

MỤC LỤC

LỜI NÓI ĐẦU.....	2
BÀI 1: SỬ DỤNG HỆ ĐIỀU HÀNH MÃ NGUỒN MỞ	3
BÀI 2: COMPILER VỚI GCC VÀ G++	12
BÀI 3: QUẢN LÝ TIẾN TRÌNH.....	20
BÀI 4: QUẢN LÝ TIẾN TRÌNH – TIẾN TRÌNH MỒ CÔI – TIẾN TRÌNH ZOMBIE	25
BÀI 5: QUẢN LÝ TIỂU TRÌNH (LUỒNG)	30
BÀI 6: QUẢN LÝ TIỂU TRÌNH (LUỒNG)	39
BÀI 7: CPU SCHEDULING.....	43
BÀI 8: GIAO TIẾP GIỮA CÁC TIẾN TRÌNH – SỬ DỤNG ĐƯỜNG ỐNG (PIPE)	46
BÀI 9: GIAO TIẾP GIỮA CÁC TIẾN TRÌNH – MESSAGE QUEUE	54
BÀI 10: GIAO TIẾP GIỮA CÁC TIẾN TRÌNH – SHARE MEMORY.....	59
BÀI 11: SEMAPHORE.....	65
BÀI 12: SEMAPHORE (tiếp theo)	71
BÀI 13: MUTEX LOCK.....	74
BÀI 14: BÀI TẬP TỔNG HỢP.....	80
BÀI 15: ÔN TẬP – KIỂM TRA.....	81
TÀI LIỆU THAM KHẢO	82
DANH MỤC HÌNH ẢNH.....	83

LỜI NÓI ĐẦU

Môn học hệ điều hành là môn học căn bản trong ngành công nghệ thông tin, môn học này cung cấp các kiến thức về cách hoạt động của hệ điều hành. Những kiến thức này hầu như khó hiểu nếu chỉ trình bày thuần túy lý thuyết. Do đó, để sinh viên có thể nắm bắt kiến thức rõ ràng hơn, cần có một hệ thống bài thực hành chi tiết, chính xác, để thông qua đó, sinh viên có thể nắm vững kiến thức môn học của mình và vận hành trong các lĩnh vực chuyên môn của mình.

Với mong muốn giúp sinh viên khoa công nghệ thông tin, trường đại học Sài Gòn có được một tài liệu có sự hướng dẫn theo từng chủ đề, có bài tập thực hành sau từng chủ đề để nắm bắt và hiểu vấn đề một cách rõ ràng nhất. Tài liệu thực hành hệ điều hành đã được soạn. Tài liệu được soạn theo từng tuần học, ứng với học kỳ là 15 tuần thì tài liệu thực hành sẽ có 15 bài. Mỗi bài đều có hướng dẫn thực hiện, sau hướng dẫn thực hiện là bài tập thực hành.

Tài liệu sẽ hướng dẫn sinh viên các cách thức cơ bản từ việc sử dụng hệ điều hành Linux cơ bản, sử dụng biên dịch chương trình trên Linux, quản lý tiến trình, quản lý tiểu trình, lập lịch CPU, quản lý giao tiếp giữa các tiến trình, sử dụng bộ nhớ dùng chung. Những kiến thức này tương ứng với các kiến thức trọng tâm của đề cương môn học hệ điều hành, sử dụng cho sinh viên khoa công nghệ thông tin trường đại học Sài Gòn.

Trong quá trình biên soạn, xin chân thành cảm ơn quý thầy cô trong khoa công nghệ thông tin đã đóng góp các ý kiến để tài liệu có thể được biên soạn hoàn chỉnh hơn.

Trong quá trình biên soạn, dù đã rất cố gắng nhưng chắc chắn sẽ khó tránh được những thiếu sót. Rất mong muốn quý thầy cô và các bạn sinh viên khi sử dụng tài liệu có thể phản hồi để tác giả có thể chỉnh sửa cho tài liệu ngày một tốt hơn và dễ hiểu hơn với các bạn sinh viên.

Xin chân thành cảm ơn!

BÀI 1: SỬ DỤNG HỆ ĐIỀU HÀNH MÃ NGUỒN MỞ

I. Mục tiêu:

Hướng dẫn sinh viên làm quen với các câu lệnh của hệ điều hành mã nguồn mở (Linux), làm quen với phần mềm ảo hóa VMWARE. Giúp sinh viên chuẩn bị cho các bài tập ở các tuần sau.

II. Hướng dẫn thực hiện:

Sinh viên sử dụng hệ điều hành Ubuntu để thực hiện.

Sinh viên tự tạo máy ảo Ubuntu trên VMWare để thực hành.

Sinh viên download bảng hướng dẫn một số câu lệnh trên Linux để tiện thực hành.

<https://drive.google.com/file/d/1TVeNkPxCGg96oqdOwqpgS6Q8M08i7OL2/view?usp=sharing>

Hệ thống Linux phân biệt chữ hoa chữ thường.

Một lệnh cơ bản của Linux có định dạng chung như sau:

<command_name> <options> <arguments>

Ví dụ:

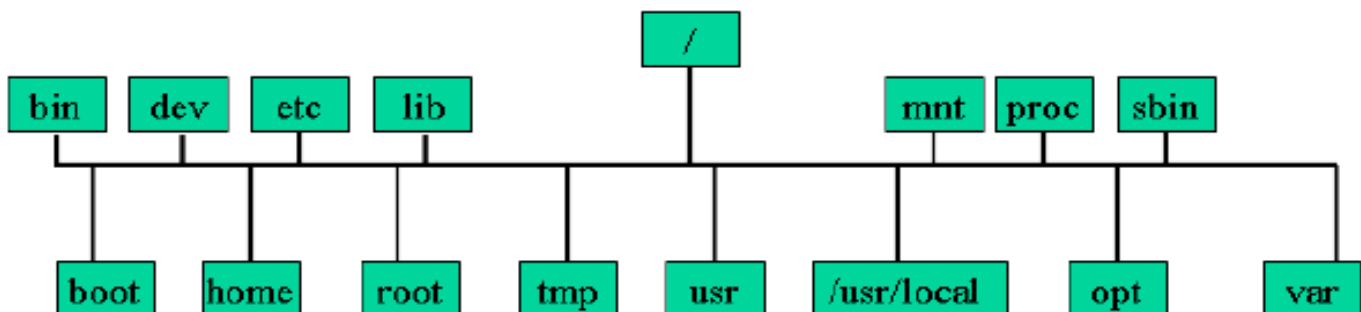
\$ ls -l /etc

\$ cd /tmp không có tùy chọn

\$ whoami không có tùy chọn và đối số

Hệ thống tập tin dạng cây:

- Nút: thư mục (directory, folder) .
- Lá: tập tin (file).
- Điểm bắt đầu: gốc (root), kí hiệu /



Hình 1. 1 Sơ đồ minh họa hệ thống tập tin trên Linux

Đường dẫn: Dùng để chỉ định một phần tử (nút) trên cây thư mục: thư mục hoặc tập tin.

Đường dẫn tuyệt đối: bắt đầu từ gốc (/) đi qua các nút trung gian và sau cùng đến phân tử quan tâm (sử dụng dấu / để phân cách các nút trên đường dẫn).

Ví dụ: /home/vunguyen/os/lab1.ppt

Đường dẫn tương đối: bắt đầu từ thư mục hiện hành đi đến phân tử quan tâm (có thể đi ngược lên thư mục cha).

Ví dụ: giả sử thư mục hiện hành là /home/vunguyen, ta có thể sử dụng đường dẫn tương đối: os/lab1.ppt.

Một số thư mục chuẩn:

- /bin, /sbin: chứa các lệnh cần thiết cho hệ thống.
- /dev: tập tin thiết bị hoặc các file đặc biệt.
- /etc: chứa các file cấu hình của Linux.
- /lib: kernel modules, thư viện chia sẻ cho các lệnh nằm trong /bin, /sbin.
- /mnt, /media: (mount point) dùng để ánh xạ các phân vùng đĩa.
- /proc: những thông số của kernel.
- /boot: Linux kernel, system map cho bước 2 của bootloader.
- /home: thư mục người dùng.
- /root: thư mục của root (admin, người quản trị).
- /tmp: thư mục tạm.
- /usr: tài nguyên (tĩnh, chia sẻ) cho người dùng.
- /usr/local, /opt: phần mềm, thư viện chia sẻ.
- /var: dữ liệu thay đổi, thư mục spool (máy in), tập tin nhật ký (logs), thư mục chia sẻ và không chia sẻ.

Một số câu lệnh trên Linux:

Lệnh	Mô tả
pwd	Xem thư mục hiện hành
file	Xem kiểu file
ls	Liệt kê file và thư mục
cd	Chuyển thư mục làm việc
mkdir	Tạo thư mục

rmmdir	Xóa thư mục
rm	Xóa file hay thư mục
cp	Copy file hay thư mục
chmod	Đổi quyền trên file hay thư mục
Lệnh	Mô tả
cat	Tạo và xem nội dung file
touch	Tạo file rỗng
more	Xem nội dung file trên 1 màn hình
head	Xem n dòng đầu tiên của file
tail	Xem n dòng cuối cùng của file
find	Tìm một file trong hệ thống cây thư mục
grep	Tìm cụm từ trong file
which	Tìm đường dẫn chứa file thực thi lệnh

Ví dụ về câu lệnh ls:

Lệnh: ls [option] path_name

- i: liệt kê inode.
- h: in ra kích thước dễ đọc.
- l: liệt kê mỗi mục trên một dòng.
- n: liệt kê cả UID và GID.

- p: hiển thị cả các ký hiệu mô tả (/ , = , @)
- R: recursive để liệt kê cả những thư mục con.
- S: sắp xếp kết quả theo kích thước.
- t (-c): sắp xếp kết quả theo thời gian cập nhật.
- u: hiển thị thời gian của lần truy cập sau cùng.

Xem thêm các options khác: man ls

\$ ls

```

root@PC: /
root@PC:/# ls
b1  cdrom  home  lost+found  opt  root  srv  usr  v3
bin  dev  initrd.img  media  p2  run  sys  v1  var
boot  etc  lib  mnt  proc  sbin  tmp  v2  vmlinuz
  
```

Hình 1. 2 Minh họa câu lệnh ls

\$ ls -l

```

root@PC: /
root@PC:/# ls -l
total 112
drwxr-xr-x  4 root root 4096 Th09  9  2019 b1
drwxr-xr-x  2 root root 4096 Th09  3  2019 bin
drwxr-xr-x  3 root root 4096 Th09  3  2019 boot
drwxrwxr-x  2 root root 4096 Th09  3  2019 cdrom
drwxr-xr-x 16 root root 3940 Th05 23 21:23 dev
drwxr-xr-x 129 root root 12288 Th05 23 21:26 etc
drwxr-xr-x  3 root root 4096 Th09  3  2019 home
lrwxrwxrwx  1 root root   33 Th09  3  2019 initrd.img -> boot/initrd.img-4.4.0-142-generic
drwxr-xr-x 23 root root 4096 Th02 25  2020 lib
drwx----- 2 root root 16384 Th09  3  2019 lost+found
drwxr-xr-x  4 root root 4096 Th09  3  2019 media
drwxr-xr-x  3 root root 4096 Th09  9  2019 mnt
drwxr-xr-x  2 root root 4096 Th03  5  2019 opt
drwxr-xr-x  3 root root 4096 Th09  9  2019 p2
dr-xr-xr-x 274 root root    0 Th05 23 21:22 proc
drwx----- 2 root root 4096 Th09  3  2019 root
drwxr-xr-x 23 root root  780 Th05 23 21:33 run
drwxr-xr-x  2 root root 12288 Th09  9  2019 sbin
drwxr-xr-x  2 root root 4096 Th03  5  2019 srv
dr-xr-xr-x 13 root root    0 Th05 23 21:22 sys
  
```

Hình 1. 3 Minh họa câu lệnh ls -l

Lệnh: cd <path>

Path: đường dẫn tương đối (tính từ thư mục hiện hành) hoặc tuyệt đối (tính từ thư mục gốc).

Thư mục đặc biệt:

. : thư mục hiện hành.

.. :Thư mục gốc.

~ hoặc ~username: thư mục home.

/ : root

Ví dụ:

```
$ cd os
```

```
$ cd ~
```

```
$ cd /
```

Lệnh tạo thư mục:

Lệnh: `mkdir [option] <path_name>`

Ví dụ:

```
$ pwd
```

```
/home/test
```

```
$ mkdir t1
```

```
$ mkdir t2
```

```
$ ls -l
```

Cần tạo thư mục test/v1/v2/v3

Ví dụ:

```
$ mkdir v1
```

```
$ mkdir v1/v2
```

```
$ mkdir v1/v2/v3
```

Hoặc

```
$ mkdir -p v1/v2/v3
```

Xóa thư mục:

Xóa thư mục rỗng: `rmdir <thư mục>`

Xóa thư mục không rỗng: `rm -r <thư mục>`

Xóa file: `rm [option] <file_name>`

option: -f: xóa không cần hỏi.

Ví dụ:

```
$ rm -r v1
```

```
$ rm -f cx1.doc
```

Lệnh copy:

Lệnh: `cp [OPTIONS] <nguồn1> <nguồn 2> ... <đích>`

option:

-r, hoặc -R: đệ quy (dùng để copy cả thư mục).

-d: bỏ qua các liên kết khi sử dụng -R.

-f: ép buộc phải làm (force).

-I: hiện dấu nhắc khi ghi đè.

-p: duy trì thuộc tính file.

Ví dụ:

```
$ cp -r t1 t2
```

```
$ cp -f cx1.doc t2/t3
```

Lệnh di chuyển hoặc đổi tên thư mục, tập tin:

Lệnh:

`mv [options] <OldName> <NewName>`

`mv [options] <Source> <Dest>`

option:

--f: ép buộc phải làm (force)

-I: hiện dấu nhắc khi ghi đè

Ví dụ:

```
$ mv cx1.doc cx2.txt
```

```
$ mv -f cx2.txt t2/cx1.doc
```


Lệnh tạo tập tin:

Tạo file và nhập vào nội dung.

```
$ cat > name_of_file
```

Nhập nội dung ở đây.

Nhấn <Enter> để xuống dòng.

Nhấn Ctrl-D để ghi nội dung vào file và kết thúc thao tác.

Ví dụ:

```
$ cat > c1.doc <Enter>
```

```
Test cat <Enter>
```

```
Test cat <Enter>
```

```
Ctrl + D
```

Tạo file rỗng: touch <file_name>

```
$ touch t1.txt
```

Lệnh xem nội dung tập tin:

Lệnh: cat <file_name>

Nội dung file sẽ cuộn sang màn hình trước khi ta có thể xem.

Lệnh: more <file_name>

Phím tắt

Space bar: trang tiếp

Return: dòng kế tiếp

q: thoát

b: trang trước

h: giúp đỡ

Hiển thị n dòng đầu tiên của một file:

```
head -n <filename> mặc định n = 10
```

Hiển thị n dòng cuối cùng của file:

```
tail -n <filename> mặc định n = 10
```

Lệnh thay đổi mật khẩu người dùng:

Để thay đổi mật khẩu cho người sử dụng: lệnh passwd.

Ví dụ

```
$ passwd
```

```
(current) UNIX password:
```

```
Enter new UNIX password:
```

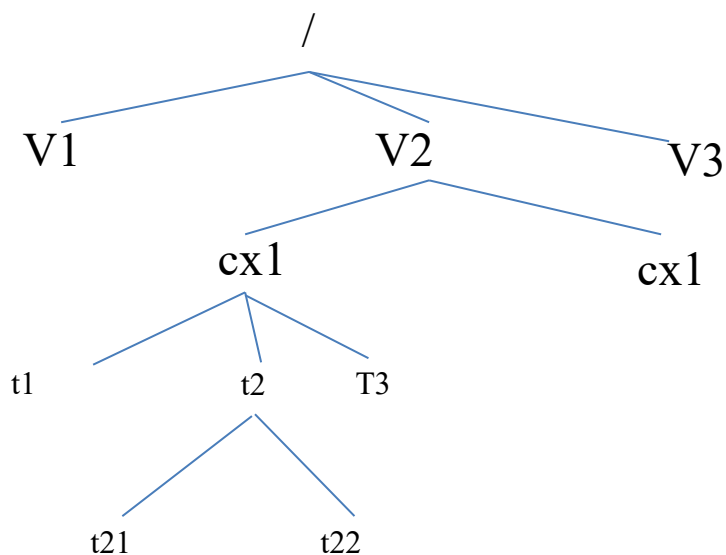
```
Retype new UNIX password:
```

III. Bài tập thực hành:

Sinh viên thực hành theo các yêu cầu sau:

Bài 1:

1. Sử dụng lệnh id để hiển thị định danh người dùng
2. Hiển thị thông tin người sử dụng lệnh whoami và who am i
3. Hiển thị các người sử dụng đang đăng nhập vào hệ thống lệnh who
4. Hiển thị thư mục home
5. Hiển thị thư mục đang làm việc
6. Đổi mật khẩu người sử dụng
7. Từ thư mục hiện hành tạo cấu trúc thư mục như sau



8. Liệt kê các thư mục trong thư mục cx1 (sử dụng các tùy chọn)
9. Chuyển đổi qua lại giữa các thư mục
10. Xóa thư mục t22

- 11.Xóa thư mục t2
- 12.Tạo file cv1.doc trong thư mục T3
- 13.Copy file cv1.doc đến thư mục cx1
- 14.Di chuyển file cv1.doc đến thư mục t1 đồng thời đổi tên thành cv2.txt
- 15.Tạo file thuchanh.doc với nội dung sau
 Hôm nay thuc hanh bai dau tien ve linux

 Cam thay co nhieu thu vi

Bài 2:

1. Xác định thư mục hiện hành mà người dùng root đang truy cập.
2. Tạo các thư mục /root/software, /root/dataserver.
3. Cho biết hai tập tin passwd và group được lưu tại vị trí nào. Sau đó copy chúng (chỉ copy passwd và group trong thư mục /etc thôi) vào thư mục /root/dataserver.
4. Tạo thư mục /root/data. Sau đó copy hai tập tin trong thư mục dataserver về thư mục này với tên là pwd và grp.
5. Tạo tập tin lylich.txt lưu trong thư mục data với nội dung ít nhất 5 dòng.
6. Thêm nội dung sau vào cuối tập tin lylich.txt: “Chao cac ban lop”
7. Gom các nội dung trong thư mục data thành tập tin backup.tar. Sau đó nén tập tin này thành backup.tar.gz.
8. Giải nén và bung nén file backup.tar.gz vào thư mục /root.
9. Xem trợ giúp các lệnh head, tail, cal. Sau đó tạo file có tên /mancmd có nội dung giải thích công dụng của các lệnh này. (man head tail cal >mancmd)
10. Xem toàn bộ nội dung tập tin /etc/passwd.
11. Hiển thị 10 dòng đầu tiên của tập tin /etc/group.

```
#head -n 10 /etc/passwd
```
12. Hiển thị 10 dòng cuối cùng của tập tin /etc/group.
13. Xem nội dung của 2 tập tin pwd và grp đồng thời
14. Tính tổng số dòng và tổng số ký tự trong tập tin pwd và grp. (wc pwd)
15. Tìm trong tập tin /etc/passwd có chuỗi “root” hay không?
16. Đếm số lượng tệp và thư mục trong một thư mục
17. Đếm số lượng thư mục con của một thư mục

BÀI 2: COMPILER VỚI GCC VÀ G++

I. Mục tiêu:

Hướng dẫn sinh viên thực hiện chạy các chương trình C và C++ trên Linux. Giúp sinh viên nắm được khi thực hiện compile một file C hoặc C++ sẽ lần lượt qua từng bước như thế nào, sinh ra các file ra sao. Từ đó, giúp sinh viên hiểu hơn về quá trình hệ điều hành thực thi một chương trình C hoặc C++.

II. Hướng dẫn thực hiện.

Để thực hiện chạy các chương trình C hay C++ trên Ubuntu, ta có thể dùng công cụ gcc cho các chương trình viết bằng C chuẩn hoặc g++ cho các chương trình viết bằng C++.

Trước khi chạy chương trình, ta phải thực hiện kiểm tra phiên bản của gcc hoặc g++

```
$gcc -version
```

Hoặc

```
$g++ --version
```

Trong trường hợp nếu chưa có gcc hoặc g++ ta thực hiện cài đặt như sau:

```
$sudo apt-get install gcc
```

Hoặc

```
$sudo apt-get install g++
```

Tuy nhiên, trong một số tình huống, với các dòng Ubuntu quá cũ, ta phải thực hiện update cho Ubuntu trước bằng câu lệnh:

```
$sudo apt-get update
```

Sau khi đã chắc chắn cài đặt xong, ta có thể cài thêm bộ build-essential để chắc chắn các thư viện chuẩn đã được đưa vào sử dụng.

