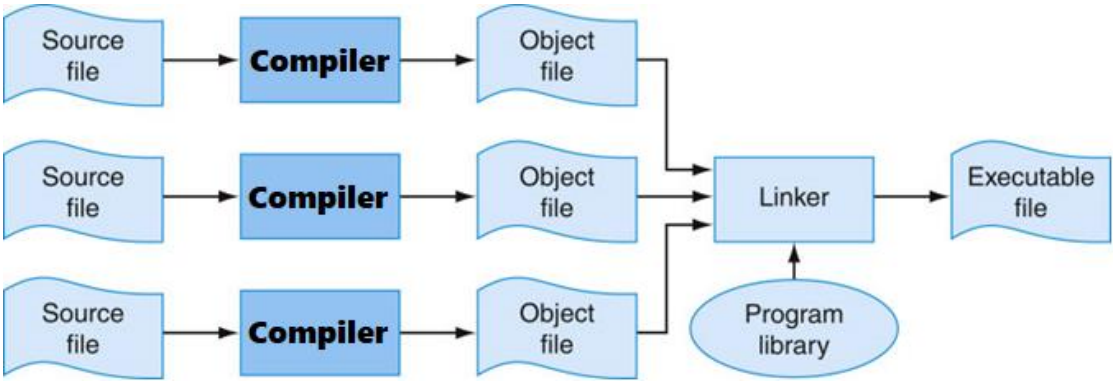


Hình 3. Tìm extension C/C++ trên VSCode

3.3.2 Quá trình biên dịch

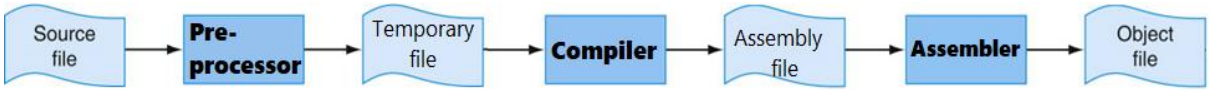
Hình 4 trình bày quá trình biên dịch một chương trình từ mã nguồn thành chương trình có thể thực thi được trên máy tính.

.



Hình 4. Quá trình biên dịch

Quá trình từ tệp mã nguồn tới tệp đối tượng có thể trình bày chi tiết hơn trong Hình 5.



Hình 5. Quá trình biên dịch chi tiết

Bộ tiền xử lý (Preprocessor) sẽ lấy bản sao của tệp mã nguồn (*.c) nằm tại thư mục của project và sao chép nội dung các tệp tiêu đề (*.h) nằm tại /usr/include vào bản sao ấy, bản sao này trở thành một chương trình hoàn chỉnh ở mức ngôn ngữ lập trình cấp cao. Tiếp theo, trình biên dịch sẽ biên dịch bản sao này thành một chương trình ở mức hợp ngữ (*.asm) theo kiến trúc tập lệnh mà máy tính đang sử dụng. Sau đó, trình biên dịch hợp ngữ (Assembler) sẽ thông dịch chương trình ở mức hợp ngữ này thành các tệp đối tượng (*.o). Cuối cùng, trình liên kết (Linker) sẽ liên kết các tệp đối tượng và các thư viện (*.a, *.so, *.sa) nằm tại /usr/lib hoặc /lib để tạo thành một chương trình ở mức nhị phân có thể thực thi được trên máy tính:

- Các tệp mã nguồn (*.c) là các tệp tin do người viết chương trình viết nhằm phục vụ mục đích chuyên biệt nào đó và thường được cập nhật trong quá trình phát triển phần mềm.
- Các tệp tiêu đề (*.h) là các tệp tin dùng để định nghĩa hàm và các khai báo cần thiết cho quá trình biên dịch. Dựa vào những thông tin này, trình biên dịch sẽ đưa ra cảnh báo hoặc lỗi cú pháp, kiểu dữ liệu, hoặc tạo ra các tệp đối tượng (*.o).

- Các tệp đối tượng (*.o) là các tệp tin chứa các khối mã nhị phân thuần túy.
- Các tệp tin thư viện (lib*.a, lib*.sa, lib*.so) là các tệp tin thiết yếu để biên dịch chương trình. Trên môi trường Linux có 2 loại thư viện liên kết là thư viện liên kết tĩnh và thư viện liên kết động:
  - ❖ Thư viện liên kết tĩnh là các thư viện khi liên kết trình biên dịch sẽ lấy bản sao toàn bộ mã thực thi của hàm trong thư viện đưa vào chương trình chính. Chương trình sử dụng thư viện này chạy độc lập với thư viện sau khi biên dịch xong. Khi nâng cấp và sửa đổi, muốn tận dụng những chức năng mới của thư viện thì chúng ta phải biên dịch lại chương trình, ngoài ra việc sử dụng thư viện liên kết tĩnh dẫn đến tiêu tốn không gian bộ nhớ và khó nâng cấp chương trình.
  - ❖ Thư viện liên kết động (lib*.so) là các thư viện không được đưa trực tiếp vào chương trình lúc biên dịch và liên kết, trình liên kết (linker) chỉ lưu thông tin tham chiếu đến các hàm trong thư viện liên kết động. Khi chương trình thực thi, Hệ điều hành sẽ

nạp các chương trình liên kết cần tham chiếu vào bộ nhớ, nhờ đó, nhiều chương trình có thể sử dụng chung các hàm trong một thư viện duy nhất.

Soạn chương trình `hello.c` như sau:

```
/*#####################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: hello.c                                   #
#####################################*/

#include <stdio.h>

int main()
{
        printf("Hello, I am <your name>,\n");
        printf("Welcome to IT007!\n");
        return 0;
}
```

Biên dịch và chạy chương trình bằng 2 dòng lệnh sau:

```
$ gcc hello.c -o hello
$ ./hello
```

.

