GIAO TIẾP GIỮA CÁC TIẾN TRÌNH TRONG LINUX

I. Khái quát

Linux cung cấp một số cơ chế giao tiếp giữa các tiến trình gọi là IPC (Inter-Process Communication):

- Trao đổi bằng tín hiệu (signals handling)
- Trao đổi bằng cơ chế đường ống (pipe)
- Trao đổi thông qua hàng đợi thông điệp (message queue)
- Trao đổi bằng phân đoạn nhớ chung (shared memory segment)
- Giao tiếp đồng bộ dùng semaphore
- Giao tiếp thông qua socket

II. Xử lý tín hiệu (signals handling)

1. Khái niệm

- Tín hiệu là các thông điệp khác nhau được gởi đến tiến trình nhằm thông báo cho tiến trình một tình huống. Mỗi tín hiệu có thể kết hợp hoặc có sẵn bộ xử lý tín hiệu (signal handler). Tín hiệu sẽ ngắt ngang quá trình xử lý của tiến trình, bắt hệ thống chuyển sang gọi bộ xử lý tín hiệu ngay tức khắc. Khi kết thúc xử lý tín hiệu, tiến trình lại tiếp tục thực thi.
- Mỗi tín hiệu được định nghĩa bằng một số nguyên trong **`/urs/include/signal.h`**. Danh sách các hằng tín hiệu của hệ thống có thể xem bằng lệnh **`kill -l`**.

2. Gởi tín hiệu đến tiến trình

Tiến trình có thể nhận tín hiệu từ hệ điều hành hoặc các tiến trình khác gởi đến. Các cách gởi tín hiệu đến tiến trình:

a) Từ bàn phím

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

b) Từ dòng lệnh

- Lệnh **`kill -<signal> <PID>`**

Ví dụ: **`kill -INT 1234`** dùng gởi tín hiệu **`INT`** ngắt tiến trình có **`PID 1234`**.
Nếu không chỉ định tên tín hiệu, tín hiệu **`TERM`** được gởi để kết thúc tiến trình.

- Lệnh **`fg`**: gởi tín hiệu **`CONT`** đến tiến trình, dùng đánh thức các tiến trình tạm dừng do tín hiệu **`TSTP`** trước đó.

c) Bằng các hàm hệ thống `kill()`:

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>           /* macro xử lý tín hiệu và hàm kill() */
…
pid_t my_pid = getpid()       /* lấy định danh tiến trình */
kill( my_pid, SIGSTOP );      /* gửi tín hiệu STOP đến tiến trình */
```

3. Đón bắt xử lý tín hiệu

- Một số tín hiệu hệ thống (như **`KILL`**, **`STOP`**) không thể đón bắt hay bỏ qua được.
- Tuy nhiên, có rất nhiều tín hiệu mà bạn có thể đón bắt, bao gồm cả những tín hiệu nổi tiếng như SEGV và BUS.

a) Bộ xử lý tín hiệu mặc định

Hệ thống đã dành sẵn các hàm mặc định xử lý tín hiệu cho mỗi tiến trình. Ví dụ, bộ xử lý mặc định cho tín hiệu **`TERM`** gọi là hàm **`exit()`** chấm dứt tiến trình hiện hành. Bộ xử lý dành cho tín hiệu **`ABRT`** là gọi hàm hệ thống **`abort()`** để tạo ra file core lưu xuống thư mục hiện hành và thoát chương trình. Mặc dù vậy đối với một số tín hiệu bạn có thể cài đặt hàm thay thế bộ xử lý tín hiệu mặc định của hệ thống. Chúng ta sẽ xem xét vấn đề này ngay sau đây:

b) Cài đặt bộ xử lý tín hiệu

Có nhiều cách thiết lập bộ xử lý tín hiệu (signal handler) thay cho bộ xử lý tín hiệu mặc định. Ở đây ta dùng cách cơ bản nhất đó là gọi hàm **`signal().`**

