CHƯƠNG III

LẬP TRÌNH NHÚNG NÂNG CAO

Lời đầu chương

Sau khi nghiên cứu chương II, lập trình nhúng căn bản, chúng ta đã có được những kiến thức cần thiết để tiến sâu hơn vào thế giới lập trình phần mềm hệ thống nhúng Linux đầy thú vị. Trước khi đi vào nghiên cứu những nội dung trong chương III, người học phải có những kiến thức nền tảng sau đây:

- Sử dụng được các lệnh trong ngôn ngữ lập trình C như: if, if...else, if...else if..., switch...case, ...; hiểu và vận dụng được các khái niệm trong ngôn ngữ lập trình C như: Con trỏ, cấu trúc, mảng, ... vào chương trình;
- Hiểu một cách tổng quát các kiến thức cơ sở lý thuyết về hệ điều hành nói chung. Phải giải thích được các khái niệm về tiến trình, tuyến, chia khe thời gian, ...trong hệ điều hành.
- Và một số những kỹ năng cần thiết như biên dịch chương trình ứng dụng, driver viết bằng ngôn ngữ lập trình C trong Linux; Sử dụng thành thạo các lệnh trong môi trường Shell của Linux;

Trong chương III-Lập trình nhúng nâng cao-chúng ta sẽ được tìm hiểu lý thuyết về hai lớp trong hệ thống phần mềm nhúng là user application và kernel driver cũng như một số những thao tác lập trình để xây dựng chương trình trong hai lớp này.

Phần lập trình nhúng nâng cao được biên soạn để viết hoàn chỉnh các driver và chương trình ứng dụng cho một số những thiết bị ngoại vi phổ biến thường được sử dụng trong việc dạy học như: LED đơn, LED 7 đoạn, LED ma trận, LCD, ADC,... Do đó những kiến thức được trình bày trong chương này chỉ là những kiến thức cần thiết để phục vụ cho việc lập trình dự án điều khiển các thiết bị ngoại vi trên. Những kiến thức chuyên sâu khác người học sẽ tự tìm hiểu trong những nguồn thông tin khác nhau được chúng tôi giới thiệu trong phần tài liệu tham khảo.

Các bài luyện tập trong quyển sách này bao gồm:

Phần A: Lập trình user application;

Phần này trình bày ví dụ về cách viết chương trình ứng dụng trong user, nhắc lại cách biên dịch và thực thi chương trình trong hệ điều hành. Bên cạnh đó các bạn sẽ tiếp xúc với hệ thống thời gian thực trong Linux thông qua các hàm trì hoãn thời gian, các hàm thao tác với thời gian thực, các hàm tạo tín hiệu định thời theo chu kỳ. Chúng ta cũng sẽ

nghiên cứu về lập trình đa tiến trình, đa tuyến về nguyên lý hoạt động và cách sử dụng vào các chương trình ứng dụng thời gian thực.

Phần B: Căn bản lập trình driver;

Sau khi lập trình chương trình ứng dụng trong lớp User thành thạo, chúng ta sẽ tiếp tục làm việc với lớp sâu hơn là Driver. Qua đó, người học sẽ hiểu vai trò của Driver, cách phân phối nhiệm vụ giữa Driver và Application sao cho đạt hiệu quả cao nhất. Các lệnh lập trình driver, các thao tác cài dặt và sử dụng driver cũng sẽ được chúng tôi trình bày trong chương này.

Trên đây là nội dung tổng quát của phần II-Lập trình hệ thống nhúng nâng cao. Mỗi bài luyện tập điều bao gồm trình bày lý thuyết và chương trình ví dụ, các chương trình đều được giải thích rõ ràng từng dòng lệnh đồng thời cũng có đưa ra kết quả thực thi chương trình trong hệ thống. Người học không những chỉ đọc mà còn phải kiểm tra kết quả thực thi bằng cách lập trình lại các chương trình ví dụ được nêu trong từng bài học để có thể hiểu được vai trò cốt lõi của chúng trong giáo trình.

PHẦN A

LẬP TRÌNH USER APPLICATION

BÀI 1

CHƯƠNG TRÌNH HELLOWORLD

I. Mở đầu:

"*Hello world!*" là một chương trình quan trọng và căn bản đầu tiên khi bước vào thế giới của bất kỳ một ngôn ngữ lập trình nào. Mặc dù đơn giản nhưng bài này sẽ cho chúng ta làm quen với các bước lập trình, biên dịch và thực thi một chương trình ứng dụng trong hệ thống Linux. Học bài này chúng ta có dịp ôn lại những thao tác biên dịch chương trình đã được học trong phần lập trình hệ thống nhúng căn bản.

II. Nội dung:

1. Hàm printf:

a. Mô tả lệnh:

```
int printf ( const char* format, …, <parameter_n>, …);
```

b. Giải thích chức năng:

Hàm `printf()` dùng in một thông tin cần hiển thị ra màn hình đầu cuối. Thông thường những thông tin này là thông báo của người lập trình về quá trình xử lý của chương trình đang đi đến đâu, có xảy ra lỗi hay không.

Hàm trả về một số kiểu `integer`. Nếu quá trình hiển thị thành công, hàm trả về một số dương. Ngược lại sẽ trả về số âm.

Tham số const `char* format` là định dạng chuỗi thông tin cần hiển thị.

Tùy theo chuỗi thông tin cần hiển thị, thì `<parameter_n>` là một giá trị thuộc dạng nào đó.

c. Ví dụ:

```
#include <stdio.h>
int main() {
   if  ( printf("Characters %d: %c, %c, %c, %c\n", 1, 'A', 65, 'B',
   66) == -1) {
         printf("Display error!\n");
   } else {
         printf("Display no error!\n");
   }
}
```

Kết quả xuất ra màn hình như sau:

```
Characters 1: A, A, B, B
Display no error!
```

**Chúng ta thấy trong câu lệnh `printf()` đầu tiên, ta xuất chuỗi "`Characters …`" ra màn hình. Chuỗi này có định dạng hiển thị ban đầu là một số nguyên `(%d)`, tiếp theo là 4 ký tự `(%c)`. Mặc dù các tham số không phải lúc nào cũng là nhưng ký tự thuần túy, nhưng hàm `printf()` cũng vẫn hiểu là những ký tự, vì hàm này đã dược định dạng phải hiểu là một ký tự. Ta ghi 65, hàm sẽ tự động chuyển số này thành ký tự '`A`' trong mã ascii. Tương tự cho các tham số khác. Hàm thực thi không có lỗi, vì vậy in ra câu thông báo "`Display no error!`".

2. Hàm exit:

d. Mô tả lệnh:

```
exit (int n);
```

**Hàm chứa trong thư viện `stdlib.h`

a. Giải thích chức năng:

Hàm `exit(n)` dùng để thoát khỏi một chương trình dang thực thi. Thông thường hàm này dùng cho trường hợp khẩn cấp, người lập trình muốn thoát khỏi chương trình một cách đột ngột khi phát sinh ra một lỗi nào đó đã được dự đoán trước.

Trong trường hợp xảy ra lỗi, người lập trình muốn biết đó là lỗi gì thì sẽ thêm vào thông tin cho n. n là một số nguyên integer do người lập trình định nghĩa, cho biết thông tin kèm theo khi thực hiên lệnh `exit(n);`

Lệnh `exit(n)` thường đặt cuối chương trình, giá trị trả về n được gán bằng 0, cho biết là chương trình thực thi không bi lỗi.

b. Ví dụ:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
if  ( printf("Characters %d: %c, %c, %c, %c\n", 1, 'A', 65, 'B', 66)
== -1) {
   printf("Display error!\n");
   exit(1); //Báo lỗi thoát là 1
} else {
```

```
    printf("Display no error!\n");
}
/*Chương trình thực thi không bị lỗi, trả về mã lỗi là 0*/
exit (0);
}
```

3. Chương trình helloworld:

a. Mã chương trình:

```
/*Khai báo thư viện chuẩn vào ra-do sử dụng lệnh printf*/
#include <stdio.h>
#include <stdlib.h>
/*Chương trình chính*/
int main() {
   /*In ra chuỗi "Hello world!"*/
   printf ("Hello world!");
   /*Thoát khỏi chương trình, trả về mã lỗi là 0*/
   exit(0);
}
```

b. Biên dịch và chạy chương trình:

- Chép chương trình đã hoàn thành vào máy Linux ảo;
- Biên dịch chương trình dùng lệnh sau:

```
arm-none-Linux-gnueabi-gcc HelloWorld.c -o HelloWorld
```

***Lệnh này sử dụng chương trình biên dịch chéo trong chương trình hỗ trợ biên dịch cho ARM. Ý nghĩa của lệnh này là biên dịch một tập tin chương trình có tên là HelloWorld.c, kết quả cho ra một tập tin mã máy có thể chạy được trên ARM có tên là HelloWorld.*

***Chú ý trước khi sử dụng lệnh này cần phải khai báo biến môi trường PATH trong Linux, sao cho trỏ đến nơi chứa các tập tin thực thi gcc cho ARM.*

- Chép tập tin đã biên dịch HelloWorld vào kit;
- Tiến hành chạy chương trình bằng dòng lệnh sau:

```
./HelloWorld
```

Khi nhấn enter dể chạy chương trình, hệ thống Linux sẽ xuất hiện thông báo lỗi như sau:

```
-sh: ./HelloWorld: Permission denied.
```

Nguyên nhân xảy ra lỗi là do tập tin chương trình `HelloWorld` vừa chép qua không có quyền thực thi trong hệ thống Linux. Để sửa lỗi này chúng ta cần phải cho phép tập tin chương trình `HelloWorld` thực thi được bằng dòng lệnh sau:

```
chmod 777 HelloWorld
```

Sau đó tiến hành chạy chương trình, chương trình chạy thành công. Lúc này màn hình hiển thị chuỗi ký tự "`Hello World!`" đã được lập trình trước đó.

```
./HelloWorld
   Hello world!
```

c. Giải thích chương trình:

Đây là một chương trình đơn giản, nhiệm vụ của chương trình là in ra chuỗi "`Hello world!`" ra màn hình hiển thị (console).

Phần đầu của chương trình là khai báo hai thư viện chuẩn `<stdio.h>` và `<stdlib.h>` dùng cho các hàm `printf` và `exit`.

Tiếp theo là khai báo và định nghĩa phần chương trình chính `main()`. Hàm `main()` có kiểu dữ liệu trả về là `int` và hàm này không có tham số.

Trong hàm `int main()`, ta sử dụng hai hàm `printf` và `exit`. Hàm `printf` (Định nghĩa trong thư viện `stdio.h`) dùng xuất một chuỗi ký tự được định dạng ra màn hình. Hàm `exit(n)` (định nghĩa trong thư viện `stdlib.h`) dùng thoát chương trình chính và trả về mã lỗi là 0.

III. Tổng kết:

Sau bài học này chúng ta đã biết cách xây dựng cho mình một chương trình ứng dụng Linux đơn giản xuất ký tự ra màn hình hiển thị. Sử dụng thành thạo hơn những thao tác biên dịch chương trình. Chúng ta còn biết cách cho phép một tập tin trong Linux được quyền thực thi để khắc phục lỗi "`permission denied`" trong hệ thống Linux. Kỹ thuật sử dụng hàm `printf()` rất quan trong trong việc thông báo lỗi trong quá trình lập trình ứng dụng, phục vụ cho việc sửa lỗi lập trình. Chúng ta cũng đã biết cách sử dụng hàm `exit(n)` trong việc bẫy lỗi chương trình.

Trong bài tiếp theo, chúng ta sẽ đi vào tìm hiểu những lệnh về trì hoãn thời gian trong hệ thống Linux. Các lệnh này rất quan trọng trong quá trình lập trình những ứng dụng có liên quan đến định thời chính xác về thời gian, hay nói đúng hơn là những công việc mang tính chu kỳ.

BÀI 2

TRÌ HOÃN THỜI GIAN TRONG USER APPLICATION

I. Kiến thức ban đầu:

Trì hoãn thời gian (delay), là một trong những thao tác quan trong nhất trong quá trình lập trình ứng dụng giao tiếp với các thiết bị ngoại vi, lập trình hoạt động mang tính chất chu kỳ. Trong các ứng dụng, chúng ta sử dụng lệnh delay để làm những công việc như: định thời gian cập nhật thông tin về ngày-tháng-năm hiện tại, thông tin về nhiệt độ hiện tại, hay định thời gian kiểm tra tình trạng hoạt động của một bộ phần nào đó trong hệ thống nhằm điều khiển những ngõ vào ra thích hợp. Các thiết bị ngoại vi (như LCD, các thiết bị lưu trữ (ổ đĩa VCD, ổ đĩa cứng, thẻ nhớ, …))thông thường có thời gian đáp ứng thấp hơn rất nhiều lần so với tốc độ làm việc của CPU. Nhằm mục đích dồng bộ hóa hoạt dộng giữa CPU và các thiết bị ngoại vi, một trong những biện pháp đơn giản nhất là trì hoãn hoạt động của CPU, làm cho CPU phải chờ các thiết bị làm việc xong rồi mới tiếp tục công việc tiếp theo.

***Trong thực tế người ta dùng biện pháp khác hiệu quả hơn rất nhiều. Đó là dùng ngắt điều khiển hoạt dộng của CPU. Thay vì bắt CPU phải chờ, CPU sẽ tiến hành làm công việc khác. Khi thiết bị ngoại vi thực thi xong nhiệm vụ nó phát sinh ra một tín hiệu báo cho CPU biết để CPU nhận thông tin hay ra lệnh tiếp theo cho thiết bị hoạt động. Vấn đề ngắt sẽ được trình bày trong những phần tiếp theo của quyển sách này.*

Trì hoãn thời gian trong hệ thống Linux khác với trì hoãn thời gian thông thường chúng ta đã hiểu là trong lúc trì hoãn thời gian, CPU không làm gì cả ngoài việc tiêu tốn thời gian trong việc chờ đợi. Trong hệ điều hảnh Linux nói riêng và các hệ điều hành khác nói chung, điều có một cơ chế quản lý thời gian hoạt động của từng tiến trình và từng tuyến. Nghĩa là các tiến trình và tuyến được cho phép hoạt động song song trong hệ thống. *(**Định nghĩa tiến trình và tuyến sẽ được trình bày trong những bài tiếp theo).* Khi một tiến trình sử dụng lệnh trì hoãn thời gian, CPU không tiêu tốn thời gian hoạt động cho việc chờ đợi, mà nó sẽ thực hiện tiến trình khác, khi thực hiện xong tất cả những tiến trình đó, nó quay trở lại tiến trình cũ có trì hoãn thời gian và kiểm tra xem lệnh trì hoãn thời gian có hoàn thành hay chưa. Nếu chưa hoàn thành, hệ điều hành tiếp tục thực hiện những tiến trình khác. Nếu hoàn thành, hệ điều hành sẽ thực hiện những

lệnh tiếp theo trong tiến trình hiện tại. Quá trình chia khe thời gian vẫn tiếp tục. Hoạt động của hệ điều hảnh cũng tương tự khi có nhiều tiến trình sử dụng hàm trì hoãn thời gian. Trong hệ điều hành nhúng Linux dùng cho Kit KM9260, lệnh trì hoãn thời gian sử dụng hoạt dộng của timer dịnh thời. Khi một lệnh trì hoãn thời gian được khởi động, timer sẽ được gán một giá trị cùng với những thông số cài đặt phù hợp sao cho có khả năng trì hoãn đúng bằng khoảng thời gian mong muốn. Timer bắt đầu chạy, khi đạt đến điều kiện định thời, timer sinh ra một ngắt báo cho CPU biết để tiến hành thoát khỏi lệnh khi hoãn hiện tại.

Bài này sẽ giới thiệu cho chúng ta những phương pháp trì hoãn thời gian cơ bản thường được sử dụng trong quá trình lập trình ứng dụng cho một hệ thống nhúng. Các ưu và nhược điểm khi dùng từng phương pháp để người lập trình có thể lựa chọn cho mình phương pháp hiểu quả nhất phù hợp với yêu cầu ứng dụng. Bên cạnh đó chúng ta sẽ được tìm hiểu các lệnh về Realtime trong hệ điều hành Linux. Những kiến thức này có vai trò quan trong trong việc thiết kế ứng dụng điều khiển LCD sau này và những ứng dụng có liên quan đến thời gian thực khác;

Có hai lớp chúng ta có thể thực hiện những lệnh về thời gian thực, lớp driver và user application. Ở đây trong phần lập trình ứng dụng chúng ta chỉ đề cập những lệnh thời gian thực trong lớp user application. Còn những lệnh thời gian thực trong driver sẽ được đề cập trong phần lập trình driver của quyền sách này.

II. Nội dung:

1. Hàm sleep:

a. Cú pháp:

```
#include <unitstd.h>
void sleep (unsigned int seconds);
```

hay

```
unsigned int sleep(unsigned int seconds);
```

b. Giải thích:

Hàm sleep làm nhiệm vụ trì hoãn một khoảng thời gian, có đơn vị là giây. Trì hoãn ở đây không phải là tiêu tốn thời gian làm việc của CPU trong việc chờ đợi thời gian trì hoãn kết thúc mà hệ điều hành vẫn tiếp tục chia khe cho CPU thực hiện những công việc khác. Nghĩa là trong lúc trì hoãn, CPU vẫn làm những công việc khác do hệ điều hành phân công thay vì chỉ chờ dợi hết thời gian.

Hàm `sleep` có một đối số và dữ liệu trả về. Đối số `unsigned int seconds` là khoảng thời gian cần trì hoãn, tính bằng đơn vị giây. Hàm `sleep` trả về giá tri 0 khi thời gian trì hoãn đã hết, trả về số giây còn lại khi chưa kết thúc thời gian trì hoãn.

Chúng ta có thể không cần sử dụng dữ liệu trả về của hàm, bằng cách dùng trường hợp thứ nhất `void sleep (unsigned int seconds);`

Hàm `sleep` được dùng trong những trường hợp trì hoãn thời gian không đòi hỏi tính chính xác cao. Thông thường là những khoảng thời gian lớn, vì đơn vị trì hoãn nhỏ nhất là giây.

c. Ví dụ:

Đoạn chương trình sau sẽ minh họa cho chúng ta cách sử dụng hàm `sleep()`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
   printf ("Delay in 10s …\n");
   sleep(10);
   printf ("Delay finished.\n");
   exit(0);
}
```

Chương trình này làm nhiệm vụ in ra chuỗi "`Delay in 10s …`" để thông báo rằng hàm `sleep` bắt đầu thực thi. Sau khoảng thời gian trì hoãn 10s, chương trình tiếp tục in ra chuỗi "Delay finished." thông báo rằng hàm `sleep` chấm dứt thực thi.

Chúng ta tiến hành biên dịch chương trình trên nạp vào kit. Kết quả khi thực thi chương trình như sau:

```
Delay in 10s …
(Chương trình trì hoãn 10s)
Delay finished.
```

2. Hàm usleep:

a. Cú pháp:

```
#include <unistd.h>
void usleep (unsigned long usec);
```

hay

```
int usleep (usigned long usec);
```

b. Giải thích:

Cũng tương tự như hàm `sleep()`, hàm usleep cũng làm nhiệm vụ trì hoãn một khoảng thời gian theo yêu cầu của người lập trình. Tuy nhiên có một điểm khác biệt là đơn vị thời gian trì hoãn tính bằng micrô giây. Nghĩa là độ phân giải nhỏ hơn 1 triệu lần so với hàm sleep(). Hàm usleep ứng dụng trong việc trì hoãn thời gian một cách chính xác cao và không quá dài (nhỏ hơn 1 giây).

Hàm `usleep()` có hai dạng sử dụng. Một dạng không có giá trị trả về và một dạng có giá trị trả về. Giá trị trả của hàm usleep là 0 nếu hoàn tất thời gian trì hoãn, ngược lại sẽ trả về thởi gian trì hoãn còn lại. Đối số của hàm usleep là một số nguyên không dấu có độ lớn là 4 bytes (dạng unsigned long) chính là số micrô giây cần trì hoãn.

c. Ví dụ:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
   printf ("Delay in 10us …\n");
   usleep(10);
   printf ("Delay finished.\n");
   exit(0);
}
```

Chương trình này làm nhiệm vụ in ra chuỗi "Delay in 10us …" để thông báo rằng hàm `usleep` bắt đầu thực thi. Sau khoảng thời gian trì hoãn 10s, chương trình tiếp tục in ra chuỗi "Delay finished." thông báo rằng hàm `sleep` chấm dứt thực thi.

Chúng ta tiến hành biên dịch chương trình trên nạp vào kit. Kết quả khi thực thi chương trình như sau:

```
Delay in 10us …
(Chương trình trì hoãn 10us)
Delay finished.
```

Do thời gian trì hoãn là rất ngắn nên chúng ta chỉ thấy hai dòng chữ xuất hiện cùng một lúc. Nếu muốn nhìn thấy hai dòng chữ trên xuất hiện tuần tự, chúng ta tiến hành thay đổi thông số như sau: usleep(1000000); thì hệ thống sẽ trì hoãn 1s.

3. Hàm nanosleep:

a. Cú pháp:

```
#include <time.h>
int nanosleep (const struct timespec *req, struct timespec *rem);
```

b. Giải thích:

Đề hiểu cách sử dụng của hàm `nanosleep` trước tiên chúng ta phải hiểu rõ cấu trúc `timespec` trong thư viện `<time.h>`.

Cấu trúc struct `timespec` được Linux định nghĩa như sau:

```
struct timespec {
__kernel_time_t tv_sec;     /* seconds */
long        tv_nsec;        /* nanoseconds */
};
```

Cấu trúc `timespec` bao gồm có hai phần tử con, `__kernel_time_t tv_sec` và `long tv_nsec`. `timespec` là cấu trúc thời gian trong Linux, bao gồm có giây (`tv_sec`) và nano giây (`tv_nsec`) của giây hiện tại. Ví dụ để tạo một khoảng thời gian là 1000 nano giây, chúng ta làm các bước như sau:

```
/*Định nghĩa biến thuộc cấu trúc timespec*/
struct timespec time_delay;
/*Gán giá trị thời gian cần trì hoãn cho biến*/
time_delay.tv_sec = 0;
time_delay.tv_nsec = 1000;
```

Như vậy thông qua việc gán giá trị cho các phần tử trong cấu trúc `timespec`, chúng ta đã tạo ra được một khoảng thời gian định thời cần thiết cho việc sử dụng hàm `nanosleep` sau đây.

Bây giờ chúng ta sẽ đi vào tìm hiểu các tham số trong hàm `nanosleep`. Hàm `nanosleep` có hai tham số cần quan tâm. `const struct timespec *req` là con trỏ trỏ đến địa chỉ của cấu trúc `timespec` được định nghĩa trước đó. Theo như các bước làm trước thì chúng ta đã định nghĩa biến `time_delay` có thời gian định thời là 1000 nano giây. Địa chỉ của biến này sẽ được gán vào đối số thứ nhất của hàm `nanosleep`. Đối số thứ hai của hàm `nanosleep` cũng là một cấu trúc thời gian `timespec` trong Linux, struct `timespec *rem`. rem cũng là một con trỏ trỏ đến địa chỉ của một biến cấu trúc `timespec` đã dịnh nghĩa trước đó. `*rem` làm nhiệm vụ lưu giá trị thời gian còn lại khi quá trình trì

hoãn chưa kết thúc (kết thúc bị lỗi). Nếu không muốn lưu giá trị thời gian này, chúng ta có thể không cần sử dụng tham số `struct timespec *rem` của `nanosleep` bằng cách khai báo trong tham số thứ hai hằng số NULL. Hàm `nanosleep` có giá trị trả về kiểu `int`, trả về giá trị 0 nếu quá trình trì hoãn không có lỗi. Ngược lại, trả về giá trị -1. Trong trường hợp này, giá trị thời gian còn lại sẽ được lưu vào trong biến con trỏ `*rem` nếu đã khai báo trong đối số thứ hai. Người lập trình có thể sử dụng giá trị này để gọi một hàm trì hoãn khác, trì hoãn phần còn lại chưa hoàn thành.

Hàm `nanosleep` có một ưu điểm là thời gian trì hoãn có độ phân giải cao hơn rất nhiều so với hàm `sleep` và `usleep`. Hàm `nanosleep` sử dụng trong những ứng dụng đòi hỏi trì hoãn với độ chính xác rất cao (tính bằng một phần tỷ của giây).

c. Ví dụ:

Đoạn chương trình sau sẽ giúp cho chúng ta biết cách sử dụng hàm nanosleep.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
/*Khai báo cấu trúc thời gian dùng lưu trữ giá trị khoảng thời gian cần trì hoãn*/
   struct timespec rq, rm;
   /*Cài đặt khoảng thời gian cần trì hoãn*/
   rq.tv_sec = 0;
   rq.tv_nsec = 1000;
   /*In thông báo quá trình trì hoãn bắt đầu*/
   printf ("Delay has just begun …\n");
/*Gọi hàm nanosleep trì hoãn một khoảng thời gian đã cài đặt*/
   nanosleep (&rq, &rm);
/*In thông báo quá trình trì hoãn kết thúc*/
   printf ("Delay has just finished. \n");
   /*Kết thúc chương trình*/
   exit (0);
}
```

Như vậy để sử dụng được hàm nanosleep chúng ta phải làm theo những bước sau:

- Khai báo biến struct timespec để lưu giá trị khoảng thời gian cần trì hoãn, và biền struct timespec để chứa giá trị trả về của hàm `nanosleep`;

```
struct timespec rq, rm;
```

- Gán giá trị thời gian cần trì hoãn;

```
rq.tv_sec = 0;
rq.tv_nsec = 1000;
```

- Gọi hàm nanosleep; Gán địa chỉ của biến vào các con trỏ tham số của hàm;

```
nanosleep (&rq, &rm);
```

Biên dịch và chạy chương trình trên kit, màn hình hiển thị kết quả như sau:

```
Delay has just begun …
```

(Chương trình trì hoãn 1000ns)

```
Delay finished.
```

Do thời gian trì hoãn rất nhỏ nên hai chuỗi thông tin xuất hiên gần như dồng thời. Có thể tăng thời gian trì hoãn bằng cách thay đổi thông số của `tv_sec` và `tv_nsec`, chẳng hạn như:

```
req.tv_sec = 1;
req.tv_nsec = 0;
```

thì thời gian trì hoãn sẽ tăng lên 1s, lúc này chúng ta có thể thấy 2 dòng thông tin xuất hiện tuần tự cách nhau 1s.

4. Hàm alarm:

a. Cú pháp:

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

b. Giải thích:

Hàm `alarm()` tạo một tín hiệu có tên là `SIGALRM` sau một khoảng thời gian là `seconds` giây, được quy định bởi người lập trình. Nếu giá trị `seconds` bằng 0 thì hàm `alarm` bị vô hiệu hóa.

Hàm `alarm` thường được dùng kết hợp với hàm xử lý tín hiệu ngắt, hàm này làm nhiệm vụ thực hiện những công việc do người lập trình định nghĩa khi tín hiệu ngắt xảy ra. Thông thường chúng ta sử dụng hàm sigaction() hay signal() để thực hiện nhiệm vụ trên.

c. Ví dụ:

Đoạn chương trình sau dây sẽ cho chúng ta biết cách tạo một khoảng thời gian định thời cho tín hiệu SIGALRM tác động:

```
alarm (30); /*Sau một khoãng thời gian là 30s, thì tín hiệu SIGALRM sẽ tác động tích cực, công việc của chúng ta là xây dựng một hàm thực thi khi có tín hiệu tác động*/

/*Sau khi tín hiệu SIGALRM tác động thì cần phải tạo lại định thời cho tín hiệu này bằng cách gọi lại hàm SIGALRM*/
```

Hàm `alarm()` dùng để định thời tạo ra một tín hiệu ngắt. Thông thường hàm này không dùng như một hàm riêng biệt mà nó phải kết hợp với những hàm khác. Đó là những hàm xử lý tín hiệu ngắt.

5. Đồng hồ thời gian thực trong Linux:

Thời gian thực trong hệ điều hành mang một ý nghĩa rất rộng. Trong phần này chúng ta nói đến thời gian thực với ngụ ý là đồng hồ thời gian thực trong hệ thống Linux. Trong phần này chúng ta sẽ nghiên cứu những lệnh truy xuất những thông tin về thời gian trong hệ thống (bao gồm ngày, tháng, năm, giờ, phút, giây, …);

Những hàm truy xuất về thời gian được định nghĩa trong thư viện `<time.h>`. Các hàm trong thư viện <time.h> bao gồm:

- Hàm thao tác với thời gian thực;
  - clock: Trả về khoảng thời gian (đơn vị là ticks) kể từ khi chương trình thực thi;
  - difftime: Trả về khoảng cách thời gian giữa hai thời điểm;
  - time: Trả về thời gian hiện tại của hệ thống;
- Hàm chuyển đổi thời gian:
  - asctime: chuyển đổi kiểu cấu trúc thời gian tm thành chuỗi thời gian;
  - ctime: chuyển đổi kiểu dữ liệu thời gian time_t thành chuỗi thời gian;
  - gmtime: chuyển đổi kiểu dữ liệu thời gian time_t thành cấu trúc thời gian tm theo giờ UTC;
  - mktime: Chuyển đổi cấu trúc thời gian tm thành loại dữ liệu thời gian time_t;

- localtime: Chuyển đổi loại dữ liệu thời gian time_t thành cấu trúc thời gian tm theo thời gian cục bộ;
- strftime: Chuyển cấu trúc thời gian struct tm thành chuỗi theo định dạng của người dùng;

- Các hằng số xây dựng sẵn bao gồm:
  - CLOCKS_PER_SEC: là khoảng thời gian 1s tính bằng đơn vị ticks;
  - NULL: Con trỏ rỗng;
- Các loại dữ liệu thời gian chuẩn:
  - clock_t: Loại dữ liệu thời gian dạng xung ticks;
  - size_t;
  - time_t: kiểu dữ liệu thời gian;
  - struct tm: cấu trúc thời gian;

Chúng ta sẽ đi sâu tìm hiểu từng nội dung đã liệt kê trên đây:

5.1. Các loại dữ liệu thời gian chuẩn:

5.1.1. clock_t:

Theo thư viện `<time.h>` thì `clock_t` được định nghĩa như sau:

```
typedef __kernel_clock_t      clock_t;
```

*/*Trong thư viện <types.h>*/*

Mà `__kernel_clock_t` được định nghĩa:

```
typedef long        __kernel_clock_t;
```

*/*Trong thư viện <posix_types.h>*/*

Như vậy, `clock_t` thực chất là một kiểu số nguyên `long`. Nhiệm vụ của biến này là lưu những giá trị khoảng thời gian (tính bằng ticks) do các hàm thao tác trên thời gian trả về.