Trong đó gcc là tên lệnh, hello.c là tệp tin đầu vào và hello là tệp tin đầu ra. `./hello` dùng để chạy chương trình. Hãy kiểm chứng kết quả hiển thị trên màn hình là:

```
Hello, I am <your name>,
Welcome to IT007!
```

Nếu muốn quan sát rõ hơn quá trình biên dịch thì có thể sử dụng các lệnh tương ứng sau:

Mục đích	**Lệnh**
Xem tệp tin tạm (sau Preprocessor)	`gcc -E hello.c -o hello_tmp.c`
Xem tệp tin hợp ngữ (sau Compiler)	`gcc -S hello.c`
Xem tệp tin đối tượng (Sau Assembler)	`gcc -c hello.c`
Tạo tệp tin thực thi từ tệp tin đối tượng	`gcc hello.o -o hello`

Bây giờ hãy quay lại với chương trình hello của chúng ta với cách mà các lập trình viên áp dụng trong thực tế bằng cách thêm hàm main.c và chuyển chương trình hello.c ở trên thành một thư viện như sau:

.

```
/*#######################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: hello.h                                   #
#######################################*/

#ifndef __HELLO_H
#define __HELLO_H

void    hello();

#endif
```

```
/*#######################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: hello.c                                   #
#######################################*/

#include <stdio.h>
#include "hello.h"

void hello()
{
      printf("Hello, I am <your name>,\n");
        printf("Welcome to IT007!\n");
```

.

```
}
```

```
/*#####################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: main.c                                    #
#####################################*/

#include "hello.h"

int main()
{
      hello();
      return 0;
}
```

Trong tệp hello.c ở trên, có một sự khác biệt nhỏ tại chỉ thị `#include`, sự khác nhau giữa `#include <filename>` và `#include "filename"` nằm ở khâu tìm kiếm tệp tiêu đề của bộ tiền xử lý trước quá trình biên dịch:

- `#include <filename>`: Bộ tiền xử lý (Pre-processor) sẽ chỉ tìm kiếm tệp tin tiêu đề (.h) trong thư mục chứa tệp tiêu đề của thư viện ngôn ngữ C. Vì thế nếu cần sử dụng thư viện được cung cấp kèm sẵn

ngôn ngữ C thì nên sử dụng #include <filename> để cải thiện tốc độ biên dịch chương trình.

- `#include "filename"`: Trước tiên, bộ tiền xử lý tìm kiếm tệp tin tiêu đề (.h) trong thư mục đặt project. Nếu không tìm thấy, bộ tiền xử lý tìm kiếm tệp tin tiêu đề trong thư mục chứa tệp tiêu đề của thư viện ngôn ngữ C. Vì thế nếu cần sử dụng thư viện tự viết thì phải sử dụng `#include "filename"`.

Biên dịch và chạy chương trình bằng 2 dòng lệnh sau:

```
$ gcc main.c hello.c -o hello
$ ./hello
```

Hãy kiểm chứng lại là kết quả hiển thị trên màn hình vẫn giống như lần trước:

```
Hello, I am <your name>,
Welcome to IT007!
```

3.3.3 Makefile

Nếu chúng ta thường xuyên chèn thêm tệp vào project hay cần thay đổi các tham số biên dịch thì việc biên dịch và chạy chương trình bằng cách sử dụng 2 dòng lệnh trên trở nên dài dòng, dễ xảy ra lỗi và rất khó để sửa lại câu lệnh bằng tay. Để khắc phục tình

.

trạng này, một chương trình có tên là make đã được ra đời để tự động hóa các thao tác có tính lặp đi lặp lại. Một điều may mắn là make đã được cài tự động cùng với gói build-essential khi chúng ta cài đặt gcc.

Mặc định, make sẽ thực thi một tệp tin là Makefile trong thư mục hiện hành gọi make, vì thế hãy sử dụng vim để soạn thảo tệp Makefile nằm trong thư mục project với nội dung sau (chú ý sau khi nhấn ':' xuống dòng thì phải bắt đầu bằng 1 dấu TAB):

```
all: hello run

hello:
	gcc main.c hello.c -o hello

run:
	./hello

clean:
	rm -f hello
```

Bây giờ biên dịch và chạy chương trình chỉ cần sử dụng câu lệnh:

```
$ make all
```

.

Makefile bao gồm các target (những gì muốn thực hiện). Makefile ở trên có 4 target là all, hello, run, và clean. Sau dấu ':' của all có hello và run, nghĩa là all phụ thuộc hello và run, khi thực thi all thì sẽ chuyển thành thực thi hello và run trước sau đó mới quay lại thực thi all. Nếu chạy make không có tham số thì target đầu tiên sẽ được thực thi. Các target trong Makefile thực chất là tên các tệp tin. Nếu chạy make all lại một lần nữa thì target hello sẽ không được thực thi do tệp tin hello đã tồn tại trong hệ thống, target hello sẽ được thực thi lại nếu một trong các phụ thuộc của nó là hello.c và main.c thay đổi, điều này giúp giảm rất nhiều thời gian nếu project có nhiều tệp tin không cần phải thay đổi nội dung trước mỗi lần biên dịch. Cũng chính vì lý do này, nên để biên dịch chính xác, thì trong thư mục hiện tại không được chứa các tệp tin có tên all, run, hay clean. Ngược lại, nếu vẫn muốn tồn tại các tệp tin nói trên thì phải thêm `.PHONY: <files>` vào cuối Makefile để ép buộc make luôn thực thi `<files>`.

```
all: hello run

hello:
	gcc main.c hello.c -o hello

run:
	./hello
```

.

```
clean:
	rm -f hello
.PHONY: hello
```

3.3.4 Trình gỡ lỗi (đọc thêm)

Trình gỡ lỗi là công cụ hỗ trợ tìm kiếm các nguy cơ tiềm tàng hoặc các vấn đề khiến chương trình chạy không như mong muốn. Việc kiểm soát lỗi và nhiều dòng code là một việc khó khăn đối với người viết chương trình, vì thế việc rà soát lỗi để tiến hành điều chỉnh lại cần sự trợ giúp đắc lực từ trình gỡ lỗi. Trong bài thực hành này gdb được sử dụng làm công cụ thực hành.

