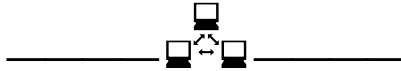


Báo cáo đồ án nguyên lý hệ điều hành

TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA CÔNG NGHỆ THÔNG TIN
BỘ MÔN MẠNG VÀ TRUYỀN THÔNG



ĐỒ ÁN HỆ ĐIỀU HÀNH

ĐỀ TÀI: TÌM HIỂU CƠ CHẾ GIAO TIẾP GIỮA 2 TIẾN TRÌNH DÙNG MESSAGE QUEUE

Sinh viên : Hồ Tấn Dục
Nguyễn Xuân Thạch
Lớp : 07T3
Cán bộ hướng dẫn : Nguyễn Văn Nguyên

Đà Nẵng 2010

MỤC LỤC

CHƯƠNG 1. TIẾN TRÌNH TRONG LINUX.....	7
1.1. TIẾN TRÌNH.....	7
1.1.1. Định nghĩa.....	7
1.1.2. Nhận biết tiến trình.....	7
1.1.3. Các trạng thái của tiến trình:.....	7
1.2. DỪNG MỘT TIẾN TRÌNH.....	8
1.3. TẠO TIẾN TRÌNH BẰNG HÀM SYSTEM().....	9
CHƯƠNG 2. GIAO TIẾP TIẾN TRÌNH BẰNG MESSAGE QUEUE.....	11
2.1. GIAO TIẾP GIỮA CÁC TIẾN TRÌNH.....	11
2.1.1. Nhu cầu giao tiếp giữa các tiến trình	11
2.1.2. Các cơ chế giao tiếp.....	11
2.2. MESSAGE QUEUE	12
2.2.1. Cấu trúc mô tả message queue.....	13
2.2.2. Thiết lập message queue.....	14
2.2.3. Gửi và nhận message.....	15
2.2.4. Điều khiển message queue.....	18
CHƯƠNG 3. CHƯƠNG TRÌNH MINH HỌA.....	20
3.1. MỤC ĐÍCH.....	20
3.2. YÊU CẦU.....	20
3.3. CHƯƠNG TRÌNH.....	20
3.3.1. Công việc của các tiến trình.....	20
3.3.2. Mã nguồn các hàm chính của chương trình.....	21
3.3.3. Kết quả chạy chương trình.....	24

TỔNG QUAN VỀ ĐỀ TÀI

1. Bối cảnh và lý do thực hiện đề tài

Nguyên lý hệ điều hành cung cấp cho chúng ta những khái niệm cơ bản về hệ điều hành máy tính: phân loại, nguyên lý, cách làm việc, phân tích thiết kế và chi tiết về một số hệ điều hành cụ thể. Tuy nhiên môn học nguyên lý hệ điều hành chỉ cung cấp những kiến thức chung, trong khi đồ án nguyên lý hệ điều hành đi vào cụ thể một hệ điều hành hoặc một phần nào đó của hệ điều hành. Điều đó giúp cho sinh viên thực hiện đồ án có cái nhìn sâu và chuyên môn hơn.

Chúng ta đã nghe đến nhiều hệ điều hành MS-DOS, windows, MAC...đó là những hệ điều hành ta phải trả phí để sử dụng. Bên cạnh đó, Linux nổi lên là một hệ điều hành của phần mềm tự do, của việc phát triển mã nguồn mở (Linux cũng là tên gọi của tên hạt nhân của hệ điều hành), điều đó gây hứng thú với chúng em, việc chọn đề tài “Tìm hiểu cơ chế giao tiếp giữa 2 tiến trình dùng message queue” vừa giúp chúng em tìm hiểu sâu một phần của hệ điều hành Linux, đồng thời có cơ hội trải nghiệm và làm quen với môi trường làm việc Linux mới mẻ này.

2. Phương pháp triển khai đề tài

Việc thực hiện đề tài được chia làm 4 giai đoạn

- Giai đoạn 1: làm quen môi trường làm việc linux, tổng hợp các lý thuyết liên quan đến đề tài.
- Giai đoạn 2: Chọn lọc, sắp xếp và tìm hiểu các lý thuyết nhằm chuẩn bị cho việc viết chương trình ứng dụng
- Giai đoạn 3: Viết chương trình ứng dụng các lý thuyết đã tìm hiểu.
- Giai đoạn 4: Hoàn thành báo cáo.

3. Kết cấu của đồ án

Đồ án gồm 2 phần:

- Phần 1: Cơ sở lý thuyết

Trong phần này tập trung vào tìm hiểu lý thuyết về tiến trình trong linux (chương 1) và cơ chế giao tiếp giữa các tiến trình bằng message queue (chương 2).

- Phần 2: Chương trình minh họa (chương 3).

Chương 1. TIỀN TRÌNH TRONG LINUX

1.1. Tiến trình

1.1.1. Định nghĩa

Tiến trình là một môi trường đang được thực hiện, nó bao gồm mã lệnh và những dữ liệu riêng.

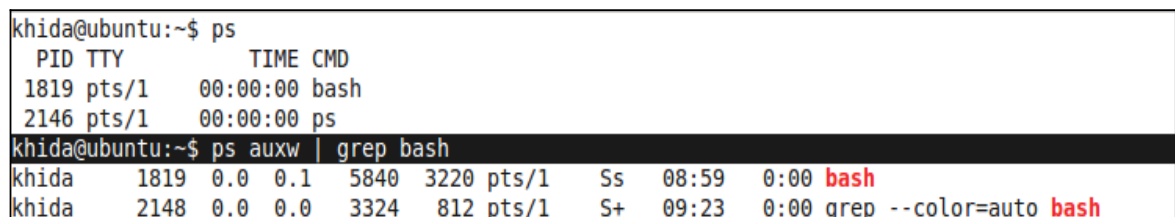
Tiến trình khác với chương trình ở chỗ, chương trình chỉ gồm tập hợp lệnh.