5.1.2. size_t:

Theo thư viện `<time.h>` thì `size_t` được định nghĩa như sau:

```
typedef __kernel_size_t size_t;
```

*/*Trong thư viện <types.h>*/*

mà `__kernel_size_t` được định nghĩa:

```
typedef unsigned long   __kernel_size_t;
```

*/*Trong thư viện <posix_types.h>*/*

Như vậy, `size_t` thực chất là một kiểu số nguyên `unsigned long`. Nhiệm vụ của biến này là lưu những giá trị kích thước của một biến nào đó, do hàm sizeof, maxsize, … trả về.

5.1.3. time_t:

Kiểu dữ liệu time_t đươc định nghĩa trong thư viện `<time.h>` như sau:

```
typedef __kernel_time_t time_t;
```

*/*Trong thư viện <types.h>*/*

mà `__kernel_time_t` được định nghĩa:

```
typedef long __kernel_time_t;
```

*/*Trong thư viện <posix_types.h>*/*

Như vậy, `size_t` thực chất là một kiểu số nguyên `long`. Nhiệm vụ của loại biến này là lưu giá trị thời gian được trả về bởi các hàm về thời gian như time, difftime, …

5.1.4. struct tm:

Trong thư viện <time.h> struct tm được định nghĩa như sau:

```
struct tm {
/*
* Số giây trong một phút
* giá trị nằm trong khoảng từ 0 đến 59
*/
int tm_sec;
/* Số phút trong một giờ, có giá trị từ 0 đến 59*/
int tm_min;
/* Số giờ trong một ngày có giá trị từ 0 đến 23 */
int tm_hour;
/* Số ngày trong một tháng, giá trị trong khoảng 0 đến 31 */
int tm_mday;
/* Số tháng trong một năm, giá trị nằm trong khoảng 0 đến 11 */
int tm_mon;
/* Số năm kể từ năm 1900 */
long tm_year;
/* Số ngày trong tuần, có giá trị từ 0 đến 6 */
int tm_wday;
/* Số ngày trong một năm, tính từ ngày 1 tháng 1, có giá trị từ  0 đến 365 */
int tm_yday;
};
```

Chúng ta thấy cấu trúc struct tm là một cấu trúc xây dựng sẵn với tất cả những thông tin liên quan đền thời gian thực: Năm, tháng, ngày, giờ, phút, giây, ngày trong tuần, … Cấu trúc này dúng làm kiểu dữ liệu trả về cho các hàm thao tác vời thời gian. Chúng ta sẽ nghiên cứu ngay trong phần sau của chương.

5.2. Các hằng số xây dựng sẵn trong <time.h>:

5.2.1. CLOCKS_PER_SEC:

Đây là một hằng số được định nghĩa sẵn quy định số ticks trong một giây. Như vậy trong một giây sẽ có CLOCKS_PER_SEC ticks. Chúng ta dùng hằng số này để quy đổi ra giây giá trị trả về của hàm clock(). Hay dùng làm hằng số qui đổi ra giây của tham số của một số hàm trì hoãn thời gian tính bằng ticks.

5.2.2. NULL:

NULL là một hằng số quy định một con trỏ rỗng, hay nói cách khác con trỏ này không trỏ đến bất kỳ một đối tượng nào cả. Hằng số này sử dụng trong những trường hợp khi muốn vô hiệu hóa một tham số nào đó của hàm (khi không muốn sử dụng một tham số của hàm nào đó).

***Trên đây chúng ta đã tìm hiểu những loại dữ liệu, hằng số quan trọng của thời gian thực trong Linux. Sau đây là những hàm sử dụng những loại dữ liệu này. Mỗi hàm chúng ta sẽ có một ví dụ minh họa cách sử dụng, các bạn cần phải đọc hiểu và biên dịch chạy trên hệ thống Linux nhằm nắm vững thêm nguyên lý hoạt động của chúng. Từ đó, sẽ có thể ứng dụng những hàm này trong những ứng dụng khác liên quan đến thời gian thực.*

5.3. Các hàm thao tác với thời gian thực:

5.3.1. clock:

a. Cú pháp:

```
#include <time.h>
clock_t clock (void);
```

b. Giải thích:

Hàm `clock` có kiểu dữ liệu trả về là `clock_t` và không có tham số.

Nhiệm vụ của hàm `clock` là trả về khoảng thời gian tính từ khi chương trình chứa hàm `clock` thực thi cho đến khi hàm `clock` được gọi.

Chương trình ví dụ sau sẽ cho chúng ta biết rõ cách sử dụng hàm này hơn.

c. Chương trình ví dụ:

Mục đích của chương trình này là tạo ra một chương trình trì hoãn. Thời gian trì hoãn tính bằng giây do người lập trình quy định.

```
/*chương trình trì hoãn thời gian ứng dụng hàm clock*/
/*Khai báo thư viện dùng cho hàm printf*/
#include <stdio.h>
/*Khai báo thư viện dùng cho các hàm thời gian*/
#include <time.h>
/*Tạo hàm wait trì hoãn thời gian. Hàm này có tham số là seconds giây cần trì hoãn, không có giá trị trả về */
void wait (int seconds) {
/*Khai báo biến loại clock_t để lưu dữ liệu kiểu ticks*/
   clock_t endwait;
/*Chọn khoảng thời gian kết thúc là thời gian hiện tại, cộng với khoảng thời gian là một giây*seconds*/
   endwait = clock () + seconds * CLOCKS_PER_SEC;
/*Chờ cho đến khi thời gian hiện tại lớn hơn thời gian kết thúc  thì thoát*/
   while (clock () < endwait) {}
}
/*Chương trình chính ứng dụng hàm wait(seconds)*/
int main () {
/*Khai báo biến int n để lưu giá trị đếm xuống*/
   int n;
/*In câu thông báo bắt đầu đếm xuống*/
   printf ("Starting countdown…\n");
/*Vòng lặp for đếm xuống*/
   for (n = 10; n>0; n--) {
/*In giá trị hiện tại của n để theo dõi*/
   printf("%d\n",n);
/*Gọi hàm wait đã được lập trình trước đó trì hoãn một khoảng  thời gian là 1 giây*/
   wait(1);
}
```

```
/*Sau khi thoát khỏi vòng lặp, in ra câu thông báo kết thúc*/
printf("FIRE!!!\n");
/*Trả về giá trị 0 cho hàm main, để thông báo là không có lỗi  xảy ra*/
return 0;
}
```

Ý nghĩa từng dòng lệnh đã được giải thích rất kỹ trong chương trình. Chúng ta tiến hành biên dịch và sao chép vào kít. Sau khi tiến hành thực thi chương trình chúng ta sẽ có kết quả như sau:

```
 ./Time_0
Starting countdown...
10
9
8
7
6
5
4
3
2
1
FIRE!!!
```

Như vậy chúng ta đã thực hiện thành công chương trình trì hoãn thời gian sử dụng hàm clock để lấy thời gian hiện tại tính từ lúc chương trình thực thi và sử dụng hằng số CLOCKS_PER_SEC. Tuy nhiên mục đích của chương trình trên không phải là để tạo ra một cách trì hoãn thời gian khác mà chỉ giúp chúng ta hiểu rõ cách sử dụng hàm clock. Thực ra, trì hoãn thời gian theo cách này không hiệu quả cho những ứng dụng lớn, vì tiêu tốn thời gian hoạt động của tiến trình vô ích.

5.3.2. difftime:

a. Cú pháp:

```
#Include <time.h>
double difftime (time_t time2, time_t time1);
```

b. Giải thích:

Hàm difftime có hai đối số kiểu `time_t`. Đối số thứ nhất `time_t time2` là thời điểm ban đầu, đối số thứ hai `time_t time1` là thời điểm lúc sau. Hàm `difftime` có kiểu dữ liệu trả về là số thực dạng `double` là độ chênh lệch thời gian tính bằng giây giữa hai thời điểm `time2` và `time1`.

c. Chương trình ví dụ:

Chương trình này làm nhiệm vụ tính toán thời gian nhập dữ liệu của người dùng, sử dụng hàm difftime để tìm sự chênh lệch thời gian giữa hai lần cập nhật (trước và sau khi nhập dữ liệu).

```
/*chương trình ví dụ hàm difftime*/
/*Khai báo thư viện dùng cho hàm printf và hàm gets*/
#include <stdio.h>
/*Khai báo thư viện dùng cho các hàm về thời gian*/
#include <time.h>
/*Chương trình chính*/
int main (void) {
/*Khai báo biến start: lưu thời điểm ban đầu, end: lưu thời  điểm lúc sau thuộc kiểu dữ liệu time_t*/
time_t start, end;
/*Khai báo biến lưu tên người dùng nhập vào, chứa tối đa 256 ký  tự*/
char szInput [256];
/*Khai báo biến số thực lưu giá trị khoảng thời gian khác biệt  của hai thời điểm start và end*/
/*Cập nhật thời gian hiện tại, trước khi người dùng nhập thông  tin. Sử dụng hàm time(), là hàm trả về thời gian hiện tại của hệ  thống, kiểu dữ liệu trả về là con trỏ time_t*/
time (&start);
/*In ra thông báo yêu cầu người sử dụng nhập thông tin cho hệ  thống.*/
printf ("Please, enter your name: ");
/*Yều cầu nhập vào biến mảng szInput*/
gets(szInput);
/*Cập nhật thời gian hiện tại, sau khi người dùng đã nhập thông  tin. Sử dung hàm time(time_t *time_val) để lấy thời gian hiện  tại*/
```

```
time(&end);
/*Sử dụng hàm difftime để tính thời gian chênh lệch giữa start và end*/
dif = difftime (end,start);
/*In ra thông báo đã nhận được thông tin người dùng*/
printf ("Hi %s. \n", szInput);
/*In ra thông báo khoảng thời gian tiêu tốn khi đợi nhập dữ liệu*/
printf  ("It took you %.21f seconds to type your name. \n", dif);
/*Trả về 0 cho hàm main báo rằng quá trình thực thi không bị lỗi.*/
return 0;
}
```

Tiến hành biên dịch và chạy chương trình trên kit, chúng ta có kết quả như sau:

```
./time_difftime_linux
Please, enter your name: Nguyen Tan Nhu
Hi Nguyen Tan Nhu.
It took you 5.00 seconds to type your name.
```

Trước khi gọi hàm gets(), chúng ta gọi hàm time() để lấy về thời gian hiện tại của hệ thống, sau khi người sử dụng nhập xong dữ liệu, ta lại gọi hàm time() để cập nhật thời gian lúc sau. Tiếp đến gọi hàm difftime để lấy về khoảng thời gian sai biệt giữa hai thời điểm. Phần sau chúng ta sẽ giải thích rõ hơn hoạt động của hàm time trong hệ thống linux.

5.3.3. time:

a. Cú pháp:

```
#include <time.h>
time_t time(time_t* timer);
```

b. Giải thích:

Hàm `time` dùng để lấy về thời gian hiện tại của hệ thống linux. Hàm time có đối số `time_t* time` là con trỏ lưu giá trị trả về của hàm. Đối số này có thể bỏ trống bằng cách điền hằng `NULL` lúc này thời gian hiện tại sẽ là giá trị trả về của hàm.

c. Chương trình ví dụ:

Chương trình ví dụ sau đây sẽ cho chúng ta hiểu rõ cách sử dụng hàm time (Chúng ta có thể tham khảo ví dụ trên để xem cách sử dụng hàm time). Chương trình này sẽ in ra số giờ của hệ thống kể từ ngày 1 tháng 1 năm 1970.

```
/*Chương trình ví dụ hàm time*/
```

```
/*Khai báo các thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <time.h>
/*Bắt đầu chương trình chính*/
int main () {
/*Khai báo biến time_t seconds làm giá trị trả về cho hàm time*/
   time_t seconds;
/*Sử dụng hàm time lấy về thời gian hiện tại của hệ thống, là số giây kể từ ngày 1 tháng 1 năm 1970*/
   seconds = time (NULL);
/*In ra thời gian hiện tại*/
   printf ("%ld hours since January 1st, 1970\n", seconds/3600);
/*Trả về giá trị 0 cho hàm main, thông báo quá trình thực thi không có lỗi*/
return 0;
}
```

Biên dịch và thực thi chương trình trên kit, chúng ta có kết quả như sau:

```
./time_time_linux
366958 hours since January 1, 1970
```

Như vậy hàm `time` làm nhiệm vụ trả về thời gian hiện tại của hệ thống tính bằng giây kể từ ngày 1 tháng 1 năm 1970. Chúng ta có thể chuyển thành dạng ngày tháng năm, giờ phút giây bằng cách sử dụng hàm `gmtime` hay `localtime`.

5.4. Các hàm chuyển đổi thời gian thực:

5.4.1. asctime:

a. Cú pháp:

```
#include <time.h>
char* asctime (const struct tm * timeptr);
```

b. Giải thích:

Hàm asctime có chức năng biến đổi cấu trúc thời gian dạng struct tm thành chuỗi thời gian có dạng Www Mmm dd hh:mm:ss yyyy. Trong đó Www là ngày trong tuần, Mmm là tháng trong năm, dd là ngay trong tháng, hh:mm:ss là giờ:phút:giây trong ngày, cuối cùng yyyy là năm.

Chuỗi thời gian được kết thúc bằng 2 ký tự đặc biệt: ký tự xuống dòng "\n" và ký tự NULL.

Hàm có tham số là con trỏ cấu trúc `struct tm * timeptr`, giá trị trả về là con trỏ ký tự `char *`

Ví dụ sau đây sẽ cho chúng ta hiểu rõ cách sử dụng hàm asctime.

c. Chương trình ví dụ:

Nhiệm vụ của đoạn chương trình này là in ra chuỗi thời gian trả về của hàm asctime.

```
/*Chương trình ví dụ hàm asctime*/
/*Khai báo các thư viện cho các hàm dùng trong chương trình*/
#inlcude <stdio.h>
#include <time.h>
/*Chương trình chính*/
int main () {
/*Khai báo biến time_t rawtime để lưu thời gian hiện tại của hệ thống*/
time_t rawtime;
/*Khai báo biến con trỏ cấu trúc struct tm để lưu thời gian*/
struct tm * timeinfo;
/*Đọc về thời gian hiện tại của hệ thống*/
 time ( &rawtime );
/*Chuyển đổi kiểu thời gian time_t thành cấu trúc thời gian struct tm*/
timeinfo = localtime ( &rawtime );
/*In thời gian hệ thống dưới dạng chuỗi*/
printf ("The curent date/time is: %s", asctime (timeinfo) );
/*Trả về giá trị 0 cho hàm main để thông báo không có lỗi xảy ra trong quá trình thực thi lệnh*/
return 0;
}
```

Sau khi biên dịch và chạy chương trình trên kit, chúng ta có kết quả sau:

```
./time_asctime_linux
The current date/time is: Sat Nov 12 06:04:50 2011
```

Hàm asctime thường dùng trong những trường hợp cần biến một cấu trúc thời gian thành chuỗi ký tự ascii phục vụ cho việc hiển thị trên các thiết bị đầu cuối, hay trong các tập tin cần quản lý về thời gian.