Sử dụng vim để viết chương trình tính giai thừa factorial.c như sau:

```
/*#######################################
# University of Information Technology          #
# IT007 Operating System                        #
# <Your name>, <your Student ID>                #
# File: factorial.c                             #
#######################################*/

#include <stdio.h>

int main()
```

.

```
{
    int i, num, j;
    printf("Enter the number: ");
    scanf ("%d", &num );

    for (i=1; i<num; i++)
        j=j*i;

    printf("The factorial of %d is %d\n", num, j);
    return 0;
}
```

Biên dịch và chạy chương trình trên. Kết quả của chương trình SAI phải không? Để thu thập thông tin gỡ lỗi một chương trình thì trong khi biên dịch, thêm tùy chọn -g vào trước tệp mã nguồn như sau:

```
$ gcc -g factorial.c -o factorial
```

Chạy gdb để gỡ lỗi chương trình bằng câu lệnh sau:

```
$ gdb factorial
```

Tạo một breakpoint trong chương trình bằng break <dòng cần dừng>, chẳng hạn:

```
(gdb) break 10
```

Nếu project có nhiều tệp thì sử dụng cú pháp: `break <file> <dòng cần dừng>`. Hoặc thậm chí có thể tạo breakpoint ngay tại một hàm bằng cú pháp: `break <func_name>`.

Sau khi đặt breakpoint, trong lúc thực thi chương trình, dgb sẽ dừng tại breakpoint và đưa ra nhắc lệnh để gỡ lỗi. Để tiếp tục thực thi chương trình, chạy câu lệnh sau:

```
(gdb) run
```

Sau khi sử dụng lệnh `run`, `gdb` có thể sẽ thông báo như sau:

```
Breakpoint 1, main () at factorial.c:10
10          j=j*i;
```

Để gỡ lỗi, chúng ta nên kiểm tra giá trị các biến hiện tại để tìm lỗi gây ra bởi biến nào, sử dụng lệnh `print <tên biến>` để xem giá trị các biến:

```
(gdb) print i
$1 = 1
(gdb) print j
$2 = 32767
(gdb) print num
$3 = 2
(gdb)
```

.

Đặc biệt, nếu muốn in giá trị theo mã hexa, có thể sử dụng cú pháp: `print/x <var_name>`.

Như chúng ta thấy, giá trị của biến j là một con số không được mong đợi. Nguyên nhân chính là chúng ta đã không khởi tạo giá trị ban đầu cho j, khiến j nhận một giá trị rác. Để sửa lỗi này, đơn giản chúng ta chỉ cần khởi tạo giá trị j=1 trong chương trình, biên dịch và chạy lại chương trình. Chương trình vẫn SAI!!!

Có 4 loại thao tác phổ biến trong gdb mà chúng ta có thể sử dụng khi chương trình dừng tại breakpoint:

- c hoặc continue: gdb sẽ tiếp tục thực thi cho tới breakpoint tiếp theo
- n hoặc next: gdb sẽ thực thi dòng tiếp theo như là một lệnh duy nhất
- s hoặc step: tương tự như next, nhưng thay vì thực thi dòng tiếp theo như là một lệnh duy nhất thì gdb sẽ xem như vào mã nguồn của một function và thực hiện từng dòng
- l hoặc layout: gdb sẽ hiển thị mã nguồn xung quanh breakpoint

Nếu không chắc chắn về bất kỳ thao tác nào, có thể sử dụng lệnh help <command> để chắc chắn hơn, ví dụ:

```
(gdb) help next
```

Tiếp tục đặt breakpoint tại dòng 10 và tiến hành gỡ lỗi cho đến khi chương trình chạy như mong muốn.

Ngoài việc tạo breakpoint, gdb còn cho phép chúng ta theo dõi một biến bằng cú pháp `watch <var_name>`. Mỗi khi `<var_name>` thay đổi giá trị, chương trình sẽ bị dừng và in ra giá trị cũ và mới của `<var_name>`.

3.4 Hướng dẫn thực hành

3.4.1 Tiến trình

3.4.1.1 Tiến trình trong môi trường Linux

Tiến trình trên môi trường Linux có các trạng thái:

- Đang chạy (running): đây là lúc tiến trình chiếm quyền xử lý CPU dùng tính toán hay thực thi các công việc của mình.
- Chờ (waiting): tiến trình bị Hệ điều hành tước quyền xử lý CPU và chờ đến lượt cấp phát khác.

- Tạm dừng (suspend) hay ngủ (sleep): Hệ điều hành tạm dừng tiến trình. Tiến trình được đưa vào trạng thái ngủ. Khi cần thiết và có nhu cầu, Hệ điều hành sẽ đánh thức (wake up) hay nạp lại mã lệnh của tiến trình vào bộ nhớ. Cấp phát tài nguyên CPU để tiến trình tiếp tục hoạt động.
- Không thể dừng hẳn (zombie): tiến trình đã bị Hệ điều hành chấm dứt nhưng vẫn còn sót lại một số thông tin cần thiết cho tiến trình cha tham khảo.