1.1.2. Nhận biết tiến trình

Mỗi tiến trình trong Linux được xác định bằng 1 số hiệu (process ID) duy nhất, viết tắt là PID. Khi một tiến trình được tạo nó sẽ được gán cho 1 PID.

Mỗi tiến trình cũng có thể có 1 tiến trình cha (parent process ID hoặc là PPID) là tiến trình sinh ra tiến trình này.

Để hiển thị thông tin của các tiến trình trong linux ta có thể dùng lệnh ps với các tùy chọn của nó.



```
khida@ubuntu:~$ ps
  PID TTY          TIME CMD
 1819 pts/1    00:00:00 bash
 2146 pts/1    00:00:00 ps
khida@ubuntu:~$ ps auxw | grep bash
khida    1819  0.0  0.1  5840  3220 pts/1    Ss   08:59   0:00 bash
khida    2148  0.0  0.0  3324   812 pts/1    S+   09:23   0:00 grep --color=auto bash
```

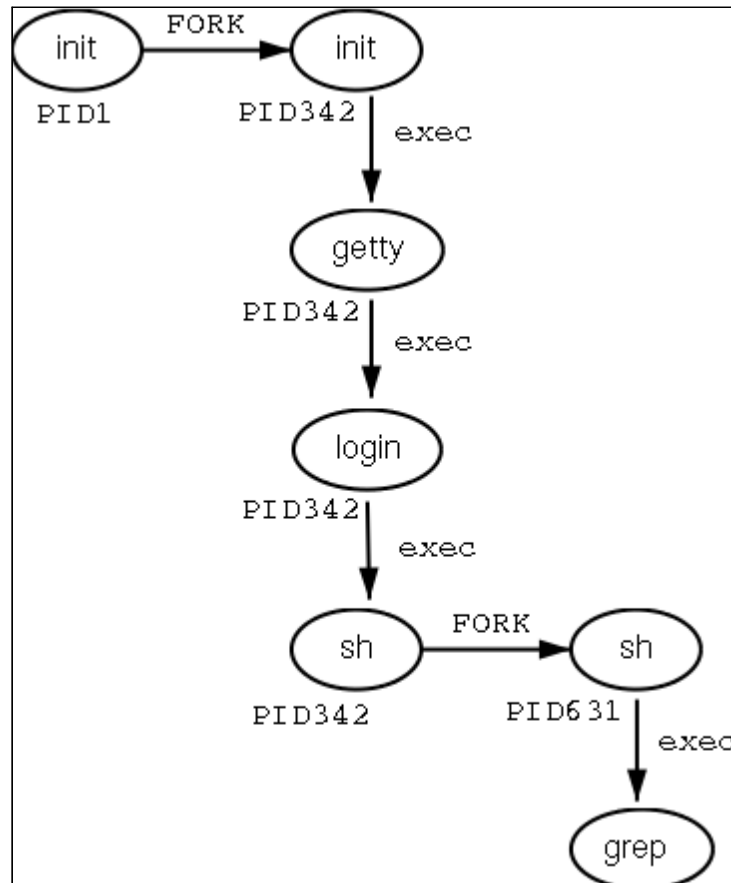
Hình 1: Hiển thị thông tin của tiến trình bằng lệnh ps

1.1.3. Các trạng thái của tiến trình:

Khởi tạo tiến trình:

Khi một tiến trình được tạo bởi một tiến trình đã tồn tại nó sẽ tạo ra một bản sao của tiến trình tạo ra nó. Tiến trình con này có một môi trường giống cha nó, chỉ khác nhau về PID. Phương thức này được gọi là forking.

Sau khi tiến trình con được tạo ra, dữ liệu của nó có thể được thay đổi bằng cách gọi lệnh exec đến hệ thống.



Hình 2: Minh họa về fork – exec

Kết thúc tiến trình:

Khi một tiến trình kết thúc bình thường (nó không bị dừng bởi lệnh kill hoặc bị đứt quãng), chương trình sẽ trả về exit status cho tiến trình cha nó cung cấp kết quả thực thi của chương trình.

1.2. Dừng một tiến trình

Ta có thể dừng một tiến trình bằng cách gửi tín hiệu đến cho nó. Chúng ta sử dụng lệnh **kill** để gửi một tín hiệu cho một tiến trình. Để xem danh sách các tín hiệu ta dùng lệnh **kill -l**. Hầu hết các tín hiệu được sử dụng bên trong bởi hệ thống hoặc bởi những lập trình viên khi viết mã.

Cú pháp:

```
kill [signal] <PID>
```

- signal: là một số hay tên của tín hiệu được gửi đến tiến trình
- PID: số hiệu của tiến trình muốn dừng

Theo mặc định lệnh kill sẽ gửi tín hiệu 15.

Một vài tín hiệu thông dụng:

Tên tín hiệu	Mã tín hiệu	Nghĩa
SIGTERM	15	Kết thúc chương trình
SIGINT	2	Ngắt từ bàn phím (Giống Ctrl + C)
SIGKILL	9	Hủy tiến trình
SIGHUP	1	Gọi lại tiến trình

1.3. Tạo tiến trình bằng hàm system()

Hàm system() trong thư viện C chuẩn cho phép ta dễ dàng thực thi một lệnh (command) bên trong chương trình. Giống như lệnh được gõ trong shell. Thực sự, system tạo một tiến trình con chạy một Bourne Shell chuẩn (/bin/sh) và cho shell đó thực thi câu lệnh.

Ví dụ: thực thi lệnh hiển thị nội dung của thư mục gốc, cũng giống như khi chạy lệnh ls -l / trong một shell.

