

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
KS Trần Hoàng Lộc

MỤC LỤC

BÀI 3.	TIẾN TRÌNH VÀ TIÊU TRÌNH	1
3.1	Mục tiêu.....	1
3.2	Nội dung thực hành	1
3.3	Sinh viên chuẩn bị	2
3.4	Hướng dẫn thực hành	20
3.5	Bài tập ôn tập.....	51

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 VÀ TIỂU 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
- ✚ Thực hành với tiểu trình thông qua thư viện pthread
- ✚ Giới thiệu về cơ chế truyền thông: signal

3.2 Nội dung thực hành

- ✚ Sử dụng trình soạn thảo vim để soạn thảo mã nguồn
- ✚ 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
- ✚ Tạo, dừng, hợp và gỡ tiểu trình với thư viện pthread
- ✚ Truyền thông giữa các tiến trình bằng cơ chế signal

3.3 Sinh viên chuẩn bị

3.3.1 Trình soạn thảo văn bản vim

Trình soạn thảo văn bản thuần túy không sử dụng các định dạng văn bản để trình bày, những văn bản được tạo ra từ trình soạn thảo này rất nhẹ, được đọc dễ dàng mà không cần xử lý nhiều. Trong bài thực hành này, vim (Visual Improved) được sử dụng làm công cụ thực hành. Sinh viên có thể cài đặt vim và các gói liên quan bằng câu lệnh sau:

```
apt-get install vim
```

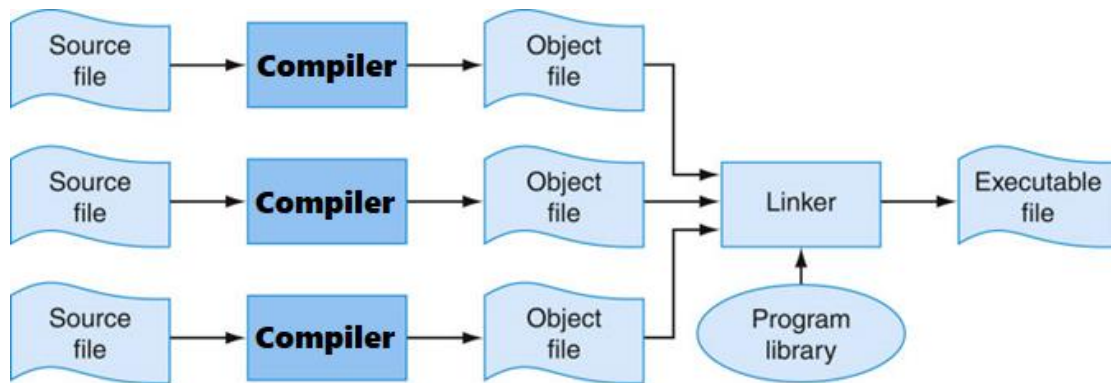
Sau khi cài đặt, vim sẽ thường được đặt tại /user/bin/vim (sử dụng which vim để biết chính xác vị trí gcc trong mỗi máy tính). Một chương trình đi kèm là vimtutor sẽ được tự động cài vào hệ thống, chạy vimtutor và hoàn thành các bài hướng dẫn làm quen với vim.

3.3.2 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 bằng câu lệnh sau:

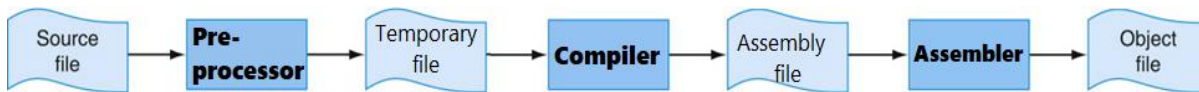
`apt-get install build-essential`

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). Hình 3-1 trình bày quá trình biên dịch một chương trình từ mã nguồn thành chương trình có thể thực thi được trên máy tính.



Hình 3-1 Quá trình biên dịch

Quá trình từ tệp mã nguồn tới tệp đối tượng có thể trình bày chi tiết hơn trong Hình 3-2.



Hình 3-2 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.