Trên môi trường Linux, có thể sử dụng lệnh top để xem những tiến trình nào đang hoạt động trên hệ thống. Hình 6 thể hiện kết quả sau khi chạy lệnh top.

```
top - 01:16:45 up  7:54,  1 user,  load average: 0.11, 0.03, 0.04
Tasks: 276 total,   1 running, 209 sleeping,   0 stopped,   0 zombie
%Cpu(s):  4.3 us,  2.9 sy,  0.0 ni, 92.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2017320 total,    81800 free,  1434624 used,   500896 buff/cache
KiB Swap:   969960 total,   889064 free,    80896 used.   397304 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 23386 duongun+  20   0  400820  54548  10420 S   3.6  2.7   1:18.08 Xorg
 24275 duongun+  20   0  807452  41552  27860 S   3.6  2.1   1:21.19 gnome-termina
 32581 duongun+  20   0   51336   4184   3432 R   3.6  0.2   0:00.51 top
 23518 duongun+  20   0 3441468 254024  47852 S   1.5 12.6   3:58.90 gnome-shell
 23550 duongun+  20   0  435300   7756   6304 S   1.5  0.4   0:30.46 ibus-daemon
   633 root      20   0  110508   3156   3052 S   0.7  0.2   0:11.44 irqbalance
   651 root      20   0    4552    656    656 S   0.7  0.0   0:00.97 acpid
     1 root      20   0  225392   8116   5672 S   0.0  0.4   0:14.61 systemd
     2 root      20   0       0      0      0 S   0.0  0.0   0:00.02 kthreadd
     4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:0H
```

Hình 6. Kết quả khi sử dụng lệnh top

Lệnh top cho ta biết khá nhiều thông tin của các tiến trình:

- Dòng thứ nhất cho biết thời gian uptime (từ lúc khởi động) cũng như số người dùng thực tế đang hoạt động.
- Dòng thứ hai là thống kê về số lượng tiến trình, bao gồm tổng số tiến trình (total), số đang hoạt động (running), số đang ngủ (sleeping), số đã dừng (stopped) và số không thể dừng hẳn (zombie).
- Dòng thứ 3-5 lần lượt cho biết thông tin về CPU, RAM và bộ nhớ Swap
- Các dòng còn lại liệt kê chi tiết về các tiến trình như định danh (PID), người dùng thực thi (USER), độ ưu tiên (PR), dòng lệnh thực thi (COMMAND) ...

Một lệnh khác là ps cũng giúp ta liệt kê được chi tiết của tiến trình, tuy nhiên có một vài điểm khác với top:

- Hiện thị ít thông tin hơn lệnh top.
- Nếu top hiển thị thời gian thực các tiến trình thì ps chỉ hiển thị thông tin tại thời điểm khởi chạy lệnh.

Dưới đây là sự mô tả các thông hiển thị bởi lệnh ps -f:

Cột	**Mô tả**

.

UID	ID người sử dụng mà tiến trình này thuộc sở hữu (người chạy nó).
PID	ID của tiến trình.
PPID	ID của tiến trình cha.
C	CPU sử dụng của tiến trình.
STIME	Thời gian bắt đầu tiến trình.
TTY	Kiểu terminal liên kết với tiến trình.
TIME	Thời gian CPU bị sử dụng bởi tiến trình.
CMD	Lệnh bắt đầu tiến trình này.

Những tùy chọn khác có thể được sử dụng song song với lệnh ps:

Tùy chọn	Mô tả
-a	Chỉ thông tin về tất cả người sử dụng.
-x	Chỉ thông tin về các tiến trình mà không có terminal.
-u	Chỉ thông tin thêm vào như chức năng -f.
-e	Hiển thị thông tin được mở rộng.

Có 2 cách để chạy một tiến trình, đó là foreground và background. Theo mặc định, mọi tiến trình mà chúng ta bắt đầu chạy là tiến trình foreground, nó nhận đầu vào từ bàn phím và gửi đầu ra tới màn hình. Khi một chương trình đang chạy trong foreground và cần một khoảng thời gian dài, chúng ta không thể

chạy bất kỳ lệnh nào khác (bắt đầu một tiến trình khác) bởi vì dòng nhắc không có sẵn tới khi chương trình đang chạy kết thúc tiến trình và thoát ra. Ngược lại, tiến trình background chạy mà không được kết nối với bàn phím. Nếu tiến trình background yêu cầu bất cứ đầu vào nào từ bàn phím, nó sẽ đợi cho đến khi được chuyển thành foreground và nhận đầu vào từ bàn phím. Lợi thế của chạy một chương trình trong background là bạn có thể chạy các lệnh khác; bạn không phải đợi tới khi nó kết thúc để bắt đầu một tiến trình mới! Cách đơn giản nhất để bắt đầu một tiến trình background là thêm dấu và (&) tại phần cuối của lệnh.

Thông thường các công việc thực tế cần làm với một hệ thống Linux sẽ không cần quan tâm lắm đến các tiến trình foregorund và background. Tuy nhiên có một vài trường hợp đặc biệt cần sử dụng đến tính năng này:

- Một chương trình cần mất nhiều thời gian để sử dụng, nhưng bạn lại muốn ngay lập tức được chạy một chương trình khác.
- Bạn đang chạy một chương trình nhưng lại muốn tạm dừng nó lại để chạy một chương trình khác rồi quay lại với cái ban đầu.

- Khi bạn xử lý một tệp có dung lượng lớn hoặc biên dịch chương trình, bạn không muốn phải bắt đầu quá trình lại từ đầu sau khi kết thúc nó.