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

Tới đây, ta đã bắt đầu chạy các chương trình trên Ubuntu.

Giả sử ta soạn thảo một chương trình đơn giản để in ra màn hình chữ Hello..... Được lưu trong file hello.c

```
#include <stdio.h>
void main()
{
    printf("Hello .....\\n");
}
```

Ta bắt đầu compile chương trình như sau:

Ta thực hiện qua 2 bước:

Bước 1:

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

Tại bước này, 1 file hello.o sẽ được sinh ra. Đây là file object sinh ra tương ứng với file hello.c

Bước 2:

```
$gcc -o hello.out hello.o
```

Tại đây, 1 file hello.out sẽ được sinh ra. Đây chính là file thực thi chương trình.

Khi cần chạy chương trình, ta sẽ gọi:

```
./hello.out
```

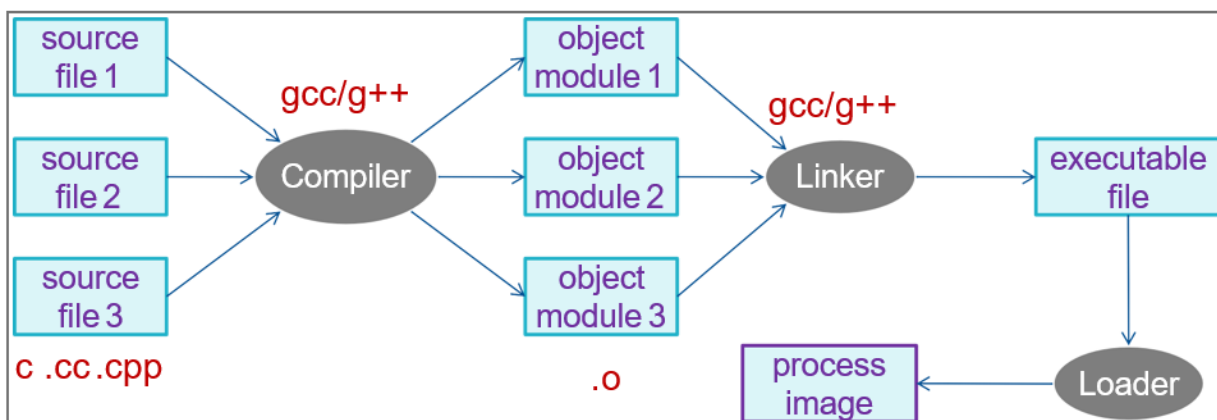
Đoạn chương trình sẽ thực hiện yêu cầu.

```
root@PC:/home/huan/Desktop/bai_tap/hello# ls
hello.c hello.c~
root@PC:/home/huan/Desktop/bai_tap/hello# gcc -c hello.c -o hello.o
root@PC:/home/huan/Desktop/bai_tap/hello# ls
hello.c hello.c~ hello.o
root@PC:/home/huan/Desktop/bai_tap/hello# gcc -o hello.out hello.o
root@PC:/home/huan/Desktop/bai_tap/hello# ls
hello.c hello.c~ hello.o hello.out
root@PC:/home/huan/Desktop/bai_tap/hello# ./hello.out
Hello.....
root@PC:/home/huan/Desktop/bai_tap/hello#
```

Hình 2. 1 Minh họa cách gọi chương trình

Tương tự, nếu đoạn code chạy bằng C++, ta có thể thực hiện bằng g++.

Quá trình biên dịch của 1 file:



Hình 2. 2 Sơ đồ quá trình biên dịch của file

Truyền đối số từ command.

Ta có thể truyền đối số từ dòng lệnh vào chương trình, tuy nhiên để thực hiện truyền đối số vào chương trình ta phải sử dụng cách thức:

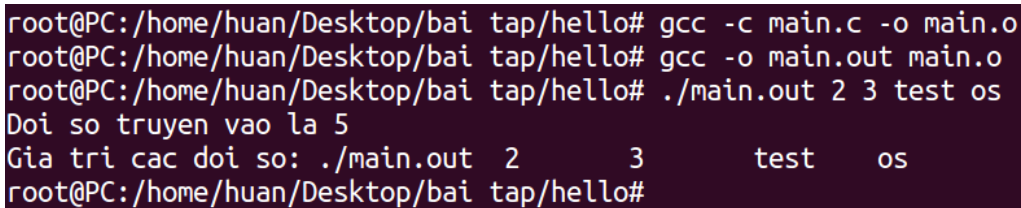
```
int main (int argc, char **argv)
```

Với argc là số lượng đối số truyền vào.

Và argv là 1 mảng các giá trị đối số truyền vào.

Ví dụ:

```
#include <stdio.h>
int main(int argc, char ** argv)
{
    printf("Doi so truyen vao la %i\n", argc);
    printf("Gia tri cac doi so: ");
    int i;
    for (i=0; i < argc; i++)
        printf("%s\t", argv[i]);
    printf("\n");
    return 1;
}
```



```
root@PC:/home/huan/Desktop/bai tap/hello# gcc -c main.c -o main.o
root@PC:/home/huan/Desktop/bai tap/hello# gcc -o main.out main.o
root@PC:/home/huan/Desktop/bai tap/hello# ./main.out 2 3 test os
Doi so truyen vao la 5
Gia tri cac doi so: ./main.out 2      3      test      os
root@PC:/home/huan/Desktop/bai tap/hello#
```

Hình 2. 3 Minh họa cách chạy chương trình

Trong tình huống trên khi ta gọi thực thi main.out ta truyền các đối số là “2 3 test os”.

Có tất cả 4 đối số, tuy nhiên chương trình sẽ tính luôn ./main.out là đối số đầu tiên nên giá trị argc sẽ là 5.

Các giá trị lần lượt sẽ là 5 đối số như hình.

Tạo các thư viện liên kết tĩnh, động

Thư viện liên kết tĩnh

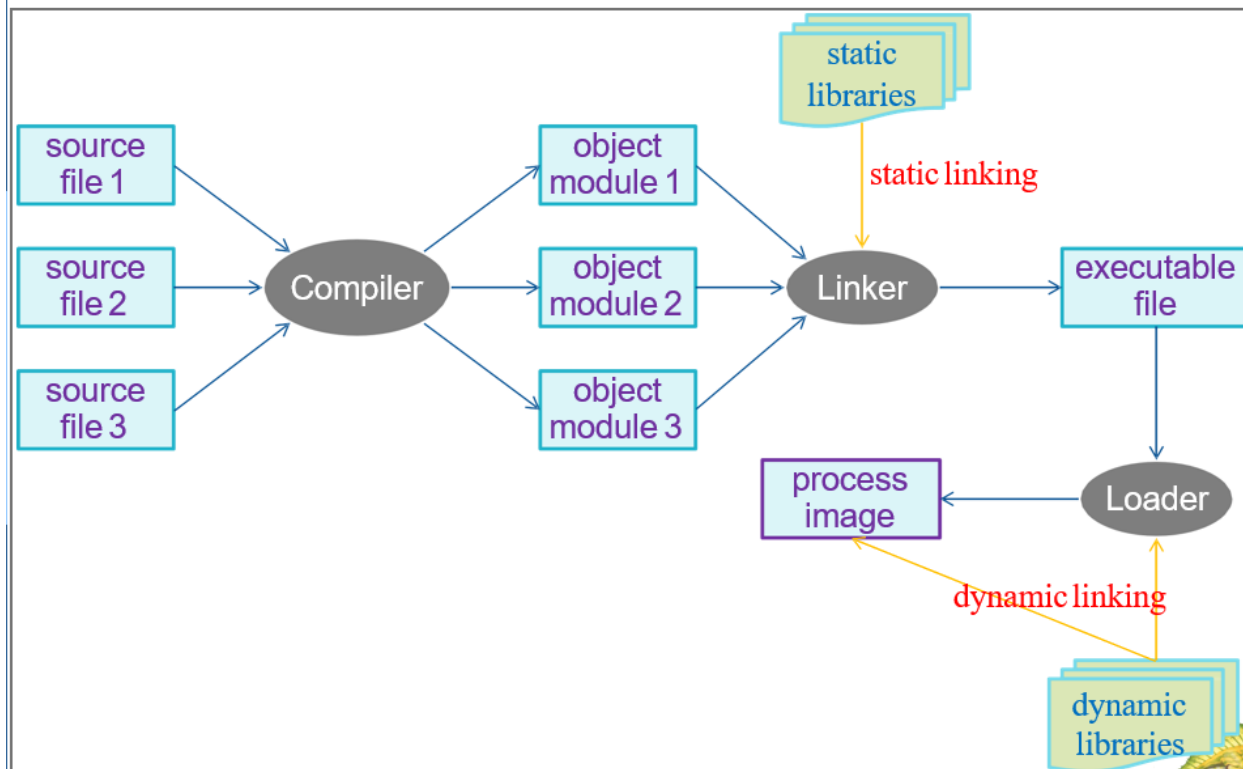
- Là tập hợp các file object tạo thành một file đơn nhất
- Tương tự file .LIB trên windows
- Khi chỉ định một liên kết ứng dụng với một static library thì linker sẽ tìm trong thư viện đó để trích những file object mà bạn cần. Sau đó, linker sẽ tiến hành liên kết các file object đó vào chương trình của bạn.
- gcc -o main\$(OBJ)

Thư viện liên kết động

- Tương tự thư viện dạng .DLL của windows

Thư mục chứa thư viện chuẩn

- /usr/lib, /lib



Hình 2. 4 Mô hình thư viện liên kết tĩnh và động

Xây dựng thư viện liên kết tĩnh

Giả sử ta có 2 file hello1.c và hello2.c như sau:

```

#include <stdio.h>

void hello1(int i)
{
    printf("Hello parameter 1 = %d \n", i);
}

#include <stdio.h>

void hello2(int i)
{
    printf("Hello parameter 2 = %d \n", i);
}
  
```

Tạo thư viện liên kết tĩnh libh.a từ 2 file hello1.c và hello2.c

- Biên dịch và tạo file object
\$ gcc -c hello1.c hello2.c
- Dùng lệnh ar để tạo thư viện tĩnh tên libh.a
\$ ar cr libh.a hello1.o hello2.o
- Dùng lệnh nm để xem kết quả
\$ nm libh.a
- Dùng lệnh file để xem libh là file gì
\$ file libh.a

```

root@PC:/home/huan/Desktop/bai tap# gcc -c hello1.c hello2.c
root@PC:/home/huan/Desktop/bai tap# ar cr libh.a hello1.o hello2.o
root@PC:/home/huan/Desktop/bai tap# nm libh.a

hello1.o:
00000000 T hello1
          U printf

hello2.o:
00000000 T hello2
          U printf
root@PC:/home/huan/Desktop/bai tap# file libh.a
libh.a: current ar archive
root@PC:/home/huan/Desktop/bai tap#

```

Hình 2. 5 Minh họa kiểm tra thư viện vừa tạo

- Tạo hàm main có sử dụng hàm trong thư viện libh.a

```

#include <stdio.h>
int main(int argc, char ** argv)
{
    int i = atoi(argv[1]);
    int k = atoi(argv[2]);
    hello1(i);
    hello2(k);
}

```

- Biên dịch không link đến thư viện liên kết tĩnh -> báo lỗi
\$ gcc -c main.c
\$ gcc -o main.out main.o


```

root@PC:/home/huan/Desktop/bai_tap# gcc -c main.c
root@PC:/home/huan/Desktop/bai_tap# gcc -o main.out main.o
main.o: In function `main':
main.c:(.text+0x39): undefined reference to `hello1'
main.c:(.text+0x45): undefined reference to `hello2'
collect2: error: ld returned 1 exit status
root@PC:/home/huan/Desktop/bai_tap#

```

Hình 2. 6 Minh họa chạy chương trình khi không gọi thư viện

- Biên dịch có link đến thư viện liên kết tĩnh

```

root@PC:/home/huan/Desktop/bai_tap# gcc -o main.out main.o libh.a
root@PC:/home/huan/Desktop/bai_tap# ./main.out 4 5
parameter 1 = 4
parameter 2 = 5
root@PC:/home/huan/Desktop/bai_tap#

```

Hình 2. 7 Minh họa chạy chương trình khi có gọi thư viện

Tạo thư viện liên kết động

- Khi thực hiện thư viện liên kết tĩnh, chương trình bắt buộc phải lưu trữ thư viện cần dùng ngay trong nơi lưu trữ chương trình. Nếu muốn tạo ra một thư viện, sau đó lưu trữ vào thư mục /lib của Linux, để sau này có thể sử dụng mà không cần copy thư viện vào nơi đang làm việc. Vậy ta cần phải tạo ra thư viện động.
- Tạo thư viện liên kết động libd.a từ 2 file hello1.c và hello2.c
- Biên dịch các file object sử dụng tùy chọn -fPIC

```
$ gcc -c -fPIC hello1.c hello2.c
```

- Tạo thư viện liên kết động tên libd.a

```
$ gcc -shared -fPIC -o libd.a hello1.o hello2.o
```

- Để sử dụng được thư viện liên kết động, trước hết chúng ta phải đưa thư viện vào thư mục /lib sử dụng lệnh copy.

```
$ sudo cp libd.a /lib (lưu ý: nếu sử dụng user root thì không cần dùng câu lệnh sudo, nếu sử dụng user khác root thì phải dùng lệnh cp).
```

```
$ gcc -c main.c
```

```
$ gcc -o main.out main.o libd.a
```

```
$ ./main.out 2 3
```

```

root@PC:/home/huan/Desktop/bai_tap# gcc -c -fPIC hello1.c hello2.c
root@PC:/home/huan/Desktop/bai_tap# gcc -shared -fPIC -o libd.a hello1.o hello2.o
root@PC:/home/huan/Desktop/bai_tap# cp libd.a /lib
root@PC:/home/huan/Desktop/bai_tap# gcc -c main.c
root@PC:/home/huan/Desktop/bai_tap# gcc -o main.out main.o libd.a
root@PC:/home/huan/Desktop/bai_tap# ./main.out 2 3
parameter 1 = 2
parameter 2 = 3
root@PC:/home/huan/Desktop/bai_tap#

```

Hình 2. 8 Minh họa chạy chương trình khi gọi thư viện liên kết động

III. Bài thực hành

Bài 1

Viết chương trình sau cho khi truyền đối số n vào thì xuất ra tổng $S = 1 + 2 + \dots + n$

- Báo lỗi nếu lời gọi có đối số không phải là một số nguyên dương.
- Báo lỗi nếu có nhiều hơn 2 đối số (là main.out và n).

```

> ./main.out 8
> S = 36
> ./main.out abc
> Doi so khong phai la so nguyen duong
> ./main.out 8 19 ab
> Co qua nhieu doi so

```

Bài 2

Viết chương trình truyền vào một danh sách số nguyên, và in ra dãy số này theo thứ tự tăng dần.

- Bỏ qua các đối số không phải là số nguyên.
- Hãy áp dụng các thuật toán sắp xếp đã học.

```

> ./main.out 8 3 1 ab -12
> Day tang la -12 1 3 8

```

Bài 3

Viết các tập tin add.c, sub.c lần lượt chứa 2 hàm số `int add(int a, int b)` và `int sub(int a, int b)`.

- Xây dựng một thư viện liên kết tĩnh từ 2 tập tin add.c và sub.c
- Xây dựng một thư viện liên kết động từ 2 tập tin add.c và sub.c

Lần lượt sử dụng các thư viện đã tạo ở trên, viết hàm main truyền vào hai số nguyên và dấu phép tính “+” hay “-”, và in ra kết quả tương ứng. Chương trình báo lỗi nếu các đối số truyền vào không đúng theo qui tắc.

```
> ./main.out 1 2 +  
> Ket qua: 3  
> ./main.out 3 4 -  
> Ket qua: -1  
> ./main.out 1 2 *  
> Doi so truyen khong dung.
```

BÀI 3: QUẢN LÝ TIẾN TRÌNH

I. Mục tiêu:

Hướng dẫn sinh viên thực hiện việc quản lý các tiến trình, bao gồm: tạo ra, quản lý, xóa bỏ các tiến trình. Quản lý các tiến trình cha, tiến trình con. Áp dụng vào các bài toán để giải quyết.

II. Hướng dẫn thực hiện:

Tiến trình là một chương trình đang thực thi, là một thực thể hoạt động, với con trỏ lệnh xác định chỉ thị kế tiếp sẽ thi hành, kèm theo tập các tài nguyên phục vụ cho hoạt động của tiến trình. Mỗi tiến trình sẽ được hệ điều hành gán cho các chỉ số ID tương ứng với hoạt động của nó.

Một tiến trình cũng trải qua các quá trình như con người: Nó được sinh ra, nó có thể có một cuộc đời ít ý nghĩa hoặc nhiều ý nghĩa, nó có thể sinh ra một hoặc nhiều tiến trình con, và thậm chí, nó có thể chết đi. Điều khác biệt nhỏ duy nhất là: tiến trình không có giới tính. Mỗi tiến trình chỉ có một tiến trình cha (hoặc có thể gọi là mẹ, ở trong khoa học sẽ thống nhất gọi là cha) duy nhất.

Dưới góc nhìn của kernel, tiến trình là một thực thể chiếm dụng tài nguyên của hệ thống (Thời gian sử dụng CPU, bộ nhớ, ...)

Khi một tiến trình con được tạo ra, nó hầu như giống hệt như tiến trình cha. Tiến trình con sao chép toàn bộ không gian địa chỉ, thực thi cùng một mã nguồn như tiến trình cha, và bắt đầu chạy tiếp những mã nguồn riêng cho tiến trình con từ thời điểm gọi hàm tạo tiến trình mới.

Mặc dù tiến trình cha và tiến trình con cùng chia sẻ sử dụng phần mã nguồn của chương trình, nhưng chúng lại có phần dữ liệu tách biệt nhau (stack và heap). Điều này có nghĩa là, những sự thay đổi dữ liệu của tiến trình con không ảnh hưởng đến dữ liệu trong tiến trình cha.

Để có thể quản lý được các tiến trình, ta phải sử dụng thư viện:

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

Để thực hiện kiểm soát các tiến trình, ta có thể sử dụng câu lệnh:

- Lấy ID của tiến trình hiện hành `pid_t getpid(void)`
- Lấy ID của tiến trình cha `pid_t getppid(void)`

Ví dụ:

- Giả sử có file `p1.c` chứa hàm `main` như sau:

```
#include <stdio.h>
#include <unistd.h>
void main()
{
    printf ("Current process id %d\n", getpid());
    printf ("Parent process id %d\n", getppid());
}
```

Khi thực thi ta sẽ có kết quả như sau:

```
root@PC:/home/huan/Desktop/bai_tap/Process# gcc -c p1.c
root@PC:/home/huan/Desktop/bai_tap/Process# gcc -o p1.out p1.o
root@PC:/home/huan/Desktop/bai_tap/Process# ./p1.out
Current process id 3525
Parent process id 2710
root@PC:/home/huan/Desktop/bai_tap/Process#
```

Hình 3. 1 Minh họa chạy chương trình lấy ProcessID

Trong đó số 3525 là chỉ số ID của tiến trình hiện hành, ID của tiến trình cha của tiến trình hiện hành là 2710.

Mỗi tiến trình cũng có thể sinh ra tiến trình con của nó. Để sinh ra tiến trình con, ta phải sử dụng các hàm tạo tiến trình như sau:

Hàm tạo tiến trình pid_t fork(void)

- pid_t pid=fork():
- Nếu thành công:
 - pid =0: trong thân process con
 - pid>0: xử lý trong thân process cha
- Nếu thất bại: pid=-1 kèm lý do
 - ENOMEM: không đủ bộ nhớ
 - EAGAIN: số tiến trình vượt quá giới hạn cho phép

Ví dụ:

```
pid_t pid = fork();
if(pid==0){
    Thân chương trình con ở đây
}
else
    if (pid>0)
    {
        Thân chương trình cha ở đây
    }
    else {
```

Lỗi ở đây

}

Mỗi tiến trình có thể quản lý một phần việc riêng rẽ với nhau.

Khi ta muốn chủ động kết thúc một tiến trình ta có thể dùng hàm:

```
exit(0);
```

Trong một số trường hợp, tiến trình cha cần chờ tiến trình con kết thúc trước khi tiếp tục thực hiện công việc của nó. Các *system calls* như **wait**, **waitpid** được xây dựng để phục vụ việc này.