5.4.2. ctime:

a. Cú pháp:

```
#include <time.h>
char * ctime ( const time_t * timer );
```

b. Giải thích:

Hàm ctime làm nhiệm vụ chuyển thông tin về thời gian dạng time_t thành thông tin về thời gian dạng chuỗi ký tự mã ascii. Chuỗi thông tin thời gian này có dạng Www Mmm dd hh:mm:ss yyyy. Trong đó Www là ngày trong tuần, Mmm là tháng trong năm, dd là ngay trong tháng, hh:mm:ss là giờ:phút:giây trong ngày, cuối cùng yyyy là năm.

Chuỗi thời gian được kết thúc bằng 2 ký tự đặc biệt: ký tự xuống dòng "\n" và ký tự NULL.

Hàm ctime có tham số là con trỏ biến thời gian dạng `time_t * timer`, giá trị trả về của hàm là con trỏ ký tự `char *`

Chương trình ví dụ sau đây sẽ cho chúng ta hiểu rõ cách sử dụng hàm ctime.

c. Chương trình ví dụ:

Tương tự như chương trình ví dụ trên, chương trình này cũng lấy thời gian hiện tại của hệ thống chuyền sang chuỗi thông tin về thời gian cuối cùng in ra màn hình hiển thị. Nhưng lần này chúng ta sử dụng hàm ctime.

```
/*Chương trình ví dụ hàm ctime*/
/*Khai báo các thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <time.h>
/*Chương trình chính*/
int main () {
   /*Khai báo biến kiểu thời gian time_t lưu thời gian hiện tại của hệ thống*/
time_t rawtime;
/*Đọc về giá trị thời gian hiện tại của hệ thống*/
time ( &rawtime );
/*In chuỗi thông tin thời gian ra màn hình*/
printf ( "The current local time is: %s", ctime (&rawtime) );
   /*Trả về giá trị 0 cho hàm main, thông báo rằng quá trình thực thi lệnh không có lỗi*/
return 0;
```

```
}
```

Biên dịch và chạy chương trình trên kit, chúng ta có kết quả như sau:

```
./time_ctime_linux
The current local time is: Sat Nov 12 07:03:28 2011
```

Kết quả hiển thị cũng tương tự như hàm asctime. Nhưng hàm ctime có nhiều ưu điểm hơn hàm `asctime`. Chúng ta không cần phải chuyển thông tin thời gian dạng time_t thàng dạng `struct tm` rồi thực hiện chuyển đổi sang thông tin thời gian dạng chuỗi. Hàm `ctime` sẽ chuyển trực tiếp `time_t` thành chuỗi thời gian.

5.4.3. gmtime:

a. Cú pháp:

```
#include <time.h>
struct tm * gmtime ( const time_t * timer);
```

b. Giải thích:

Hàm gmtime làm nhiệm vụ chuyển đổi kiểu thời gian dạng time_t thành cấu trúc thời gian `struct tm` phục vụ cho các hàm thao tác với thời gian thực khác.

Hàm có tham số là con trỏ thời gian kiểu time_t. Dữ liệu trả về là con trỏ cấu trúc thời gian dạng `struct tm`.

c. Chương trình ví dụ:

Nhiệm vụ của chương trình ví dụ này là in ra giá trị thời gian giữa các muối giờ khác nhau, sử dụng hàm gmtime để trả về cấu trúc thời gian struct tm.

```
/*Chương trình ví dụ sủ dụng hàm gmtime*/
/*Khai báo các thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <sdtio.h>
#include <time.h>
/*Khai báo các hằng số dùng trong quá trình lập trình*/
#define MST (-7)
#define UTC (0)
#define CCT (+8)
/*Chương trình chính*/
int main () {
      /*Khai báo biến time_t để lưu thời gian hiện tại của hệ thống*/
time_t rawtime;
/*Khai báo biến con trỏ struct tm chứa dữ liệu trả về sau khi  chuyển đổi */
```

```
struct tm * ptm;
/*Cập nhật giá trị thời gian hiện tại*/
time (&rawtime);
/*Chuyển sang cấu trúc thời gian tm*/
ptm = gmtime ( &rawtime );
/*In các giá trị thời gian ra màn hình*/
puts ("Current time arround the world");
printf ("Phoenix, AZ (U.S.): %2d:%2d\n", (ptm->tm_hour+MST)%24,
ptm->tm_min);
printf ("Reykjavik (Iceland): %2d:%2d\n", (ptm->tm_hour+UTC)%24,
ptm->tm_min);
printf ("Beijing (China), AZ (U.S.): %2d:%2d\n", (ptm-
>tm_hour+CCT)%24, ptm->tm_min);
/*Trả về giá trị 0 cho hàm main*/
return 0;
}
```

Biên dịch và chạy chương trình trên kit, ta có kết quả sau:

```
./time_gmtime_linux
Current time around the World:
Phoenix, AZ (U.S.) :  -5:20
Reykjavik (Iceland) :  2:20
Beijing (China) :     10:20
```

5.4.4. mktime:

a. Cú pháp:

```
#include <time.h>
time_t mktime ( struct tm * timeptr );
```

b. Giải thích:

Nhiệm vụ của hàm mktime là chuyển thông tin thời gian dạng cấu trúc `struct tm` thành thông tin thời gian dạng `time_t`; Bên cạnh đó những thông tin có liên quan trong cấu trúc struct tm chưa được cập nhật cũng sẽ được thay đổi phù hợp với ngày tháng năm đã thay đổi. Cụ thể như thế nào sẽ được trình bày trong ví dụ sử dụng hàm `mktime`.

Hàm `mktime` có tham số là con trỏ cấu trúc `struct tm *` là thời gian muốn chuyển đổi. Giá trị trả về là thời gian dạng `time_t`.

Sau đây là ví dụ về cách sử dụng hàm `mktime`.

c. Chương trình ví dụ:

Nhiệm vụ của chương trình ví dụ này, người sử dụng sẽ nhập thông tin về ngày tháng năm, chương trình sẽ cho biết ngày đó là thứ mấy.

```
/*Chương trình ví dụ hàm mktime*/
/*Khai báo các thư viện cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <time.h>
/*Chương trình chính*/
int main () {
/*Khai báo biến kiểu time_t lưu thông tin thời gian*/
time_t rawtime;
/*Khai báo biến cấu trúc lưu cấu trúc thời gian cần chuyển đổi*/
struct tm * timeinfo;
/*Khai báo biến lưu các ngày trong tuần*/
char * weekday[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday");
/*Khai báo các số nguyên lưu thông tin nhập từ người dùng*/
int year, month, day;
/*Yêu cầu nhập thông tin từ người dùng*/
printf ("Enter year: "); scanf ("%d",&year);
printf ("Enter month: "); scanf ("%d",&month);
printf ("Enter day: "); scanf ("%d",&day);
/*Cập nhật thời gian hiện tại*/
time (&rawtime );
/*Chuyển thành cấu trúc thời gian*/
timeinfo = localtime ( &rawtime );
/*Cập nhật thời gian do người dùng định nghĩa*/
timeinfo->tm_year = year - 1900;
timeinfo->tm_mon = month - 1;
timeinfo->tm_mday = day;
/*Khi gọi hàm mktime thì những thông tin còn lại như tm_wday sẽ được hệ thống
tự động cập nhật*/
mktime ( timeinfo );
printf ( "That day is a %s. \n", weekday[timeinfo->wday]);
```

```
/*Trả về giá trị 0 cho hàm main, thông báo không có lỗi xảy ra  trong quá trình thực thi lệnh*/
return 0;
}
```

Khi biên dịch và chạy chương trình, chúng ta sẽ có kết quả như sau:

```
./time_mktime
Enter year: 2011
Enter month: 11
Enter day: 17
That day is a Thursday.
```

5.4.5. localtime:

a. Cú pháp:

```
#include <time.h>
struct tm * localtime ( const time_t * timer );
```

b. Giải thích:

Hàm localtime dùng chuyển đổi thông tin thời gian dạng `time_t` thành thông tin thời gian dạng `struct tm`.

Hàm localtime có tham số là con trỏ đến biến thời gian `const time_t timer` cần chuyển đổi.

Giá trị trả về của hàm là con trỏ cấu trúc `struct tm *`.

c. Chương trình ví dụ:

Sau đây là chương trình ví dụ sử dụng hàm `localtime`. Chương trình này sẽ cập nhật thời gian hiện tại dạng `time_t`, chuyển sang cấu trúc `struct tm`, cuối cùng in ra màn hình dưới dạng chuỗi thời gian bằng hàm `asctime`.

```
/*Chương trình ví dụ hàm localtime*/
/*Khai báo các thư viện cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <time.h>
/*chương trình chính*/
int main () {
/*Khai báo biến thời gian dạng time_t để lưu thời gian hiện tại*/
time_t rawtime;
```

```
/*Khai báo con trỏ cấu trúc thời gian struct tm lưu thông tin thời gian sau khi chuyển đổi*/
struct tm * timeinfo;
/*Cập nhật thông tin thời gian hiện tại*/
time ( &rawtime);
timeinfo = localtime ( &rawtime );
/*Chuyển đổi cấu trúc thời gian thành chuỗi sau đó in ra màn hình*/
printf ("Current local time and date: %s", asctime ( timeinfo ));
/*Trả về giá trị 0, thông báo quá trình thực thi lệnh không bị lỗi*/
return 0;
}
```

Sau khi biên dịch và thực thi chương trình, chúng ta có kết quả sau:

```
./time_localtime_linux
Current local time and date: Thu Nov 17 11:54:42 2011
```

5.4.6. strftime:

a. Cú pháp:

```
#include <time.h>
   size_t strftime ( char * ptr, size_t maxsize, const char * format, const struct tm * timeptr );
```

b. Giải thích:

Hàm `strftime` làm nhiệm vụ chuyển thông tin thời gian dạng cấu trúc struct tm thành thông tin thời gian dạng chuỗi có định dạng do người lập trình định nghĩa.

Hàm `strftime` có 4 tham số: `char * ptr`, là con trỏ ký tự dùng để lưu chuỗi thông tin thời gian trả về của hàm sau khi đã định dạng; `size_t maxsize`, là kích thước tối đa của chuỗi thông tin thời gian trả về; `const char * format`, là định dạng chuỗi thông tin thời gian mong muốn (tương tự như phần định dạng thông tin xuất ra trong hàm printf); cuối cùng, `const struct tm * timeptr`, là con trỏ cấu trúc nguồn cần chuyển đổi.

Các định dạng được quy định trong bảng sau:

Ký hiệu	*Được thay thế bằng*	*Ví dụ*
%a	Tên viết tắt của ngày trong tuần;	Mon
%A	Tên đầy đủ của ngày trong tuần;	Monday
%b	Tên viết tắt của tháng;	Aug
%B	Tên đầy đủ của tháng;	August
%c	Chuỗi ngày giờ chuẩn	Thu Aug 07:55:02
%d	Ngày của tháng (01-31)	17
%H	Giờ trong ngày (00-23)	23
%I	Giờ trong ngày (01-12)	12
%j	Ngày trong năm (001-366)	245
%m	Tháng trong năm (01-12)	08
%M	Phút trong giờ (00-59)	45
%p	AM hoặc PM	AM
%S	Giây trong phút (00-61)	02
%U	Số tuần trong năm (00-53) với Chúa nhật là ngày đầu tiên trong tuần;	34
%w	Số ngày trong tuần (0-6), Chúa nhật: 0	4
%W	Số tuần trong năm (00-53) với thứ Hai là ngày đầu tiên trong tuần;	34
%x	Chuỗi ngày tháng năm (dd/mm/yy)	18/11/11
%X	Chuỗi giờ phút giây (HH:MM:SS)	07:32:34
%y	Hai số cuối của năm	11
%Y	Số Năm hoàn chỉnh	2011
%Z	Tên múi giờ	CDT
%%	Ký tự đặc biệt	%

Bảng 3-1- Các ký hiệu định dạng chuỗi thời gian trong hàm strftime()

Hàm strftime có giá trị trả về là một số kiểu size_t. Bằng 0 nếu quá trình chuyển đổi xảy ra lỗi (do kích thước quy định không đủ,…). Bằng một số dương kích thước chuỗi ký tự đã chép vào con trỏ char * ptr nếu quá trình chuyển đổi thành công;

Chương trình ví dụ sau sẽ giúp cho chúng ta hiểu rõ cách sử dụng hàm strftime.

c. Chương trình ví dụ:

Chương trình ví dụ này có nhiệm vụ in ra chuỗi thông tin về giờ phút giây hiện tại theo dạng 12H, có thêm AM và PM phía sau.

```
/*Chương trình ví dụ hàm strftime*/
/*Khai báo các thư viện cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <time.h>
/*Chương trình chính*/
int main () {
/*Khai báo biến kiểu time_t để lưu thời gian hiện tại của hệ thống*/
time_t rawtime;
/*Khai báo con trỏ cấu trúc thời gian làm tham số cho hàm strftime*/
struct tm * timeinfo;
/*Khai báo vùng nhớ có kích thước 80 ký tự lưu chuỗi thông tin thời gian*/
char buffer [80];
/*Đọc thời gian hiện tại của hệ thống*/
time (&rawtime);
/*Chuyển thông tin thời gian dạng time_t thành thông tin thời gian dạng cấu trúc
struct tm, theo thời gian cục bộ*/
timeinfo = localtime ( &rawtime );
/*Thực hiện chuyển đổi cấu trúc thởi gian struct tm thành chuỗi thời gian theo
định dạng người dùng*/
strftime (buffer, 80, "Now it's %I:%M%p.", timeinfo);
/*Trả về giá trị 0 thông báo quá trình thực thi lệnh không xảy ra lỗi*/
return 0;
}
```

Biên dịch và chạy chương trình chúng ta có kết quả sau:

```
./time_strftime
Now it's 08:02AM.
```

Như vậy đến đây chúng ta có thể tạo ra một chuỗi thông tin về thời gian với định dạng linh động, phù hợp với mục đích người dùng, thay vì sử dụng dịnh dạng có sẵn của những hàm chuyền đổi thời gian khác.

III. Tổng kết:

Trì hoãn thời gian là một trong những vấn đề quan trọng trong lập trình ứng dụng điều khiển các thiết bị ngoại vi. Trong chương này chúng ta đã tìm hiểu những lệnh căn bản nất về trì hoãn thời gian. Tùy theo từng ứng dụng cụ thể, người lập trình sẽ chọn cho mình cách trì hoãn thời gian thích hợp sao cho không làm ảnh hưởng nhiều đến tính thời gian thực của chương trình. Chúng ta cũng đã tìm hiểu lệnh `alarm` tạo lập định thời cho tín hiệu SIGALRM, những lệnh thao tác với thời gian thực trong hệ thống linux được định nghĩa trong thư viện time.h.