Một số lệnh hữu dụng giúp ta xử lý các trường hợp này là:

- jobs: liệt kê danh sách các công việc đang chạy
- &: với việc sử dụng từ khóa này khi kết thúc câu lệnh, một chương trình có thể bắt đầu trong background thay vì foreground như mặc định.
- fg <job_number>: dùng để đưa một chương trình background trở thành chương trình foreground.
- Ctrl+z: ngược lại với fg, đưa một chương trình foreground trở thành chương trình background.

Mỗi một tiến trình có hai ID được gán cho nó: ID của tiến trình (pid) và ID của tiến trình cha (ppid). Mỗi tiến trình trong hệ thống có một tiến trình cha (ngoại trừ tiến trình init). Hầu hết các lệnh mà chúng ta chạy có Shell như là parent của nó. Kiểm tra ví dụ ps -f mà tại đây lệnh này liệt kê cả ID của tiến trình và ID của tiến trình cha.

Thông thường, khi một tiến trình con bị khử, tiến trình cha được thông báo thông qua tín hiệu SIGCHLD. Sau đó, tiến trình cha có thể thực hiện một vài công việc khác hoặc bắt đầu lại tiến trình con nếu cần thiết. Tuy nhiên, đôi khi tiến trình cha bị khử trước khi tiến trình con của nó bị khử. Trong trường hợp này, tiến trình cha của

tất cả các tiến trình, "tiến trình init" trở thành PPID mới. Đôi khi những tiến trình này được gọi là tiến trình orphan. Khi một tiến trình bị khử, danh sách liệt kê ps có thể vẫn chỉ tiến trình với trạng thái Z. Đây là trạng thái Zombie, hoặc tiến trình không tồn tại. Tiến trình này bị khử và không được sử dụng. Những tiến trình này khác với tiến trình orphan. Nó là những tiến trình mà đã chạy hoàn thành nhưng vẫn có một cổng vào trong bảng tiến trình.

3.4.1.2 Tạo tiến trình

a) Sử dụng hàm fork()

Có thể tạo tiến trình bằng hàm `fork()`, hàm `fork()` tạo một tiến trình mới bằng cách tạo một bản sao của nó. Tiến trình gọi hàm `fork()` được gọi là tiến trình cha, tiến trình mới được tạo ra là tiến trình con. Tiến trình cha quay lại việc thực thi và tiến trình con bắt đầu thực thi tại cùng một nơi (nơi mà `fork()` trả về). Nếu kết quả trả về của hàm `fork()` là 0 thì nghĩa là chúng ta đang trong tiến trình con, nếu kết quả trả về > 0 thì nghĩa là chúng ta đang trong tiến trình cha và kết quả trả về là PID của tiến trình con, nếu kết quả trả về -1 thì nghĩa là hàm `fork()` thất bại. Tiến trình cha và tiến trình con được phân biệt bởi PID của chúng, PID của tiến trình cha sẽ được gán cho PPID của tiến trình con.

Ví dụ 3-1:

```
/*#####################################
# University of Information Technology          #
```

.

```
# IT007 Operating System                                #
# <Your name>, <your Student ID>                        #
# File: test_fork.c                                     #
########################################*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
    __pid_t pid;
    pid = fork();

    if (pid > 0)
    {
        printf("PARENTS | PID = %ld | PPID = %ld\n",
(long)getpid(), (long)getppid());
        if (argc > 2)
            printf("PARENTS | There are %d arguments\n",
argc - 1);
        wait(NULL);
    }

    if (pid == 0)
    {
        printf("CHILDREN | PID = %ld | PPID = %ld\n",
(long)getpid(), (long)getppid());
```

```
        printf("CHILDREN | List of arguments: \n");
        for (int i = 1; i < argc; i++)
        {
            printf("%s\n", argv[i]);
        }
    }

    exit(0);
}
```

Biên dịch và thực thi ví dụ trên với câu lệnh:

```
./test_fork ThamSo1 ThamSo2 ThamSo3
```

b) *Sử dụng họ hàm exec()*

Một cách khác để tạo tiến trình đó là sử dụng họ hàm `exec()`, họ hàm `exec()` được sử dụng để thay thế tiến trình hiện tại bằng cách nạp chương trình được chỉ định tới không gian địa chỉ của nó, sau đó tiến trình gọi hàm `exec()` sẽ tự hủy. Nghĩa là số lượng tiến trình sẽ giữ nguyên, tiến trình mới sẽ thay thế tiến trình cũ tại không gian địa chỉ đã cấp phát và PID không đổi. Họ hàm `exec()` bao gồm: `execl()`, `execlp()`, `execle()`, `exect()`, `execv()`, `execvp()`.

Ví dụ 3-2:

Script bên dưới thực hiện việc đếm biến i từ `1` đến `$1`, kết quả sẽ được ghi vào file text **count.txt**, mỗi lần đếm cách nhau 1 giây:

```
#!/bin/bash

echo "Implementing: $0"
```

.

```
echo "PPID of count.sh: "
ps -ef | grep count.sh

i=1

while [ $i -le $1 ]
do
    echo $i >> count.txt
    i=$((i + 1))
    sleep 1
done
exit 0
```

Chương trình sau sử dụng hàm execl() để thực thi file script vừa tạo ở trên:

```
/*#######################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: test_execl.c                              #
#######################################*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(int argc, char* argv[])
{
    __pid_t pid;
```

.

```
    pid = fork();

    if (pid > 0)
    {
        printf("PARENTS | PID = %ld | PPID = %ld\n",
(long)getpid(), (long)getppid());
        if (argc > 2)
            printf("PARENTS | There are %d arguments\n",
argc - 1);
        wait(NULL);
    }

    if (pid == 0)
    {
        execl("./count.sh", "./count.sh", "10", NULL);

        printf("CHILDREN | PID = %ld | PPID = %ld\n",
(long)getpid(), (long)getppid());
        printf("CHILDREN | List of arguments: \n");
        for (int i = 1; i < argc; i++)
        {
            printf("%s\n", argv[i]);
        }
    }

    exit(0);
}
```