Hàm **wait**, hàm này cho phép tiến trình cha (tiến trình hiện tại) chờ (tạm ngưng tiến trình hiện tại ngay tại lời gọi hàm **wait**) cho đến khi bất kỳ một tiến trình con nào của nó kết thúc. Nguyên mẫu của hàm này như sau:

```
pid_t wait(int *status);
```

Ví dụ sau đây tạo ra một số lượng tiến trình con (được truyền vào qua đối số hàm **main**), in ra ID của các tiến trình con và lời gọi chờ của tiến trình cha.

```
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int pnum, count, retval, child_no;
    pnum = atoi(argv[1]);
    if(pnum <= 0) {
        printf("So tien trinh con phai lon hon 0");
        return -1;
    }
    else {
        retval=1;
        for(count=0, count<pnum; count++) {
            if(retval!=0)
                retval=fork();
            else
                break;
        }
        if(retval == 0) {
            child_no = count;
            printf("Tien trinh %d, PID %d\n",child_no, getpid());
        }
        else {
            for(count=0; count<pnum; count++)
                wait(NULL);
            printf("Tien trinh cha PID %d", getpid());
        }
    }
}
```

```
    return 0;
}
```

III. Bài tập thực hành

Bài 1:

Tạo 1 thư mục có tên là phân số. Trong thư mục này tạo 3 file đặt tên như sau: khaibaops.c, tinhtoanps.c và xulyps.c

Trong file khaibaops.c xây dựng các hàm gồm nhập phân số và xuất phân số.

Trong file tinhtoanps.c xây dựng các hàm cộng, trừ, nhân chia phân số để tính toán phân số.

Từ 2 file khaibaops.c và tinhtoanps.c hãy xây dựng thư viện tĩnh có tên là pstinh.a, xây dựng thư viện động có tên là psdong.a. Chép psdong.a sang thư viện /lib.

Tại file xulyps.c hãy xây dựng 1 cấu trúc phân số. Sau đó sử dụng các hàm có trong bộ thư viện vừa mới xây dựng, hãy cho người dùng nhập phân số. Sau đó lần lượt tính các phép toán cộng, trừ, nhân, chia phân số và xuất kết quả.

Tạo 1 thư mục mới nằm ngoài thư mục phân số có tên là hỗn số. Trong thư mục này tạo file: xulyhonso.c

Trong file xulyhonso.c hãy gọi thư viện đã tạo ở trên. Hãy xây dựng thêm hàm đổi hỗn số sang phân số, sau đó thực hiện các phép tính cộng trừ nhân chia hỗn số này.

Hãy cho biết sự khác biệt khi sử dụng thư viện tĩnh và động trong việc áp dụng qua file xulyhonso.c.

Bài 2:

Hãy viết 1 chương trình: xuất ra ID của tiến trình hiện tại và ID tiến trình cha của nó. Sau đó, hãy nhân đôi tiến trình này. Tại tiến trình cha và con, hãy xuất ra ID của tiến trình của nó và ID tiến trình cha của nó.

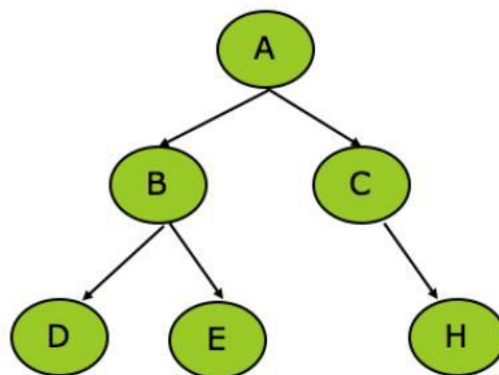
Hãy thêm câu lệnh wait(NULL) vào phần thân của tiến trình cha. Hãy cho biết có điều gì khác biệt.

Bài 3:

Hãy viết 1 hàm để xuất ra thông tin tiến trình hiện tại và tiến trình cha của nó.

Nếu 1 tiến trình sinh ra 1 tiến trình con, hãy xuất thông tin của tiến trình con (đó chính là chỉ số PID).

Hãy tạo cây tiến trình sau:



Bài 4:

Viết chương trình để truyền đối vào số nguyên dương n vào và

- Tiến trình cha tiếp tục tính rồi xuất ra tổng $S = 1 + 2 + \dots + n$

- Đồng thời tạo một tiến trình con tính tổng các ước số của n và in ra màn hình.

Hãy sử dụng lời gọi `wait()` để chắc chắn rằng tiến trình con không trở thành một tiến trình mồ côi. Nếu không sử dụng lời gọi `wait()` thì rủi ro nào có thể xảy ra cho chương trình này?

Bài 5:

Phỏng đoán Collatz tin rằng dãy số sinh ra sẽ luôn tiến về 1 với bất kỳ số nguyên dương nào được tạo ra ở bước đầu. Dãy số được tạo ra theo giải thuật sau:

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

Ví dụ với $n=35$, dãy số sinh ra là 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Viết một chương trình nhận số nguyên dương n vào thông qua đối số, kiểm tra tính đúng của giá trị này. Tạo ra một tiến trình con để tính và in ra dãy số, trong lúc tiến trình cha chờ tiến trình con hoàn thành thông qua lời gọi `wait()`.

BÀI 4: QUẢN LÝ TIẾN TRÌNH – TIẾN TRÌNH MỒ CÔI – TIẾN TRÌNH ZOMBIE

I. Mục tiêu:

Hướng dẫn sinh viên kiểm soát các loại tiến trình mồ côi, tiến trình zombie. Thực hiện phát hiện và hủy các loại tiến trình mồ côi, tiến trình zombie.

II. Hướng dẫn làm bài:

Khi một tiến trình tạo ra tiến trình con, trong một số trường hợp, tiến trình đó sẽ tạo ra tiến trình mồ côi, hoặc tiến trình zombie.

Orphaned process: process cha kết thúc trước; process con sau đó sẽ có cha là init (PID=1).

Zombied process:

- Tiến trình con kết thúc nhưng chưa báo trạng thái cho tiến trình cha.
- Dùng hàm wait() hoặc waitpid() ở tiến trình cha để lấy trạng thái trả về từ tiến trình con

Lưu ý 1: Chúng ta có thể chạy một chương trình ở chế độ “background” bằng cách thêm “&” vào sau lời gọi.

```
> ./main.out &
```

Khi đó main.out thực thi bên dưới và chúng ta có thể gõ lệnh hay chạy một tiến trình khác ở bên trên terminal.

Lưu ý 2: Khi một tiến trình đang thực thi, chúng ta có thể đưa tiến trình này vào “background” thông qua việc gửi tín hiệu SIGTSTP bằng cách nhấn Ctrl + Z, khi đó chúng ta có thể gõ lệnh hoặc chạy một tiến trình khác.

Để xem danh sách tiến trình kèm thông tin, sử dụng

```
> ps
```

```
> ps -l -y //long format and not show flags
```

Từ 2 lưu ý trên, ta có thể tạo ra tiến trình zombie bằng cách tạo ra 1 tiến trình cha, sau đó cho tiến trình này về trạng thái background, và chờ cho tiến trình con kết thúc. Lúc đó tiến trình sẽ trở thành zombie.

Ví dụ:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
void main()
{
    pid_t pid;
    pid=fork();
    if(pid>0)
    {
        printf("hello %d \n",getpid());
        sleep(10);
        exit(0);
    }
    else
    if (pid==0)
    {
        printf("child %d \n",getpid());
        exit(0);
    }
    else
        printf ("error \n");
}
```

Khi thực thi, tại tiến trình cha, bấm ctrl z, sau đó hãy dùng lệnh ps để kiểm tra.

Lệnh liệt kê tiến trình:

> ps [options]

Các option:

- a: hiển thị các tiến trình của user được liên kết
- e: hiển thị tất cả tiến trình
- f: hiển thị PID của tiến trình cha và thời điểm bắt đầu
- l: tương tự -f

Lệnh hủy tiến trình

```
> kill [-signal | -s signal] pid
```

Thường kết hợp với lệnh ps để lấy ID tiến trình

Các signal:

2: tương đương CTRL + C.

9: bắt buộc kết thúc.

5: mặc định – kết thúc bình thường.

19: tạm dừng .

Kill – l: liệt kê tất cả các signal

Ví dụ:

```
kill -9 11234 (kill tiến trình có pid = 11234)
```

Hàm kết thúc tiến trình

Hàm void exit(int status) kết thúc ngay lập tức tiến trình đang gọi.

Bất cứ file nào được mở bởi tiến trình thì được đóng và bất cứ tiến trình con nào được kế thừa bởi tiến trình ban đầu và tiến trình cha được gửi một tín hiệu SIGCHILD.

Khai báo hàm exit() trong C

```
void exit(int status)
```

Tham số

status: Đây là giá trị trạng thái được trả về tới tiến trình cha.

Ví dụ:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    printf("Bat dau thuc thi chuong trinh ...\n");
    printf("Thoat chuong trinh ...\n");
    exit(0);
    printf("Ket thuc chuong trinh ...\n");
    return(0);
}
```

Hàm system

Cú pháp: int system(const char *string);

Thực thi lệnh trong đối số string và trả về kết quả khi thực hiện lệnh xong. Khi gọi hàm system(string), hệ thống sẽ thực hiện lệnh sh -c string.

Giá trị trả về:

0: thành công

-127: Không khởi động shell để thực hiện lệnh

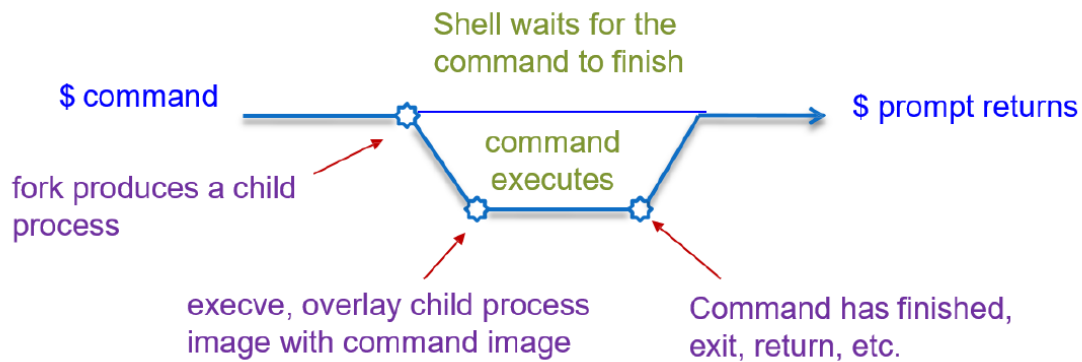
-1: lỗi khác

-1: mã trả về khi thực hiện lệnh string.

Ví dụ:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int re;
    printf("Call system to execute ls -a\n");
    re=system("ls -a");
    if(re != -1)
        printf("System call ls is done!\n");
    printf("Call system to execute ps -a\n");
    re=system("ps -a");
    if(re != -1)
        printf("System call ps is done!\n");
    return 0;
}
```

Các hàm exec... Thực hiện theo cơ chế sau:



Hình 4. 1 Minh họa cơ chế vận hành các hàm exec

Các hàm exec... sẽ thay thế tiến trình gọi hàm bằng chương trình tương ứng trong tham số nhập của hàm. Vùng text, data, stack bị thay thế.

Chương trình được gọi bắt đầu thực thi ở hàm main, có thể nhận tham số nhập thông qua các tham số truyền.

III. Bài tập thực hành:

Bài 1:

Viết một chương trình lặp vô tận với lời gọi `while(1)`; và thực thi nó. Đưa tiến trình này vào “background” thông qua việc gửi tín hiệu `SIGTSTP` bằng cách nhấn `Ctrl + Z`. Sử dụng lệnh `ps` để xác định PID của nó và sử dụng `kill` để kết thúc nó.

Bài 2:

Viết chương trình mà khi chạy nó sinh ra tiến trình con, để tiến trình con trở thành zombie. Tiến trình zombie này cần tồn tại trong hệ thống tối thiểu 10 giây (bằng cách dùng lời gọi `sleep(10)`). Sau đó dùng lệnh `ps -l` để xem trạng thái các tiến trình. Kết liễu zombie này bằng cách xác định PID của tiến trình cha nó và sử dụng lệnh `kill`.

Bài 3:

Sử dụng lời gọi `system()` để viết chương trình thực thi các lệnh Linux như sau:

- Tạo 1 thư mục “BaiTap” tại Desktop
- Tạo 2 thư mục “LyThuyet”, “ThucHanh” trong “BaiTap”
- Tạo 1 file rỗng với tên là “test” trong “ThucHanh”.
- Khi thực hiện xong các lệnh trên, nếu thực hiện thành công, hãy thực hiện thông báo đã hoàn thành.

Bài 4:

Xây dựng 1 chương trình theo ý tưởng như sau:

Xây dựng 1 mảng chứa 10 phần tử số nguyên. Khởi tạo các phần tử đều mang giá trị 0. Mảng này gọi là hàng chờ.

Xây dựng 1 hàm `Producer`, hàm này sẽ thêm `n` giá trị vào cuối mảng đã tạo, các giá trị do người dùng nhập. Nếu nhập quá 10 thì sẽ không cho nhập nữa.

Xây dựng 1 hàm `consumer`, hàm này sẽ lấy `n`, với `n` là số giá trị cuối mảng ra khỏi mảng. Lưu ý, sau khi lấy giá trị ra, giá trị này sẽ bị loại bỏ khỏi mảng (trả lại giá trị 0 cho phần tử đó), các giá trị còn lại sẽ được đưa về cuối mảng.

Tại hàm `main`:

- Cho tạo 2 tiến trình. Tiến trình cha là `producer`, tiến trình con là `consumer`.
- Khi tiến trình cha thực hiện, hãy nhập các số, sau đó chép vào hàng chờ. Thực hiện xong xuất ra thông tin của hàng chờ.
- Dùng hàm `wait` để chờ tiến trình con tiêu thụ `n` sản phẩm, với `n` được nhập vào từ bàn phím. Sau khi tạo, in ra các sản phẩm còn trong hàng chờ.
- Sau khi tiến trình con hoàn thành, quay ra `main`, tại đây hỏi người dùng có kết thúc chương trình không, nếu không lại tiếp tục tạo 2 tiến trình như trên và xử lý tiếp tục, với hàng chờ đã có. Lưu ý, giả sử nếu hàng chờ còn 3 sản phẩm, `producer` phải nhập sản phẩm từ vị trí thứ 4 tính từ cuối mảng. Nếu hàng chờ còn full thì không cho `producer` nhập. Nếu `producer` không nhập, hàng chờ không còn sản phẩm, thì thông báo không có giá trị nào để tiêu thụ. `Consumer` không được phép tiêu thụ quá số sản phẩm trong hàng chờ.

BÀI 5: QUẢN LÝ TIỂU TRÌNH (LUỒNG)

I. Mục tiêu:

Hướng dẫn sinh viên thực hiện việc quản lý tiểu trình (luồng). Bao gồm việc thực hiện tạo ra các tiểu trình (thread – luồng), sử dụng tiểu trình, kết thúc tiểu trình.

II. Hướng dẫn thực hiện:

1. Tạo threads

Thư viện hàm sử dụng:

```
#include <pthread.h>
```

Hàm khởi tạo thread

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

Kết quả trả về

- 0: Thành công, tạo ra một thread mới, ID của thread được trả về qua đối số thread
- <>0: thất bại

Đối số truyền:

- thread: dùng để giữ tham khảo đến threadID nếu hàm thành công. Kiểu pthread_t
- attr: giữ thuộc tính của thread, set NULL nếu muốn sử dụng các thuộc tính mặc định của hệ thống.
- start_routine: là một hàm do người sử dụng định nghĩa mà sẽ được thực thi bởi thread mới.
- Hàm này nên có kiểu trả về là một con trỏ kiểu void, nếu không thì phải ép kiểu về con trỏ void khi gọi thread. Hàm này nên có một đối số truyền cũng có kiểu con trỏ void. Đối số truyền của hàm này sẽ được truyền thông qua tham số thứ 4 (arg).
- arg: là đối số truyền cho hàm start_routine
- Trong trường hợp đối số truyền cần nhiều hơn 1 tham số có thể sử dụng struct, các đối số kiểu này phải được cấp phát tĩnh và được khởi tạo trước.

Khi thread được tạo ra, nó có những thuộc tính riêng, một stack thực thi. Nó kế thừa các tín hiệu và độ ưu tiên từ chương trình gọi.

2. Kết thúc threads

Khi thread được tạo ra, nó thực thi hàm start_routine cho đến khi hoàn tất.

- Có lời gọi hàm thread_exit tương minh
- Bị cancel bởi hàm thread_cancel
- Tiến trình tạo ra thread kết thúc
- Có một thread gọi system call exec

Ví dụ 1: Hãy biên dịch, chạy thử, bỏ hàm sleep() ở dòng 17, hoặc tăng giá trị đối số sleep() và giải thích kết quả chạy cho từng tình huống.

```
#include <pthread.h>
#include <stdio.h>
void* thr1(void* ar){
printf("This is thread %d\n",*((int*) ar));
sleep(2);
}
int main(int argc, char* argv[]){
int i;
int num=atoi(argv[1]);
pthread_t tid[num];
for(i=0; i<num; i++){
pthread_create(&tid[i], NULL, thr1, (void*) &tid[i]);
}
sleep(3);
return(0);
}
```

Kết quả thực hiện:

```
$ gcc -o t01.out t01.o -lpthread
$ ./t01.out 3
This is thread 1547249408
This is thread 1555654312
This is thread 1564399124
```

Ví dụ 2:

Truyền dữ liệu cho thread

Đối số thứ 4 là một kiểu struct

Lưu ý phân ép kiểu trong thr1

Lưu ý đối số truyền thứ 4 của hàm pthread_create

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
struct arr{
    int n;
    int a[10];
};
void* thr1(void* ar){
    int count;
    struct arr *ap=(struct arr*) ar;
    for(count=0;count<ap->n; count++){
        printf("%d\t",ap->a[count]);
        printf("\n");
    }
}
int main(int argc, char* argv[]){
    struct arr ar;
    ar.n=5;
```

```

    int i;
    for(i=0;i<ar.n;i++)
        ar.a[i]=i+1;
    pthread_t tid;
    pthread_create(&tid,NULL,&thr1,&ar);
    sleep(2);
    return 0;
}

```

Kết quả thực hiện:

```

$ gcc -o t02.out t02.o -lpthread
$ ./t02.out
1 2 3 4 5

```

Đội tiểu trình hoàn tất

Tiến trình gọi có thể sử dụng lệnh `pthread_join` để đợi tiểu trình con hoàn tất.

```
int pthread_join(pthread_t threadid, void **status)
```

Kết quả trả về:

- Thành công: 0
- Thất bại: <>0

Đối số:

- threadid: một threadid đã tồn tại
- status: trạng thái trả về, 0 nếu thành công.