Sử dụng vim để 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

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, gdb 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 Khái niệm

Hệ thống sẽ không thực sự quản lý toàn bộ các chương trình nằm trên ổ đĩa, mà chỉ quản lý khi nó được thực thi. Một chương trình để có thể thực thi được trên bất cứ một Hệ điều hành nào thì nó đều phải ở dạng mã máy, mỗi chương trình chứa rất nhiều các đoạn mã máy (các mã nhị phân 0 và 1) giúp cho máy tính có thể biết được chương trình sẽ làm gì. Các đoạn mã này sẽ được nạp vào bộ nhớ khi thực thi, được cấp phát vùng hoạt động, thời gian thực thi... Khi một chương trình đang được thực thi thì ta gọi nó là tiến trình. Mỗi tiến trình có một mã định danh duy nhất (PID) là một biến nguyên 32 bit, Hệ điều hành sẽ cấp phát tài nguyên cho tiến trình được tạo ra và mỗi tiến trình này sở hữu những tài nguyên này cho đến khi Hệ điều hành thu hồi lại hoặc chúng bị kết thúc.

Vì chương trình sẽ được nạp vào bộ nhớ khi thực thi, nghĩa là ta có thể nạp chương trình nhiều lần để thực thi, hay nói cách khác, ta có thể chạy một chương trình n lần cùng một lúc và sẽ có n tiến trình được tạo ra. Một trường hợp phổ biến đó là chương trình MS

office WORD, chỉ có một chương trình WORD được cài trên máy tính, nhưng chúng ta có thể mở chương trình này nhiều lần (tạo nhiều tiến trình) để đọc nhiều tệp tin cùng một lúc.

3.4.1.2 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 3-3 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-terminal
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 3-3: Kết quả sau 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 foreground 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.

3.4.1.3 Tạo tiến trình

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.

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

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

Hoặc có thể tạo tiến trình bằng 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()`, `exec_t()`, `execv()`, `execvp()`.

Chương trình bên dưới là ví dụ cho việc tạo một tệp với tên là foo.c, đồng nghĩa với việc ta gõ lệnh gedit foo.c trong terminal, ta có thể truyền các tham số cho hàm execl:

```
/*#####  
# University of Information Technology      #  
# IT007 Operating System                  #  
# <Your name>, <your Student ID>         #  
# File: example_execl.c                  #  
#####*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
int main(){  
    execl("/usr/bin/gedit", "gedit", "foo.c", NULL);  
    printf("ERROR!!! execl() is failed!\n");  
    exit(1);  
}
```

Nếu muốn tạo mới hoàn toàn một tiến trình, có thể sử dụng hàm system(). Ví dụ bên dưới trình bày cách sử dụng hàm system để tạo một tiến trình mới dùng để mở tệp abc.txt bằng vim editor.

```

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

#include <stdio.h>
#include <stdlib.h>

int main(){
    printf("Hello IT007! I wil open vim editor now ^_^\n");
    system("vi abc.txt");
    return 0;
}

```

3.4.1.4 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: example_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: gửi tín hiệu INT (SIGINT) đến tiến trình, ngắt ngay tiến trình (interrupt).

✚ Ctrl+Z: gửi tín hiệu TSTP (SIGTSTP) đến tiến trình, dừng tiến trình (suspend).

✚ Ctrl+\: gửi 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 Tiểu trình

3.4.2.1 Khái niệm

Tiểu trình là các luồng điều khiển riêng biệt, thường là trong một chương trình (hoặc trong một nhân Hệ điều hành). Nhiều tiểu trình cùng chia sẻ không gian địa chỉ và các tài nguyên khác, nhờ thế chúng có nhiều ưu điểm như:

- ✚ Truyền thông tốc độ cao giữa các tiểu trình
- ✚ Chuyển đổi ngữ cảnh nhanh giữa các tiểu trình
- ✚ Được sử dụng nhiều trong các chương trình yêu cầu xử lý lớn
- ✚ Một chương trình có thể sử dụng nhiều CPU cùng một lúc (lập trình song song)

Nhờ những ưu điểm của nó, tiểu trình ngày nay trở thành mức trừu tượng lập trình hiện đại và phổ biến. Chúng cung cấp đa luồng (đa tiểu trình) cùng thực thi trong một chương trình trong một không gian địa chỉ (chia sẻ bộ nhớ). Chúng cũng có thể chia sẻ việc mở tệp tin và sử dụng chung các tài nguyên khác. Tiểu trình đang trở thành mức lập trình song song chủ yếu trong các hệ thống đa bộ xử lý.