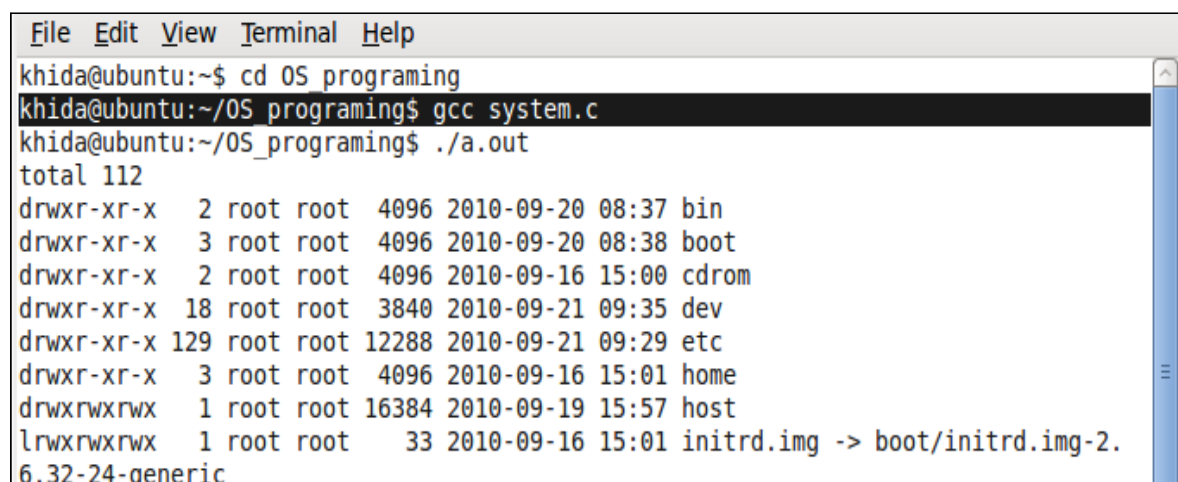
```
#include <stdio.h>

void main(){
    int return_value;

    return_value = system("ls -l /");

    printf("Ma tra ve %d\n",return_value);
}
```

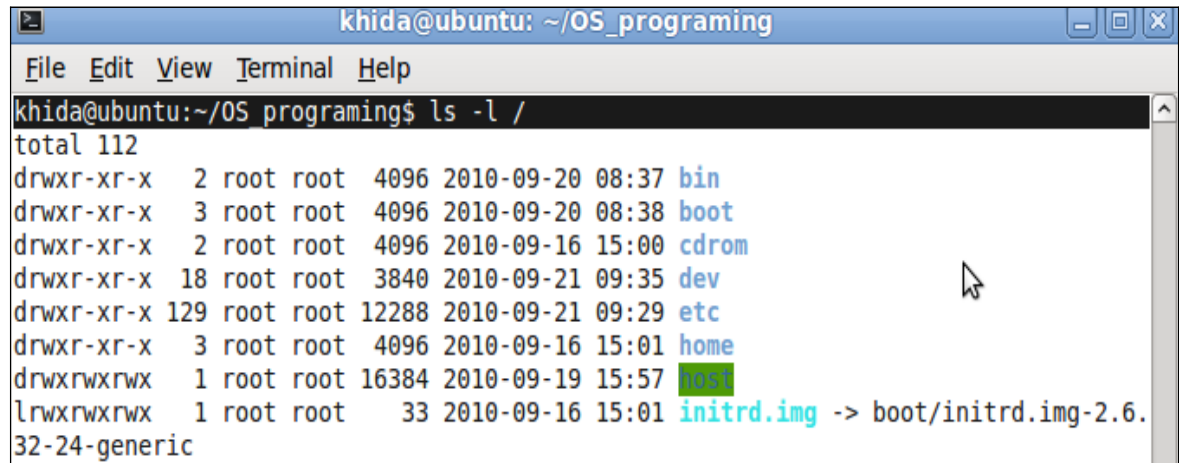
Kết quả biên dịch và chạy chương trình:



```
File Edit View Terminal Help
khida@ubuntu:~$ cd OS programing
khida@ubuntu:~/OS programing$ gcc system.c
khida@ubuntu:~/OS programing$ ./a.out
total 112
drwxr-xr-x  2 root root  4096 2010-09-20 08:37 bin
drwxr-xr-x  3 root root  4096 2010-09-20 08:38 boot
drwxr-xr-x  2 root root  4096 2010-09-16 15:00 cdrom
drwxr-xr-x 18 root root 3840 2010-09-21 09:35 dev
drwxr-xr-x 129 root root 12288 2010-09-21 09:29 etc
drwxr-xr-x  3 root root  4096 2010-09-16 15:01 home
drwxrwxrwx  1 root root 16384 2010-09-19 15:57 host
lrwxrwxrwx  1 root root   33 2010-09-16 15:01 initrd.img -> boot/initrd.img-2.
6.32-24-generic
```

Hình 3: Kết quả chạy chương trình C sử dụng system() và ls

Kết quả chạy lệnh trực tiếp từ terminal:



```
khida@ubuntu: ~/OS_programing
File Edit View Terminal Help
khida@ubuntu:~/OS_programing$ ls -l /
total 112
drwxr-xr-x  2 root root  4096 2010-09-20 08:37 bin
drwxr-xr-x  3 root root  4096 2010-09-20 08:38 boot
drwxr-xr-x  2 root root  4096 2010-09-16 15:00 cdrom
drwxr-xr-x 18 root root 3840 2010-09-21 09:35 dev
drwxr-xr-x 129 root root 12288 2010-09-21 09:29 etc
drwxr-xr-x  3 root root  4096 2010-09-16 15:01 home
drwxrwxrwx  1 root root 16384 2010-09-19 15:57 host
lrwxrwxrwx  1 root root    33 2010-09-16 15:01 initrd.img -> boot/initrd.img-2.6.
32-24-generic
```

Hình 4: Kết quả thực hiện lệnh ls từ terminal

Hàm system trả về kết quả exit status của câu lệnh shell. Nếu bản thân shell không thể chạy, system sẽ trả về 127, nếu vì lỗi khác nó sẽ trả về -1.