Ví dụ 3:

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
void* thr1(void* ar){
    int count;
    printf("this is thread %d\n",*((int*)ar));
    sleep(2);
}
int main (int argc, char* argv[]){
    int i;
    pthread_t tid[3];
    int status, *pstatus = &status;
    for(i=0;i<3;i++){
        pthread_create(&tid[i],NULL,thr1,(void*) &tid[i]);
    }
    for(i=0;i<3;i++){
        if(pthread_join(tid[i],(void**) pstatus)>0){
            printf("pthread_join for thread %d failure\n",
                (int)tid[i]);
        }
    }
}

```



```

    }
    printf("pthread_waited of %d OK, return code: %d\n",
(int)tid[i], status);
    sleep(1);
}
sleep(1);
return 0;
}

```

Kết quả thực hiện:

```

$ gcc -o t03.out t03.o -lpthread
$ ./t03.out
This is thread 1547249408
This is thread 1555654312
This is thread 1564399124
pthread_waited of -1547249408 OK, return code: 0
pthread_waited of -1547249408 OK, return code: 0
pthread_join for thread -1555654312 failure

```

Ví dụ 4:

Hàm thr1 sử dụng để khởi tạo mảng

Hàm thr2 sử dụng để tính tổng các phần tử của mảng.

Hàm thr3 sử dụng để ghi mảng ra file

Lưu ý việc sử dụng hàm pthread_join cho dòng 41. Ví dụ này khi chạy thì tiểu trình thr3 sẽ không ghi được kết quả tính sum của tiểu trình thr2 do tại dòng 41 cần được thực thi sau dòng 27. Hãy viết thêm câu lệnh đồng bộ.

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
struct arr{
    int n;
    int a[10];
};
struct file {
    struct arr ar;
    char* filename;
};
static int sum =0;
void* thr1(void* ar){
    struct arr *ap = (struct arr*) ar;
    ap->n=3; int i=0;
    for(i=0;i<ap->n;i++)
        ap->a[i] = i+1;
}
void* thr2(void* ar){

```

```

    struct arr *ap = (struct arr*) ar;
    int i, s=0;
    for(i=0;i<ap->n;i++)
        s=s + ap->a[i];
    sum=s;
}
void* thr3 (void* ar){
    struct file *fi = (struct file*) ar;
    FILE *out; int count;
    out= fopen(fi->filename,"wb");
    fprintf(out,"number element or array: %d\n", fi->ar.n);
    for(count=0; count<fi->ar.n; count++){
        fprintf(out,"%d\t",fi->ar.a[count]);
    }
    fprintf(out,"\n");
    fprintf(out,"sum=%d\n",sum);
    fclose(out);
}
int main (int argc,char * argv[]){
    int i;
    pthread_t tid[3];
    struct arr ar;
    int status, *pstatus= &status;
    pthread_create(&tid[0],NULL,thr1,(void*) &ar);
    sleep(1);
    if(pthread_join(tid[0],(void**) pstatus)==0)
    {
        pthread_create(&tid[1],NULL,thr2,(void*) &ar);
        if(pthread_create(&tid[1],NULL,thr2,(void*) &ar)==0)
        {
            struct file arf;
            arf.ar=ar;
            arf.filename=argv[1];
            pthread_create(&tid[2],NULL,thr3,(void*) &arf);
        }
    }
    sleep(2);
    return 0;
}

```

Kết quả thực hiện:

```

$ gcc -o t04.out t04.o -lpthread
$ ./t04.out tf1
$ cat tf1
Number element of array: 3
1 2 3
sum = 6

```

3. Thao tác trên tập tin

- Creation of a new file (**fopen with attributes as “a” or “a+” or “w” or “w++”**)
- Opening an existing file (**fopen**)
- Reading from file (**fscanf or fgetc**)
- Writing to a file (**fprintf or fputs**)
- Moving to a specific location in a file (**fseek, rewind**)
- Closing a file (**fclose**)

File operation	Declaration & Description
fopen() - To open a file	<p>Declaration: FILE *fopen (const char *filename, const char *mode)</p> <p>fopen() function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist.</p> <pre>FILE *fp; fp=fopen ("filename", "mode");</pre> <p>Where, fp - file pointer to the data type "FILE". filename - the actual file name with full path of the file. mode - refers to the operation that will be performed on the file. Example: r, w, a, r+, w+ and a+. Please refer below the description for these mode of operations.</p>
fclose() - To close a file	<p>Declaration: int fclose(FILE *fp);</p> <p>fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below.</p> <pre>fclose (fp);</pre>
fgetc() - To read a file	<p>Declaration: char *fgetc(char *string, int n, FILE *fp)</p> <p>fgetc function is used to read a file line by line. In a C program, we use fgetc function as below.</p> <pre>fgetc (buffer, size, fp);</pre> <p>where, buffer - buffer to put the data in. size - size of the buffer fp - file pointer</p>
fprintf() - To write into a file	<p>Declaration:</p> <pre>int fprintf(FILE *fp, const char *format, ...);</pre> <p>fprintf() function writes string into a file pointed by fp. In a C program, we write string into a file as below. fprintf (fp, "some data"); or</p> <pre>fprintf (fp, "text %d", variable_name);</pre>

Ví dụ: Mở file, nhập nội dung lên file đó.

```
// C program to Open a File,
// Write in it, And Close the File
#include <stdio.h>
#include <string.h>
int main( )
```

```
{
// Declare the file pointer
FILE *filePointer ;
// Get the data to be written in file
char dataToBeWritten[50] = "GeeksforGeeks-A Computer Science Portal for
Geeks";
// Open the existing file GfgTest.c using fopen()
// in write mode using "w" attribute
filePointer = fopen("GfgTest.c", "w") ;
// Check if this filePointer is null
// which maybe if the file does not exist
if ( filePointer == NULL )
{
    printf( "GfgTest.c file failed to open." ) ;
}
else
{
    printf("The file is now opened.\n") ;
// Write the dataToBeWritten into the file
    if ( strlen ( dataToBeWritten ) > 0 )
    {
// writing in the file using fputs()
        fputs(dataToBeWritten, filePointer) ;
        fputs("\n", filePointer) ;
    }
// Closing the file using fclose()
    fclose(filePointer) ;
    printf("Data successfully written in file GfgTest.c\n");
    printf("The file is now closed.") ;
}
return 0;
}
```

Ví dụ: Mở file, đọc file đó

```
// C program to Open a File,
// Read from it, And Close the File
#include <stdio.h>
#include <string.h>
int main( )
{
// Declare the file pointer
    FILE *filePointer ;
// Declare the variable for the data to be read from file
    char dataToBeRead[50];
// Open the existing file GfgTest.c using fopen()
// in read mode using "r" attribute
    filePointer = fopen("GfgTest.c", "r") ;
// Check if this filePointer is null
```

```
// which maybe if the file does not exist
    if ( filePointer == NULL )
    {
        printf( "GfgTest.c file failed to open." ) ;
    }
    else
    {
        printf("The file is now opened.\n") ;
// Read the dataToBeRead from the file
// using fgets() method
        while( fgets ( dataToBeRead, 50, filePointer ) != NULL )
        {
// Print the dataToBeRead
            printf( "%s" , dataToBeRead ) ;
        }
// Closing the file using fclose()
        fclose(filePointer) ;
        printf("Data successfully read from file GfgTest.c\n");
        printf("The file is now closed.") ;
    }
    return 0;
}
```

III. Bài tập thực hành

Bài 1:

Viết chương trình đa luồng tính toán các giá trị thống kê khác nhau từ một danh sách các số được truyền vào thông qua đối số của dòng lệnh. Chương trình sau đó sẽ tạo ba tiểu trình tính toán riêng biệt. Một tiểu trình sẽ xác định trung bình cộng của các số, tiểu trình thứ hai sẽ xác định giá trị lớn nhất và tiểu trình thứ ba sẽ xác định giá trị nhỏ nhất. Ví dụ:

```
>./bai22.out 90 81 78 95 79 72 85
Gia tri trung binh: 82
Gia tri lon nhat: 95
Gia tri nho nhat: 72
```

Các biến số đại diện cho các giá trị trung bình, nhỏ nhất và lớn nhất sẽ được lưu trữ trên toàn cục. Các tiểu trình sẽ tính toán các giá trị này và tiến trình cha sẽ xuất ra các giá trị kết quả khi tiểu trình kết thúc.

Bài 2:

Viết chương trình đa luồng để xuất ra số nguyên tố. Người dùng chạy chương trình và nhập vào một số nguyên thông qua đối số tại dòng lệnh. Chương trình sau đó sẽ tạo ra một tiến trình riêng biệt xuất ra tất cả các số nguyên tố nhỏ hơn hoặc bằng số được nhập bởi người dùng.

Bài 3:

Hãy thực hiện lại ví dụ số 4 trên phần hướng dẫn thực hiện. Xử lý tiếp vấn đề đã được đề cập tại ví dụ trên.

Bài 4:

Cho một tập tin có cấu trúc sau:

- Dòng đầu tiên chứa số phần tử mảng
- Dòng còn lại chứa các phần tử là số nguyên

Viết chương trình gồm các thread thực hiện các công việc sau:

- Thread thứ nhất đọc file đầu vào là đối số thứ nhất từ biến môi trường
- Thread thứ hai tính tổng các số nguyên tố trong mảng
- Thread thứ ba tính sắp xếp mảng tăng dần
- Thread thứ tư thực hiện việc ghi file result. Nội dung file đầu vào và đầu ra như sau:

Ví dụ: file đầu vào có dạng sau:

```
10
4 5 7 8 11 9 20 13 2 3
```

Kết quả trong file result có dạng sau:

```
So phan tu mang: 10
4 5 7 8 11 9 20 13 2 3
Mang cac so nguyen to:
5 7 11 13 2 3
Tong cac so nguyen to: 41
Mang cac so nguyen to da duoc sap xep
2 3 5 7 11 13
```

BÀI 6: QUẢN LÝ TIỂU TRÌNH (LUỒNG)

(Tiếp theo)

I. Mục tiêu:

Hướng dẫn sinh viên áp dụng tiểu trình cho các bài toán sắp xếp.

II. Hướng dẫn thực hiện:

Sinh viên đọc lại hướng dẫn quản lý tiểu trình ở bài 5.

Ý tưởng thuật toán Merge sort

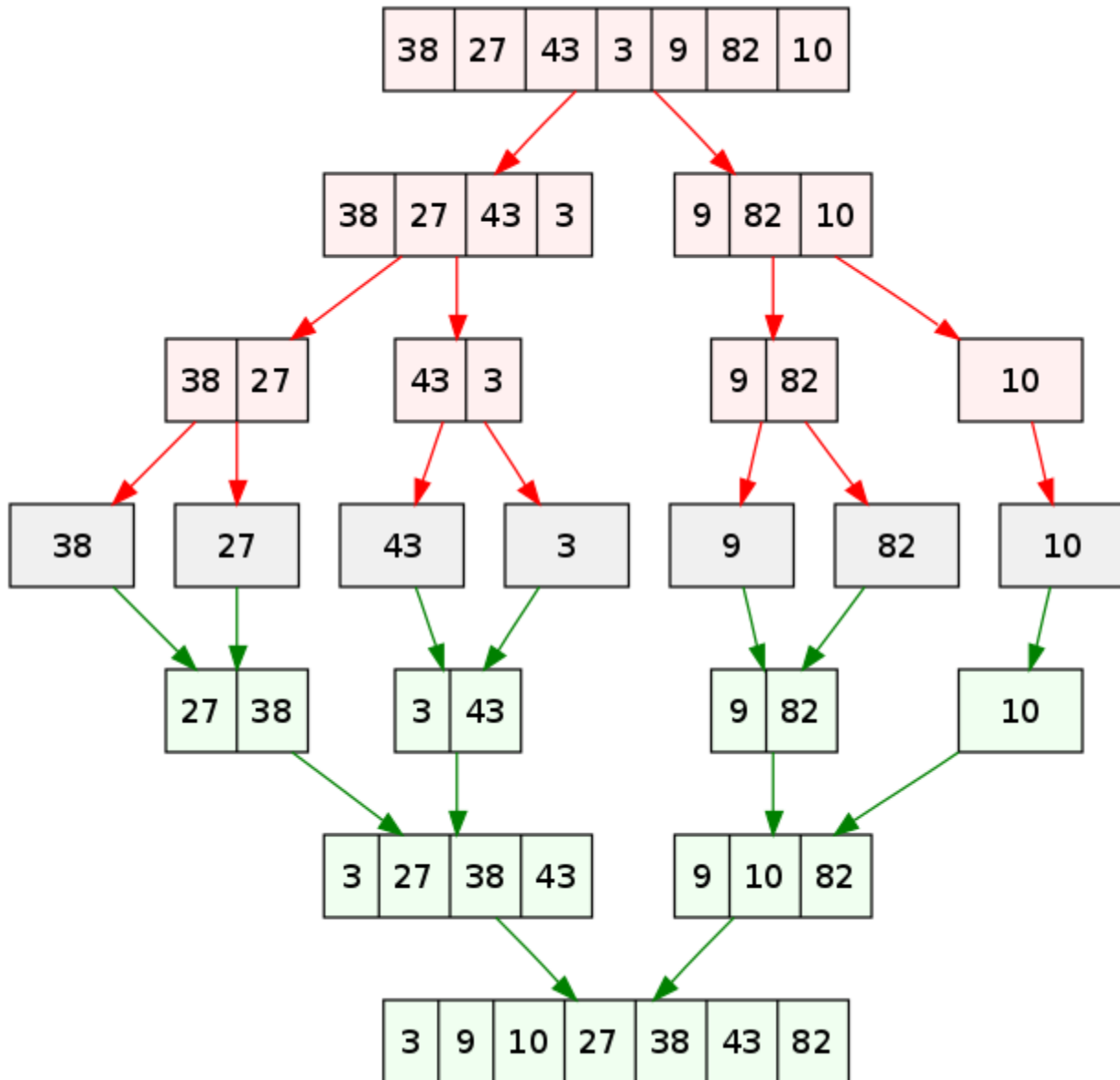
Thuật toán này chia mảng cần sắp xếp thành 2 nửa. Tiếp tục lặp lại việc này ở các nửa mảng đã chia. Sau cùng gộp các nửa đó thành mảng đã sắp xếp. Hàm merge() được sử dụng để gộp hai nửa mảng. Hàm merge(arr, l, m, r) là tiến trình quan trọng nhất sẽ gộp hai nửa mảng thành 1 mảng sắp xếp, các nửa mảng là arr[l...m] và arr[m+1...r] sau khi gộp sẽ thành một mảng duy nhất đã sắp xếp.

Hãy xem ý tưởng triển khai code dưới đây để hiểu hơn

```

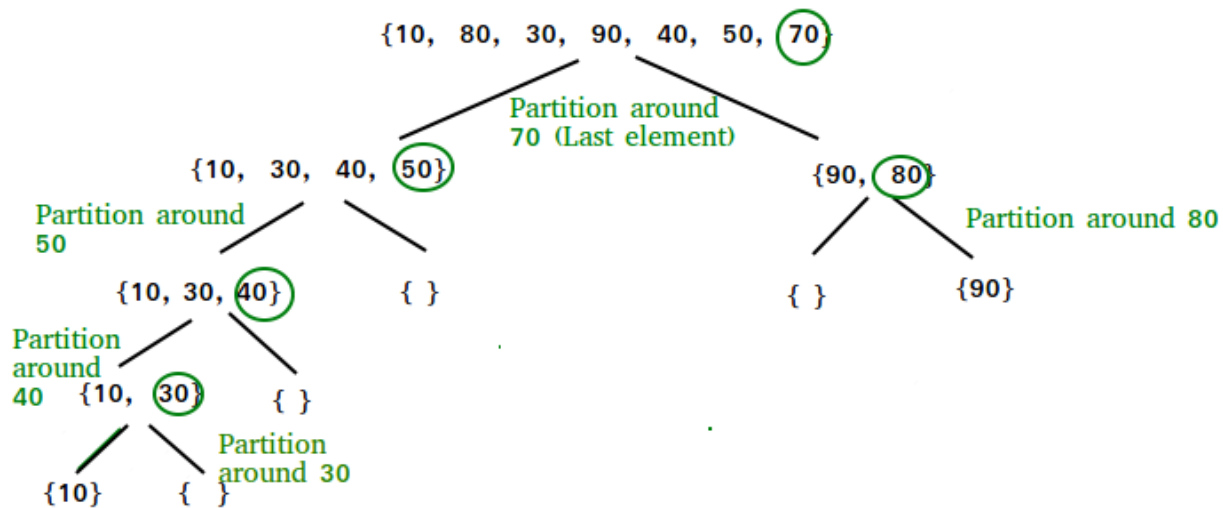
0   mergeSort(arr[], l, r)
1   If r > l
2       1. Tìm chỉ số nằm giữa mảng để chia mảng thành 2 nửa:
3           middle m = (l+r)/2
4       2. Gọi đệ quy hàm mergeSort cho nửa đầu tiên:
5           mergeSort(arr, l, m)
6       3. Gọi đệ quy hàm mergeSort cho nửa thứ hai:
7           mergeSort(arr, m+1, r)
8       4. Gộp 2 nửa mảng đã sắp xếp ở (2) và (3):
9           merge(arr, l, m, r)
```

Hình ảnh dưới đây từ wikipedia sẽ hiển thị toàn bộ sơ đồ tiến trình của thuật toán merge sort cho mảng {38, 27, 43, 3, 9, 82, 10}. Nếu nhìn kỹ hơn vào sơ đồ này, chúng ta có thể thấy mảng ban đầu được lặp lại hành động chia cho tới khi kích thước các mảng sau chia là 1. Khi kích thước các mảng con là 1, tiến trình gộp sẽ bắt đầu thực hiện gộp lại các mảng này cho tới khi hoàn thành và chỉ còn một mảng đã sắp xếp.



Hình 6. 1 Minh họa thuật toán Merge Sort

Ý tưởng của thuật toán sắp xếp quick sort



Hình 6. 2 Minh họa thuật toán sắp xếp quick sort

Giống như Merge sort, thuật toán sắp xếp quick sort là một thuật toán chia để trị(Divide and Conquer algorithm). Nó chọn một phần tử trong mảng làm điểm đánh dấu(pivot). Thuật toán sẽ thực hiện chia mảng thành các mảng con dựa vào pivot đã chọn. Việc lựa chọn pivot ảnh hưởng rất nhiều tới tốc độ sắp xếp. Nhưng máy tính lại không thể biết khi nào thì nên chọn theo cách nào. Dưới đây là một số cách để chọn pivot thường được sử dụng:

- Luôn chọn phần tử đầu tiên của mảng.
- Luôn chọn phần tử cuối cùng của mảng. (Được sử dụng trong bài viết này)
- Chọn một phần tử random.
- Chọn một phần tử có giá trị nằm giữa mảng(median element).

III. Bài tập thực hành:

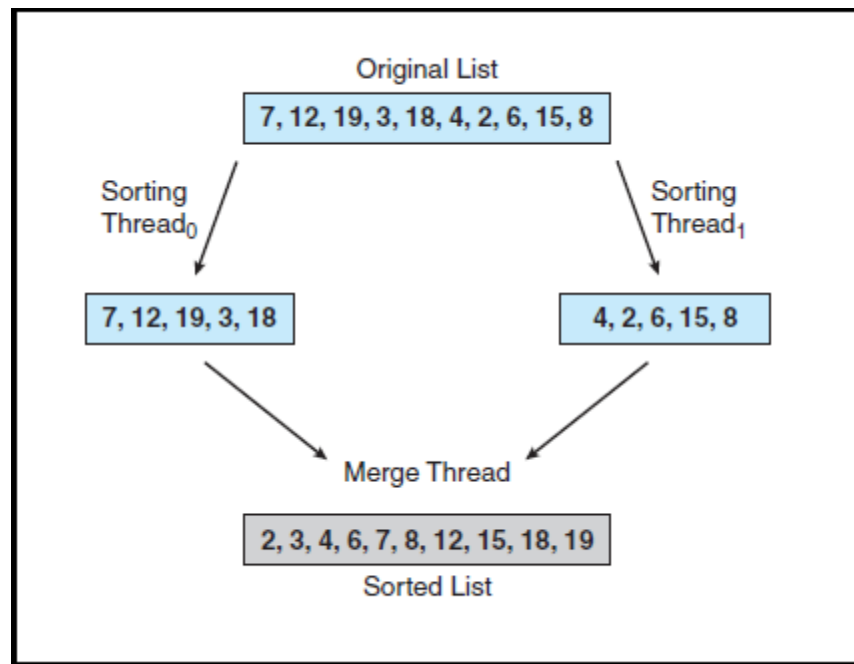
Bài 1:

Tạo 1 mảng các phần tử chưa được sắp xếp. Hãy xây dựng 1 hàm sắp xếp dựa trên các thuật toán đã học. Hãy tách mảng này thành 2 mảng con. Sau đó, với mỗi mảng con, gọi 1 tiểu trình thực hiện sắp xếp mảng đó. Sau khi thực hiện xong, thực hiện nối 2 mảng này.

Bài 2:

Áp dụng ý tưởng lập trình đa luồng, sắp xếp một mảng theo phương pháp chia để trị

1. Quick sort
2. Merge sort



Hình 6. 3 Minh họa Merge Sort với 2 Thread

BÀI 7: CPU SCHEDULING

I. Mục tiêu:

Hướng dẫn sinh viên thực hiện các giải thuật CPU Scheduling. Giúp sinh viên hiểu rõ hơn về việc thực hiện CPU Scheduling trong hệ điều hành.

II. Hướng dẫn thực hiện:

Hãy download tài liệu sau để thực hiện các bài tập về lập lịch CPU.

<https://drive.google.com/open?id=1SawEyNB1Y-7uGyzzFwb0MfYyVerGILdw>

Lập lịch CPU (CPU Scheduling) là việc thực hiện điều phối các tiến trình vào CPU để thực hiện tính toán, giúp khai thác hệ thống hiệu quả hơn.