Trong bài sau, chúng ta sẽ bước vào những vấn đề lý thú của hệ điều hành Linux nói riêng và các hệ điều hành thời gian thực khác nói chung, đó là các hàm thao tác với tiến trình và tuyến phục vụ cho việc thiết kế các hệ thống thời gian thực một cách dễ dàng hơn.

BÀI 3

LẬP TRÌNH ĐA TIẾN TRÌNH TRONG USER APPLICATION

I. Kiến thức ban đầu:

1. Định nghĩa tiến trình trong Linux:

Trong hầu hết các hệ điều hành đa nhiệm tựa UNIX như Linux, thì khả năng xử lý đồng thời nhiều tiến trình là một trong những đặc điểm quan trọng. Như vậy tiến trình được định nghĩa như thế nào trong hệ thống Linux.

Giả sử khi chúng ta thực hiện câu lệnh `ls -l` thì Linux sẽ tiến hành liệt kê tất cả những thông tin về tập tin và thư mục có trong thư mục hiện hành. Như vậy hệ điều hành Linux đã thực hiện một tiến trình. Cùng một lúc hai người sử dụng lệnh `ls -l` (chẳng hạn một người đăng nhập trực tiếp và một người đăng nhập qua mạng) hệ điều hành cũng sẽ đáp tất cả những yêu cần trên. Như vậy ta nói rằng hệ điều hành đã thực hiện song song hai tiến trình. Thậm chí ngay cả một chương trình, cũng có thể có nhiều tiến trình cùng một lúc xảy ra. Ví dụ như chương trình ghi dữ liệu vào đĩa CD, có rất nhiều tiến trình xảy ra cùng một lúc, một tiến trình làm nhiệm vụ chép nội dung dữ liệu từ đĩa cứng sang đĩa CD và một tiến trình tính toán thời gian còn lại để hoàn thành công việc hiển thị trên thanh trạng thái … Tiến trình và chương trình là hai định nghĩa hoàn toàn khác nhau, nhiều khi cũng có thể coi là một. Khi chương trình có duy nhất một tiến trình xử lý thì lúc đó tiến trình và chương trình là một. Nhưng khi có nhiều tiến trình cùng thực hiện để hoàn thành một chương trình thì lúc đó tiến trình chỉ là một phần của chương trình.

Mỗi một tiến trình trong Linux điều được gán cho một con số đặc trưng duy nhất. Số này gọi là định danh của tiến trình, PID (Process Identification Number). PID dùng để phân biệt giữa các tiến trình cùng tồn tại song song trong hệ thống. Mỗi một tiến trình được cấp phát một vùng nhớ lưu trữ những thông tin cần thiết có liên quan đến tiến trình. Các tiến trình được điều phối xoay vòng thực hiện bởi hệ thống phân phối thời gian của hệ điều hành. Các tiến trình được trao đổi thông tin với nhau thông qua cơ chế giao tiếp giữa các tiến trình (IPC-Inter Process Communication). Các cơ chế giao tiếp giữa các tiến trình bao gồm những kỹ thuật như đường ống, socket, message, chuyển hướng xuất nhập, hay phát sinh tín hiệu …

Như vậy, tiến trình được hiểu là một thực thể điều khiển đoạn mã lệnh có riêng một không gian địa chỉ vùng nhớ, có ngăn xếp riêng lẽ, có bảng chứ các số mô tả tập tin được mở cùng tiến trình và đặc biệt là có một định danh PID (Process Identify) duy nhất trong toàn bộ hệ thống vào thời điểm tiến trình đang chạy. (Theo định nghĩa trong sách lập trình Linux của nhà xuất bản Minh Khai)

2. Cấu trúc tiến trình:

Phần trên chúng ta đã khái quát được những kiến thức căn bản nhất trong một tiến trình. Trong phần này chúng ta sẽ đi sâu vào tìm hiển cấu trúc của tiến trình, cách xem thông tin của các tiến trình đang chạy trong hệ thống Linux và ý nghĩa của những thông tin đó.

Như đinh nghĩa trong phần trên mỗi tiến trình được cấp phát một vùng nhớ riêng để lưu những thông số cần thiết. Những thông tin đó là gì?

PID
Code
Data
Library
FileDes

Thông tin đầu tiên là PID (Process Identify). Thông tin này là số định danh cho mỗi tiến trình. Mỗi tiến trình dang chạy trong hệ thống điều có một định danh riêng biệt không trùng lắp với bất kỳ tiến trình nào. Số PID được hệ điều hành cung cấp một cách tự động. PID là một số nguyên dương từ 2 đến 32768. Riêng tiến trình `init` là tiến trình quản lý và tạo ra mọi tiến trình con khác được hệ điều hành gán cho PID là 1. Đó là nguyên nhân tại sao PID của các tiến trình là một số nguyên dương bắt đầu từ 2 đến 32768.

Thông tin thứ hai là Code. Code là vùng nhớ chứa tập tin chương trình thực thi chứa trong đĩa cứng được hệ điều hành nạp vào vùng nhớ (cụ thể ở đây là RAM). Ví dụ khi chúng ta đánh lệnh ls –l thì hệ điều hành sẽ chép mã lệnh thực thi của lệnh `ls` trong thư muc bin của `Root File System` vào RAM nội nói đúng hơn là chép vào vùng nhớ Code của tiến trình. Từ đó hệ điều hành sẽ chia khe thời gian thực thi đoạn chương trình trình này.

Thông tin thứ ba là Library. Library là thư viện chứa những đoạn mã dùng chung cho các tiến trình hay chỉ dùng riêng cho một tiến trình nào đó. Các thư viện dùng chung gọi là các thư viện chuẩn tương tự như thư viện DLLs trong hệ điều hành Windows. Những thư viện dùng riêng có thể là những thư viện do người lập trình định nghĩa trong quá trình xây dựng chương trình trình (bao gồm thư viện liên kết động và thư viện liên kết chuẩn).

Thông tin thứ tư là FileDes. FileDes là viết tắt của từ tiếng Anh File Description tạm dịch là bảng mô tả tập tin. Như tên gọi của nó, bảng thông tin mô tả tập tin chứa các số mô tả tập tin được mở trong quá trình thực thi chương trình. Mỗi một tập tin trong hệ thống Linux khi được mở điều chứa một số mô tả tập tin duy nhất, bao gồm những thông tin như dường dẫn, cách thức truy cập tập tin (chỉ đọc, chỉ ghi, hay cả hai thao tác đọc và ghi, …).

Để xem những thông tin trên, chúng ta thực hiên lệnh shell sau:

```
ps –af
```

Sau khi thực hiện xong chúng ta sẽ có kết quả như sau:

```
root@:/home/arm/project/application# ps -af
UID         PID  PPID  C STIME TTY          TIME CMD
root       2731  2600  0 21:55 pts/2    00:00:00 ps –af
```

Cột thứ nhất UID là tên của người dùng đã gọi tiến trình. PID là số định danh mà hệ thống cung cấp cho tiến trình. PPID là số dịnh danh của tiến trình cha đã gọi tiến trình PID. C là CPU thực hiện tiến trình. STIME là thời điểm tiến trình được đưa vào sử dụng. TIME là thời gian chiếm dụng CPU của tiến trình. TTY là terminal ảo nơi gọi thực thi tiến trình. CMD là chi tiết dòng lệnh được triệu gọi thực thi.

Ngoài ra chúng ta cũng có thể dùng lệnh `ps –ax` để xem các tiến trình đang thực thi của toàn bộ hệ thống.

II. Nội dung:

Phần này chúng ta sẽ tìm hiểu những hàm thao tác với tiến trình. Những hàm này sẽ giúp rất nhiều trong việc thiết kế một hệ thống hoạt động đa tiến trình sao cho hiệu quả nhất. Mỗi hàm sẽ bao gồm 3 phần thông tin. Thứ nhất là cú pháp sử dụng hàm, thứ hai là ý nghĩa các tham số trong hàm, thứ ba là chương trình ví dụ cách sử dụng hàm. Để có thể ứng dụng ngay những kiến thức đã học vào thực tế, chúng ta phải tiến hành đọc hiểu mã lệnh, sau đó lập trình trên máy tính, biên dịch và chạy chương trình, quan sát so sánh với dự đoán ban đầu. Những hàm liệt kê sẽ theo thứ tự từ dễ đến khó, mang tính chất kế thừa

thông tin của nhau. Vì thế nên chúng ta phải nghiên cứu theo trình tự của bài học nếu các bạn là người mới bắt đầu.

1. Hàm system():

a. Cú pháp:

```
#include <stdlib.h>
int system ( const char *cmdstr);
```

b. Giải thích:

Hàm system sẽ thực thi chuỗi lệnh cmdstr do người lập trình nhập vào. Chuỗi lệnh nhập vào `cmdstr` là lệnh trong môi trường shell, chẳng hạn ls –l hay `cd <directory>` … Hàm `system` khi được gọi trong một tiến trình, nó sẽ tiến hành tạo và chạy một tiến trình khác (là tiến trình chứa trong tham số) khi thực hiện xong tiến trình này thì hệ thống mới tiếp phân công thực hiện những lệnh tiếp theo sau hàm `system`. Nếu tiến trình trong hàm system được gán cho hoạt động hậu tiến trình thì những lệnh theo sau hàm `system` vẫn thực hiện song song với tiến trình đó. Chúng ta sẽ làm rõ hơn trong phần chương trình ví dụ.

Hàm `system` có tham số `const char *cmdstr` là chuỗi lệnh người lập trình muốn thực thi.

Hàm `system` có giá trị trả về kiểu `int`. Trả về giá trị lỗi 127 nếu như không khởi động được shell, mã lỗi `-1` nếu như gặp các lỗi khác. Những trường hợp còn lại là mã lỗi do lệnh trong tham số `cmdstr` trả về.

c. Chương trình ví dụ:

Viết chương trình in ra 10 lần chuỗi helloworld như sau:

*/*chương trình helloworld system_helloworld.c*/*

```
#include <stdio.h>
int main (void) {
   int i;
   for (i = 0; i < 10; i++) {
         printf("Hello world %d!\n", i);
         sleep(1);
   }
}
```

Biên dịch và lưu tập tin kết xuất với tên system_helloworld;

```
arm-none-linux-gnueabi-gcc system_helloworld.c –o system_helloworld
```

Viết chương trình khai báo sử dụng hàm system gọi thực thi tiến trình system_helloworld đã biên dịch:

```
/*Chương trình ứng dụng hàm system trường hợp 1  system_main_1.c*/
#include <stdio.h>
/*Khai báo thư viện sử dụng hàm system*/
#include <stdlib.h>
int main (void) {
   /*In thông báo hàm system bắt đầu thực thi*/
   printf ( "System has called a process... \n");
   /*Thực thi hàm system gọi tiến trình system_helloworld đã  được biên dịch trước đó*/
system ( "./system_helloworld" );
/*Thông báo tiến trình trong hàm system thực thi xong*/
   printf ( "Done.\n");
   return 0;
}
```

Biên dịch và lưu tập tin kết xuất với tên system_main;

```
arm-none-linux-gnueabi-gcc system_main_1.c –o system_main_1
```

Chạy chương trình system_main_1 vừa biên dịch trên kit, ta có kết quả sau:

```
./system_main_1
System has called a process...
Hello world 0!
Hello world 1!
Hello world 2!
Hello world 3!
Hello world 4!
Hello world 5!
Hello world 6!
Hello world 7!
Hello world 8!
Hello world 9!
Done.
```

Giải thích chương trình:

Ở chương trình `system_main_1.c`, hàm `system()` gọi tiến trình "./system_helloworld" chương trình này có nhiệm vụ in ra 10 lần chuỗi `"Hello world"`. Chương trình chứa hàm `system` đợi cho đến khi thực hiện xong việc in 10 lần chuỗi `"Hello world"` sau đó mới thực hiện tiếp lệnh phía sau hàm `system`. Như chúng ta thấy, chuỗi `"Done"` được in phía sau cùng tất cả các dòng thông tin.

Trong trường hợp thứ hai, chúng ta sử dụng hàm `system` để gọi một tiến trình chạy hậu cảnh. Nghĩa là lúc này, những câu lệnh phía sau hàm system sẽ thực thi song song với tiến trình được gọi bởi hàm system.

Viết chương trình ví dụ sau:

```
/*Chương trình ứng dụng hàm system trường hợp 2  system_main_2.c*/
#include <stdio.h>
/*Khai báo thư viện sử dụng hàm system*/
#include <stdlib.h>
int main (void) {
   /*In thông báo hàm system bắt đầu thực thi*/
   printf ( "System has called a process... \n");
   /*Thực thi hàm system gọi tiến trình system_helloworld đã  được biên dịch trước đó*/
system ( "./system_helloworld &" );
/*Thông báo tiến trình trong hàm system thực thi xong*/
   printf ( "Done.\n");
   return 0;
}
```

Tiến hành biên dịch chương trình:

```
arm-none-linux-gnueabi-gcc system_main_2.c –o system_main_2
```

Chạy chương trình system_main_2 vừa biên dịch trên kit, ta có kết quả sau:

```
./system_main_2
System has called a process...
Done.
# Hello world 0!
Hello world 1!
Hello world 2!
Hello world 3!
```

```
Hello world 4!
Hello world 5!
Hello world 6!
Hello world 7!
Hello world 8!
Hello world 9!
```

Cũng tương tự trong chương trình `system_main_1.c` nhưng trong hàm system ta gọi tiến trình chạy hậu cảnh system (`"./system_helloworld &"`); kết quả là chuỗi thông tin "Done" được in ra ngay phía sau dòng thông báo "`System has called a process...` " mà không phải nằm tại vị trí cuối như trường hợp 1. Hơn nữa cũng trong trường hợp này, chúng ta thấy lệnh trong hàm `system` thực thi chậm hơn lệnh phía sau hàm `system`. Đây cũng là một yếu điểm lớn nhất khi triệu gọi tiến trình bằng hàm system đó là thời gian thực thi chương trình chậm vì system gọi lại shell của hệ thống và chuyển giao trách nhiệm thực thi chương trình cho lệnh shell.

Trong các phần sau, chúng ta sẽ học cách triệu gọi một tiến trình thực thi sao cho ít tốn thời gian hơn.