Chương 2. GIAO TIẾP TIỀN TRÌNH BẰNG MESSAGE QUEUE

2.1. Giao tiếp giữa các tiến trình

2.1.1. Nhu cầu giao tiếp giữa các tiến trình

Các tiến trình trên nguyên tắc là độc lập với nhau, nhưng đôi khi chúng cần phải liên lạc với nhau để:

- Chia sẻ thông tin: nhiều tiến trình có thể cùng truy cập vào các dữ liệu chung
- Hợp tác hoàn thành tác vụ: đôi khi để đạt được kết quả nhanh chóng, ta chia tác vụ ra làm các công việc nhỏ để tiến hành song song với nhau ví dụ dữ liệu ra của tiến trình này là dữ liệu vào cho tiến trình khác xử lý.

Khi đó, hệ điều hành cần cung cấp cơ chế để các tiến trình có thể trao đổi thông tin với nhau.

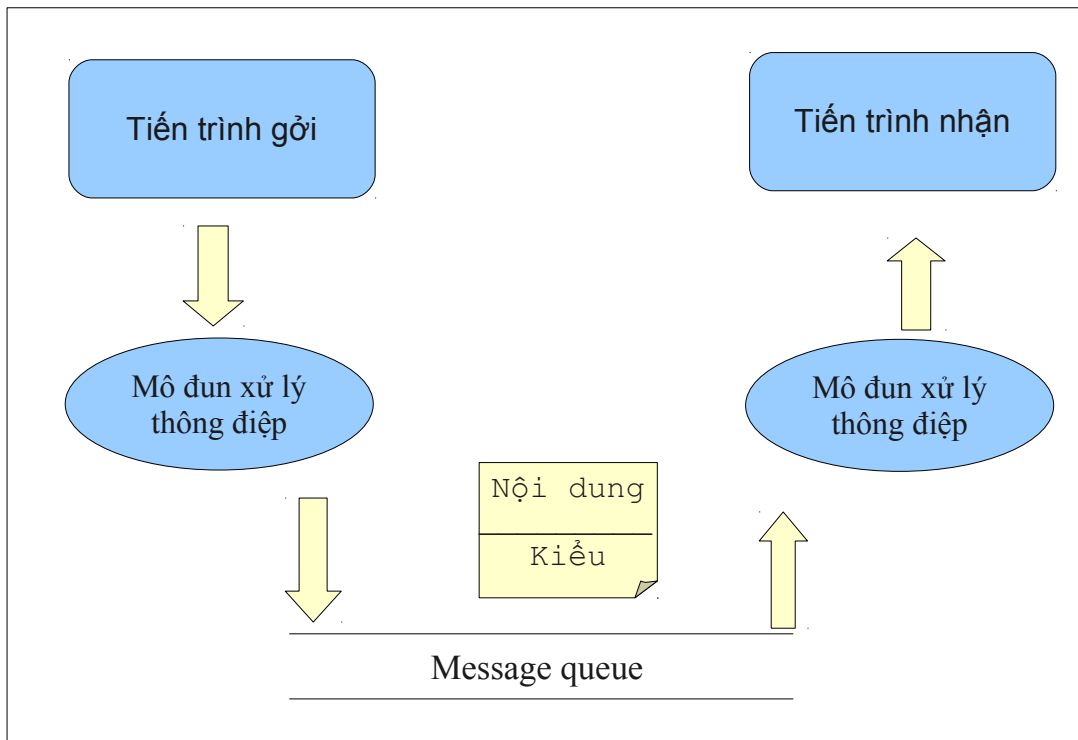
2.1.2. Các cơ chế giao tiếp

- *Tín hiệu (Signal)* : là một cơ chế phần mềm tương tự như các ngắt cứng tác động đến tiến trình.
- *Pipe*: là một kênh liên lạc trực tiếp giữa hai tiến trình, khi một pipe được thiết lập, một tiến trình sẽ ghi dữ liệu vào pipe và tiến trình kia sẽ đọc dữ liệu từ pipe. Thứ tự truyền dữ liệu qua pipe theo nguyên tắc FIFO.
- *Vùng nhớ chia sẻ*: cơ chế này cho phép nhiều tiến trình cùng truy xuất đến một vùng nhớ chung. Các tiến trình chia sẻ một vùng nhớ vật lý thông qua không gian địa chỉ của chúng.
- *Message queue*: hệ điều hành cung cấp cơ chế liên lạc giữa các tiến trình thông qua việc chia sẻ một tài nguyên chung thông qua việc gửi thông điệp. Để hỗ trợ cơ chế này hệ điều hành cung cấp các hàm IPC chuẩn (Interprocess Communication) cơ bản là hai hàm send(message) và receive(message). Đơn vị truyền thông trong cơ chế trao đổi thông điệp là một thông điệp, do đó các tiến trình có thể trao đổi thông điệp ở dạng có cấu trúc.
- ...

2.2. Message queue

Hai hoặc nhiều tiến trình có thể trao đổi thông tin với nhau bằng cách sử dụng một hệ thống message queue chung. Tiến trình gửi thông điệp qua mô đun xử lý thông điệp sẽ gửi thông điệp vào hàng đợi và tiến trình khác sẽ đọc nó. Mỗi thông

điệp có 1 đặc điểm nhận biết (kiểu) để các tiến trình khác có thể lựa chọn được thông điệp thích hợp.



Hình 5: Mô hình gửi và nhận thông điệp

Trước khi một tiến trình gửi hoặc nhận thông điệp, message queue phải được thiết lập bằng cách sử dụng hàm `msgget()`. Việc gửi và nhận thông điệp được thực hiện bằng các hàm `msgsnd()` và `msgrcv()`.

Khi một thông điệp được gửi, nội dung của nó sẽ được sao chép vào message queue. Thao tác `msgsnd()` và `msgrcv()` có thể được thực hiện như là chức năng bị chặn hoặc không bị chặn.

Chức năng bị chặn yêu cầu việc trao đổi phải đồng bộ, tiến trình gửi không thể tiếp tục gửi cho đến khi thông điệp đã được di chuyển hoặc đã được tiến trình nhận công nhận.