Để thực hiện lập lịch CPU có nhiều giải thuật:

- FCFS (First Come, First Served) – Đến trước phục vụ trước.
- SJF (Shortest Job First) – Công việc ngắn trước.
- SRTF (Shortest Remain Time First) – Công việc nào còn ít thời gian nhất thực hiện trước.
- Priority – Điều phối có độ ưu tiên.
- Round Robin – Luân chuyển.
- Multilevel queue Scheduling – Hàng đợi đa mức.
-

Ví dụ thuật toán FCFS:

```
#include<stdio.h>
#include<conio.h>
void main() {
    int n,i,j,sum=0;
    int arrv[10], ser[10], start[10], finish[10],wait[10], turn[10];
    float avgtturn=0.0,avgwait=0.0;
    start[0]=0;
    clrscr();
    printf("\n ENTER THE NO. OF PROCESSES:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\n ENTER THE ARRIVAL TIME AND SERVICE TIME OF PROCESS
%d: ",i+1);
        scanf("%d%d",&arrv[i],&ser[i]); }
    for(i=0;i<n;i++)
    {
        sum=0;
        for(j=0;j<i;j++)
            sum=sum+ser[j];
        start[i]=sum;
    }
    for(i=0;i<n;i++)
```

```

{
    finish[i]=ser[i]+start[i];
    wait[i]=start[i];
    turn[i]=ser[i]+wait[i];
}
for(i=0;i<n;i++)
{
    avgwait+=wait[i] ;
    avgturn+=turn[i];
}
avgwait/=n;
avgturn/=n;
printf("\n PROCESS ARRIVAL SERVICE START FINISH WAIT TURN \n");
for(i=0;i<n;i++)
{
    printf("\n\tP%d\t%d \t %d \t %d \t %d \t %d \t %d \n",i ,arrv[i],
ser[i], start[i], finish[i],wait[i],turn[i]);
}
printf("\n AVERAGE WAITING TIME = %f tu",avgwait);
printf("\n AVERAGE TURN AROUND TIME = %f tu", avgturn);
getch();
}

```

Ví dụ thuật toán SJF:

```

#include<stdio.h>
#include<conio.h>
void main() {
    int n,i,j,temp1,temp2,sum=0;
    int  pro[10],  arrv[10],  ser[10],  start[10],  finish[10],
wait[10],  turn[10];
    float avgturn=0.0,avgwait=0.0; start[0]=0; clrscr();
    printf("\n ENTER THE NO. OF PROCESSES:");
    scanf("%d",&n);
    for(i=0;i<n;i++) {
        printf("\n ENTER THE ARRIVAL TIME AND SERVICE TIME OF PROCESS
%d:",i+1);
        scanf("%d%d",&arrv[i],&ser[i]); pro[i]=i;
    }
    for(i=0;i<n;i++) {
        for(j=0;j<n-i-1;j++) {
            if(ser[j]>ser[j+1]) {
                temp1=ser[j];
                ser[j]=ser[j+1];
                ser[j+1]=temp1;
                temp2=arrv[j];
                arrv[j]=arrv[j+1];
                arrv[j+1]=temp2;
            }
        }
    }
}

```

```

    }
    for(i=0;i<n;i++) {
        sum=0;
        for(j=0;j<i;j++)
            sum=sum+ser[j];
        start[i]=sum;
    }
    for(i=0;i<n;i++) {
        finish[i]=ser[i]+start[i];
        wait[i]=start[i];
        turn[i]=ser[i]+wait[i];
    }
    for(i=0;i<n;i++) {
        avgwait +=wait[i] ;
        avgturn +=turn[i];
    }
    avgwait/=n;
    avgturn/=n;
    printf("\n PROCESS  ARRIVAL  SERVICE  START  FINISH  WAIT  TURN
\n");
    for(i=0;i<n;i++){
        printf("\n\tP%d\t%d \t %d \t %d \t %d \t %d \t %d \n",
        pro[i],arrv[i], ser[i], start[i], finish[i],wait[i],turn[i]);
    }
    printf("\n AVERAGE WAITING TIME = %f tu",avgwait);
    printf("\n AVERAGE TURN AROUND TIME = %f tu" ,avgturn);
    getch();
}

```

III. Bài tập thực hành

Bài 1:

Hãy thực hiện lại thuật toán FCFS và SJF theo hướng dẫn trên, và thực thi để khảo sát kết quả. Từ kết quả, hãy so sánh hai thuật toán trên.

Bài 2:

Từ file tài liệu đã đưa, hãy thực hiện lập trình mô tả thuật toán FCFS, SJF, Round Robin, SRTF. Trong mỗi thuật toán hãy thực hiện các việc như sau:

1. Hãy lập trình để xuất ra mô tả bằng sơ đồ gantt.
2. Hãy cho chương trình nhập dữ liệu từ file, và xuất ra file.

BÀI 8: GIAO TIẾP GIỮA CÁC TIẾN TRÌNH – SỬ DỤNG ĐƯỜNG ỐNG (PIPE)

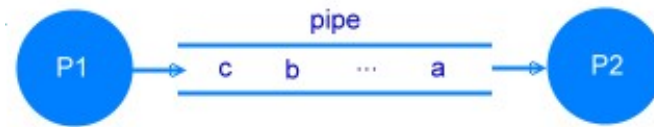
I. Mục tiêu:

Hướng dẫn sinh viên thực hiện giao tiếp giữa các tiến trình bằng cách sử dụng đường ống (pipe). Quản lý đường ống không tên (unnamed pipe) và có tên (named pipe).

II. Hướng dẫn thực hiện:

Giới thiệu: Một pipe là một kênh liên lạc trực tiếp giữa hai tiến trình : dữ liệu xuất của tiến trình này được chuyển đến làm dữ liệu nhập cho tiến trình kia dưới dạng một dòng các byte.

Khi một pipe được thiết lập giữa hai tiến trình, một trong chúng sẽ ghi dữ liệu vào pipe và tiến trình kia sẽ đọc dữ liệu từ pipe. Thứ tự dữ liệu truyền qua pipe được bảo toàn theo nguyên tắc FIFO. Một pipe có kích thước giới hạn (thường là 4096 ký tự).



Hình 8. 1 Liên lạc qua pipe (đường ống)

Một tiến trình chỉ có thể sử dụng một pipe do nó tạo ra hay kế thừa từ tiến trình cha. Hệ điều hành cung cấp các lời gọi hệ thống read/write cho các tiến trình thực hiện thao tác đọc/ghi dữ liệu trong pipe. Hệ điều hành cũng chịu trách nhiệm đồng bộ hóa việc truy xuất pipe trong các tình huống:

Tiến trình đọc pipe sẽ bị khóa nếu pipe trống, nó sẽ phải đợi đến khi pipe có dữ liệu để truy xuất.

Tiến trình ghi pipe sẽ bị khóa nếu pipe đầy, nó sẽ phải đợi đến khi pipe có chỗ trống để chứa dữ liệu.

Khi thực hiện pipe sẽ có 2 loại pipe: unname pipe (pipe không tên) và name pipe (pipe có tên).

Khởi tạo pipe:

Thư viện:

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

Hàm ghi / đọc:

```
ssize_t write(int fd, const void *buf, size_t count)
ssize_t read(int fd, const void *buf, size_t count)
```

Kết quả trả về:

- -1 thất bại.
- Thành công: số byte ghi được hoặc đọc được.

Unnamed pipe:

Thường được sử dụng cục bộ.

Dành cho các tiến trình có quan hệ cha con.

Hàm tạo unnamed pipe:

```
int pipe(int filedes[2]);
```

Kết quả

- Thành công: 0; hai file mô tả tương ứng được trả về trong file filedes[0] và filedes[1].
- Với hệ thống cũ, filedes[0] được sử dụng để đọc, filedes[1] được sử dụng để ghi
- Trong các hệ thống mới sử dụng fullduplex, nếu filedes[0] được dùng để đọc thì filedes[1] được dùng để ghi và ngược lại.
- Thất bại trả về -1.

Ví dụ 1.1:

Tiến trình con đọc dữ liệu từ đối số truyền, ghi vào pipe.

Tiến trình cha đọc từ pipe và xuất ra màn hình.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main(int argc, char* argv[])
{
    char result[100];
    int fp[2];
    int pid;
    if(argc<2) {
        printf("Doi so thieu.\n");
        return -1;
    }
    if(pipe(fp)==0) {
        pid = fork();
        if(pid<0) {printf("Fork failed\n"); return -1;}
        else if(pid==0) {
            printf("Data from child: %s\n", argv[1]);
            close(fp[0]);
            write(fp[1], argv[1], strlen(argv[1]));
        }

        else {
            close(fp[1]);
            read(fp[0], result, strlen(argv[1]));
            printf("Read from child: %s\n", result);
        }
    } // of row 14
    else {printf("Pipe failed\n"); return -2;}
}
```

Trong ví dụ trên, ta truyền đối số dưới dạng ký tự. Nếu muốn truyền đối số dưới dạng số nguyên, ta có thể dùng biến bình thường, tuy nhiên, ta phải dùng hàm `sizeof(var)` để xác định độ lớn của biến.

Ví dụ 1.2:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main(int argc, char* argv[])
{
    int fp[2];
    int pid;
    if(argc<2) {
        printf("Doi so thieu.\n");
        return -1;
    }
    if(pipe(fp)==0) {
        pid = fork();
        if(pid<0) {printf("Fork failed\n"); return -1;}
        else if(pid==0) {
            int n=atoi(argv[1]);
            printf("Data from child: %d\n", n);
            close(fp[0]);
            write(fp[1], &n, sizeof(n));
        }

        else {
            close(fp[1]);
            int tam;
            read(fp[0], &tam, sizeof(tam));
            printf("Read from child: %d\n", tam);
        }
        // of row 14
    } else {printf("Pipe failed\n"); return -2;}
}
```

Tuy nhiên, trong trường hợp 2 tiến trình đều có nhu cầu gửi dữ liệu cho nhau. Ta phải sử dụng 2 đường ống. Một đường ống chuyên gửi dữ liệu từ tiến trình 1 qua tiến trình 2, và ngược lại.

Ví dụ 1.3: Tiến trình cha cho người dùng nhập vào 1 số, gửi số này xuống tiến trình con. Tiến trình con nhận số từ tiến trình cha, tiến trình con cho người dùng nhập vào 1 số, gửi sang tiến trình cha. Tiến trình cha xuất số này ra.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main(int argc, char* argv[])
```



```

{
    int fp1[2],fp2[2];
    int pid;
    if(argc<2) {
        printf("Doi so thieu.\n");
        return -1;
    }
    if((pipe(fp1)==0)&&(pipe(fp2)==0)) {
        pid = fork();
        if(pid<0) {printf("Fork failed\n"); return -1;}
        else if(pid==0) {
            close(fp1[1]);
            int tam;
            read(fp1[0],&tam , sizeof(tam));
            printf("Read from parents: %d\n", tam);
            printf("Reply from child: ");
            scanf("%d",tam);
            close(fp2[0]);
            write(fp2[1],&tam,sizeof(tam));
        }
        else {
            int n;
            scanf("%d",n);
            printf("Data from parent: %d\n", n);
            close(fp1[0]);
            write(fp1[1],&n, sizeof(n));
            close(fp2[1]);
            read(fp2[0],&n,sizeof(n));
            printf("data from child %d\n",n);
        }
    } // of row 14
    else {printf("Pipe failed\n"); return -2;}
}

```

Named Pipe

Mang ý nghĩa toàn cục.

Tương tự như unnamed pipe.

Được ghi nhận trên file system.

Có thể sử dụng với các tiến trình không có quan hệ cha con.

Có thể tạo từ dấu nhắc lệnh.

Thư viện:

```

#include <sys/types.h>
#include<sys/stat.h>

```

Hàm khởi tạo:

```
int mknod(const char *path, mode_t mode, dev_t dev)
```

Đối số:

- path: đường dẫn đến pipe; kết hợp S_IFIFO với quyền khác.
- mode: quyền truy cập trên file.
- dev: mặc định là 0.

Kết quả trả về:

- Thất bại: -1; Thành công: 0.

Hàm sử dụng:

```
int mkfifo(const char *pathname, mode_t mode)
```

Đối số:

- path: đường dẫn đến pipe.
- mode: quyền truy cập trên file.

Kết quả trả về:

- Thất bại: -1
- Thành công: 0

Tiến trình cha ghi dữ liệu vào pipe

Tiến trình vào đọc, hiển thị dữ liệu, ghi dữ liệu phản hồi.

Tiến trình cha đọc dữ liệu phản hồi và hiển thị ra màn hình.

Ví dụ 2.1: Cho phép người dùng nhập vào 1 chuỗi, gửi chuỗi này qua tiến trình thứ 2. Tiến trình thứ 2 cho phép nhập lại 1 chuỗi khác, gửi ngược lại tiến trình đầu tiên.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#define FIFO1 "/tmp/ff.1"
#define FIFO2 "/tmp/ff.2"
#define PM 0666
extern int errno;
#define PIPE_BUF 4096
int main(int argc, char* argv[])
{
    char s1[PIPE_BUF], s2[PIPE_BUF];
    int childpid, readfd, writefd;
    //kiểm tra 2 file FIFO1 và FIFO2 có khởi tạo thành công?
    if((mknod(FIFO1, S_IFIFO | PM, 0)<0)&&(errno!=EEXIST)){
        printf("Fail to create FIFO 1. Aborted.\n");
        return -1;
    }
    if((mknod(FIFO2, S_IFIFO | PM, 0)<0)&&(errno!=EEXIST)){
        unlink(FIFO1);
```

```

        printf("Fail to create FIFO 2. Aborted.\n");
        return -1;
    }
    childpid=fork();
    if(childpid==0){ //child
        //kiểm tra có thể mở 2 file FIFO1, FIFO2
        if((readfd=open(FIFO1, 0))<0)
            perror("Child cannot open readFIFO.\n");
        if((writefd=open(FIFO2, 1))<0)
            perror("Child cannot open writeFIFO.\n");
        //xử lý chương trình.
        read(readfd, s2, PIPE_BUF);
        printf("Child read from parent: %s\n", s2);
        printf("Enter response: ");
        gets(s1);
        write(writefd, s1, strlen(s1));
        close(readfd);
        close(writefd);
        return 1;
    }
    else if(childpid>0) { //parent
        //Kiểm tra có mở thành công 2 file FIFO1, FIFO2.
        if((writefd=open(FIFO1, 1))<0)
            perror("Parent cannot open writeFIFO.\n");
        if((readfd=open(FIFO2, 0))<0)
            perror("Child cannot open readFIFO.\n");
        // xử lý chương trình.
        printf("Enter data to FIFO1: ");
        gets(s1);
        write(writefd, s1, strlen(s1));
        read(readfd, s2, PIPE_BUF);
        printf("Parent read from child: %s\n", s2);
        while(wait((int*) 0)!=childpid);
        close(readfd);
        close(writefd);
        if(unlink(FIFO1)<0)
            perror("Cannot remove FIFO1.\n");
        if(unlink(FIFO2)<0)
            perror("Cannot remove FIFO2.\n");
        return 1;
    }
    else { printf("Fork failed\n"); return -1;}
}

```

Tương tự, ta cũng có thể làm tương tự bài trên, nhưng dùng số nguyên thay vì dùng chuỗi.

Ví dụ 2.2:

```

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

```

```

#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#define FIFO1 "/tmp/ff.1"
#define FIFO2 "/tmp/ff.2"
#define PM 0666
extern int errno;

int main(int argc, char* argv[])
{
    int childpid, readfd, writefd;
    //kiểm tra 2 file FIFO1 và FIFO2 có khởi tạo thành công?
    if((mknod(FIFO1, S_IFIFO | PM, 0)<0)&&(errno!=EEXIST)){
        printf("Fail to create FIFO 1. Aborted.\n");
        return -1;
    }
    if((mknod(FIFO2, S_IFIFO | PM, 0)<0)&&(errno!=EEXIST)){
        unlink(FIFO1);
        printf("Fail to create FIFO 2. Aborted.\n");
        return -1;
    }
    childpid=fork();
    if(childpid==0){ //child
        //kiểm tra có thể mở 2 file FIFO1, FIFO2
        if((readfd=open(FIFO1, 0))<0)
            perror("Child cannot open readFIFO.\n");
        if((writefd=open(FIFO2, 1))<0)
            perror("Child cannot open writeFIFO.\n");
        //xử lý chương trình.
        int num;
        read(readfd, &num, sizeof(num));
        printf("Child read from parent: %d\n", num);
        printf("Enter response: ");
        scanf("%d",&num);
        write(writefd, &num, sizeof(num));
        close(readfd);
        close(writefd);
        return 1;
    }
    else if(childpid>0) { //parent
        //Kiểm tra có mở thành công 2 file FIFO1, FIFO2.
        if((writefd=open(FIFO1, 1))<0)
            perror("Parent cannot open writeFIFO.\n");
        if((readfd=open(FIFO2, 0))<0)
            perror("Child cannot open readFIFO.\n");
        // xử lý chương trình.
        printf("Enter number to FIFO1: ");
    }
}

```

```

    int num;
    scanf("%d", &num);
    write(writefd, &num, sizeof(num));
    read(readfd, &num, sizeof(num));
    printf("Parent read from child: %d\n", num);
    while(wait((int*) 0) != childpid);
    close(readfd);
    close(writefd);
    if(unlink(FIFO1) < 0)
        perror("Cannot remove FIFO1.\n");
    if(unlink(FIFO2) < 0)
        perror("Cannot remove FIFO2.\n");
    return 1;
}
else { printf("Fork failed\n"); return -1;}
}

```

III. Bài tập thực hành:

Bài 1:

Tiến trình cha chuyển đổi số đầu tiên (argv [1]) là một số nguyên lớn hơn 3 cho tiến trình con thông qua đường ống. Tiến trình con nhận, tính giá trị $n! = 1 * 2 * \dots * n$ và ghi nó vào đường ống. Tiến trình cha nhận và xuất dữ liệu ra màn hình. Sử dụng đường ống vô danh (Unnamed Pipe).

```

> ./baitap1A.out 4
4! = 24

```

Giải lại vấn đề với đường ống có tên (Named Pipe).

Bài 2:

Tiến trình cha đọc hai số nguyên và một thao tác +, -, x, / và chuyển tất cả cho tiến trình con. Quá trình con tính toán kết quả và trả về cho tiến trình cha. Quá trình cha ghi kết quả vào một tệp.

```

> ./baitap2A.out 4 6 +
4 + 6 = 10

```

Giải lại vấn đề với đường ống có tên (Named Pipe).

BÀI 9: GIAO TIẾP GIỮA CÁC TIẾN TRÌNH – MESSAGE QUEUE

I. Mục tiêu:

Hướng dẫn sinh viên thực hiện quản lý giao tiếp giữa các tiến trình sử dụng phương pháp message queue.

II. Hướng dẫn thực hiện:

Message queue là một hộp thư, cho phép các thành phần/service trong một hệ thống (hoặc nhiều hệ thống), gửi thông tin cho nhau.

Sở dĩ gọi nó là queue (hàng đợi) vì nó thực hiện việc lấy message theo cơ chế FIFO – First In First Out, tức đút vào trước thì rút ra trước.

Trong các hệ thống dùng kiến trúc microservice, ta sử dụng message queue để giúp các service liên hệ với nhau **một cách bất đồng bộ**. *Service A* làm xong việc có thể gửi message queue để *service B* biết mà xử lý, **không cần phải chờ** *service B* làm xong.

Các thông tin cần để giao tiếp được đặt trong một cấu trúc thông điệp định nghĩa trước.

Tiến trình tạo thông điệp phải xác định được kiểu và vị trí của thông điệp.

Các tiến trình truy xuất hàng đợi của thông điệp thông qua hàng đợi theo cơ chế FIFO

- `msgget()`: sử dụng để tạo Message queue
- `msgctl()`: sử dụng để điều khiển MQ
- `msgsnd()`: sử dụng để gửi thông điệp
- `msgrcv()`: sử dụng để nhận thông điệp

Thư viện:

```
#include <sys/types.h>
#include<sys/ipc.h>
```

Prototype của hàm:

```
key_t ftok ( char *pathname, char proj );
```

Được sử dụng để tạo key cho các thao tác trên MQ. Hàm `ftok` có thể tạo ra cùng key value.

Đối số:

- `pathname`: tham chiếu đến một file đã tồn tại. Thường sử dụng “.” là tham chiếu đến chính thư mục hiện tại.
- `proj`: là một ký tự đơn định danh cho project.

msgget – tạo thông điệp

Thư viện:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

Prototype của hàm:

```
int msgget (key_t key,int msgflg);
```

Được sử dụng để tạo thông điệp.

Đối số:

msgflg: các bit thấp xác định quyền truy cập đến MQ.