2. Các dạng hàm exec:

Các hàm tạo tiến trình dạng như `system()` thông thương tiêu tốn thời gian hoạt động của hệ thống nhất. Mỗi một tiến trình khi khởi tạo và thực thi, hệ điều hành sẽ cấp phát riêng một không gian vùng nhớ cho tiến trình đó sử dụng. Khi tiến trình A đang hoạt động, gọi hàm `system()` để tạo một tiến trình B. Thì hệ diều hành sẽ cấp phát một vùng nhớ cho tiến trình B hoạt động, điều này làm thời gian thực hiện của tiến trình B trở nên chậm hơn. Để giải quyết trường hợp này, hệ diều hành Linux cung cấp cho chúng ta một số hàm dạng `exec` có thể thực hiện tạo lập một tiến trình mới nhanh chóng bằng cách tạo một không gian tiến trình mới từ không gian của tiến trình hiện có. Nghĩa là tiến trình hiện có sẽ nhường không gian của mình cho tiến trình mới tạo ra, tiến trình cũ sẽ không hoạt động nữa. Sau đây chúng ta sẽ nghiên cứu hàm execlp() phục vụ cho việc thay thế tiến trình.

a. Cú pháp:

```
#include <stdio.h>
int execlp ( const char *file, const char *arg, …, NULL);
```

b. Giải thích:

Hàm `execlp ()` có nhiệm vụ thay thế không gian của tiến trình cũ (tiến trình triệu gọi hàm) bằng một tiến trình mới. Tiến trình mới tạo ra có ID là ID của tiến trình cũ nhưng mã lệnh thực thi, cấu trúc tập tin, thư viện là của tiến trình mới. Tiến trình cũ sẽ bị mất không gian thực thi, tất nhiên sẽ không còn thực thi trong hệ thống nữa.

Hàm `execlp` có các tham số: `const char *file` là đường dẫn đến tên tập tin chương trình thực thi trên đĩa. Nếu tham số này để trống, hệ điều hành sẽ tìm kiếm trong những đường dẫn khác chứa trong biến môi trường `PATH`; các tham số khác dạng `const char *arg` là những tên lệnh, tham số lệnh thực thi trong môi trường `shell`. Chẳng hạn, nếu chúng ta muốn thực thi lệnh `ps -ax` trong môi trường shell thì tham số thứ hai của lệnh là "`ps`", tham số thứ ba là "`-ax`". Tham số kết thúc là `NULL`.

Hàm này có giá trị trả về là một số `int`. Hàm trả về mã lỗi `127` khi không triệu gọi được shell thực thi, `-1` khi không thực thi được chương trình, các mã lỗi còn lại do chương trình được gọi thực thi trả về.

c. Chương trình ví dụ:

Thiết kế chương trình thực hiện công việc sau: in ra chuỗi "`Hello world`" 10 lần. Mỗi lần in ra cách nhau 1 giây.

Chương trình mẫu mang tên là: `execlp_helloworld.c`

```
/*Chương trình execlp_helloworld.c */
#include <stdio.h>
int main (void) {
   int i;
   for (i = 0; i < 10; i++) {
        printf("Hello world %d!\n", i);
        sleep(1);
   }
   return 0;
}
```

Thực hiện biên dịch chương trình với tập tin ngõ ra là execlp_helloworld.

```
arm-none-linux-gnueabi-gcc execlp_helloworld.c –o execlp_helloworld
```

Thiết kế chương trình sử dụng hàm `execlp` để gọi chương trình `execlp_helloworld` vừa biên dịch mang tên `execlp_main.c`.

```
/*Chương trình execlp_main.c*/
/*Khai báo thư viện cho các lệnh sử dụng trong chương trình*/
#include <stdio.h>
#include <stdlib.h>
/*Chương trình chính*/
int main() {
/*Thông báo hàm execlp bắt dầu thực thi*/
   printf ("Execlp has just start ...");
/*Tiến hành gọi chương trình execlp_helloworld từ thư mục hiện  tại*/
   execlp ("execlp_helloworld", "execlp_helloworld", NULL);
/*In ra thông báo kết thúc chương trình execlp_main, chúng ta  sẽ không thấy lệnh
này thực thi*/
   printf ("Done.");
/*Trả về giá trị 0 cho hàm main, thông báo quá trình thực thi  chương trình không có
lỗi*/
   return 0;
}
```

Biên dịch chương trình `execlp_main.c` với tập tin ngõ ra là `execlp_main`, lưu vào cùng thư mục với chương trình `execlp_helloworld`:

```
arm-none-linux-gnueabi-gcc execlp_main.c –o execlp_main
```

Tiến hành chạy chương trình chúng ta có kết quả sau:

```
./execlp_main
Execlp has just start ...
```

**Chúng ta thấy chương trình `execlp_helloworld` không thể thực thi được. Nguyên nhân vì khi gọi chương trình "`execlp_helloworld`" hệ điều hành sẽ tiến hành kiếm trong biến môi trường `PATH` đường dẫn chứa chương trình này. Mà chương trình "`execlp_helloworld`" không chứa trong bất kỳ đường dẫn nào trong biến môi trường mà chỉ chứa trong thư mục hiện tai. Để chương trình gọi được chương trình "`execlp_helloworld`" chúng ta phải thêm dường dẫn thư mục hiện hành vào biến môi trường bằng lệnh sau: `export PATH=$PATH:.` trong shell.

Khi đó tiến hành chạy lại chương trình execlp_main thì chương trình `execlp_helloworld` sẽ được gọi thực thi. Sau đây là kết quả:

```
./execlp_main
```

```
Execlp has just start ...
Hello world 0!
Hello world 1!
Hello world 2!
Hello world 3!
Hello world 4!
Hello world 5!
Hello world 6!
Hello world 7!
Hello world 8!
Hello world 9!
```

Do câu lệnh `execlp ("execlp_helloworld", "execlp_helloworld", NULL);` tiến hành thay thế không gian thực thi của tiến trình hiện tại bằng tiến trình mới `execlp_helloworld`. Vì thế, tất cả những lệnh phía sau hàm `execlp` điều không thực thi. Chúng ta thấy chuỗi `"Done."` không thể nào in ra màn hình được.

Đôi khi thay thế tiến trình đang chạy bằng một tiến trình khác lại không tốt. Giả sử chúng ta muốn quay lại chương trình chính khi chương trình mới tạo thành thực thi xong thì sao. Hệ điều hành Linux cung cấp cho chúng ta những lệnh có khả năng làm được những công việc trên.

3. Hàm fork():

a. Cú pháp:

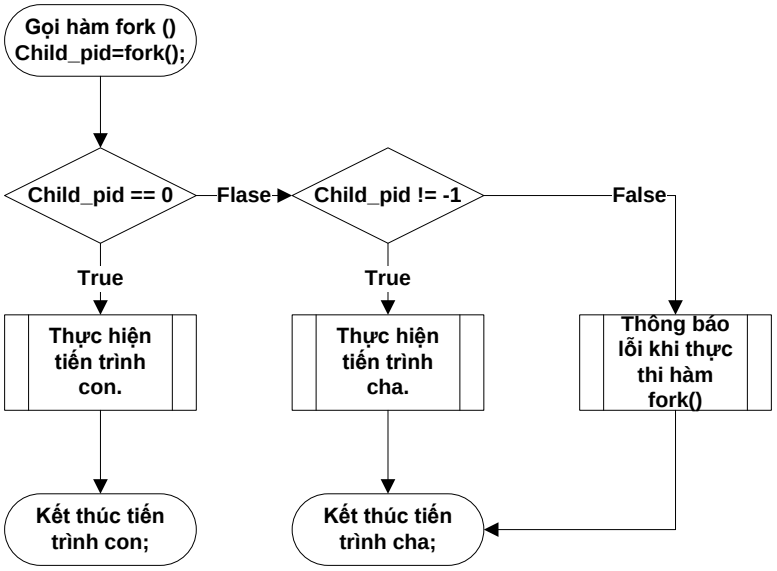
```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

b. Giải thích:

Hàm `fork()` cũng nằm trong nhóm hàm tạo tiến trình mới trong tiến trình đang thực thi. Hàm `fork()` làm nhiệm vụ tạo một tiến trình mới là con của tiến trình đang thực thi. Nghĩa là toàn bộ không gian của tiến trình cha sẽ được sao chép qua không gian của tiến trình con, chỉ khác nhau ở mã lệnh thực thi và các giá trị ID của chúng. Tiến trình con có hai thông số ID, PPID là ID của tiến trình cha, và PID là ID của tiến trình con này.

Hàm `fork()` sẽ quay trở về thời điểm gọi hai lần. Lần thứ nhất, hàm `fork()` trả về giá trị 0 để báo hiệu những lệnh tiếp theo sau là thuộc về tiến trình con. Lần thứ hai, hàm `fork()` trả về một giá trị khác 0. Giá trị này chính là ID của tiến trình con mới vừa tạo

thành. Hàm `fork ()` trả về giá trị -1 khi quá trình tạo lập tiến trình con xảy ra lỗi. Hàm `fork ()` không có tham số. Chúng ta có thể dùng đặc điểm trên để tiến hành tạo lập một tiến trình mới ngay trong một tiến trình hiện tại theo lưu đồ sau:



Hình 3-1- Lưu đố tạo lập tiến trình con trong hàm fork()

Với lưu đồ trên chúng ta có thể thiết kế một doạn chương trình chuẩn thao tác và sử dụng hàm `fork()` như sau:

```
pid_t   child_pid;
child_pid = fork();
switch (child_pid) {
   case -1:   printf ("Cannot create child process.");
              /*Thông báo lỗi cho người dùng khi không tạo lập được tiến trình
              con*/
              break;
   case 0:    printf ("This is the child process.");
              /*Những lệnh thuộc về tiến trình con*/
              break;
   default:   printf ("This is the parent process.");
              /*Những lệnh thuộc về tiến trình cha*/
              break;
}
```

Sau đây là chương trình ví dụ minh họa cách sử dụng hàm fork() để tạo lập tiến trình.

c. Chương trình ví dụ:

Chương trình ví dụ sau đây sẽ làm nhiệm vụ tạo lập một tiến trình con bên trong tiến trình cha đang thực thi. Tiến trình cha sẽ in ra chuỗi "Parent process: Hello" 5 lần. Trong khi đó tiến trình con cũng in ra chuỗi thông tin "Child process: Hello" nhưng với 10 lần. Mỗi lần cách nhau 1 giây.

Thực hiện thiết kế chương trình với những yêu cầu trên:

```
/*Chương trình ví dụ hàm fork() fork_main.c*/
/*Khai báo thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h> //Thư viện sử dụng cho hàm printf();
#include <unistd.h>       //Thư viện sử dụng cho hàm fork();
#include <sys/types.h> //Thư viện sử dụng cho kiểu pid_t;
/*Chương trình chính*/
int main(void) {
   /*Khai báo biến lưu trữ id cho tiến trình con*/
pid_t   child_pid;
/*Khai báo các biến sử dụng đếm số lần in chuỗi ký tự*/
   int n, m;
/*Thực hiện chia tiến trình bằng hàm fork()*/
/*Chương trình sẽ quy lại đây hai lần*/
   child_pid = fork();
   /*Tách tiến trình bằng cách kiểm tra child_pid*/
   switch (child_pid) {
         /*Khi chương trình không thể phân chia tiến trình con*/
         case -1: printf ("Cannot create process.");
                  return -1;
         /*Những lệnh trong trường hợp này thuộc về tiến trình  con*/
         case 0: printf ("This is the child");
                 for (n = 0; n < 10; n++) {
                    printf ("Child process: Hello %d\n", n);
                    sleep (1);
                 }
                 return 0;
         /*Những lệnh trong trường hợp này thuộc về tiến trình  cha*/
```

```
        default:   printf ("This is the parent");
                   for (m=0; m < 5; m++) {
                   printf ("Parent process: Hello %d\n", m);
                   sleep(1);
                   }
                   break;
    }
return 0;
}
```

Thực hiện biên dịch chương trình với tập tin chương trình kết xuất là `fork_main`.

```
arm-none-linux-gnueabi-gcc fork_main.c –o fork_main
```

Chép tập tin chương trình fork_main vào kít và tiến hành chạy chương trình. Chúng ta có kết quả sau:

```
./fork_main
This is the parentParent process: Hello 0
This is the childChild process: Hello 0
Parent process: Hello 1
Child process: Hello 1
Parent process: Hello 2
Child process: Hello 2
Parent process: Hello 3
Child process: Hello 3
Parent process: Hello 4
Child process: Hello 4
# Child process: Hello 5
Child process: Hello 6
Child process: Hello 7
Child process: Hello 8
Child process: Hello 9
```

Với kết quả trên, chúng ta thấy tiến trình cha và tiến trình con chạy đồng thời với nhau và trình tự xuất ra chuỗi thông tin. Hai tiến trình này được hệ điều hành chia thời gian thực hiện đồng thời, độc lập với nhau. Do đó khi tiến trình cha kết thúc, tiến trình con vẫn tiếp tục thực hiện. Chúng ta có thể kiểm tra sự thực thi của hai tiến trình trên bằng lệnh shell sau.

```
./fork_main & ps –af
[1] 3264
```

```
This is the parentParent process: Hello 0
This is the childChild process: Hello 0
UID        PID  PPID  C STIME TTY          TIME CMD
root      3264  3173  0 19:28 pts/1    00:00:00 ./fork_main
root      3265  3173  8 19:28 pts/1    00:00:00 ps -af
root      3266  3264  0 19:28 pts/1    00:00:00 ./fork_main
```

Chúng ta thấy, tiến trình fork_main được gọi hai lần trong hệ thống. Lần thứ nhất là tiến trình cha, số ID là 3264. Lần thứ 2 là tiến trình con với số PPID là 3264 cùng với số PID của tiến trình cha, và số PID là 3266.

Đôi khi chúng ta muốn tiến trình cha trong lúc thực thi khi cần thực hiện một nhiệm vụ cụ thể sẽ gọi một tiến trình con khác thực hiện thay và chờ tiến trình con kết thúc tiến trình cha mới làm việc tiếp. Trong trường hợp trên thì yêu cầu này không thực hiện được. Tiến trình cha thậm chí kết thúc tiến trình con vẫn tiếp tục thực hiện. Hiện tượng này gọi là hiện tượng tiến trình con bị bỏ rơi. Chúng ta không muốn trường hợp này xảy ra. Hệ thống Linux cung cấp cho chúng ta hàm `wait` để thực hiện công việc này.

4. Hàm wait():

a. Cú pháp:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait ( int  &stat_loc );
```

b. Giải thích:

Như trên đã đề cập, tiến trình cha và tiến trình con đòi hỏi phải có sự đồng bộ nhau trong việc xử lý thông tin. Tiến trình cha phải đợi tiến trình con kết thúc mới tiến hành làm những công việc tiếp theo, sử dụng kết quả của tiến trình con. Hàm wait() được sử dụng với mục đích này.

Hàm `wait()` có tham số `int &stat_loc` là con trỏ để lưu trạng thái của tiến trình con khi thực hiện xong. Hàm `wait()` trả về là số ID của tiến trình con hoàn thành.