Chức năng không bị chặn cho phép việc trao đổi thông điệp bất đồng bộ.

2.2.1. Cấu trúc mô tả message queue

Bên trong hệ thống, mỗi đối tượng IPC có một cấu trúc dữ liệu để lưu trữ thông tin về chúng. Đối với message queue cấu trúc đó là `msqid_ds`. Nhân hệ thống tạo, lưu trữ và bảo quản một thể hiện của cấu trúc này cho mỗi message queue được tạo ra trong hệ thống. Nó được định nghĩa trong `sys/msg.h` như dưới đây:

Báo cáo đồ án nguyên lý hệ điều hành

```
/* one msqid structure for each queue on the system */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first; /* first message on queue */
    struct msg *msg_last; /* last message in queue */
    time_t msg_stime;      /* last msgsnd time */
    time_t msg_rtime;      /* last msgrcv time */
    time_t msg_ctime;      /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes;     /* max number of bytes on queue
*/
    ushort msg_lspid;      /* pid of last msgsnd */
    ushort msg_lrpid;      /* last receive pid */
};
```

Mô tả ngắn về từng thành phần

msg_perm	Thể hiện của cấu trúc ipc_perm, nó được định nghĩa trong sys/ipc.h. Nó chứa thông tin về cấp phép cho message queue, bao gồm cả quyền truy cập và thông tin về người tạo ra hàng đợi.
msg_first	Trỏ đến message đầu tiên trong hàng đợi
msg_last	Trỏ đến message cuối cùng trong hàng đợi
msg_stime	Thời điểm của message cuối được gửi vào hàng đợi
msg_rtime	Thời điểm của message cuối được lấy khỏi hàng đợi
msg_ctime	Thời điểm của lần thay đổi cuối của hàng đợi
wwait, rwait	Trỏ đến wait queue của nhân, chúng được sử dụng khi một

	message queue nghĩ rằng tiến trình đang trong trạng thái sleep (ví dụ: hàng đợi đã đầy và tiến trình đang đợi để mở)
msg_cbytes	Tổng số bytes trong hàng đợi (tổng của tất cả message)
msg_qnum	Số lượng messages hiện tại trong hàng đợi
msg_qbytes	Số bytes tối đa trong hàng đợi
msg_lspid	PID của tiến trình gửi message cuối
msg_lrpid	PID của tiến trình nhận message cuối

2.2.2. Thiết lập message queue

Hàm `msgget()` thiết lập một message queue.

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

#include <sys/ipc.h>

#include <sys/msg.h>

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

Hàm này trả về ID của hàng đợi (`msgid`) đã được liên kết với tham số `key`, hoặc `-1` nếu thất bại.

Tham số:

- `key`: là một giá trị cụ thể tùy ý để liên kết với hàng đợi.
- `msgflg`: là một cờ để chỉ dẫn các tùy chọn và điều kiện thiết lập hàng đợi.

Một cách để tạo ra `key` là dùng hàm `ftok(const char *path, int id)`, nó chuyển đổi tên file ra một giá trị `key` duy nhất trong hệ thống. Nếu `ftok()` được gọi với những giá trị `id` khác nhau hoặc `path` chỉ đến những file khác nhau nó sẽ trả về những `key` khác nhau.

Nếu `key` có giá trị `IPC_PRIVATE`, hoặc `key` không phải là `IPC_PRIVATE` và chưa có message queue nào gắn với `key` và `(msgflg & IPC_CREATE)` là non-zero thì một message queue mới được tạo.

Khi giá trị cờ `IPC_CREAT` được truyền vào tham số cờ, `msgget()` sẽ tạo mới một message queue, với giá trị cờ `IPC_EXCL`, `msgget()` sẽ tạo mới một message queue nếu nó chưa tồn tại. Khi cả giá trị `IPC_CREAT` và `IPC_EXCL` được truyền vào `msgflg`, hàm sẽ thất bại nếu message queue liên kết với `key` đã tồn tại.

Việc này sẽ hữu ích khi có nhiều hơn 1 tiến trình cố gắng khởi tạo kênh liên lạc. Có trường hợp, nhiều tiến trình có thể truy cập vào cùng một kênh, nếu chúng đều cố tạo kênh liên lạc với giá trị cờ `IPC_EXCL`, chỉ có tiến trình đầu tiên thành công.

Báo cáo đồ án nguyên lý hệ điều hành

Những cờ điều khiển này có thể tổ hợp với nhau, sử dụng phép logic OR cùng với tham số cấp phép hệ bất phân.

Ví dụ:

```
msqid = msgget(ftok("/tmp",key), (IPC_CREAT|IPC_EXCL|0400));
```

Khi một message queue đã được tạo, một cấu trúc dữ liệu liên kết với message queue mới được khởi tạo theo cách sau:

- `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid`, và `msg_perm.gid` được gán bằng user ID, group ID tương ứng với tiến trình được gọi.
- Các bits cấp phép của `msg_perm.mode` được gán bằng các bits cấp phép của `msgflg`.
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, và `msg_rtime` được gán bằng 0.
- `msg_ctime` được gán bằng thời điểm hiện tại.
- `msg_qbytes` được gán bằng giới hạn của hệ thống.

2.2.3. Gởi và nhận message

Việc gởi và nhận thông điệp được thực hiện bằng hai hàm riêng biệt `msgsnd()` và `msgrcv()`.

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

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

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

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

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

Tiến trình gọi hàm phải định rõ kiểu của message theo cấu trúc sau.

```
struct msgbuf {  
    long mtype; /* message type, must be > 0 */  
    char mtext[1]; /* message data */  
};
```

`mtext` là 1 mảng (hoặc cấu trúc khác) có kích thước được chỉ định bởi `msgsz`, giá trị nguyên không âm. Message có độ dài bằng 0 cũng được chấp thuận.