Thao tác trên Message Queue:

Message Queue được sử dụng để gửi và nhận thông điệp. Thông điệp được định nghĩa bởi cấu trúc sau:

```
struct msgbuf {
    long int mtype; /* type of received/sent message */
    char mtext[1]; /* text of the message */
};
```

Thư viện:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

Gửi message:

```
int msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int
msgflg);
```

Giá trị trả về: Thành công 0; thất bại -1

Đối số:

- msqid: id hợp lệ của MQ
- msgp: tham chiếu đến message sẽ được gửi
- msgsz: kích thước của message
- msgflg: hành động sẽ thực hiện nếu hệ thống có giới hạn cho MQ. Các giá trị có thể:
 - IPC_NOWAIT: khi hệ thống đạt đến giới hạn, msgsnd sẽ không gửi, trả lỗi về cho tiến trình gọi
 - 0: msgsnd sẽ khóa cho đến khi giới hạn được gỡ bỏ.

Nhận message:

```
int msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz, long
msgtyp, int msgflg);
```

Giá trị trả về: thành công 0; thất bại -1

Đối số:

- msqid: id hợp lệ của MQ
- msgp: tham chiếu đến message sẽ được nhận
- msgsz: kích thước tối đa của message nhận
- msgtyp: loại của message nhận được
- msgflg: hành động sẽ thực hiện nếu msgtyp nhận được Message không có trong queue. Các giá trị có:
 - IPC_NOWAIT: block message type
 - MSG_EXCEPT and msgtyp>0: trả về message đầu tiên.

Ví dụ: Tạo 2 tiến trình hoàn toàn độc lập, cho phép gửi thông điệp từ tiến trình này sang tiến trình khác:

Ta tạo ra tiến trình writer để gửi dữ liệu:

```
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok(".", 1);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;

    printf("Write Data : ");
    gets(message.mesg_text);

    // msgsnd to send message
    msgsnd(msgid, &message, sizeof(message), 0);
    // display the message
    printf("Data send is : %s \n", message.mesg_text);
    return 0;
}
```

Tạo ra tiến trình reader để nhận dữ liệu.

```
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
```



```

int msgid;

// ftok to generate unique key
key = ftok(".", 1);

// msgget creates a message queue
// and returns identifier
msgid = msgget(key, 0666 | IPC_CREAT);

// msgrcv to receive message
msgrcv(msgid, &message, sizeof(message), 1, 0);

// display the message
printf("Data Received is : %s \n", message.mesg_text);

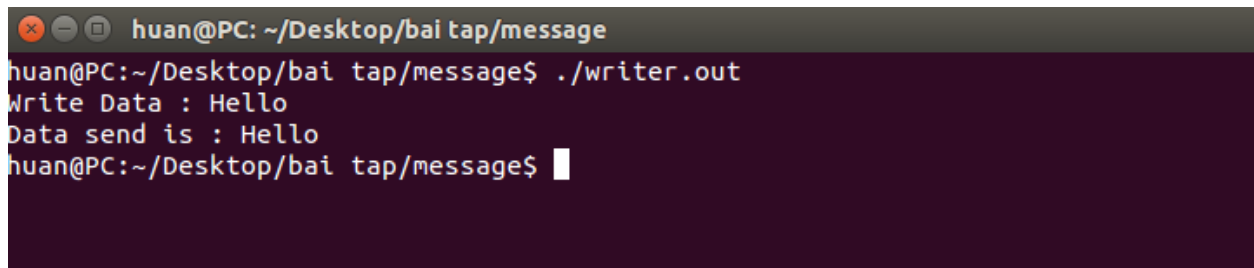
// to destroy the message queue
msgctl(msgid, IPC_RMID, NULL);

return 0;
}

```

Khi thực hiện, compile cho cả 2 tiến trình, tiến trình writer sẽ gửi dữ liệu cho tiến trình reader.

Tiến trình writer:



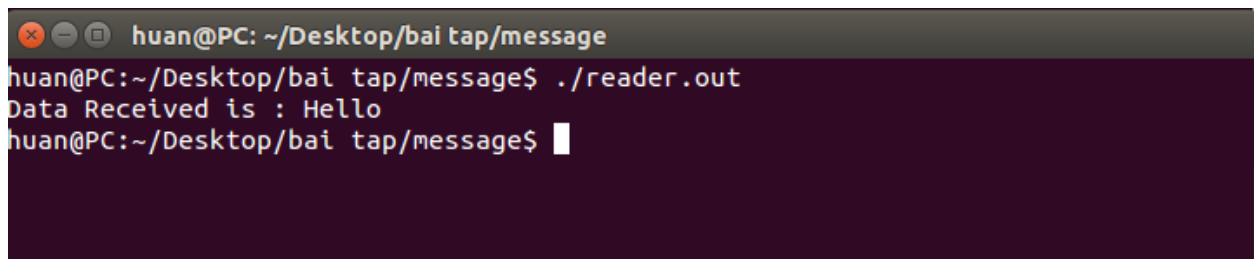
```

huan@PC: ~/Desktop/bai tap/message
huan@PC:~/Desktop/bai tap/message$ ./writer.out
Write Data : Hello
Data send is : Hello
huan@PC:~/Desktop/bai tap/message$

```

Hình 9. 1 Minh họa chạy tiến trình writer

Tiến trình reader:



```

huan@PC: ~/Desktop/bai tap/message
huan@PC:~/Desktop/bai tap/message$ ./reader.out
Data Received is : Hello
huan@PC:~/Desktop/bai tap/message$

```

Hình 9. 2 Minh họa chạy tiến trình reader

Hai tiến trình này sẽ chạy cùng lúc với nhau. Tiến trình reader sẽ chờ tiến trình writer gửi dữ liệu để đọc và xuất ra màn hình.

III. Bài tập thực hành:

Bài 1:

a. Tiến trình cha chuyển đổi số đầu tiên (`argv[1]`) là một số nguyên lớn hơn 3 cho tiến trình con thông qua message queue. Tiến trình con nhận, tính giá trị $n! = 1 * 2 * \dots * n$ và ghi nó vào message queue. Tiến trình cha nhận và xuất dữ liệu ra màn hình. Sử dụng message queue để giải quyết vấn đề.

```
> ./baitap1A.out 4
4! = 24
```

b. Hãy thực hiện 2 tiến trình hoàn toàn riêng biệt, không sử dụng tiến trình cha con. Giải quyết lại vấn đề trên.

Bài 2:

Tiến trình cha đọc hai số nguyên và một thao tác $+$, $-$, x , $/$ và chuyển tất cả cho tiến trình con. Quá trình con tính toán kết quả và trả về cho tiến trình cha. Quá trình cha xuất kết quả

```
> ./baitap2A.out 4 6 +
4 + 6 = 10
```

Hãy giải quyết vấn đề trên với kỹ thuật message queue.

Hãy tách 2 tiến trình trên thành 2 tiến trình hoàn toàn độc lập. Giải quyết lại vấn đề trên.

Bài 3:

Tạo ra 2 tiến trình P0 và P1. Tiến trình P0 đọc từ file nhiều dãy số nguyên liên tiếp (mỗi dãy có thể có số phần tử khác nhau). Sau đó tiến trình P0 lần lượt gửi các dãy sang P1. P1 thực hiện sắp xếp mỗi dãy theo thứ tự tăng dần, đồng thời tính tổng của dãy đó, sau đó gửi dãy kết quả và tổng tương ứng về P0. P0 sẽ ghi lại kết quả vào file sao cho: dãy có tổng nhỏ nhất sẽ được ghi đầu tiên, sau đó các dãy có tổng lớn hơn lần lượt ghi tiếp theo. Hãy thực hiện bài toán với message queue.

BÀI 10: GIAO TIẾP GIỮA CÁC TIẾN TRÌNH – SHARE MEMORY

I. Mục tiêu:

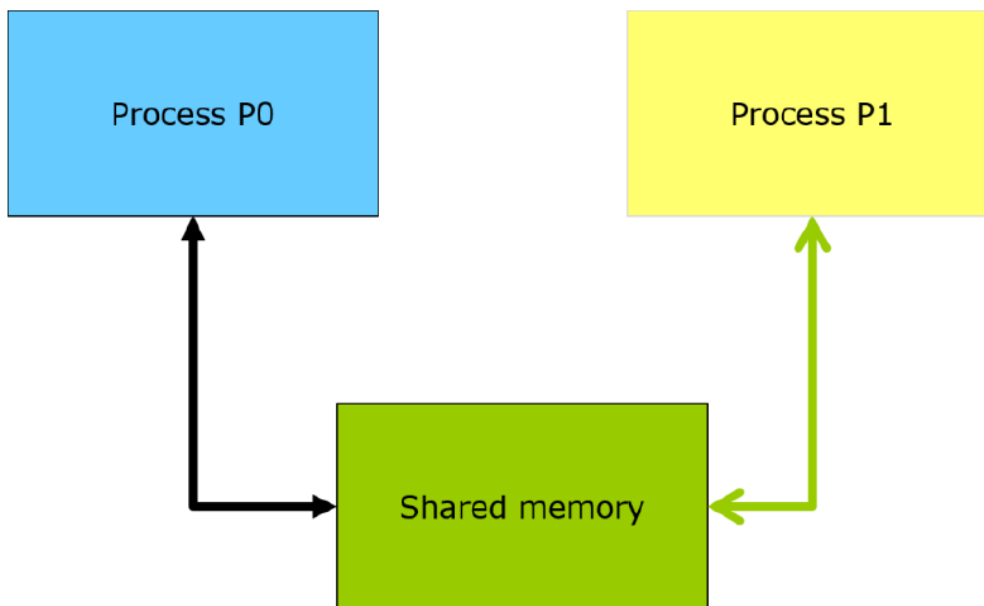
Hướng dẫn sinh viên thực hiện quản lý giao tiếp giữa các tiến trình, sử dụng kỹ thuật share memory – chia sẻ vùng nhớ.

II. Hướng dẫn thực hiện:

Các process sử dụng chung một vùng không gian nhớ chung. Việc tương tác giữa hai process sẽ hoàn toàn do việc đọc/ghi trên vùng nhớ chung này. OS không hề can thiệp vào quá trình này, thế nên shared memory là phương pháp nhanh nhất (nhanh về mặt tốc độ) để các process nói chuyện với nhau. Tuy nhiên, nhược điểm của phương pháp này là các process phải tự quản lý việc đọc ghi dữ liệu, và quản lý việc tranh chấp tài nguyên.

Điều gì sẽ xảy ra nếu cả 2 processes cùng ghi vào vùng nhớ chung? Dữ liệu ghi sau sẽ đè lên dữ liệu ghi trước, và có thể dẫn đến sai lệch dữ liệu. Ngoài ra, khi sử dụng shared memory cũng cần tránh việc trở tới dữ liệu nằm ngoài vùng nhớ chung, vì không gian nhớ của các process là isolate với nhau.

Thông thường, tốt nhất để sử dụng shared memory là process cha, tạo ra vùng nhớ chung, rồi ghi dữ liệu vào đó. Các process con đơn thuần chỉ là đọc dữ liệu từ vùng nhớ chung.



Hình 10. 1 Minh họa các tiến trình sử dụng vùng nhớ dùng chung

Cho phép các tiến trình sử dụng chung vùng nhớ.

- Kích thước tối thiểu: 1 Byte
- Kích thước tối đa: 4 MB
- Số vùng nhớ chia sẻ tối đa: 4096
- Sử dụng:

- Tạo vùng nhớ chia sẻ
- Các tiến trình phải gắn vùng nhớ chia sẻ vào không gian địa chỉ của mình trước khi sử dụng
- Sau khi sử dụng xong, có thể gỡ vùng nhớ ra khỏi không gian địa chỉ của tiến trình

shmget(): sử dụng để tạo SM
 shmctl(): sử dụng để điều khiển SM
 shmat(): sử dụng để gắn SM vào tiến trình
 shmdt(): sử dụng để gỡ SM khỏi tiến trình

Thư viện:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

Prototype của hàm:

```
int shmget (key_t key, int size, int shmflg);
```

Được sử dụng để tạo SM (Share Memory).

Đối số:

- key: key tương ứng với SM
- size: kích thước SM theo Byte
- shmflg: tương tự semflg của semget(). Không có IPC_EXCL

Thư viện:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

Prototype của hàm: Gắn SM vào tiến trình.

```
void *shmat (int shmid, void *shmaddr, int shmflg);
```

Đối số:

- shmid: shmid tương ứng
- shmaddr: địa chỉ gắn SM
- shmflg: SHM_RDONLY hoặc 0

Thư viện:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

Prototype của hàm: Gỡ SM khỏi tiến trình.

```
int shmdt (void *shmaddr);
```

Đối số:

- shmaddr: địa chỉ nhớ gắn SM

Thư viện:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

Prototype của hàm: sử dụng để thay đổi quyền trên SM.

```
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

Đối số:

- shmid: id of SM.
- cmd: sử dụng để xác định hoạt động cụ thể của shmctl. cmd bao gồm các giá trị sau:
- cmd: sử dụng để xác định hoạt động cụ thể của shmctl. cmd bao gồm các giá trị sau:
 - IPC_STAT: trả về giá trị hiện thời của mỗi thành phần trong shmid_ds. Thông tin trả về được lưu giữ trong đối số thứ 3
 - IPC_SET: hạn chế số của những giá trị thành phần của shmid_ds. Các thành phần có thể hiệu chỉnh:shm_perm.uid; shm_perm.gid,...
 - IPC_RMID: xóa SM có id tương ứng.
 - SHM_LOCK: khóa SM
 - SHM_UNLOCK: mở khóa SM

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* operation permission struct
*/
    size_t shm_segsz; /* size of segment in bytes */
    __time_t shm_atime; /* time of last shmat() */
    unsigned long int __unused1;
    __time_t shm_dtime; /* time of last shmdt() */
    unsigned long int __unused2;
    __time_t shm_ctime; /* time of last change by shmctl()
*/
    unsigned long int __unused3;
    __pid_t shm_cpid; /* pid of creator */
    __pid_t shm_lpid; /* pid of last shmop */
    shmatt_t shm_nattch; /* number of current attaches */
    unsigned long int __unused4;
    unsigned long int __unused5;
};
```

Ví dụ 1 : Tiến trình con đọc vào 2 số nguyên từ đối số truyền, ghi vào SM, tiến trình cha thực hiện tính tổng và ghi lại vào SM. Tiến trình con đọc kết quả và xuất ra màn hình.

```
#include <stdio.h>
#include <unistd.h>
#include <limits.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```

#define SIZE 256
int main(int argc, char* argv[])
{
    int *shm, shmid, k,pid;
    key_t key;
    key=ftok(".", "a")==-1)
    shmid = shmget(key, SIZE, IPC_CREAT | 0666);
    shm = (int*) shmat(shmid, 0, 0);
    pid = fork();
    if(pid==0) { // child
        shm[0] = atoi(argv[1]);
        shm[1] = atoi(argv[2]);
        sleep(3);
        printf("%d + %d = %d\n", shm[0], shm[1], shm[2]);
        shmdt((void*) shm);
        shmctl(shmid, IPC_RMID, (struct shmctl) 0);
        return 0;
    }
    else if(pid >0) { // parent
        sleep(1);
        shm[2] = shm[1] + shm[0];
        shmdt((void*) shm);
        sleep(5);
        return 0;
    }
    else { perror("Fork failed."); return 4; }
    return 0;
}

```

Ví dụ 2: viết 2 tiến trình riêng biệt: 1 tiến trình dùng để gửi dữ liệu, 1 tiến trình dùng để nhận dữ liệu và xuất ra màn hình.

Tiến trình writer dùng để gửi dữ liệu:

```

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile", 65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key, 1024, 0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid, (void*) 0, 0);

    printf("Write Data : \n");
}

```

```
    gets(str);

    printf("Data written in memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    return 0;
}
```

Tiến trình reader dùng để nhận dữ liệu:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Data read from memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    // destroy the shared memory
    shmctl(shmid,IPC_RMID,NULL);

    return 0;
}
```

Thực thi 2 tiến trình trên để thấy kết quả.

III. Bài tập thực hành:**Bài 1:**

Hãy thực hiện lại các ví dụ ở phần hướng dẫn thực hiện, kiểm tra kết quả thực hiện.

Bài 2:

Tiến trình con ghi một mảng vào SM_0, với SM[0] chứa số phần tử mảng. Tiến trình cha thực hiện, tính tổng các phần tử của mảng và ghi vào cuối SM_1. Trình con trình nhận và xuất dữ liệu ra màn hình (lưu ý sử dụng 2 SM).

Bài 3:

Viết 1 chương trình, xây dựng 3 tiến trình P0, P1, P2. Tạo ra 1 file “data” file này chứa các dãy số nguyên, dãy số này có thể có số phần tử khác nhau.

- P0 đọc các dãy số, sau đó chép lên share memory.
- P1 truy cập vào share memory, sắp xếp các dãy số theo thứ tự tăng dần, và tính tổng cho dãy số đó.
- Sau khi P1 làm xong, P2 sẽ truy cập vào share memory, sắp xếp các dãy số này sao cho dãy số nào có tổng nhỏ nhất, đặt ở trên, dãy nào có tổng lớn hơn đặt ở dưới.
- Sau khi làm xong, P0 sẽ chép vào file “result”.

BÀI 11: SEMAPHORE

I. Mục tiêu:

Hướng dẫn sinh viên quản lý việc đồng bộ hóa các tiến trình, tiểu trình sử dụng phương pháp semaphore.

II. Hướng dẫn thực hiện:

Semaphores được sử dụng để đồng bộ các tiến trình và tiểu trình. Semaphores được kết hợp với hàng đợi thông điệp và bộ nhớ dùng chung trong các IPC cơ sở trong các hệ thống Unix. Có hai loại semaphores, semaphores System-V truyền thống và semaphores POSIX. Trong bài học này, semaphores POSIX sẽ được giới thiệu.

Có hai loại semaphores POSIX - được đặt tên và vô danh. Như thuật ngữ cho thấy, semaphores được đặt tên có một tên, đó là định dạng / somename. Ký tự đầu tiên là dấu gạch chéo về phía trước, theo sau là một hoặc nhiều ký tự, không có ký tự nào là dấu gạch chéo. Đầu tiên chúng ta sẽ xem xét các semaphores được đặt tên và sau đó là semaphore vô danh.

Các chương trình sử dụng ngữ nghĩa POSIX cần được liên kết với thư viện pthread.

Một biến semaphore sẽ được gán đến các tài nguyên. Khi một tiến trình muốn sử dụng tài nguyên nó sẽ kiểm tra biến semaphore của tài nguyên này. Nếu giá trị của biến này khác 0, tài nguyên đang có sẵn. Nếu giá trị biến này bằng 0, tài nguyên đang được sử dụng.

Thư viện và các lời gọi:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
sem_open
sem_post
sem_wait
sem_trywait
sem_timedwait
sem_getvalue
sem_unlink
sem_init
sem_destroy
```

Lời gọi sem_open

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
sem_t *sem_open (const char *name, int oflag); //form #1
sem_t *sem_open (const char *name, int oflag,
mode_t mode, unsigned int value); //form #2
```

sem_open là lời gọi để bắt đầu một semaphore. sem_open mở một semaphore hiện có hoặc tạo một semaphore mới và mở nó cho các hoạt động tiếp theo. Tham số đầu tiên, name, là tên của semaphore, một chuỗi ký tự hằng số. oflag có thể chứa O_CREAT, trong trường hợp đó, semaphore được tạo nếu nó chưa tồn tại. Nếu cả O_CREAT và O_EXCL được chỉ định, lời gọi sẽ báo lỗi nếu semaphore có tên được chỉ định đã tồn tại. Nếu tham số oflags có thiết lập

O_CREAT, hình thức thứ hai của sem_open phải được sử dụng, có thêm 2 tham số: chế độ mode và giá trị nguyên không âm. Tham số mode chỉ định các quyền cho semaphore, được che dấu bằng umask cho tiến trình, tương tự như mode trong lời gọi hệ thống open() cho các tập tin. Tham số cuối cùng là giá trị khởi tạo cho semaphore. Nếu O_CREAT được chỉ định trong oflag và semaphore đã tồn tại, cả tham số chế độ và giá trị đều bị bỏ qua.

sem_open trả về một con trỏ tới semaphore khi thành công. Con trỏ này phải được sử dụng trong các cuộc gọi tiếp theo cho semaphore. Nếu cuộc gọi thất bại, sem_open trả về SEM_FAILED và errno được đặt lỗi tương ứng.

Trong Linux, các ngữ nghĩa POSIX được tạo trong thư mục /dev/shm. Các semaphores được đặt tên với một tiền tố, sem. theo sau là tên được truyền trong lệnh gọi sem_open.

Lời gọi sem_post