Nhưng chính vì do các tiểu trình cùng chia sẻ tài nguyên nên tiểu trình có một nhược điểm lớn đó là sự tranh chấp tài nguyên

giữa các tiểu trình, đòi hỏi nỗ lực lớn trong việc đồng bộ hóa tiểu trình.

3.4.2.2 Tiểu trình trong Linux

Trong nhân Hệ điều hành Linux, tiểu trình được hiện thực như tiến trình, tiểu trình đơn thuần là tiến trình mà có thể chia sẻ một số tài nguyên nhất định với các tiến trình khác. Đối với một số Hệ điều hành khác, ví dụ như MS Windows, tiểu trình và tiến trình đều là các khái niệm riêng biệt và được hỗ trợ đầy đủ.

Trong bài thực hành này POSIX thread (pthread) sẽ được sử dụng để lập trình tiểu trình. Nó cho phép chúng ta tạo ra các ứng dụng chạy song song theo luồng, phù hợp với các hệ thống đa bộ xử lý. POSIX là viết tắt của Portable Operating Systems Interface là mô tả các API (Application Programming Interface) bao gồm hàm và chức năng của chúng.

Các thao tác của tiểu trình bao gồm: tạo tiểu trình, đồng bộ tiểu trình (hợp – join, khóa – blocking), lập lịch, quản lý dữ liệu và tương tác giữa các tiểu trình.

Mỗi tiểu trình là độc lập với nhau, nghĩa là nó không biết hệ thống có bao nhiêu tiểu trình và nó được sinh ra từ đâu.

Các tiểu trình trong cùng một chương trình sẽ chia sẻ không gian địa chỉ, PC, dữ liệu, tập tin, signal, user ID, group ID. Nhưng chúng cũng có những tài nguyên riêng của chúng, bao gồm: ID của tiểu trình, các thanh ghi, ngăn xếp, signal mask, độ ưu tiên.

3.4.2.3 Tạo tiểu trình

Để tạo tiểu trình, sử dụng hàm `pthread_create()` như bên dưới:

```
int pthread_create(pthread_t * thread, pthread_attr_t * attr, void *
(*start_routine)(void *), void * arg);
```

Trong đó:

✚ `thread` là biến tham chiếu kiểu `pthread_t` được dùng như PID

✚ `attr` là biến tham chiếu kiểu `pthread_attr_t` thể hiện thuộc tính của thread (dùng `NULL` nếu đặt thuộc tính mặc định)

✚ `*start_routine` là một con trỏ hàm kiểu `void` đến chức năng mong muốn tiểu trình thực thi

- `*arg` là con trỏ đối số cho hàm kiểu `void`

Nếu tiểu trình được tạo thành công, hàm `pthread_create()` sẽ trả về số nguyên 0, ngược lại sẽ là một số khác 0.

Dùng vim editor để soạn và dùng gcc với cờ `-pthread` để biên dịch chương trình in ra vô hạn dòng chữ: “Hello, How are you?” và “I’m fine, and you?”

```
/*#####  
# University of Information Technology      #  
# IT007 Operating System                  #  
# <Your name>, <your Student ID>         #
```

```

# File: example_thread_creation.c                                     #
#####*/

#include <pthread.h>
#include <stdio.h>
void *thread_print(void * message) {
    while(1) {
        printf("Hello, How are you?\n");
    }
}

int main() {
    pthread_t idthread;
    pthread_create(
        &idthread,
        NULL,
        &thread_print,
        NULL);

    while(1) {
        printf("I'm fine, and you?\n");
    }
    return 0;
}

```

Trong đó, idthread là tiêu trình sẽ in ra “Hello, How are you?”, main là tiêu trình sẽ in ra “I’m fine, and you?”. Nhấn CRT+C để kết thúc.

3.4.2.4 Dừng tiêu trình

Để dừng một pthread có thể sử dụng hàm pthread_exit(), nếu hàm này được pthread gọi ngoài hàm main() thì nó sẽ dừng pthread gọi hàm này; nếu hàm này được gọi trong main() thì nó sẽ đợi các pthread trong nó dừng rồi nó mới dừng.