`mtype` phải có giá trị nguyên dương, giá trị này sẽ được tiến trình nhận dùng cho việc lựa chọn message.

Báo cáo đồ án nguyên lý hệ điều hành

Tiến trình phải có quyền “ghi” để gửi và quyền “đọc” để nhận message trên hàng đợi.

Lời gọi `msgsnd` thêm một bản sao của message được trả bởi `msgp` vào message queue (message queue này được xác định bởi `msqid`).

Nếu còn đủ chỗ trống trong hàng đợi, `msgsnd` sẽ hoàn thành ngay lập tức. (Sức chứa của hàng đợi được chỉ định bởi giá trị `msg_qbytes` trong cấu trúc lưu trữ của hàng đợi. Khi hàng đợi được tạo, giá trị này được gán bằng `MSGMNB` bytes, nhưng nó có thể được thay đổi bằng cách sử dụng hàm `msgctl`). Nếu không còn đủ chỗ trong hàng đợi, theo mặc định `msgsnd` bị chặn cho đến khi có đủ chỗ để thêm message. Tuy nhiên, nếu giá trị `IPC_NOWAIT` được chỉ định trong `msgflg` thì lời gọi hàm sẽ bị lỗi `EAGAIN`.

Một lời gọi `msgsnd` đang bị chặn cũng có thể thất bại nếu hàng đợi bị gỡ bỏ (trường hợp này hệ thống gọi lỗi với `errno` được gán giá trị `EIDRM`), hoặc nó bắt được một tín hiệu (trường hợp này hệ thống gọi lỗi với `errno` được gán giá trị `EINTR`).

Sau khi gửi message thành công, dữ liệu của cấu trúc hàng đợi được cập nhật như sau:

- `msg_lspid` gán bằng PID của tiến trình gửi.
- `msg_qnum` được tăng lên 1.
- `msg_stime` gán bằng thời gian hiện hành.

Lời gọi `msgrcv` đọc một message từ message queue (message queue này được xác định bởi `msqid`) vào `msgbuf` được trả bởi `msgp`, gỡ bỏ message đã đọc ra khỏi hàng đợi.

Tham số `msgsz` chỉ định số byte tối đa cho một thành viên `mtext` của cấu trúc được trả bởi `msgp`. Nếu message có độ dài lớn hơn `msgsz`, và nếu `msgflg` được chỉ định `MSG_NOERROR`, đoạn text của message sẽ bị cắt (phần cắt sẽ bị mất đi), trường hợp ngược lại message không bị gỡ khỏi hàng đợi, và hệ thống gọi lỗi với `errno` được gán giá trị `E2BIG`.

Tham số `msgtyp` chỉ định kiểu của message được nhận như sau:

- Nếu `msgtyp` bằng 0, message đầu tiên của hàng đợi được đọc.
- Nếu `msgtyp` lớn hơn 0, message đầu tiên của hàng đợi có kiểu `msgtyp` được đọc, trừ khi `MSG_EXCEPT` được chỉ định trong `msgflg`, trong trường hợp này message đầu tiên của hàng đợi có kiểu khác với `msgtyp` sẽ được đọc.
- Nếu `msgtyp` nhỏ hơn 0, message đầu tiên của hàng đợi với kiểu nhỏ hơn hoặc bằng giá trị tuyệt đối của `msgtyp` sẽ được đọc.

Tham số `msgflg` có thể được thiết lập với một, nhiều hoặc không giá trị cờ nào dưới đây:

Báo cáo đồ án nguyên lý hệ điều hành

- `IPC_NOWAIT`: để return ngay lập tức nếu không có message nào của kiểu type trong hàng đợi. Lỗi gọi hệ thống thất bại với `errno` được gán giá trị `ENOMSG`.
- `MSG_EXCEPT`: sử dụng với `msgtyp` lớn hơn 0 để đọc message đầu tiên trong hàng đợi với kiểu khác với `msgtyp`.
- `MSG_NOERROR`: để cắt nội dung của message nếu nó dài hơn `msgsz`.

Nếu không có message nào có kiểu type và `IPC_NOWAIT` không được thiết lập trong `msgflg`, tiến trình gọi bị chặn cho đến khi một trong những điều kiện sau thỏa mãn.

- Một message có kiểu type được đặt vào hàng đợi.
- Message queue bị gỡ bỏ khỏi hệ thống. Trong trường hợp này, lời gọi hệ thống thất bại với `errno` gán bằng `EIDRM`.
- Tiến trình gọi bắt được một tín hiệu. Trong trường hợp này lời gọi hệ thống thất bại với `errno` được gán bằng `EINTR`.

Sau khi đọc thành công, cấu trúc dữ liệu của message queue được cập nhật như sau:

- `msg_lrpId` được gán bằng PID của tiến trình vừa đọc.
- `msg_qnum` giảm đi 1.
- `msg_rtime` gán bằng thời gian hiện hành.

Giá trị trả về

Nếu thất bại cả hai hàm đều trả về -1 với `errno` chỉ dẫn lỗi. Ngược lại, `msgsnd` trả về 0 và `msgsrcv` trả về số byte thực sự chép vào mảng `mtext`.

Các báo lỗi

Khi `msgsnd` thất bại, giá trị trả về `errno` sẽ được gán một trong số những giá trị sau:

- **EAGAIN** : message không thể gửi quá giới hạn `msg_qbytes` của hàng đợi và `IPC_NOWAIT` được thiết lập trong `msgflg`.
- **EACCES** : tiến trình gọi không có quyền ghi vào message queue.
- **EFAULT** : không thể truy cập địa chỉ được trả bởi `msgp`.
- **EIDRM** : message queue đã bị gỡ bỏ.
- **EINTR** : sleep trên một message queue đã đầy, tiến trình bắt được một tín hiệu.
- **EINVAL** : giá trị `msqid` không thích hợp, hoặc `msgsz` có giá trị không thích hợp (nhỏ hơn 0 hoặc lớn hơn giá trị hệ thống `MSGMAX`), hoặc `mtype` có giá trị không dương.