```
#include <semaphore.h>
int sem_post (sem_t *sem);
```

sem_post tăng semaphore. Nó cung cấp hoạt động kiểm tra cho semaphore, tương tự signal(). Nó trả về 0 khi thành công và -1 khi lỗi.

Lời gọi sem_wait / sem_trywait / sem_timedwait

```
#include <semaphore.h>
int sem_wait (sem_t *sem);
sem_timedwait
struct timespec {
time_t tv_sec; /* Seconds */
long tv_nsec; /* Nanoseconds [0 .. 999999999] */
};
```

sem_wait giảm các semaphore được chỉ đến bởi *sem. Nếu giá trị semaphore là khác không, sự sụt giảm xảy ra ngay lập tức. Nếu giá trị semaphore bằng 0, các lời gọi bị chặn cho đến khi semaphore trở nên lớn hơn 0 và việc giảm dần được thực hiện. sem_wait trả về 0 khi thành công và -1 khi lỗi. Trong trường hợp có lỗi, giá trị semaphore được giữ nguyên và errno được đặt thành số lỗi thích hợp. sem_wait cung cấp lệnh gọi cho hoạt động P cho semaphore.

sem_trywait giống như lời gọi sem_wait, ngoại trừ: nếu giá trị semaphore bằng 0, nó không chặn mà trả về ngay lập tức với errno được đặt thành EAGAIN.

sem_timedwait cũng giống như cuộc gọi sem_wait, ngoại trừ: có bộ đếm thời gian được chỉ định với con trỏ, abs_timeout. Nếu giá trị semaphore lớn hơn 0, nó sẽ bị giảm và giá trị thời gian chờ được trả bởi abs_timeout không được sử dụng. Trong trường hợp đó, lời gọi hoạt động giống như lời gọi sem_wait. Nếu giá trị semaphore bằng 0, các lời gọi bị chặn, thời lượng chặn tối đa là thời gian cho đến khi bộ hẹn giờ tắt. Nếu giá trị semaphore trở nên lớn hơn 0 trong khoảng thời gian chặn, semaphore bị giảm ngay lập tức và lời gọi được đánh thức trở lại. Nếu không, bộ hẹn giờ sẽ tắt và cuộc gọi trở lại với errno được đặt thành ETIMEDOUT. Bộ định thời được chỉ định trong struct timespec{ };

sem_getvalue

```
#include <semaphore.h>
```

```
int sem_getvalue (sem_t *sem, int *sval);
```

sem_getvalue nhận giá trị của semaphore được chỉ bởi sem. Giá trị được trả về trong số nguyên được chỉ bởi sval. Nó trả về 0 khi thành công và -1 khi lỗi, với lỗi không cho biết lỗi thực tế.

sem_unlink

```
#include <semaphore.h>
int sem_unlink (const char *name);
```

sem_unlink xoá semaphore có tên đang liên kết với name.

POSIX Unnamed Semaphore calls

Trong các đề cập ở trên, các semaphores là cục bộ của một quá trình; chúng chỉ được sử dụng bởi chủ đề của nó. Không có quá trình khác sử dụng chúng. Vì vậy, có vẻ như lãng phí nỗ lực để có tên semaphore trên toàn hệ thống và sử dụng các cuộc gọi như sem_open. Có những semaphores không tên POSIX có thể làm những gì chúng ta cần một cách đơn giản và hiệu quả hơn nhiều. Đầu tiên, hệ thống gọi,

sem_init

```
#include <semaphore.h>
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

sem_init tương đương với **sem_open** và dùng cho các semaphores không tên. Cần định nghĩa một biến kiểu sem_t và chuyển đổi số con trỏ của nó là sem trong lời gọi. Hoặc, có thể định nghĩa một con trỏ và cấp phát bộ nhớ động bằng cách sử dụng lời gọi malloc. sem_init khởi tạo semaphore được trỏ bởi sem với giá trị số nguyên value. Đối số thứ hai được pshared cho biết semaphore này được dùng chia sẻ giữa các tiểu trình của một tiến trình hay giữa các tiến trình. Nếu pshared có giá trị 0, semaphore được chia sẻ giữa các tiểu trình của một tiến trình. Semaphore nên được đặt tại một nơi mà mọi tiểu trình có thể tìm thấy. Nếu pshared có giá trị khác 0, semaphore được chia sẻ bởi các tiến trình. Trong trường hợp đó, semaphore phải được đặt trong một phân đoạn bộ nhớ dùng chung được gắn với các tiến trình liên quan.

sem_destroy

```
#include <semaphore.h>
int sem_destroy (sem_t *sem);
```

sem_destroy xoá semaphore không tên đang liên kết với name.

Ví dụ:

```
// C program to demonstrate working of Semaphores
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex;
void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered...\n");
```

```
//critical section
    sleep(4);
//signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}
int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

Giải thuật Monte Carlo để tính số pi

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#define SEED 35791246
main(int argc, char* argv)
{
    int niter=0;
    double x,y;
    int i,count=0; /* # of points in the 1st quadrant of unit circle
*/
    double z;
    double pi;
    printf("Enter the number of iterations used to estimate pi: ");
    scanf("%d",&niter);
/* initialize random numbers */
    srand(SEED);
    count=0;
    for ( i=0; i<niter; i++) {
        x = (double)rand()/RAND_MAX;
        y = (double)rand()/RAND_MAX;
        z = x*x+y*y;
        if (z<=1) count++;
    }
    pi=(double)count/niter*4;
    printf("# of trials= %d , estimate of pi is %g \n",niter,pi);
}
```

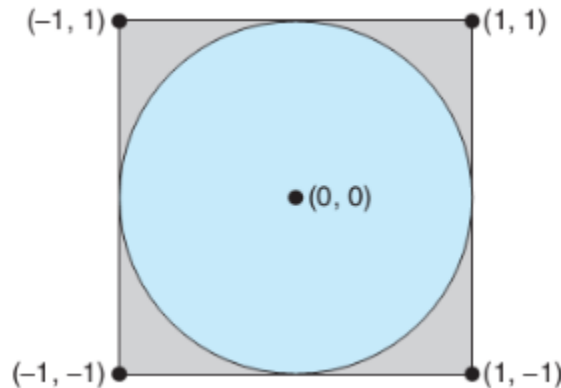
III. Bài tập thực hành:

Bài 1:

Tạo 2 tiểu trình con, một tiểu trình in ra các số lẻ từ 1 đến 11, một tiểu trình in ra số chẵn từ 2 đến 10. Hãy sử dụng semaphore sau cho màn hình in ra dãy số đúng theo thứ tự từ 1 đến 11.

Bài 2:

Một cách giá trị π khá thú vị là sử dụng kỹ thuật Monte Carlo, liên quan đến ngẫu nhiên. Kỹ thuật này hoạt động như sau: Giả sử bạn có một vòng tròn bán kính là 1 nội tiếp trong một hình vuông cạnh là 2, như thể hiện trong hình sau:



Hình 11. 1 Minh họa kỹ thuật Monte Carlo để tính số Pi

- Đầu tiên, tạo một chuỗi các điểm ngẫu nhiên dưới dạng tọa độ (x, y) đơn giản. Những điểm này phải nằm trong tọa độ Descartes bị ràng buộc hình vuông. Trong tổng số điểm ngẫu nhiên được tạo, một số sẽ xảy ra trong vòng tròn.
- Tiếp theo, ước tính π bằng cách thực hiện phép tính sau: $\pi = 4 \times (\text{số điểm trong vòng tròn}) / (\text{tổng số điểm})$.

Hãy viết một phiên bản đa luồng của thuật toán này để tạo ra một tiểu trình riêng biệt sinh ra một số lượng điểm ngẫu nhiên; sau đó tính số lượng điểm nằm trong hình và lưu trữ kết quả đó trong một biến toàn cục. Khi tiểu trình này kết thúc, tiến trình cha sẽ tính toán và xuất giá trị ước tính của π . Hãy đánh giá độ chính xác của số π với số lượng điểm ngẫu nhiên được tạo ra. Theo nguyên tắc, số lượng điểm càng lớn, giá trị tính càng tiến gần π .

Áp dụng semaphore và ví dụ ở trên để giải quyết bài toán trên.

Bài 3:

Một hãng sản xuất xe ô tô có các bộ phận hoạt động song song:

Bộ phận sản xuất khung xe

```
void SXKhung() {
    printf("San xuat khung");
}
```

Bộ phận sản xuất bánh xe

```
void SXBanhXe() {
    printf("San xuat banh xe");
}
```

Bộ phận lắp ráp: Sau khi có đủ 1 khung và 4 bánh thì tiến hành lắp ráp.

```
void LapRapXe() {
    printf("Lap rap xe");
}
```

}

Hãy đồng bộ hoạt động của các bộ phận trên theo nguyên tắc: tại mỗi thời điểm chỉ cho phép sản xuất 1 khung xe, cần chờ đủ 4 bánh xe để gắn vào khung xe hiện tại này trước khi sản xuất một khung xe khác.

BÀI 12: SEMAPHORE (tiếp theo)

I. Mục tiêu:

Sinh viên áp dụng các kiến thức đã học, giải quyết các vấn đề áp dụng sự đồng bộ hóa của các tiến trình, sử dụng semaphore.

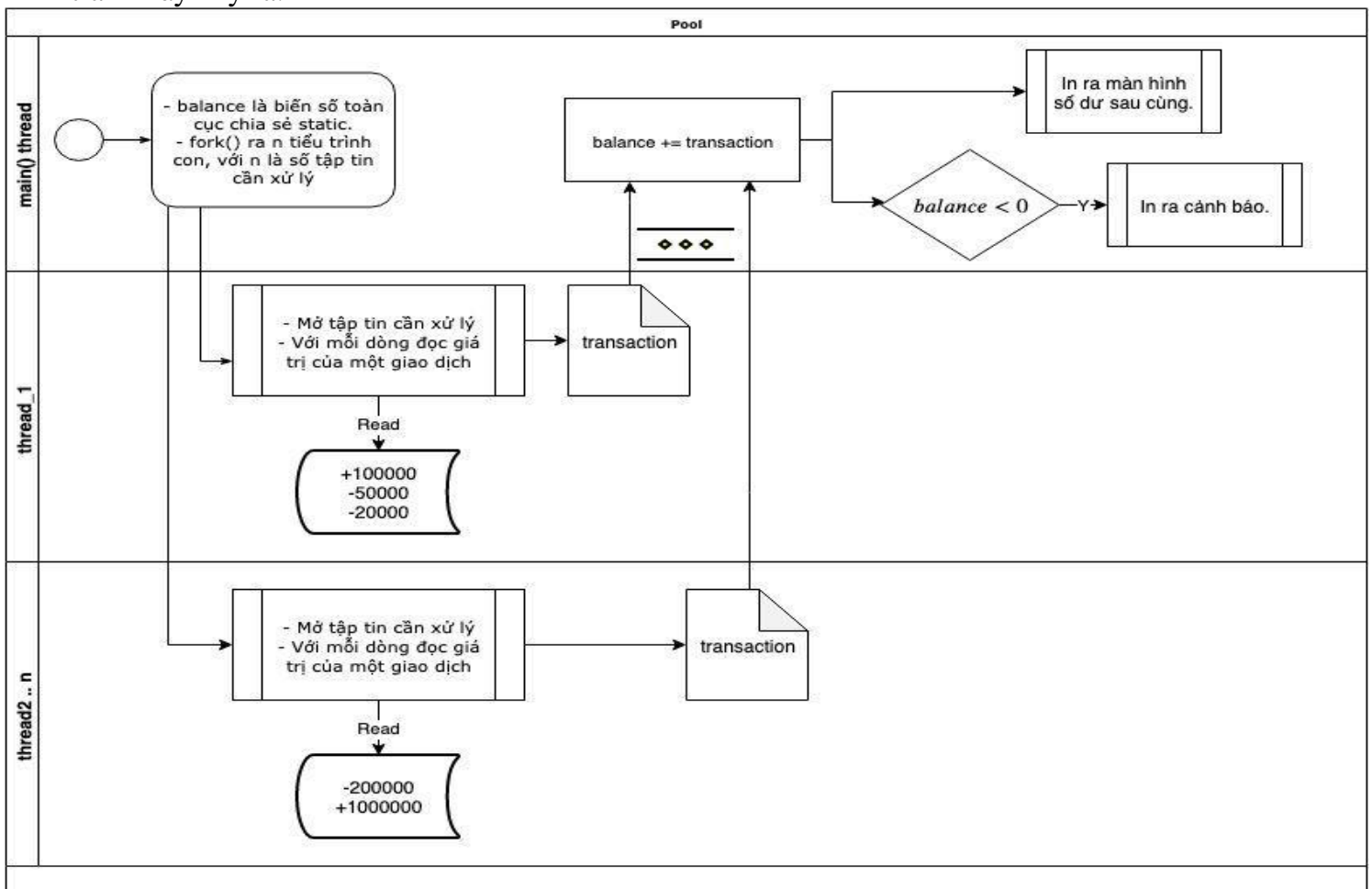
II. Hướng dẫn thực hiện:

Xem lại các hướng dẫn ở bài 11 để áp dụng giải quyết các vấn đề ở phần bài tập thực hành.

III. Bài tập thực hành:

Bài 1:

Tình trạng cạnh tranh (race condition) có thể xuất hiện trong nhiều hệ thống máy tính. Hãy xem xét một hệ thống ngân hàng duy trì số dư tài khoản với hai hàm thực thi: deposit(số tiền) và withdraw(số tiền). Hai hàm này được truyền vào số tiền sẽ được gửi hoặc rút từ số dư tài khoản ngân hàng. Giả sử rằng người chồng và người vợ chia sẻ một tài khoản ngân hàng. Một cách đồng thời, người chồng gọi hàm withdraw() và người vợ gọi hàm deposit(). Mô tả làm thế nào một tình trạng cạnh tranh có thể xảy đến và làm cách nào để ngăn chặn tình trạng cạnh tranh này xảy ra.



Hình 12. 1 Minh họa xử lý bài toán

Nội dung tập tin a.txt:

```
+10      + thể hiện gửi tiền vào.
-5        - thể hiện rút tiền ra.
+20
```

Nội dung tập tin b.txt:

```
+20      + thể hiện gửi tiền vào.
-8        - thể hiện rút tiền ra.
-10
```

Nội dung tập tin c.txt:

```
-20      + thể hiện gửi tiền vào.
+50      - thể hiện rút tiền ra.
```

Khi thực hiện:

```
>./bank.out a.txt b.txt
Final Balance = 27
>./bank c.txt
Warning: Balance under 0.00
Final Balance = 30
```

Bài 2:

Tình trạng cạnh tranh (race condition) có thể xuất hiện trong nhiều hệ thống máy tính. Hãy xem xét một hệ thống đấu giá trực tuyến trong đó giá đấu cao nhất hiện thời cho mỗi mặt hàng phải được duy trì. Một người muốn đặt giá đấu cho một mặt hàng sẽ gọi hàm bid(số_tiền), hàm này sẽ so sánh số tiền đang được đặt giá đấu với giá đấu cao nhất hiện tại. Nếu số tiền vượt quá giá đấu cao nhất hiện thời, giá đấu cao nhất sẽ được đặt thành số tiền mới. Điều này được minh họa dưới đây:

Giả sử rằng có nhiều tập tin mà mỗi tập tin (đại diện cho một người) chứa các số tiền của mỗi lần đặt lệnh. Chương trình cần tạo ra nhiều tiểu trình, mỗi tiểu trình sẽ đọc một tập tin và các tiểu trình đồng thời đặt lệnh đấu giá, có các tình huống sau có thể xảy ra:

- Giá đấu cao hơn giá hiện tại (giá hiện tại là biến số chia sẻ và khởi tạo là 0): giá hiện tại sẽ được cập nhật và tên người thắng cũng sẽ được cập nhật (là tên của tập tin chứa giá đấu đang được xử lý).
- Giá đấu thấp hơn hay bằng giá hiện tại: không cập nhật gì cả.
- Giá đấu cao hơn giá hiện tại nhưng nếu giá hiện tại cao nhất cũng là của người đấu này thì tiểu trình này bị chặn lại cho đến khi người thắng được cập nhật. (Không ai tự bỏ giá cao hơn giá vừa bỏ liền trước đó).

Nội dung tập tin andy.txt :

```
10      Giá đấu lần đầu.
50      Giá đấu tiếp theo.
60      Giả sử rằng giá đấu luôn dương và tăng dần.
100     Hàng cuối cùng thể hiện giá cao nhất mà người này có thể đấu.
```

Nội dung tập tin ben.txt :

```
15
40
```

Lời gọi:


```
>./bid.out andy.txt ben.txt
Value on bid:
Andy 10
Ben 15
Andy 50
Ben 40
Andy 60
The winner is Andy with value 60
```

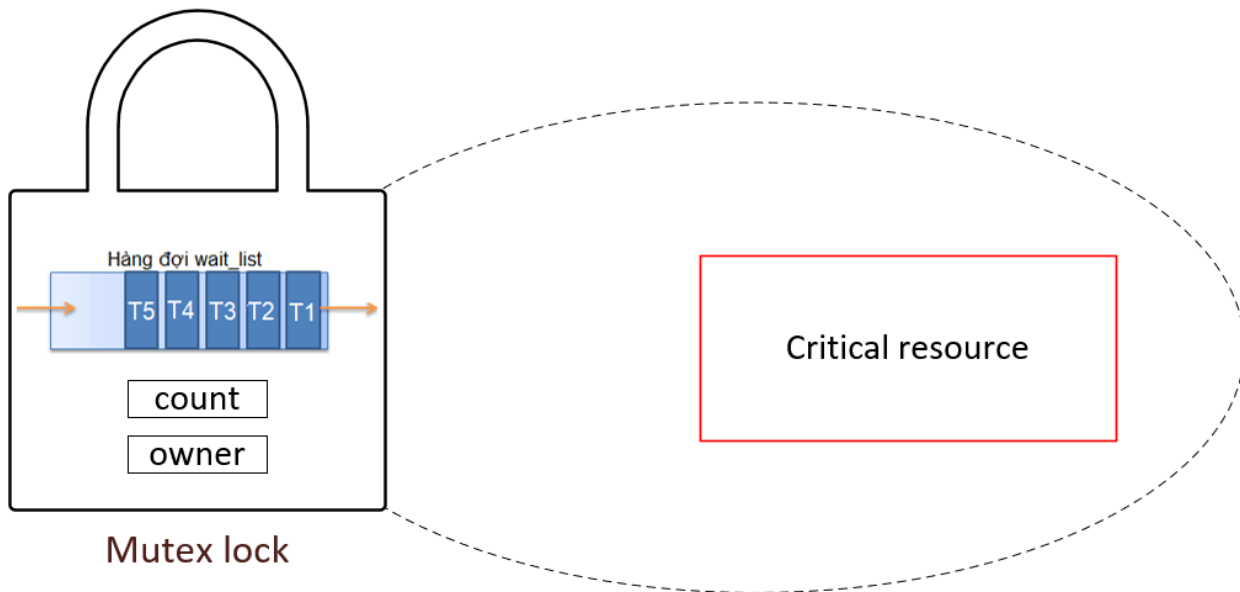
BÀI 13: MUTEX LOCK

I. Mục tiêu:

Hướng dẫn sinh viên thực hiện tránh xung đột trong truy cập critical resource, sử dụng phương pháp mutex lock.

II. Hướng dẫn thực hiện:

Mutex lock là một cấu trúc dữ liệu, được Linux kernel xây dựng theo nguyên tắc mutual exclusion, dùng để ngăn chặn race condition xảy ra trên các cấu trúc dữ liệu khác. Nói nôm na, mutex lock đảm bảo rằng: tại một thời điểm bất kì, chỉ có tối đa một thread truy cập vào critical resource.



Hình 13. 1 Mutex lock giống như một ổ khóa dùng để bảo vệ critical resource

Mutex lock có cấu tạo như thế nào?