```
#include <signal.h>
void signal( int signum, void (*sighanldler)( int ) );
```

III. Đường ống (pipe)

1. Khái niệm

- Các tiến trình chạy độc lập có thể chia sẻ hoặc chuyển dữ liệu cho nhau xử lý thông qua cơ chế đường ống (pipe).

Ví dụ: **`ps -ax | grep ls`**

- Trên đường ống dữ liệu chỉ có thể chuyển đi theo một chiều, dữ liệu vào đường ống tương đương với thao tác ghi (pipe write), lấy dữ liệu từ đường ống tương đương với thao tác đọc (pipe read). Dữ liệu được chuyển theo luồng (stream) theo cơ chế FIFO.

2. Tạo đường ống

Hệ thống cung cấp hàm **pipe()** để tạo đường ống có khả năng đọc / ghi. Sau khi tạo ra, có thể dùng đường ống để giao tiếp giữa hai tiến trình. Đọc / ghi đường ống hoàn toàn tương đương với đọc / ghi file.

```
#include <unistd.h>
int pipe( int filedes[2] );
```

Mảng **filedes** gồm hai phần tử nguyên dùng lưu lại số mô tả cho đường ống trả về sau lời gọi hàm, ta dùng hai số này để thực hiện thao tác đọc / ghi trên đường ống: phần tử thứ nhất dùng để đọc, phần tử thứ hai dùng để ghi.

```
int pipes[2];
int rc = pipe( pipes );        /*Tạo đường ống*/
if ( rc == -1 )                /*Có tạo đường ống được không?*/
{
    perror( "Error: pipe not created" );
    exit( 1 );
}
```

3. Đường ống hai chiều

Sử dụng cơ chế giao tiếp đường ống hai chiều dễ dàng cho cả hai phía tiến trình cha và tiến trình con. Các tiến trình dùng một đường ống để đọc và một đường ống để ghi. Tuy nhiên cũng rất dễ gây ra tình trạng tắc nghẽn "deadlock":

- Cả hai đường ống đều rỗng nếu đường ống rỗng hàm **read()** sẽ block cho đến khi có dữ liệu đổ vào hoặc khi đường ống bị đóng bởi bên ghi.
- Cả hai tiến trình cùng ghi dữ liệu: vùng đệm của một đường ống bị đầy, hàm **write()** sẽ block cho đến khi dữ liệu được lấy bớt ra từ một thao tác đọc **read()**.

4. Đường ống có đặt tên

Đường ống được tạo ra từ hàm pipe() được gọi là đường ống vô danh (anonymouse pipe). Nó chỉ được sử dụng giữa các tiến trình cha con do bạn chủ động điều khiển tạo ra từ hàm fork(). Một vấn đề đặt ra, nếu hai tiến trình không quan hệ gì với nhau thì có thể sử dụng được cơ chế pipe để trao đổi dữ liệu hay không ? Câu trả lời là có. Linux cho phép bạn tạo ra các đường ống đặt tên (named pipe). Những đường ống mang tên sẽ nhìn thấy và truy xuất bởi các tiến trình khác nhau.

a) Tạo pipe đặt tên với hàm mkfifo()

Đường ống có đặt tên gọi là đối tượng FIFO, được biểu diễn như một file trong hệ thống. Vì vậy có thể dùng lệnh **mkfifo()** để tạo file đường ống với tên chỉ định.

```
#include <sys/types.h>
#include <sys/stat.h>
mkfifo( const char *filename, mode_t mode );
```

Đối số thứ nhất là tên đường ống cần tạo, đối số thứ hai là chế độ đọc ghi của đường ống. Ví dụ:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{
    int res = mkfifo( "~/tmp/my_fifo", 0777 );
    if ( res == 0 ) printf( "FIFO object created" );
    exit ( EXIT_SUCCESS );
}
```

- Có thể xem file đường ống này trong thư mục tạo nó.
- Có thể tạo đường ống có đặt tên từ dòng lệnh, ví dụ:

```
mkfifo ~/tmp/my_fifo --mode=0777
```

b) Đọc / ghi trên đường ống có đặt tên

- Dùng dòng lệnh với > (ghi dữ liệu) hoặc < (đọc dữ liệu), ví dụ:

```
echo Hello world! > ~/tmp/my_fifo
cat < /tmp/my_fifo
```

hoặc:

```
echo Hello world! > ~/tmp/my_fifo  &  cat < ~/tmp/my_fifo
```

- Lập trình: thao tác trên đường ống có đặt tên giống như thao tác trên file nhưng chỉ có chế độ **O_RDONLY** (chỉ đọc) hoặc **O_WRONLY** (chỉ ghi).