Dùng vim editor để soạn và dùng gcc với cờ -pthread để biên dịch chương trình bên dưới:

```
/*#####  
# University of Information Technology      #  
# IT007 Operating System                  #  
# <Your name>, <your Student ID>         #  
# File: example_thread_selfexit.c        #  
#####*/  
  
#include <pthread.h>  
#include <stdio.h>  
#inlucde <stdlib.h>  
#inlcude <unistd.h>
```

```
#define NUM_THREADS 2

void *thread_print(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello IT007! I'm Thread #%ld ^ _^!!\n", tid);
    pthread_exit(NULL);
}

int main()
{
    pthread_t threads[NUM_THREADS];
    int check;
    long tID;

    for(tID = 0; tID < NUM_THREADS; tID++){
        printf("I'm Main Thread: create Thread: #%ld\n", tID);
        check = pthread_create(
            &threads[tID],
            NULL,
            thread_print,
            (void *)tID);
    }
}
```

```

if (check != 0){
    printf("ERROR!!! I'm Main Thread, can't create Thread #%ld
", tID);
    exit(-1);
}
}
sleep(100);
/* Last thing that main() should do */
pthread_exit(NULL);
}

```

Dùng lệnh `top/ps` để kiểm chứng các tiểu trình được tạo mới kết thúc trước khi tiểu trình `main` kết thúc (gợi ý: có thể điều chỉnh chương trình để lấy định danh của thread để tìm kiếm nhanh hơn). Tiếp tục dùng vim editor để soạn và dùng gcc với cờ `-pthread` để biên dịch chương trình bên dưới:

```

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

```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_THREADS 2

void *thread_print(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello IT007! I'm Thread #%ld ^_ ^!!!\n", tid);
    sleep(100);
}

int main()
{
    pthread_t threads[NUM_THREADS];
    int check;
    long tID;

    for(tID = 0; tID < NUM_THREADS; tID++){
        printf("I'm Main Thread: create Thread: #%ld\n", tID);
```

```
check = pthread_create(  
    &threads[tID],  
    NULL,  
    thread_print,  
    (void *)tID);  
  
if (check != 0){  
    printf("ERROR!!! I'm Main Thread, I can't create Thread  
#%ld ", tID);  
    exit(-1);  
}  
}  
  
/* Last thing that main() should do */  
pthread_exit(NULL);  
}
```

Dùng lệnh `top/ps` để kiểm chứng tiểu trình `main` kết thúc nhưng các tiểu trình được tạo mới vẫn chưa kết thúc, sau đó dùng lệnh `kill` để hủy các tiểu trình được tạo.

3.4.2.5 Hợp và gỡ tiểu trình

Để kết hợp các `pthread`, có thể sử dụng hàm `pthread_join(threadid, status)`, `pthread_join()` sẽ ngưng `pthread`

đang gọi tới khi threadid kết thúc. Khi threaded kết thúc, pthread_join() sẽ trả về giá trị 0.

Để tháo gỡ các pthread, có thể sử dụng hàm pthread_detach(threadid).

```
/*#####  
# University of Information Technology      #  
# IT007 Operating System                  #  
# <Your name>, <your Student ID>         #  
# File: example_thread_join.c            #  
#####*/  
  
#include <pthread.h>  
#include <stdio.h>  
#inlucde <stdlib.h>  
#inlcude <unistd.h>  
  
#define NUM_THREADS 2  
  
void *thread_print(void *threadid)  
{  
    long tid;  
    tid = (long)threadid;  
    printf("Hello IT007! I'm Thread #%ld ^ _^!!!\n", tid);
```

```
sleep(100);
}

int main()
{
    pthread_t threads[NUM_THREADS];
    int check;
    long tID;

    for(tID = 0; tID < NUM_THREADS; tID++){
        printf("I'm Main Thread: create Thread: #%ld\n", tID);
        check = pthread_create(
            &threads[tID],
            NULL,
            thread_print,
            (void *)tID);
        if (check != 0){
            printf("ERROR!!! I'm Main Thread, I can't create
Thread #%ld ", tID);
            exit(-1);
        } //end if
        pthread_join(threads[tID], NULL);
    } //end for
}
```