Biên dịch và thực thi ví dụ trên với câu lệnh:

```
./test_execl ThamSo1 ThamSo2 ThamSo3
```

c) *Sử dụng hàm system()*

Nếu muốn tạo mới hoàn toàn một tiến trình, có thể sử dụng hàm `system()`. Chương trình sau thực hiện lại đoạn script trong Ví dụ 3-2 bằng cách sử dụng hàm system():

Ví dụ 3-3:

```
/*#######################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: test_system.c                             #
#######################################*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(int argc, char* argv[])
{
    printf("PARENTS | PID = %ld | PPID = %ld\n",
(long)getpid(), (long)getppid());
    if (argc > 2)
        printf("PARENTS | There are %d arguments\n", argc
- 1);


    system("./count.sh 10");
```

.

```
    printf("PARENTS | List of arguments: \n");
    for (int i = 1; i < argc; i++)
    {
        printf("%s\n", argv[i]);
    }

    exit(0);
}
```

Biên dịch và thực thi ví dụ trên với câu lệnh:

```
./test_system ThamSo1 ThamSo2 ThamSo3
```

3.4.1.3 *Kết thúc tiến trình*

Hàm `exit()` dùng để kết thúc tiến trình và hoàn trả lại tài nguyên. Khi một tiến trình con kết thúc, một tín hiệu sẽ được gửi tới tiến trình cha và nó sẽ được đặt trong một trạng thái zombie đặc biệt dùng để biểu diễn các tiến trình đã kết thúc cho đến khi tiến trình cha gọi hàm `wait()` hoặc `waitpid()`.

```
/*#######################################
# University of Information Technology          #
# IT007 Operating System                        #
# <Your name>, <your Student ID>                #
# File: test_fork_wait.c                        #
#######################################*/

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

.

```
int main(){
    pid_t pid;
    pid = fork();
    if(pid == 0)
        printf("Child process, pid=%d\n",pid);
    else {
        wait(NULL);
        printf("Parent Proces, pid=%d\n",pid);
    }
    exit(0);
}
```

Nếu chương trình có GUI thì có thể kết thúc tiến trình bằng cách click chuột vào dấu X ở góc trên màn hình.

Hoặc có thể gửi tín hiệu kết thúc tiến trình đang chạy từ bàn phím:

- Ctrl+C: gởi tín hiệu INT (SIGINT) đến tiến trình, ngắt ngay tiến trình (interrupt).
- Ctrl+Z: gởi tín hiệu TSTP (SIGTSTP) đến tiến trình, dừng tiến trình (suspend).
- Ctrl+\: gởi tín hiệu ABRT (SIGABRT) đến tiến trình, kết thúc ngay tiến trình (abort).

Phương thức phổ biến khác là sử dụng lệnh kill như sẽ trình bày trong phần 3.4.3. Khi sử dụng lệnh kill với một tiến trình con thì chỉ tiến trình đó được tắt, nhưng nếu sử dụng kill với tiến trình cha thì

toàn bộ con của nó cũng được tắt theo. Lý do là vì tiến trình con muốn hoạt động được thì phải có 1 tiến trình khác yêu cầu Hệ điều hành tạo ra nó. Một tiến trình không thể tự nhiên sinh ra nếu không có một yêu cầu khởi tạo tiến trình được đáp ứng bởi Hệ điều hành. Hoặc trong lập trình, có thể sử dụng hàm return() để kết thúc một tiến trình.

3.4.2 Giao tiếp liên tiến trình (IPC - Interprocess Communication)

Các tiến trình thực thi đồng thời trong hệ thống có thể thuộc loại độc lập hoặc cộng tác. Một tiến trình độc lập là tiến trình không chia sẻ bất cứ dữ liệu nào với các tiến trình khác. Mặt khác, nếu tiến trình tác động hoặc bị tác động bởi tiến trình khác thì ta nói tiến trình đó là tiến trình cộng tác. Và rõ ràng, các tiến trình chia sẻ dữ liệu với nhau là các tiến trình cộng tác.

Các tiến trình cộng tác sẽ cần phải có cơ chế giao tiếp liên tiến trình để có thể trao đổi dữ liệu với nhau. Hai mô hình giao tiếp liên tiến trình cơ bản là:

- **Bộ nhớ được chia sẻ (shared memory):** một vùng nhớ được chia sẻ giữa các tiến trình. Các tiến trình có thể trao đổi dữ liệu bằng cách đọc và ghi dữ liệu vào vùng nhớ này. Cơ chế này có tốc độ thực thi nhanh do chỉ cần sử dụng system call để tạo vùng nhớ chia sẻ.

- **Truyền thông điệp (message passing):** các thông điệp được trao đổi giữa các tiến trình. Cơ chế này phù hợp với các dữ liệu nhỏ và cũng dễ triển khai trên các hệ thống phân tán. Tuy nhiên, cơ chế này lại tốn thời gian hơn bộ nhớ được chia sẻ do mỗi lần trao đổi thông điệp (gửi – nhận) luôn cần có sự hỗ trợ của system call.

Trong phần này, chúng ta sẽ đi tìm hiểu cách triển khai mô hình bộ nhớ chia sẻ trong Linux.

3.4.2.1 Cách cài đặt bộ nhớ chia sẻ

Việc cài đặt bộ nhớ chia sẻ được thực hiện qua 3 bước:

Bước 1: Khởi tạo vùng nhớ được chia sẻ với system call `shm_open()`

```
fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- `name`: định danh của vùng nhớ được chia sẻ
- `O_CREATE | O_RDRW`: yêu cầu tạo đối tượng vùng nhớ chia sẻ nếu chưa tồn tại và mở với chế độ đọc và ghi
- `0666`: chế độ trên vùng nhớ chia sẻ