IV. Thực hành

Bài 1: Chương trình đặt bẫy tín hiệu (hay thiết lập bộ xử lý) tín hiệu **INT**. Đây là tín hiệu gửi đến tiến trình khi người dùng nhấn **Ctrl + C**. Chúng ta không muốn chương trình bị ngắt ngang do người dùng vô tình (hay cố ý) nhấn tổ hợp phím này.

```
#include <stdio.h>     /*Hàm nhập xuất chuẩn*/
#include <unistd.h>    /*các hàm chuẩn của UNIX như getpid()*/
#include <signal.h>    /*các hàm xử lý tín hiệu()*/

/*Trước hết cài đặt hàm xử lý tín hiệu*/
void catch_int( int sig_num )
{
    signal( SIGINT, catch_int );
    /*Thực hiện công việc của bạn ở đây*/
    printf( "Do not press Ctrl+C\n" );
}
/*Chương trình chính*/
int main()
{
    int count = 0;
    /*Thiết lập hàm xử lý cho tín hiệu INT(Ctrl + C)*/
    signal( SIGINT, catch_int );      /*Đặt bẫy tín hiệu INT*/
    while ( 1 )
    {
        printf( "Counting … %d\n", count++ );
        sleep( 1 );
    }
}
```

Bài 2: Tạo đường ống, gọi hàm fork() để tạo ra tiến trình con. Tiến trình cha sẽ đọc dữ liệu nhập vào từ phía người dùng và ghi vào đường ống trong khi tiến trình con phía bên kia đường ống tiếp nhận dữ liệu bằng cách đọc từ đường ống và in ra màn hình.

```
#include <stdio.h>
#include <unistd.h>
/*Cài đặt hàm dùng thực thi tiến trình con*/
void do_child( int data_pipes[] )
{
    int c;  /*Chứa dữ liệu từ tiến trình cha*/
    int rc; /*Lưu trạng thái trả về của read()*/
    /*Tiến trình con chỉ đọc đường ống nên đóng đầu ghi do không cần*/
    close( data_pipes[1] );
    /*Tiến trình con đọc dữ liệu từ đầu đọc */
    while ( ( rc = read( data_pipes[0], &c, 1 ) ) > 0 )
    {
        putchar( c );
    }
    exit( 0 );
}

/*Cài đặt hàm xử lý công việc của tiến trình cha*/
void do_parent( int data_pipes[] )
{
    int c;  /*Dữ liệu đọc được do người dùng nhập vào*/
    int rc; /*Lưu trạng thái trả về của write()*/
    /*Tiến trình cha chỉ ghi đường ống nên đóng đầu đọc do không cần*/
    close( data_pipes[0] );
    /*Nhận dữ liệu do người dùng nhập vào và ghi vào đường ống */
    while ( ( c = getchar() ) > 0 )
    {
        /*Ghi dữ liệu vào đường ống*/
        rc = write( data_pipes[1], &c, 1 );
        if ( rc == -1 )
        {
            perror( "Parent: pipe write error" );
            close( data_pipes[1] );
            exit( 1 );
        }
    }
    /*Đóng đường ống phía đầu ghi để thông báo cho phía cuối đường ống dữ liệu đã hết*/
    close(data_pipe[1]);
    exit(0);
}

/*Chương trình chính*/
int main()
{
    int data_pipes[2];                /*Mảng chứa số mô tả đọc ghi của đường ống*/
    int pid;                          /*pid của tiến trình con*/
    int rc;                           /*Lưu mã lỗi trả về*/
    rc = pipe( data_pipes );          /*Tạo đường ống*/
    if ( rc == -1 )
```

```
    {
        perror( "Error: pipe not created" );
        exit( 1 );
    }
    /*Tạo tiến trình con*/
    pid = fork();
    switch ( pid )
    {
        case -1:              /*Không tạo được tiến trình con*/
            perror( "Child process not create" );
            exit( 1 );
        case 0:               /*Tiến trình con*/
            do_child( data_pipes );
        default:              /*Tiến trình cha*/
            do_parent( data_pipes );
    }
    return 0;
}
```