Mutex lock gồm 3 thành phần chính: biến count, biến owner và hàng đợi wait_list. Dựa vào đó, Linux kernel đã xây dựng cấu trúc mutex để biểu diễn một mutex lock.

```
struct mutex {
    /*
     * Biến count lưu trạng thái của mutex lock, cũng như trạng thái của
     * critical resource.
     * - Nếu count = 1, thì trạng thái của mutex lock đang là UNLOCKED,
     *   còn trạng thái của critical resource đang là AVAILABLE.
     * - Nếu count < 1, thì trạng thái của mutex lock đang là LOCKED,
     *   còn trạng thái của critical resource đang là UNAVAILABLE.
     *   count = 0: không có thread nào đang phải đợi để được
     *             sử dụng critical resource.
     *   count < 0: có thread đang phải đợi để được sử dụng
     *             critical resource.
     */
};
```

```

*/
atomic_t          count;

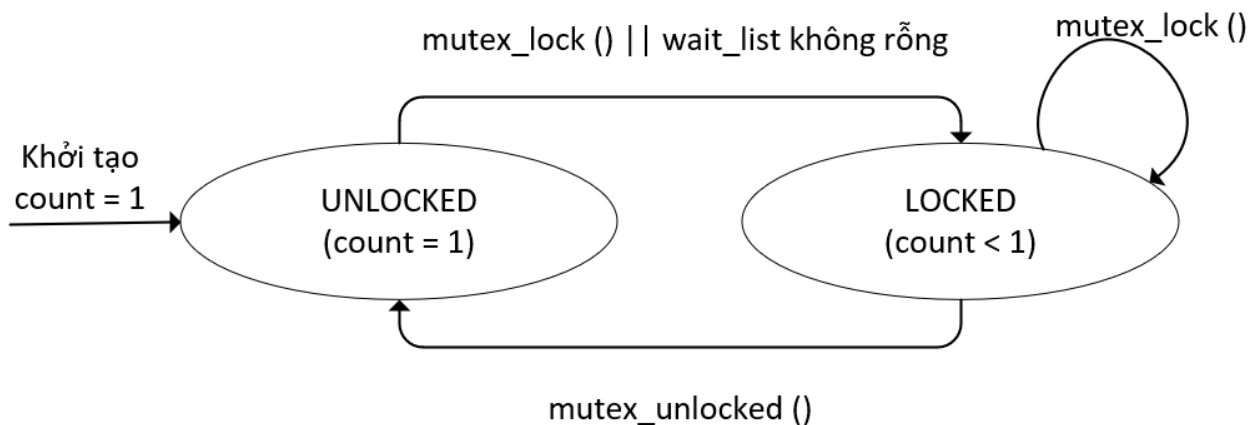
/*
 * Hàng đợi wait_list có thể bị nhiều thread truy cập đồng thời.
 * wait_lock là một spinlock bảo vệ wait_list
 */
spinlock_t        wait_lock;

/*
 * Hàng đợi wait_list chứa danh sách các thread đang phải đợi để
 * chiếm được mutex lock, cũng chính là danh sách các thread
 * đang phải đợi để được sử dụng critical resource.
 */
struct list_head   wait_list;

/*
 * Biến owner mô tả thread đang chiếm dụng mutex lock, cũng chính là
 * thread đang sử dụng critical resource.
 */
struct task_struct *owner;
...
};

```

Mutex lock hoạt động ra sao?



Hình 13. 2 Sơ đồ biểu diễn các trạng thái của một mutex lock

Khi **count** đang bằng 1 (tức là mutex lock đang ở trạng thái **UNLOCKED**), nếu một thread gọi hàm `mutex_lock`, thì:

Biến **count** bị giảm thành 0 (tức là mutex lock bị chuyển sang trạng thái **LOCKED**). Ta nói rằng thread đã khóa mutex lock lại.

Biến **owner** được thiết lập bằng thread đó. Ta nói rằng thread đã chiếm dụng mutex lock.

CPU bắt đầu thực thi critical section của thread (nói theo ngôn ngữ của CPU), hay thread đang sử dụng critical resource (nói theo ngôn ngữ của Linux kernel).

Khi **count** đang nhỏ hơn 1 (tức là đang ở trạng thái **LOCKED**), nếu một thread gọi hàm **mutex_lock**, thì:

- Biến **count** sẽ giảm xuống 1 đơn vị.
- CPU tạm ngừng thực thi thread này rồi chuyển sang thực thi thread khác (nói theo ngôn ngữ của CPU). Hay nói theo ngôn ngữ của Linux kernel, thread được thêm vào hàng đợi **wait_list** và sẽ đi ngủ, sau đó Linux kernel sẽ lập lịch cho thread khác. Do đó, ta nói rằng, mutex lock áp dụng cơ chế **sleep-waiting**, tức là mutex lock thuộc loại **sleep lock**, trái với spinlock thuộc loại busy lock.

Khi **count** đang nhỏ hơn 1 (tức là đang ở trạng thái **LOCKED**), nếu một thread A gọi hàm **mutex_unlock**, thì:

- Biến **owner** được thiết lập thành NULL. Ta nói rằng, thread A đã giải phóng mutex lock.
- Biến **count** sẽ được tăng thành 1 hoặc được thiết lập bằng 1 (tức là mutex lock chuyển sang trạng thái **UNLOCKED**). Ta nói rằng thread A đã mở khóa mutex lock.
- Nếu hàng đợi **wait_list** không rỗng và giả sử thread B nằm ở đầu hàng đợi, CPU sẽ chuyển sang thực thi thread B (nói theo ngôn ngữ của CPU). Hay nói theo ngôn ngữ của Linux kernel, Linux kernel sẽ đánh thức thread B dậy. Sau khi thức dậy, thread B sẽ chuyển mutex lock sang trạng thái **LOCKED** (thay đổi biến **count** thành -1 nếu vẫn còn các thread khác đang đợi, hoặc thành 0 nếu không còn thread nào đang đợi). Sau đó, thread B chiếm lấy mutex lock rồi bắt đầu sử dụng critical resource.

Mutex lock bảo vệ critical resource như thế nào?

Trong khi lập trình device driver, ta đặt hàm **mutex_lock** và **mutex_unlock** lần lượt vào trước và sau critical section của mỗi thread. Việc làm này giúp bảo vệ critical resource. Để thấy được điều này, ta xét ví dụ sau. Giả sử, hệ thống có kernel thread A và B được thực thi riêng biệt trên 2 lõi CPU0 và CPU1. Cả 2 thread đều có nhu cầu sử dụng critical resource R, và tài nguyên R được bảo vệ bằng mutex lock M. Xét 2 trường hợp sau:

- **Trường hợp 1:** A muốn truy cập R trong khi B đang sử dụng R.
 - Trước khi thực thi các lệnh trong critical section của thread A, CPU0 sẽ thực thi hàm **mutex_lock** và thấy rằng M đang ở trạng thái **LOCKED**. Khi đó, CPU0 sẽ dừng thực thi thread A rồi chuyển sang thực thi một thread C nào đó.
 - Sau khi thực thi xong critical section của thread B, CPU1 thực thi tiếp hàm **mutex_unlock** để chuyển M sang trạng thái **UNLOCKED**. Lúc này, thread A sẽ chiếm lấy M và CPU0 tiếp tục thực thi thread A.
- **Trường hợp 2:** cả A và B đồng thời muốn truy cập R.
 - Khi đó, cả 2 thread đồng thời thực thi hàm **mutex_lock**. Tuy nhiên, do hàm **mutex_lock** dùng thao tác atomic để thay đổi biến **count**, nên chỉ có một trong hai thread chiếm được M.
 - Thread nào chiếm được M trước thì sẽ sử dụng R trước. Thread nào không chiếm được M thì sẽ đi ngủ cho đến khi thread đầu tiên sử dụng xong R.

Như vậy, tại bất cứ thời điểm nào, tối đa chỉ có một thread được phép chiếm dụng mutex lock, đồng nghĩa với việc, tối đa chỉ có một thread được phép sử dụng critical resource. Do đó, race condition sẽ không xảy ra và critical resource được bảo vệ.

Sử dụng mutex clock:

Để sử dụng mutex, trước hết chúng ta phải khai báo và khởi tạo mutex. Trong Posix thread, biến mutex là kiểu dữ liệu có dạng *pthread_mutex_t* và có thể được khởi tạo tĩnh sử dụng macro *PTHREAD_MUTEX_INITIALIZER* hoặc khởi tạo động lúc runtime.

Khởi tạo tĩnh (statically allocation)

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Khởi tạo động (dynamically initializing)

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attr);
/*Trả về 0 nếu thành công, hoặc 1 số dương mã lỗi*/
```

Trong cách khởi tạo tĩnh, macro *PTHREAD_MUTEX_INITIALIZER* dùng để khởi tạo một mutex với các thuộc tính (thread attribute) mặc định. Trong khi hàm *pthread_mutex_init()* trong cách khởi tạo động cho phép khởi tạo và thiết lập thuộc tính cho mutex. Nếu không cần quan tâm đến thuộc tính của thread, ta có thể truyền NULL vào đối số *pthread_mutexattr_t *attr*. Khi khởi tạo động mutex bằng hàm *pthread_mutex_init()*, ta cần phải hủy mutex đó nếu không cần sử dụng nữa bằng hàm *pthread_mutex_destroy()* có prototype như sau (khởi tạo tĩnh bằng macro *PTHREAD_MUTEX_INITIALIZER* không cần destroy mutex):

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
/*Return 0 nếu thành công, hoặc một số dương mã lỗi nếu không
thành công*/
```

Lock/unlock mutex

Sau khi khởi tạo, mutex được khóa và mở khóa bởi 2 hàm sau đây:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
/*Trả về 0 nếu thành công, hoặc 1 số dương mã lỗi khi xảy ra
lỗi*/
```

Để khóa 1 mutex, ta truyền địa chỉ của mutex đó vào hàm *pthread_mutex_lock()*. Nếu một mutex đang ở trạng thái unlock, hàm này sẽ khóa mutex đó và return. Nếu mutex đó đã bị khóa bởi thread khác, hàm này sẽ bị lock cho đến khi mutex được mở. Nếu một thread khóa một mutex mà chính nó đang giữ khóa thì sẽ xảy ra deadlock (thread rơi vào trạng thái chờ vô hạn).

Ngoài ra, chuẩn Posix còn cung cấp hai hàm lock mutex sau đây:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
và
int pthread_mutex_timedlock(pthread_mutex_t *restrict
mutex, const struct timespec *restrict abs_timeout);
```

Hàm *pthread_mutex_trylock()* hoạt động khác *pthread_mutex_lock()* ở chỗ: nếu mutex đang bị khóa, nó sẽ không lock thread mà sẽ return ngay lập tức với mã lỗi là *EBUSY*. Còn

hàm `pthread_mutex_timedlock()` được thêm vào đối số `abs_timeout` để thiết lập thời gian tối đa thread có thể chờ; nếu sau khoảng thời gian "`abs_timeout`" mà thread đó chưa sở hữu được mutex, nó sẽ return và trả về mã lỗi `ETIMEDOUT`.

Ví dụ:

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

#define MAX_THREAD      2

pthread_t tid[MAX_THREAD];
/*Counter la bien toan cuc duoc 2 thread su dung*/
int counter;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;    //khai bao mutex

void *threadFunc(void *argv)
{
    pthread_mutex_lock(&mutex);
    counter += 1;
    printf("Thread %d has started\n", counter);

    sleep(1);
    printf("Thread %d has finished\n", counter);
    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}

int main(void)
{
    int i = 0;
    int ret = 0;

    for (i = 0; i < MAX_THREAD; i++)
    {
        ret = pthread_create(&(tid[i]), NULL, threadFunc, NULL);
        if (ret != 0)
        {
            printf("Thread [%d] created error\n", i);
        }
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

III. Bài tập thực hành:

Bài 1:

Hãy thực hiện lại ví dụ ở phần hướng dẫn thực hiện. Rút ra nhận xét về cách mutex lock thực hiện.

Bài 2:

Mỗi hãng máy bay sẽ có một số lượng vé nhất định, các đại lý sẽ truy cập đến hệ thống của hãng để đặt mua vé máy bay.

Hãy xây dựng một chương trình quản lý vé máy bay của hãng. Khi thực hiện chương trình này, sẽ cho người dùng nhập vào số lượng vé máy bay trong ngày của hãng. Dữ liệu này sẽ được nhập vào file: sove.txt.

Hãy xây dựng chương trình cho 2 đại lý: mỗi đại lý khi chạy sẽ đọc từ file: daily1.txt và daily2.txt. Khi chạy, mỗi đại lý sẽ mua vé từ hãng, mỗi lần mua vé thì file sove.txt của hãng sẽ bị giảm theo số lượng vé của mỗi đại lý đặt. Nếu đặt mua thành công, sẽ hiện thông báo đặt mua thành công với số vé, nếu hết vé phải báo là đã hết vé, không thể mua tiếp. Mỗi lần mua xong, bắt buộc mỗi đại lý phải tạm nghỉ 5s.

Nội dung tập tin daily1.txt:

2

4

5

10

Các số trên là số vé mà khách đến đại lý để mua ở các lần khác nhau.

Nội dung tập tin daily2.txt:

5

3

4

Hãy sử dụng công cụ share memory, mutex lock để giải quyết bài tập trên.

BÀI 14: BÀI TẬP TỔNG HỢP

I. Mục tiêu:

Áp dụng các kiến thức đã học để giải quyết các vấn đề.

II. Hướng dẫn thực hiện:

Đọc lại các kiến thức ở các bài trước để giải quyết các bài tập thực hành.

III. Bài tập thực hành:

Bài 1: Bài toán thang máy:

Một thang máy di chuyển qua các tầng khác nhau, khi có người ở 1 tầng nào đó bấm vào nút gọi thang máy, có 2 nút: lên và xuống. Thang máy sẽ di chuyển đến cho người đó chọn. Khi thang máy tới tầng này, người dùng sẽ bấm nút chọn tầng để thang máy di chuyển đến tầng muốn đến. Trong lúc vận hành sẽ có nhiều người cùng bấm nút gọi thang máy. Thang máy sẽ ưu tiên đang đi lên hoặc đang đi xuống để chọn người gọi đi lên hoặc đi xuống, và sẽ đón người ở gần nó nhất.

Hãy viết 1 chương trình tạo ra nhiều tiến trình, mỗi tiến trình ứng với 1 tầng, các tiến trình này sử dụng vùng nhớ dùng chung. Thực hiện quản lý vùng nhớ dùng chung này để các tầng có thể gọi thang máy và tránh xung đột với nhau.

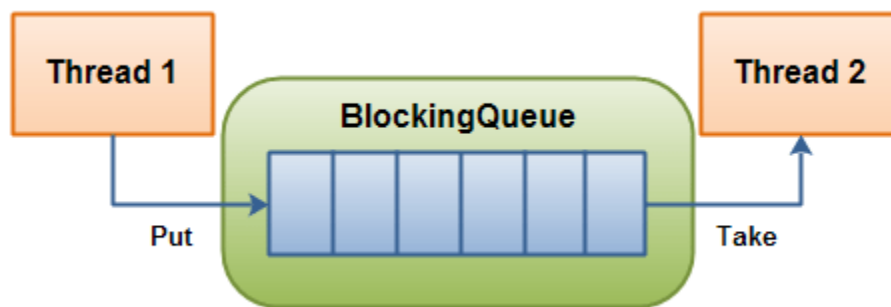
Bài 2: Bài toán producer – consumer:

Vấn đề mô tả hai đối tượng nhà sản xuất (Producer) và người tiêu dùng (Consumer), cả hai cùng chia sẻ một bộ đệm có kích thước cố định được sử dụng như một hàng đợi (queue).

Producer: công việc của nhà sản xuất là tạo dữ liệu, đưa nó vào bộ đệm và bắt đầu lại.

Consumer: công việc người tiêu dùng là tiêu thụ dữ liệu (nghĩa là loại bỏ nó khỏi bộ đệm), từng phần một và xử lý nó. Consumer và Producer hoạt động song song với nhau.

Vấn đề là đảm bảo rằng nhà sản xuất không thể thêm dữ liệu vào bộ đệm nếu nó đầy và người tiêu dùng không thể xóa dữ liệu khỏi bộ đệm trống, đồng thời đảm bảo an toàn cho luồng (thread-safe).



Hình 14. 1 Minh họa bài toán Producer - Consumer

Hãy vận dụng các kỹ thuật đã học để giải quyết bài toán này.

BÀI 15: ÔN TẬP – KIỂM TRA

Sinh viên thực hiện một số đề kiểm tra sau đây. Thời gian thực hiện là 45 phút.

Bài 1:

Sinh viên thực hiện theo yêu cầu như sau:

Hãy thực hiện một chương trình nhận 2 giá trị số nguyên dương (tạm gọi là n , point) từ đối số truyền vào.

Hãy thực hiện 1 vòng lặp thực hiện n lần, mỗi lần lặp sinh ra một số ngẫu nhiên từ 0 – 100. Các số ngẫu nhiên này sẽ thay đổi liên tục. Sau đó hãy tạo 3 tiểu trình. (2.5đ)

- 1 tiểu trình kiểm tra xem số vừa sinh ra có phải là số chẵn và lớn hơn số point được truyền vào không. Nếu thỏa mãn hãy chép vào mảng `chan[]`. (2.5đ)
- 1 tiểu trình kiểm tra xem số vừa sinh ra có phải là số lẻ và nhỏ hơn số point được truyền vào không. Nếu thỏa mãn hãy chép vào mảng `le[]`. (2.5đ)
- Sau khi 2 tiểu trình trên hoàn thành. Hãy thực hiện 1 tiểu trình để in ra 2 mảng kết quả vừa thực hiện. (2.5đ)

Với 2 mảng `chan[]` và `le[]` là 2 mảng toàn cục.

Bài 2:

Hãy thực hiện một chương trình nhận 1 giá trị số nguyên dương n từ đối số truyền vào.

Hãy thực hiện thực hiện 1 vòng lặp thực hiện n lần, mỗi lần lặp sinh ra một số ngẫu nhiên từ 0 – 100. Các số ngẫu nhiên này sẽ thay đổi liên tục. Sau đó hãy tạo 3 tiểu trình. (2.5đ)

- 1 tiểu trình kiểm tra xem số vừa sinh ra có phải là số nguyên tố hay không? Nếu là số nguyên tố thì ghi vào 1 mảng `nguyento[]`. (2.5đ)
- 1 tiểu trình kiểm tra xem số vừa sinh ra có phải là chẵn hay không? Nếu là số chẵn thì ghi vào mảng `chan[]`. (2.5đ)
- Sau khi 2 tiểu trình trên hoàn thành. Hãy thực hiện 1 tiểu trình để in ra 2 mảng kết quả vừa thực hiện. (2.5đ)

Với 2 mảng `nguyento[]` và `chan[]` là 2 mảng toàn cục.

TÀI LIỆU THAM KHẢO

- [1] Jyotirmay Patel, “Lab Manual Operating System”. Department of Computer Science & Engineer, Meetrut Institute of Technology.
- [2] G. Sunitha, “Operating Systems Lab Manual”. Department of Computer Science & Engineer, Institute of Aeronautical Engineer.

DANH MỤC HÌNH ẢNH

Bài 1:

Hình 1. 1 Sơ đồ minh họa hệ thống tập tin trên Linux	3
Hình 1. 2 Minh họa câu lệnh ls	6
Hình 1. 3 Minh họa câu lệnh ls -l	6

Bài 2:

Hình 2. 1 Minh họa cách gọi chương trình	13
Hình 2. 2 Sơ đồ quá trình biên dịch của file	13
Hình 2. 3 Minh họa cách chạy chương trình	14
Hình 2. 4 Mô hình thư viện liên kết tĩnh và động	15
Hình 2. 5 Minh họa kiểm tra thư viện vừa tạo	16
Hình 2. 6 Minh họa chạy chương trình khi không gọi thư viện	17
Hình 2. 7 Minh họa chạy chương trình khi có gọi thư viện	17
Hình 2. 8 Minh họa chạy chương trình khi gọi thư viện liên kết động	18

Bài 3:

Hình 3. 1 Minh họa chạy chương trình lấy ProcessID	21
--	----

Bài 4:

Hình 4. 1 Minh họa cơ chế vận hành các hàm exec	28
---	----

Bài 6:

Hình 6. 1 Minh họa thuật toán Merge Sort	40
Hình 6. 2 Minh họa thuật toán sắp xếp quick sort	41
Hình 6. 3 Minh họa Merge Sort với 2 Thread	42

Bài 8:

Hình 8. 1 Liên lạc qua pipe (đường ống)	46
---	----

Bài 9:

Hình 9. 1 Minh họa chạy tiến trình writer	57
Hình 9. 2 Minh họa chạy tiến trình reader	57

Bài 10:

Hình 10. 1 Minh họa các tiến trình sử dụng vùng nhớ dùng chung	59
--	----

Bài 11:

Hình 11. 1 Minh họa kỹ thuật Monte Carlo để tính số Pi	69
--	----

Bài 12:

Hình 12. 1 Minh họa xử lý bài toán	71
--	----

Bài 13:

Hình 13. 1 Mutex lock giống như một ổ khóa dùng để bảo vệ critical resource	74
Hình 13. 2 Sơ đồ biểu diễn các trạng thái của một mutex lock	75

Bài 14:

Hình 14. 1 Minh họa bài toán Producer - Consumer	80
--	----

HẾT