Bước 2: Cài đặt độ lớn của vùng nhớ chia sẻ với hàm ftruncate()

```
ftruncate(fd, 4096);
```

- `fd`: đối tượng vùng nhớ chia sẻ
- `4096`: độ lớn của vùng nhớ

Bước 3: Khởi tạo file ánh xạ bộ nhớ có chứa đối tượng vùng nhớ chia sẻ với hàm `mmap()`

```
mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

- `0`: địa chỉ vùng nhớ bắt đầu ánh xạ
- `SIZE`: kích thước vùng nhớ (ví dụ trong trường hợp trên là 4096)
- `PROT_READ | PROT_WRITE`: chế độ bảo về cho phép đọc và ghi
- `MAP_SHARED`: chế độ chia sẻ những thay đổi trên bộ nhớ với các tiến trình khác
- `fd`: đối tượng bộ nhớ chia sẻ
- `0`: Offset (độ dời) trên file, chỉ vị trí nhớ bắt đầu ánh xạ

Bước 4: Thu hồi bộ nhớ

```
munmap(ptr, SIZE);
close(fd);
shm_unlink(name);
```

3.4.2.2 Ví dụ về bộ nhớ chia sẻ

Cho 2 tiến trình Process A và Process B thực hiện như sau:

- Process A khởi tạo bộ nhớ chia sẻ, ghi vào bộ nhớ "Hello Process B", sau đó chờ bộ nhớ được cập nhật bởi Process B.
- Proces B truy cập bộ nhớ chia sẻ, đọc dữ liệu do Process A ghi vào, sau đó cập nhật bộ nhớ với chuỗi "Hello Process A"

Ví dụ 3-4:

a) Process A:

```
/*#######################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: test_shm_A.c                              #
#######################################*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <unistd.h>

#include <sys/mman.h>

int main()
{
```

.

```
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory obect */
    char *ptr;
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR,0666);
    /* configure the size of the shared memory object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    /* write to the shared memory object */
    strcpy(ptr, "Hello Process B");
    /* wait until Process B updates the shared memory
segment */
    while (strncmp(ptr, "Hello Process B", 15) == 0)
    {
        printf("Waiting Process B update shared
memory\n");
        sleep(1);
    }
    printf("Memory updated: %s\n", (char *)ptr);
    /* unmap the shared memory segment and close the
file descriptor */
    munmap(ptr, SIZE);
    close(fd);
```

```
    return 0;
}
```

b) *Process B:*

```
/*#######################################
# University of Information Technology            #
# IT007 Operating System                          #
# <Your name>, <your Student ID>                  #
# File: test_shm_B.c                              #
#######################################*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <unistd.h>

#include <sys/mman.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory obect */
    char *ptr;
```

.

```
    /* create the shared memory object */
    fd = shm_open(name, O_RDWR,0666);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    /* read from the shared memory object */
    printf("Read shared memory: ");
    printf("%s\n",(char *)ptr);
    /* update the shared memory object */
    strcpy(ptr, "Hello Process A");
    printf("Shared memory updated: %s\n", ptr);
    sleep(5);
    // unmap the shared memory segment and close the
file descriptor
    munmap(ptr, SIZE);
    close(fd);
    // remove the shared memory segment
    shm_unlink(name);
    return 0;
}
```

3.4.3 Signal (Truyền thông giữa các tiến trình)

Tín hiệu là các thông điệp khác nhau được gởi đến tiến trình nhằm thông báo cho tiến trình một tình huống. Mỗi tín hiệu có thể kết hợp hoặc có sẵn bộ xử lý tín hiệu (signal handler). Tín hiệu sẽ ngắt ngang quá trình xử lý của tiến trình, bắt hệ thống chuyển sang gọi bộ xử lý tín hiệu ngay tức khắc. Khi kết thúc xử lý tín hiệu, tiến trình lại tiếp tục thực thi.

.

Bảng dưới đây liệt kê các signal phổ biến có thể bắt gặp hoặc cần sử dụng nó trong các chương trình:

Tên signal	**Số hiệu signal**	**Mô tả**
SIGHUP	1	Trì hoãn việc kiểm tra trên quản lý terminal hoặc sự dừng của quản lý tiến trình.
SIGINT	2	Được thông báo nếu người sử dụng gửi một tín hiệu ngắt (Ctrl+C).
SIGQUIT	3	Được thông báo nếu người sử dụng gửi một tín hiệu bỏ (Ctrl+D).
SIGFPE	8	Được thông báo nếu một hoạt động thuộc về toán không hợp pháp được thử chạy.
SIGKILL	9	Nếu một tiến trình nhận signal này, nó phải thoát ra ngay lập tức và sẽ không thực hiện các hoạt động làm sạch.
SIGALRM	14	Tín hiệu báo số lần thực hiện (Alarm Clock).
SIGTERM	15	Tín hiệu kết thúc phần mềm (được gửi bởi sigkill theo mặc định).

Để xem toàn bộ các loại signal trên hệ thống, chúng ta có thể sử dụng lệnh:

```
$ kill -l
```

Mỗi signal có một hoạt động mặc định liên kết với nó. Hoạt động mặc định với một signal là hoạt động mà một script hoặc một chương trình thực hiện khi nó nhận được một signal. Một trong số các hoạt động mặc định có thể là:

- Kết thúc tiến trình
- Bỏ qua signal
- Kết xuất lõi nhớ. Nó tạo một tệp gọi là core (lõi) chứa hình ảnh bộ nhớ của tiến trình khi nó nhận được signal
- Dừng tiến trình
- Tiếp tục tiến trình bị dừng

Có một vài phương thức trong việc gửi các signal tới một tiến trình. Một trong những phương thức phổ biến nhất là cho người sử dụng nhập từ bàn phím:

- Ctrl+C: gởi tín hiệu INT (SIGINT) đến tiến trình, ngắt ngay tiến trình (interrupt).
- Ctrl+Z: gởi tín hiệu TSTP (SIGTSTP) đến tiến trình, tạm dừng tiến trình (suspend).
- Ctrl+\: gởi tín hiệu ABRT (SIGABRT) đến tiến trình, kết thúc ngay tiến trình (abort).

.

Phương thức phổ biến khác để gửi signal là sử dụng lệnh kill mà có cú pháp như sau:

```
$ kill -signal pid
```

Ở đây, signal là hoặc số hoặc tên của signal để gửi và pid là ID của tiến trình mà signal nên được gửi tới. Ví dụ:

```
$ kill -1 1001
```

Có 3 cách để bắt một tín hiệu trong lập trình. Nó có thể bị từ chối (không phải tất cả tín hiệu đều bị từ chối), nó có thể được gửi tới một handler mặc định hoặc gửi tới một handler định trước. Để có thể bắt tín hiệu định trước, ta có thể dùng hàm signal (trong thư viện signal.h) để lấy số hiệu và địa chỉ của một hàm để bắt tín hiệu. Cú pháp của hàm signal() như sau:

```
signal(int signum, sighandler handler)
```

Trong đó signum mà loại signal, handler là trình trình xử lý khi signum xảy ra.

```
/*#######################################
# University of Information Technology          #
# IT007 Operating System                        #
# <Your name>, <your Student ID>                #
# File: example_signal.c                        #
#######################################*/
```

.

```
#include <stdio.h>
#include <signal.h>

int loop_forever = 1;

void on_sigint(){
    printf("\nCRT+C is pressed!\n");
    loop_forever = 0;
}

int main(){
  loop_forever = 1;
  signal(SIGINT, on_sigint);
  while(loop_forever){}
  return 0;
}
```

3.5 Bài tập thực hành

1. Thực hiện Ví dụ 3-1, Ví dụ 3-2, Ví dụ 3-3, Ví dụ 3-4 giải thích code và kết quả nhận được?
2. Viết chương trình `time.c` thực hiện đo thời gian thực thi của một lệnh shell. Chương trình sẽ được chạy với cú pháp "`./time <command>`" với `<command>` là lệnh shell muốn đo thời gian thực thi.

Ví dụ:

```
$ ./time ls
time.c
```

```
time
```

```
Thời gian thực thi: 0.25422
```

Gợi ý: Tiến trình cha gọi hàm `fork()` tạo ra tiến trình con rồi `wait()`. Tiến trình con gọi hàm `gettimeofday()` để lấy mốc thời gian trước khi thực thi lệnh shell, sau đó sử dụng hàm `execl()` để thực thi lệnh. Sau khi tiến trình con kết thúc, tiến trình cha tiếp tục gọi hàm `gettimeofday()` một lần nữa để lấy mốc thời gian sau khi thực thi lệnh shell và tính toán.

3. Viết một chương trình làm bốn công việc sau theo thứ tự:
    - In ra dòng chữ: "Welcome to IT007, I am <your_Student_ID>!"
    - Thực thi file script count.sh với số lần đếm là 120
    - Trước khi count.sh đếm đến 120, bấm CTRL+C để dừng tiến trình này
    - Khi người dùng nhấn CTRL+C thì in ra dòng chữ: "count.sh has stopped"
4. Viết chương trình mô phỏng bài toán Producer - Consumer như sau:
    - Sử dụng kỹ thuật shared-memory để tạo một bounded-buffer có độ lớn là 10 bytes.

- Tiến trình cha đóng vai trò là Producer, tạo một số ngẫu nhiên trong khoảng [10, 20] và ghi dữ liệu vào buffer
- Tiến trình con đóng vai trò là Consumer đọc dữ liệu từ buffer, in ra màn hình và tính tổng
- Khi tổng lớn hơn 100 thì cả 2 dừng lại

3.6 Bài tập ôn tập

Phỏng đoán Collatz xem xét chuyện gì sẽ xảy ra nếu ta lấy một số nguyên dương bất kỳ và áp dụng theo thuật toán sau đây:

$$n = \begin{cases} n/2 & \text{nếu n là số chẵn} \\ 3*n+1 & \text{nếu n là số lẻ} \end{cases}$$

Phỏng đoán phát biểu rằng khi thuật toán này được áp dụng liên tục, tất cả số nguyên dương đều sẽ tiến đến 1. Ví dụ, với n = 35, ta sẽ có chuỗi kết quả như sau:

```
35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

Viết chương trình C sử dụng hàm fork() để tạo ra chuỗi này trong tiến trình con. Số bắt đầu sẽ được truyền từ dòng lệnh. Ví dụ lệnh thực thi `./collatz 8` sẽ chạy thuật toán trên `n = 8` và chuỗi kết quả sẽ ra là `8, 4, 2, 1`. Khi thực hiện, tiến trình cha và tiến trình con chia sẻ một buffer, sử dụng phương pháp bộ nhớ chia sẻ, hãy tính toán chuỗi trên tiến trình con, ghi kết quả vào buffer và dùng tiến trình cha để in kết quả ra màn hình. Lưu ý, hãy nhớ thực hiện các thao tác để kiểm tra input là số nguyên dương.

TÀI LIỆU THAM KHẢO

Tiến trình trong Hệ điều hành (Phần 3): https://viblo.asia/p/tien-trinh-trong-he-dieu-hanh-phan-3-3Q75wg6Q5Wb, truy cập ngày 02/01/2019.