```
/* Last thing that main() should do */  
pthread_exit(NULL);  
}
```

Khi có thêm `pthread_join()`, để có thể thực thi tiếp vòng lặp for thì `threads[tID]` phải kết thúc trước.

3.4.2.6 Truyền dữ liệu cho tiểu trình

Đối số cuối cùng của hàm `pthread_create()` là một con trỏ đối số cho thủ tục mà tiểu trình được tạo ra sẽ thực thi. Trong các ví dụ trước, đối số truyền vào là đơn kiểu dữ liệu, để có thể truyền nhiều đối số với đa dạng kiểu dữ liệu hơn thì chúng ta có thể sử dụng kiểu cấu trúc như bên dưới:

```
/*#####  
# University of Information Technology      #  
# IT007 Operating System                  #  
# <Your name>, <your Student ID>         #  
# File: example_thread_structure.c        #  
#####*/  
#include <pthread.h>  
#include <stdio.h>  
  
#define NUM_THREADS 2
```



```
struct struct_print_parms{
    char character;
    int count;
};

void* char_print (void* args) {
    struct struct_print_parms* p = (struct struct_print_parms*) args;
    int i;
    for (i=0; i < p->count; i++)
        printf ("%c\n", p->character);
    return NULL;
}

int main () {
    pthread_t tid;
    struct struct_print_parms th_args;
    th_args.character = 'X';
    th_args.count = 5;
    pthread_create(&tid, NULL, &char_print, &th_args);
    pthread_join (tid, NULL);
    return 0;
}
```

3.4.3 Signal (Truyền thông giữa các tiến trình)

Tín hiệu là các thông điệp khác nhau được gửi 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 lỗi nhớ. Nó tạo một tệp gọi là core (lỗi) 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: gửi tín hiệu INT (SIGINT) đến tiến trình, ngắt ngay tiến trình (interrupt).

✚ Ctrl+Z: gửi tín hiệu TSTP (SIGTSTP) đến tiến trình, tạm dừng tiến trình (suspend).

✚ Ctrl+\: gửi 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 ôn tập

1. Mối quan hệ cha-con giữa các tiến trình

- a. Vẽ cây quan hệ parent-child của các tiến trình bên dưới:

UID	PID	PPID	COMMAND
88	86	1	WindowServer
501	281	86	iTunes
501	282	86	Terminal
0	287	282	login
501	461	293	firefox-bin
501	531	86	Safari
501	726	86	Mail
501	751	293	Aquamacs
501	293	287	-bash

-
- b. Trình bày cách sử dụng lệnh ps để tìm tiến trình cha của một tiến trình dựa vào PID của nó.
 - c. Tìm hiểu và cài đặt lệnh pstree (nếu chưa được cài đặt), sau đó trình bày cách sử dụng lệnh này để tìm tiến trình cha của một tiến trình dựa vào PID của nó.

2. Chương trình bên dưới in ra kết quả gì? Giải thích tại sao?

```
/*#####  
# University of Information Technology      #  
# IT007 Operating System                  #  
# <Your name>, <your Student ID>         #  
# File: exercise_2.c                      #  
#####*/  
  
#include<stdio.h>  
  
int main(){  
    pid_t pid;  
    int num_coconuts = 17;  
    pid = fork();  
    if(pid == 0) {  
        num_coconuts = 42;  
        exit(0);  
    } else {
```

```
wait(NULL); /*wait until the child terminates */  
}  
printf("I see %d coconuts!\n", num_coconuts);  
exit(0);  
}
```

3. Trong phần thực hành, các ví dụ chỉ sử dụng thuộc tính mặc định của pthread, hãy tìm hiểu POSIX thread và trình bày tất cả các hàm được sử dụng để làm thay đổi thuộc tính của pthread, sau đó viết các chương trình minh họa tác động của các thuộc tính này và chú thích đầy đủ cách sử dụng hàm này trong chương trình. (Gợi ý các hàm liên quan đến thuộc tính của pthread đều bắt đầu bởi: pthread_attr_*)
4. Viết chương trình làm các công việc sau theo thứ tự:
 - a. In ra dòng chữ: “Welcome to IT007, I am <your_Student_ID>!”
 - b. Mở tệp abcd.txt bằng vim editor
 - c. Tắt vim editor khi người dùng nhấn CTRL+C
 - d. Khi người dùng nhấn CTRL+C thì in ra dòng chữ: “You are pressed CTRL+C! Goodbye!”

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.