Chương trình sau sẽ cho chúng ta hiểu rõ cách sử dụng hàm `wait()`.

c. Chương trình ví dụ:

Cũng giống như chương trình ví dụ trước, chương trình này cũng tạo ra hai tiến trình, tiến trình cha và tiến trình con. Nhưng lần này tiến trình cha sẽ đợi tiến trình con kết thúc sau đó mới thực hiện tiếp công việc của mình khi sử dụng hàm `wait()`.

```
/*Chương trình ví dụ hàm wait() wait_main.c*/
/*Khai báo thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h> //Thư viện sử dụng cho hàm printf();
#include <unistd.h>       //Thư viện sử dụng cho hàm fork();
#include <sys/types.h> //Thư viện sử dụng cho kiểu pid_t;
#include <sys/wait.h>     //Thư viện sử dụng cho hàm wait;
/*Chương trình chính*/
int main(void) {
   /*Khai báo biến lưu trữ id cho tiến trình con*/
pid_t   child_pid;
/*Khai báo các biến sử dụng đếm số lần in chuỗi ký tự*/
   int n, m;
   /*Khai báo biến lưu trạng thái tiến trình con khi thực hiện xong*/
   int child_status;
/*Thực hiện chia tiến trình bằng hàm fork()*/
/*Chương trình sẽ quy lại đây hai lần*/
   child_pid = fork();
   /*Tách tiến trình bằng cách kiểm tra child_pid*/
   switch (child_pid) {
         /*Khi chương trình không thể phân chia tiến trình con*/
         case -1: printf ("Cannot create process.");
                  return -1;
         /*Những lệnh trong trường hợp này thuộc về tiến trình con*/
         case 0: printf ("This is the child");
                 for (n = 0; n < 10; n++) {
                    printf ("Child process: Hello %d\n", n);
                    sleep (1);
                 }
                 return 0;
```

```
            /*Những lệnh trong trường hợp này thuộc về tiến trình cha*/
            default:   printf ("This is the parent");
            /*Chờ tiến trình con kết thúc*/
                       wait (&child_status);
                       for (m=0; m < 5; m++) {
                       printf ("Parent process: Hello %d\n", m);
                       sleep(1);
                       }
                       break;
      }
   return 0;
   }
```

Biên dịch chương trình và lưu tên chương trình đã biên dịch thành `wait_main`

```
arm-none-linux-gnueabi-gcc wait_main.c –o wait_main
```

Chạy chương trình chúng ta có kết quả sau:

```
./wait_main
This is the parent
This is the child
Child process: Hello 0
Child process: Hello 1
Child process: Hello 2
Child process: Hello 3
Child process: Hello 4
Child process: Hello 5
Child process: Hello 6
Child process: Hello 7
Child process: Hello 8
Child process: Hello 9
Parent process: Hello 0
Parent process: Hello 1
Parent process: Hello 2
Parent process: Hello 3
Parent process: Hello 4
```

Như vậy chúng ta thấy rõ ràng, tiến trình con thực hiện xong ghi 10 lần chuỗi "`Hello world`" thì tiến trình cha mới thực hiện tiếp công việc của mình ghi 5 lần chuỗi "`Hello world`" tiếp theo.

Trong hàm `wait (&child_status);` có xuất hiện biến con trỏ `&child_status`, biến này kết hợp với một số macro trong thư viện `<sys/wait.h>` sẽ cho chúng ta những thông tin thực hiện tiến trình con. Chi tiết về các macro này, các bạn hãy tham khảo trong thư viện `<sys/wait.h>` trong hệ thống linux.

III. Kết luận:

Trong bài này chúng ta đã học những định nghĩa rất căn bản về tiến trình. Tiến trình là một trong những kỹ thuật lập trình hiệu quả trong lập trình ứng dụng chạy trên hệ điều hành nhúng. Để hiểu và thao tác thành thạo với tiến trình chúng ta cũng đã nghiên cứu những hàm như tạo lập tiến tình, thay thế tiến trình và nhân đôi tiến trình. Bên cạnh đó chúng ta cũng tham khảo cách sử dụng hàm wait trong việc đồng bộ hóa hoạt động của các tiến trình với nhau.

(Thêm giới hạn trong bài này là không trình bày trao đổi thông tin giữa ác tiến trình, tranh chấp dữ liệu thực thi của các tiến trình với nhau).

Tiến trình là một kỹ thuật hiệu quả nhưng chưa phải là hiệu quả nhất về thời gian thực thi. Vì hệ điều hành còn phải tốn thời gian khởi tạo không gian thực thi cho mỗi tiến trình mới tạo ra. Đối với những ứng dụng đòi hỏi thời gian thực thi nhanh thì tiến trình không thể giải quyết được. Linux cung cấp cho chúng ta một kỹ thuật lập trình mới hiệu quả hơn đó là kỹ thuật lập trình đa tuyến mà chúng ta sẽ nghiên cứu ngay trong bài sau đây.

BÀI 4

LẬP TRÌNH ĐA TUYẾN TRONG USER APPLICATION

I. Kiến thức ban đầu về tuyến trong Linux:

Trong bài trước chúng ta đã học những khái niệm cơ bản nhất về tiến trình trong Linux. Trong hệ thống Linux, các tiến trình có một không gian vùng nhớ riêng hoạt động độc lập nhau, xử lý một nhiệm vụ cụ thể trong chương trình. Chúng ta có thể tạo lập một tiến trình mới từ tiến trình đang thực thi bằng các hàm thao tác với tiến trình. Tuy nhiên chi phí tạo lập một tiến trình mới rất lớn đặc biệt là thời gian thực thi. Vì hệ điều hành phải tạo lập một không gian vùng nhớ cho chương trình mới, hơn nữa việc giao tiếp giữa các tiến trình rất phức tạp (chúng ta sẽ nghiên cứu trong các bài sau). Hệ điều hành Linux cung cấp cho chúng ta một kỹ thuật lập trình mới ít tốn thời gian hơn và hiệu quả hơn trong lập trình ứng dụng. Đó là kỹ thuật lập trình đa tuyến.

Tuyến là một thuật ngữ được dịch từ tuyến. Một số sách khác còn gọi là tiểu trình hay luồn. Tuyến được hiểu như là bộ phận cấu thành một tiến trình. Có cùng không gian vùng nhớ với tiến trình, có khả năng sử dụng những biến cục bộ được định nghĩa trong tiến trình. Nếu như trong hệ điều hành có nhiều tiến trình cùng hoạt động song song thì trong một tiến trình cũng có nhiều tuyến hoạt động song song nhau chia sẽ không gian vùng nhớ của tiến trình đó. Tiến trình và tuyến nhiều khi cũng tương tự nhau. Nếu như trong tiến trình có nhiều tuyến cùng hoạt động song song thì tuyến là một bộ phận cấu thành tiến trình. Nếu như trong tiến trình chỉ có một tuyến thì tuyến và tiến trình là một. Như vậy đến đây chúng ta có thể hiểu hệ điều hành Linux vừa làm nhiệm vụ chia khe thời gian thực hiện tiến trình, vừa làm nhiệm vụ chia khe thực hiện tuyến trong tiến trình.

Trong bài này chúng ta sẽ nghiên cứu những hàm sử dụng trong lập trình đa tuyến:

- Các hàm tạo lập và hủy tuyến;
- Các hàm chờ đợi tuyến thực thi nhằm thực hiện đồng bộ trình tự thời gian thực thi lệnh giữa các tuyến;

II. Nội dung:

1. Hàm pthread_create():

a. Cú pháp:

```
#include <pthread.h>

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

b. Giải thích:

Hàm `pthread_create ()` được dùng để tạo một tuyến mới từ tiến trình đang chạy. Tuyến mới được tạo thành có thuộc tính quy định bởi tham số `pthread_attr_t * attr`, nếu tham số này có giá trị `NULL` thì tuyến mới tạo ra sẽ có thuộc tính mặc định (Chúng ta không đi sâu vào các thuộc tính này do đó thông thường nên sử dụng NULL). Khi tuyến được tạo ra thành công, hệ điều hành sẽ tự động cung cấp một ID lưu trong tham số `pthread_t * thread` phục vụ cho mục đích quản lý.

Tuyến mới tạo ra có tập lệnh thực thi chứa trong chương trình con quy định trong tham số `void * (*start_routine) (*void)`. Chương trình con này có tham số chứa trong `void *arg`.

Đề thoát khỏi tuyến đang thực thi một cách đột ngột và trả về mã lỗi, chúng ta dùng hàm `pthread_exit()`. Hàm này tương tự như `exit ()` trong tuyến chính `main()`. Nếu `pthread_exit()` hay `exit ()` đặt cuối tuyến thì nhiệm nhiệm vụ của chúng là trả về giá trị thông báo cho người lập trình biết trạng thái thực thi lệnh của chương trình.

Đoạn mã lệnh sau sẽ cho chúng ta hiểu rõ cách sử dụng hàm `pthread_create ()` và xem cách hàm `pthread_create ()` hoạt động trong thực tế.

c. Chương trình ví dụ:

Trong chương trình ví dụ sau đây, chúng ta tạo ra một chương trình con. Chương trình con này in ra chuỗi thông tin được cung cấp từ tham số của nó. Chương trình con này được cả hai tuyến sử dụng để in ra thông tin của mình.

Chúng ta tiến hành lập trình theo yêu cầu trên. Sau đây là chương trình mẫu:

*/*Chương trình ví dụ hàm pthread_create () pthread_create.c*/*

*/*Khai báo thư viện cần thiết cho các hàm sử dụng trong chương trình */*

```
#include <stdio.h> //Cho hàm printf ();
#include <pthread.h> // Cho hàm pthread_create();
/*Chương trình con in ra thông tin từ tiến trình triệu gọi*/
void * do_thread ( void * data) {
   /*Khai báo bộ đếm in dữ liệu ra màn hình*/
int i;
/*Chép dữ liệu cung cấp bởi chương trình triệu gọi vào vùng nhớ  trong hàm kiểu int*/
   int me = (int*)data;
   /*In chuỗi thông tin ra màn hình 5 lần cho mỗi tiến trình  triệu gọi*/
   for ( i=0; i<5; i++) {
         /*%d đầu tiên là số cho biết tiến trình nào đang triệu  gọi, %d thứ hai là số lần in thông tin ra màn hình*/
         printf (" '%d' -Got '%d' \n", me, i);
         /*Trì hoãn một khoảng thời gian là 1s*/
         sleep (1);
   }
   /*Thoát khỏi tuyến được triệu gọi, không xuất ra dữ liệu  thông báo*/
   pthread_exit(NULL);
}
/*Tuyến chính triệu gọi chương trình con do_thread*/
int main (void) {
/*Khai báo biến lưu mã lỗi trả về bởi hàm pthread_create()*/
   int thr_id;
/*Khai báo biến lưu ID cho tuyến tạo ra*/
   pthread_t p_thread;
/*Biến lưu định danh của tuyến */
   int a = 1;
   int b = 2;
/*  Sử dụng hàm pthread_create() tạo một tuyến mới
Tuyến mới có ID lưu vào p_thread, mã lỗi trả về thr_id, hàm  thực thi là do_thread, tham số là a*/
   thr_id = pthread_create ( &p_thread, NULL, do_thread, (void *)a);
/*Gọi hàm do_thread() trong tuyến chính, tham số là b*/
```

```
    do_thread ((void*)b);
/*Trả về giá trị 0 thông báo quá trình thực thi lệnh không có  lỗi*/
    return 0;
}
```

Tiến hành biên dịch chương trình lưu với tên là pthread_create bằng lệnh sau:

**Chú ý đây là chương trình có sử dụng đa tuyến nên chúng ta phải biên dịch có sự hỗ trợ của thư viện đa tuyến chứ không biên dịch chương trình giống như cách thông thường.

```
arm-none-linux-gnueabi-gcc pthread_create.c -o pthread_create -lpthread
```

Chạy chương trình trên hệ thống chúng ta có kết quả sau:

```
./pthread_create
'2' -Got '0'
'1' -Got '0'
'2' -Got '1'
'1' -Got '1'
'2' -Got '2'
'1' -Got '2'
'2' -Got '3'
'1' -Got '3'
'2' -Got '4'
'1' -Got '4'
```

Hai tuyến, một xuất phát từ tuyến chính `main()`, và một xuất phát từ hàm `pthread_create`, chạy song song nhau, cùng in ra màn hình chuỗi thông tin. Trong mỗi lần in tuyến '2' luôn thực hiện trước tuyến '1', nguyên nhân là vì tuyến '2' được gọi thực thi trước tiên trong hệ thống. Chương trình con `do_thread` mặc dù chỉ có một nhưng có khả năng chạy đồng thời trên nhiều tiến trình. Thực ra mỗi thời điểm CPU chỉ thực thi một tuyến theo sự phân chia thời gian của hệ điều hành do đó không xảy ra sự xung đột trong việc sử dụng hàm.

2. Hàm pthread_join():

Trong thực tế, chúng ta cần thực hiện đồng bộ hóa hoạt động của giữa các tuyến với nhau. Giả sử tuyến chính triệu gọi một tuyến phụ thực hiện một công việc nào đó, tuyến chính chờ tuyến phụ thực hiện song, trả về giá trị nào đó thì tuyến chính mới thực hiện những công việc tiếp theo. Linux cũng cung cấp cho chúng ta hàm `pthread_join()` để thực hiện những công việc trên.

a. Cú pháp:

```
#include <pthread.h>
int pthread_join ( pthread_t th, void *thread_return);
```

b. Giải thích:

Hàm `pthread_join ()` cung cấp cho chung ta một cong cụ đồng bộ hóa trình tự hoạt động của các tuyến.

Hàm `pthread_join ()` có hai tham số. Tham số thứ nhất `pthread_t th` là ID của tuyến muốn đợi. Tham số thứ hai `void *thread_return` là giá trị tuyến trả về từ hàm `pthreate_exit(data)` khi được sử dụng cuối chương trình.

Hàm `pthread_join ()` có giá trị trả về là `int`, trả về 0 nếu như hàm thực thi thành công, trả về giá trị khác 0 là mã lỗi thực thi hàm.

c. Chương trình ví dụ:

Chương trình ví dụ sau cho chúng ta hiểu được cách sử dụng hàm `pthread_join()` trong chương trình. Chương trình `thread_join.c` có hai tuyến tạo thành. Tuyến chính tạo ra tuyến phụ gọi chương trình con in ra chuỗi thông tin cần thiết và chờ cho tuyến phụ kết thúc, tuyến chính tiếp tục thực hiện in thông báo tuyến phụ đã kết thúc và thoát chương trình thực thi.

Mã chương trình mẫu:

```
/*Chương trình ví dụ cách sử dụng hàm pthread_join, chương trình thread_join.c */
/*Khai báo thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
/*Khai  báo biến toàn cục sử dụng làm dữ liệu chung cho cả hai tuyến*/
/*Các tuyến nằm trong cùng một tiến trình có thể sử dụng chung biến này*/
```

```
char message[] = "Hello world!";
/*Chương trình con dùng cho tạo lập tuyến*/
void * do_thread (void *data) {
/*Biến dùng cho việc đếm thông tin xuất ra màn hình*/
   int i;
/*Nạp thông tin trong tham số hàm cho bộ nhớ trong tuyến*/
   int me = (int *)data;
/*In ra thông báo tuyến đang thực thi*/
   printf ("Thread function is executing ...\n");
/*In ra giá trị của biến message trước khi bị thay đổi*/
   printf ("Thread data is %s\n", message);
/*Trì hoãn in các thông tin thí nghiệm cho người dùng quan  sát*/
   for ( i=0; i<5; i++) {
        printf (" '%d' got '%d'\n", me, i);
        sleep (1);
   }
/*Thay đỗi nội dung của biến toàn cục message*/
   strcpy(message,"Goodbye!");
/*Xuất ra thông tin trả về cho tham số của hàm pthread_join */
   pthread_exit("Thanks for your using me!");
}
/*Tuyến chính*/
int main(void) {
/*Khai báo biến lưu ID của tuyến tạo ra*/
   pthread_t a_thread;
/*Biến int lưu mã lỗi của chương trình*/
   int res;
/*Định danh cho tuyến*/
   int a = 1;
/*Khai báo con trỏ trả dữ liệu về khi đợi tuyến hoàn thành xong  nhiệm vụ*/
   void * thread_result;
/*Thực hiện tạo tuyến mới từ tuyến chính với những thuộc tính  mặc định,
Hàm thực thi là do_thread, tham số cho hàm là biến int a*/
   res = pthread_create (&a_thread, NULL, do_thread, (int *)a);
```