Báo cáo đồ án nguyên lý hệ điều hành

- **ENOMEM** : hệ thống không đủ bộ nhớ để tạo một bản sao của `msgbuf`.

Khi `msgrcv` thất bại, giá trị trả về `errno` sẽ được gán một trong các giá trị sau.

- **E2BIG** : độ dài của message text lớn hơn `msgsz` và `MSG_NOERROR` không được thiết lập trong `msgflg`.
- **EACCES** : tiến trình gọi không có quyền đọc trên message queue.
- **EFAULT** : địa chỉ được trỏ bởi `msgp` không truy cập được.
- **EIDRM** : trong khi tiến trình đang sleep để nhận một message, message queue bị gỡ khỏi hệ thống.
- **EINTR** : trong khi tiến trình đang sleep để nhận 1 message mới, tiến trình nhận một tín hiệu.
- **EINVAL** : giá trị `msgqid` không hợp hoặc `msgsz` nhỏ hơn 0.
- **ENOMSG** : `IPC_NOWAIT` được thiết lập trong `msgflg` và không có message nào với kiểu type trên message queue.

2.2.4. Điều khiển message queue

Chúng ta có thể thay đổi cấp phép và những tính chất khác của message queue bằng cách sử dụng hàm `msgctl()`. Chủ sở hữu hoặc người tạo ra hàng đợi có thể thay đổi quyền sở hữu và những cấp phép, nhiều tiến trình với cấp phép của mình cũng có thể sử dụng `msgctl()` để điều khiển hoạt động của hàng đợi.

Int msgctl(int msqid, int cmd, struct msqid_ds *buf)

`msqid`: ID của một message queue đã tồn tại.

`cmd`: có thể là những giá trị sau:

- `IPC_STAT`: đặt thông tin trạng thái của hàng đợi vào cấu trúc dữ liệu được trỏ bởi `buf`. Tiến trình gọi hàm phải có quyền đọc mới thành công.
- `IPC_SET`: Thiết lập chủ sở hữu, ID nhóm, cấp phép và kích thước (bytes) của message queue. Tiến trình gọi hàm phải có user ID của chủ sở hữu, người tạo, hoặc là `superUser`.
- `IPC_RMID`: xóa message queue được chỉ định bởi `msqid`.

Chương 3. CHƯƠNG TRÌNH MINH HỌA

3.1. Mục đích

Minh họa sử dụng message queue để giao tiếp giữa 2 tiến trình.

3.2. Yêu cầu

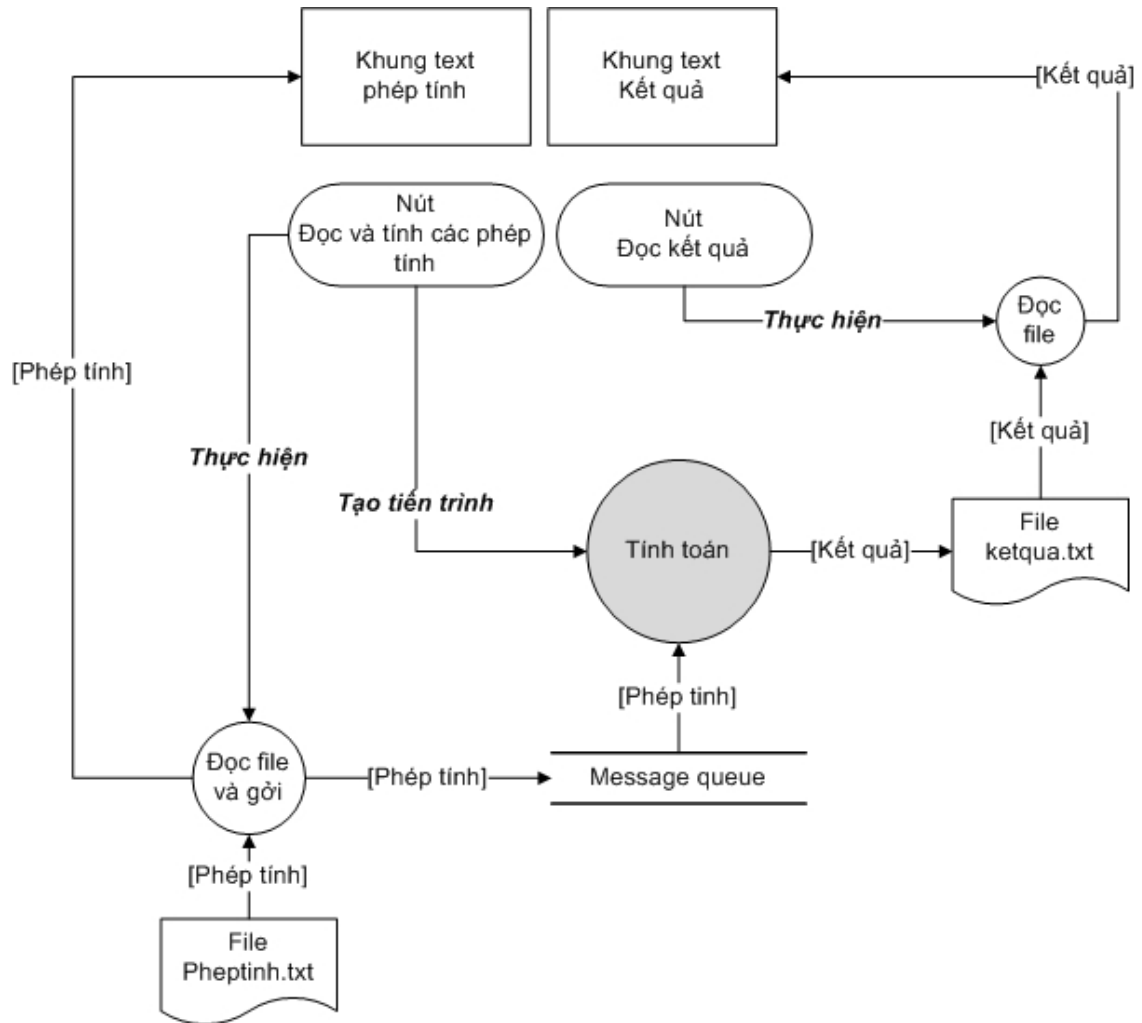
Viết chương trình đọc các phép toán từ file sau đó tính và xuất kết quả ra file khác. Một tiến trình đọc các phép toán và truyền nó vào message queue, tiến trình khác sẽ đọc các phép toán này từ message queue và tính kết quả rồi xuất ra file.