Bài 3: Chương trình sử dụng cơ chế đường ống giao tiếp hai chiều, dùng hàm **`fork()`** để nhân bản tiến trình. Tiến trình thứ nhất (tiến trình cha) sẽ đọc nhập liệu từ phía người dùng và chuyển vào đường ống đến tiến trình thứ hai (tiến trình con). Tiến trình thứ hai xử lý dữ liệu bằng cách chuyển tất cả ký tự thành chữ hoa sau đó gửi về tiến trình cha qua một đường ống khác. Cuối cùng tiến trình cha sẽ đọc từ đường ống và in kết quả của tiến trình con ra màn hình *(Sinh viên tự làm)*.

Bài 4: Tạo hai tiến trình tách biệt: **`producer.c`** là tiến trình sản xuất, liên tục ghi dữ liệu vào đường ống mang tên **`/tmp/my_fifo`** trong khi **`consumer.c`** là tiến trình tiêu thụ liên tục đọc dữ liệu từ đường ống **`/tmp/my_fifo`** cho đến khi nào hết dữ liệu trong đường ống thì thôi. Khi hoàn tất quá trình nhận dữ liệu, tiến trình **`consumer`** sẽ in ra thông báo kết thúc.

```
/* producer.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "my_fifo"           /*Tạo đường ống*/
#define BUFFER_SIZE PIPE_BUF          /*Vùng đệm dùng cho đường ống*/
#define TEN_MEG ( 1024 * 1024 * 10 )  /*Dữ liệu*/

int main() {
    int pipe_fd;
    int res;
    int open_mode = O_WRONLY;
    int bytes_sent = 0;
    char buffer[BUFFER_SIZE + 1];

    /*Tạo pipe nếu chưa có*/
    if ( access( FIFO_NAME, F_OK ) == -1 )
    {
        res = mkfifo( FIFO_NAME, (S_IRUSR | S_IWUSR) );
        if ( res != 0 )
        {
            fprintf( stderr, "FIFO object not created [%s]\n", FIFO_NAME);
            exit( EXIT_FAILURE );
        }
    }
    /*Mở đường ống để ghi*/
    printf( "Process %d starting to write on pipe\n", getpid() );
    pipe_fd = open( FIFO_NAME, open_mode);
    if ( pipe_fd != -1 )
    {
        /*Liên tục đổ vào đường ống*/
        while ( bytes_sent < TEN_MEG )
        {
            res = write( pipe_fd, buffer, BUFFER_SIZE );
            if ( res == -1 )
            {
                fprintf( stderr, "Write error on pipe\n" );
                exit( EXIT_FAILURE );
            }
            bytes_sent += res;
        }
        /*Kết thúc quá trình ghi dữ liệu*/
        ( void ) close( pipe_fd );
    }
    else
    {
```

```
        exit( EXIT_FAILURE );
    }
    printf( "Process %d finished, %d bytes sent\n", getpid(), bytes_sent );
    exit( EXIT_SUCCESS );
}
```

```
/* consumer.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "my_fifo"
#define BUFFER_SIZE PIPE_BUF

int main() {
    int pipe_fd;
    int res;
    int open_mode = O_RDONLY;
    int bytes_read = 0;
    char buffer[BUFFER_SIZE + 1];
    /* Mở đường ống để đọc */
    printf( "Process %d starting to read on pipe\n", getpid() );
    pipe_fd = open( FIFO_NAME, open_mode);
    if ( pipe_fd != -1 )
    {
        do
        {
            res = read( pipe_fd, buffer, BUFFER_SIZE );
            bytes_read += res;
        } while ( res > 0 );
        ( void ) close( pipe_fd );    /Kết thúc đọc*/
    }
    else
    {
        exit( EXIT_FAILURE );
    }
    printf( "Process %d finished, %d bytes read\n", getpid(), bytes_read );
    exit( EXIT_SUCCESS );
}
```

Chạy producer dưới nền, tiếp đến là consumer: **./producer & ./consumer**