3.3. Chương trình

3.3.1. Công việc của các tiến trình

Chương trình gồm 2 tiến trình chính:

- Tiến trình cha đọc các phép toán từ file `pheptoaan.txt` và gửi chúng vào message queue. Đồng thời nó tạo ra tiến trình con để tính toán.
- Tiến trình con đọc các phép toán từ message queue và tính rồi ghi kết quả ra file `ketqua.txt`



Hình 6: Sơ đồ thực hiện của chương trình

3.3.2. Mã nguồn các hàm chính của chương trình

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

#define MSGSIZE 100 // kích thước 1 message
#define TYPE 1

typedef struct mymsgbuf{
    long type;
    char msg[MSGSIZE];
} message_buf;
```

Hàm đọc các phép tính từ file, rồi gửi vào message queue, đồng thời tạo tiến trình con để tính toán

```
void send_and_cal(GtkWidget *widget, gpointer data){
    pid_t pid;
    pid = fork();

    if(pid>0){
        /* Tien trinh cha - Doc cac phep toan va goi vao
        message queue*/

        // Xoa sach textbox phep toan
        gtk_text_buffer_set_text( buf_pheptoan, "", 0);

        FILE *f = fopen("pheptoan.txt","r");
        message_buf mbuf;
        int msqid ;
        if((msqid= msgget(key,IPC_CREAT | 0666))<0)
            perror("[send]msgget");
        else{
            // Goi cac phep toan vao hang doi
            mbuf.type = TYPE;
            while( !feof(f) ){
                if(fscanf(f,"%s",mbuf.msg)>0){
                    if(msgsnd(msqid, &mbuf,
sizeof(mbuf),IPC_NOWAIT)<0) perror("Msgsnd");
                    else{
                        gtk_text_buffer_insert_at_cursor( bu
f_pheptoan, mbuf.msg, strlen(mbuf.msg));
                        gtk_text_buffer_insert_at_cursor( bu
f_pheptoan, "\n", 1);
                    }
                }
            }
            // Goi message ket thuc
            strcpy(mbuf.msg, "end");
            if(msgsnd(msqid, &mbuf, sizeof(mbuf),IPC_NOWAIT)<0)
perror("Msgsnd");
            else{
                gtk_text_buffer_insert_at_cursor( buf_pheptoan,
mbuf.msg, strlen(mbuf.msg));
            };

            fclose(f);
            // Doi cho tien trinh con ket thuc
            waitpid(pid,NULL,0);
        }
    }
    else{
        /* Tien trinh con */
    }
}
```

Báo cáo đồ án nguyên lý hệ điều hành

```
execlp("./tinhtoan",NULL);  
// Neu exe co loi, in ra ma loi  
perror("execlp");  
_exit(1);  
}  
  
}
```

Các hàm phục vụ cho việc đọc các phép toán từ message queue và xuất kết quả vào file

- void msq2file() : đọc các phép toán từ hàng đợi và xuất kết quả vào file.
- Việc tính toán sử dụng ký pháp nghịch đảo BaLan (Do mã dài nên sẽ không đưa vào đây, có thể tham khảo các hàm trong cd chứa mã nguồn).

```
void msq2file() {  
    FILE *f = fopen("ketqua.txt","w");  
    message_buf mbuf;  
    int msqid;  
    int ok=1;  
    float kq;  
    string bieuThuc;  
    if((msqid = msgget(key, 0666))<0) perror("[read]msgget")  
;  
    else{  
        while(ok){  
            if(msgrcv(msqid, &mbuf, sizeof(mbuf),TYPE,0)<0)  
perror("msgrcv"); // Doc ko duoc  
            else  
                if(strcmp(mbuf.msg,"end")==0) ok=0; // Doc  
duoc message ket thuc  
            else {  
                bieuThuc = mbuf.msg;  
                kq = tinhToan(bieuThuc);  
                if(kq - (int)kq !=0 ) fprintf(f,"%s=  
%f\n", mbuf.msg,kq);  
                else fprintf(f,"%s=%d\n", mbuf.msg,  
(int)kq);  
            }  
        }  
        // xoa bo hang doi  
        msgctl(msqid,IPC_RMID,NULL);  
    }  
  
    fclose(f);  
    printf("\nKet thuc viec doc msq va ghi ra file  
ketqua.txt");  
}
```

3.3.3. Kết quả chạy chương trình



Hình 7: Kết quả chạy chương trình demo

KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

1. Những kết quả đạt được

- Biết được cách hoạt động của các tiến trình trong linux, cách tạo tiến trình con.
- Đã tìm hiểu được cơ chế giao tiếp giữa các tiến trình bằng message queue.
- Hiểu và viết được chương trình ứng dụng message queue.

2. Những vấn đề tồn tại

- Việc lập trình giao diện trong linux còn gặp khó khăn.

3. Hướng phát triển

- Tìm hiểu cách tạo tiến trình con mô phỏng theo lệnh fork().
- Tìm hiểu thêm về các cơ chế giao tiếp tiến trình khác.

TÀI LIỆU THAM KHẢO

- [1] Machtelt Garrels, **Introduction to Linux A Hands on Guide.**
- [2] <http://www.mkssoftware.com>
- [3] <http://linux.about.com>
- [4] <http://linux.omnipotent.net>
- [5] <http://tldp.org>