```
   /*Thông báo khi có lỗi xuất hiện*/
      if (res != 0) {
            perror ("Thread cresated error.");
            exit (EXIT_FAILURE);
      }
   /*In thông báo nạp một tuyến vào danh sách đợi*/
      printf ("Wait for thread to finish ...\n");
   /*Tiến hành nạp tuyến vào danh sách đợi*/
      res = pthread_join(a_thread, &thread_result);
   /*Thông báo mã lỗi trong qua trình thực thi hàm pthread_join*/
      if (res != 0) {
            perror ("Thread wait error.");
            exit (EXIT_FAILURE);
      }
   /*Thông báo tuyến phụ đã hoàn thành nhiệm vụ, xuất kết quả trả về*/
      printf ("Thread completed, it returned %s\n", (char
*)thread_result);
   /*In ra biến message sau khi đã bị thay đổi bởi tuyến phụ*/
      printf ("Message is now: %s\n", message);
   /*Trả về giá trị 0 cho tuyến main() thông báo quá trình thực thi không có lỗi*/
      return 0;
   }
```

Biên dịch chương trình lưu với tên thread_join bằng dòng lệnh sau:

```
arm-none-linux-gnueabi-gcc thread_join.c –o thread_join –lpthread
```

Tiến hành chạy chương trình chúng ta có kết quả sau:

```
./thread_join
Wait for thread to finish ...
Thread function is executing ...
Thread data is Hello world!
 '1' got '0'
 '1' got '1'
 '1' got '2'
 '1' got '3'
 '1' got '4'
Thread completed, it returned Thanks for your using me!
Message is now: Goodbye!
```

Chúng ta có thể tạo ra nhiều tuyến hoạt động cùng một lúc và tuyến chính cũng có thể chờ đợi nhiều tuyến cùng một lúc. Bằng cách nạp các tuyến cần chờ đợi vào danh sách chờ đợi khi dùng hàm pthread_join(), tuyến nào được nạp vào hàng đợi trước sẽ được hoàn thành trước nhất nếu các tuyến đó có cùng thời gian thực thi.

Các tuyến có thể sử dụng chung các biến toàn cục đã định nghĩa, đây cũng là một ưu điểm của lập trình đa tuyến so với lập trình đa tiến trình. Tuy nhiên lập trình đa tuyến có một yếu điểm là tính không bền vững trong quá trình thực thi lệnh. Trong quá trình thực thi một tiến trình có nhiều tuyến chạy đồng thời, nếu có một tuyến bị lỗi thì toàn bộ tiến trình sẽ bị lỗi. Còn lập trình đa tiến trình thì không như thế, khi một tiến trình bị lỗi, các tiến trình khác điều hoạt động bình thường. Chúng ta phải biết tận dụng cả hai ưu điểm của hai phương pháp lập trình trên. Đồng thời khắc phục nhược điểm của chúng. Bằng việc phân công phù hợp nhiệm vụ của từng tác vụ trong khi lập trình.

III. Kết Luận:

Đến đây chúng ta đã biết cách khởi tạo một tuyến phụ chạy song song với một tuyến chính đang thực thi bằng hàm pthread_create (). Chúng ta cũng đã biết cách đồng bộ hóa hoạt động giữa các tuyến với nhau, tuyến chính có thể thực hiện đợi cho một hay nhiều tuyến khác thực hiện xong việc xử lý thông tin sau đó mới tiếp tục công việc xử lý của mình, bằng cách dùng hàm pthread_join().

Đến đây chúng ta đã kết thúc phần lập trình ứng dụng trong lớp user application của phần mềm hệ thống nhúng. Trong phần sau chúng ta sẽ nghiên cứu cách viết chương trình trong một môi trường khác là kernel driver, đây là một lớp trung gian giao tiếp giữa hệ điều hành và phần cứng hệ thống.

PHẦN B

CĂN BẢN LẬP TRÌNH DRIVER

BÀI 1

DRIVER VÀ APPLICATION TRONG HỆ THỐNG NHÚNG

I. Khái quát về hệ thống nhúng:

Hệ thống nhúng (embedded system) được ứng dụng rất nhiều trong cuộc sống ngày nay. Theo định nghĩa, hệ thống nhúng là một hệ thống xử lý và điều khiển những tác vụ đặc trưng trong một hệ thống lớn với yêu cầu tốc độ xử lý thông tin và độ tin cậy rất cao. Nó bao gồm phần cứng và phần mềm cùng phối hợp hoạt động với nhau, tùy thuộc vào tài nguyên phần cứng mà hệ thống sẽ có phần mềm điều khiển phù hợp. Đôi khi chúng ta thường nhầm lẫn hệ thống nhúng với máy tính cá nhân. Hệ thống nhúng cũng bao gồm phần cứng (CPU, RAM, ROM, USB, ...) và phần mềm (Application, Driver, Operate System, ...). Thế nhưng khác với máy tính cá nhân, các thành phần này đã được rút gọn, thay đổi cho phù hợp với một mục đích nhất định của ứng dụng sao cho tối ưu hóa thời gian thực hiện đáp ứng yêu cầu về thời gian thực (Real-time) theo từng mức độ.

Bài này sẽ đi sâu vào tìm hiểu cấu trúc bên trong phần mềm của hệ thống nhúng nhằm mục đích hiểu được vai trò của driver và application, phân phối nhiệm vụ hoạt động cho hai lớp này sao cho đạt hiệu quả cao nhất về thời gian.

II. Cấu trúc của hệ thống nhúng:

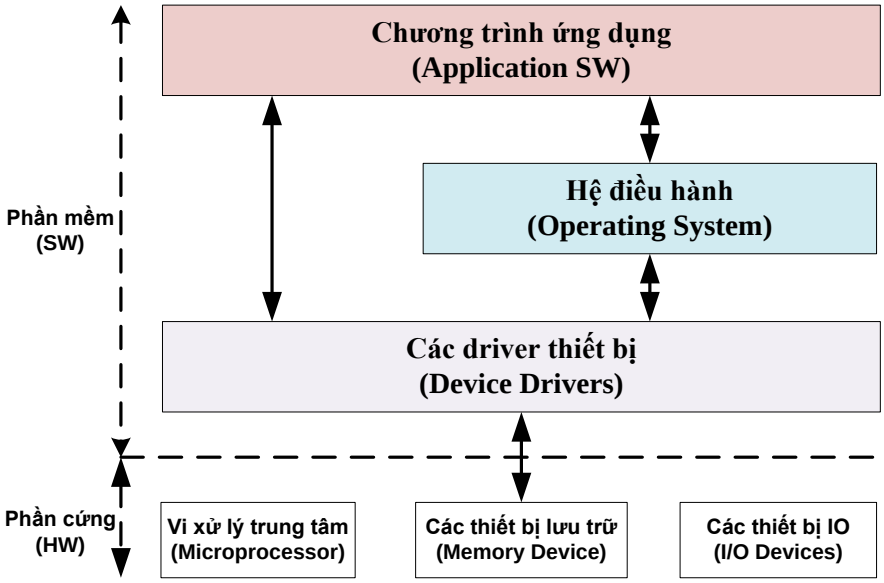
Hệ thống nhúng thông thường bao gồm những thành phần sau: Phần cứng: Bộ vi xử lý trung tâm, bộ nhớ, các thiết bị vào ra; Phần mềm: Các Driver cho thiết bị, Hệ điều hành và các chương trình ứng dụng.

Mối liên hệ giữa các thành phần được minh họa trong sơ đồ hình 3-2.

Thành phần thứ nhất trong hệ thống nhúng là phần cứng. Đây là thành phần quan trong nhất trong hệ thống. Làm nhiệm vụ thực tế hóa những dòng lệnh từ phần mềm yêu cầu. Phần cứng của hệ thống nhúng thường bao gồm những thành phần chính sau:

- Bộ xử lý trung tâm, làm nhiệm vụ tính toán thực thi các mã lệnh được yêu cầu, được xem như bộ não của toàn hệ thống. Các bộ xử lý trong hệ thống nhúng, không giống như hệ thống máy vi tính cá nhân là những con vi xử lý mạnh chuyên về xử lý dữ liệu, là những dòng vi điều khiển mạnh, được tích hợp sẵn các module ngoại vi giúp cho việc thực thi lệnh của hệ thống được thực hiện nhanh chóng hơn. Hơn nữa tập lệnh của vi điều khiển cũng trở nên gọn nhẹ hơn, ít tốn dung lượng vùng nhớ hơn phù hợp với đặc

điểm của hệ thông nhúng. Với những vi điều khiển đã tích hợp sẵn những ngoại vi mạnh, đa dạng thì kích thước mạch phần cứng trong quá trình thi công sẽ giảm rất nhiều. Đây là ưu điểm của hệ thống nhúng so với các hệ thống đa nhiệm khác.



Hình 3-2- Sơ đồ cấu trúc hệ thống nhúng

- Thành phần thứ hai là các thiết bị lưu trữ: Các thiết bị lưu trữ bao gồm có RAM, NAND Flash, NOR Flash, ... mặc dù bên trong vi điều khiển đã tích hợp sẵn ROM và RAM, nhưng những vùng nhớ này chỉ là tạm thời, dung lượng của chúng rất nhỏ, giúp cho việc thực thi những lệnh cũ nhanh hơn. Để lưu trữ những mã lệnh lớn như: Kernel, Rootfs, hay Application thì đòi hỏi phải có những thiết bị lưu trữ lớn hơn. RAM làm nhiệm vụ chứa chương trình thực thi một cách tạm thời. Khi một chương trình được triệu gọi, mã lệnh của chương trình được chép từ các thiết bị lưu trữ khác vào RAM, từ đây từng câu lệnh được biên dịch sẽ lần lượt đi vào vùng nhớ cache bên trong vi xử lý để thực thi. Các loại ROM như NAND Flash, NOR Flash, ... thường có dung lượng lớn nhất trong hệ thống nhúng, dùng để chứa những chương trình lớn (hệ điều hành, rootfs, bootstrapcode, ...) lâu dài để sử dụng trong những mục đích khác nhau khi người dùng (hệ điều hành và user) cần sử dụng đến. Chúng tương tự như ổ đĩa cứng trong máy tính cá nhân.

- Các thiết bị vào ra: Đây là những module được tích hợp sẵn bên trong vi điều khiển. Chúng có thể là ADC module, Ethenet module, USB module, ... các thiết bị này có vai trò giao tiếp giữa hệ thống với môi trường bên ngoài.

Thành phần quan trọng thứ hai trong một hệ thống nhúng là phần mềm. Phần mềm của hệ thống nhúng thay đổi theo cấu trúc phần cứng. Hệ thống chỉ hoạt động hiệu quả khi phần mềm và phần cứng có sự tương thích nhau. Đi từ thấp lên cao thông thường phần mềm hệ thống nhúng bao gồm các lớp sau: Driver thiết bị, hệ điều hành, chương trình ứng dụng.

- *Các driver thiết bị (device driver):* Đây là những phần mềm được viết sẵn để trực tiếp điều khiển phần cứng hệ thống nhúng. Mỗi một hệ thống nhúng được cấu tạo từ những phần cứng khác nhau, những vi điều khiển với những tập lệnh khác nhau, những module khác nhau của các hãng khác nhau có cơ chế giao tiếp khác nhau, device driver làm nhiệm vụ chuẩn hóa thành những thư viện chung (có mã lệnh giống nhau), phục vụ cho hệ điều hành và người viết chương trình lập trình dễ dàng hơn. Chẳng hạn, nhiều hệ thống có giao thức truy xuất dữ liệu khác nhau, nhưng device driver sẽ quy về 2 hàm duy nhất mang tên read và write để đọc và nhập thông tin cho hệ thống xử lý. Để phân biệt giữa các thiết bị với nhau, device driver sẽ cung cấp một ID duy nhất cho thiết bị đó nhằm mục đích thuận tiện cho việc quản lý. **Device driver sẽ được trình bày rất rõ trong những bài khác.

- *Hệ điều hành:* Đây cũng là một phần mềm trong hệ thống nhúng, nhiệm vụ của nó là quản lý tài nguyên hệ thống. Bao gồm quản lý tiến trình, thời gian thực, truy xuất vùng nhớ ảo và vùng nhớ vật lý, các giao thức mạng, ...

- *Chương trình ứng dụng:* Các chương trình ứng dụng là do người dùng lập trình. Thông thường trong hệ thống nhúng, công việc lập trình và biện dịch thông thường không nằm trên chính hệ thống đó. Ngược lại thường được nằm trên một hệ thống đa nhiệm khác, quá trình này gọi là biên dịch chéo (cross-compile). Sau khi biên dịch xong, chương trình đã biên dịch được chép vào bên trong ROM lưu trữ phục vụ cho quá trình sử dụng sau này. Các chương trình sẽ sử dụng những dịch vụ bên trong hệ điều hành (tạo tiến trình, tạo tuyến, trì hoãn thời gian, ...) và những hàm được định nghĩa trong device driver (giao tiếp thiết bị đầu cuối, truy xuất IO, ...) để tác động đến phần cứng của hệ thống.

***Quyển sách này chủ yếu trình bày sâu về phần mềm hệ thống nhúng. Trong phần đầu chúng ta đã nghiên cứu sơ lược về cách lập trình ứng dụng, làm thế nào để trì hoãn*

thời gian, tạo tiến trình, tạo tuyến, ... Phần này sẽ đi sâu vào lớp cuối cùng trong phần mềm là Device driver.

III. Mối quan hệ giữa Device drivers và Applications:

Application (chương trình ứng dụng) và Device drivier (Driver thiết bị) có những điểm giống và khác nhau. Tiêu chí để so sánh dựa vào nguyên lý hoạt động, vị trí, vai trò của từng loại trong hệ thống nhúng.

Application và Device driver khác nhau căn bản ở những điểm sau:

Về cách thức mỗi loại hoạt động, đa số các Application đơn nhiệm vừa và nhỏ hoạt động xuyên suốt từ câu lệnh đầu tiên cho đến câu lệnh kết thúc một cách tuần tự kể từ khi được gọi từ người sử dụng. Trong khi đó, Device driver thì hoàn toàn khác, chúng được lập trình với theo dạng từng module, nhằm mục đích phục vụ cho việc thực hiện một thao tác của Application được gọn nhẹ và dễ dàng hơn. Mỗi module có một hay nhiều chức năng riêng, được lập trình cho một thiết bị đặc trưng và được cài dặt sẵn trên hệ điều hành để sẵn sàng hoạt động khi được gọi. Sau khi được gọi, module sẽ thực thi và kết thúc ngay lập tức. Một cách khái quát, chúng ta có thể xem: Nếu Application là chương trình phục vụ người dùng, thì Device driver là chương trình phục vụ Application. Nghĩa là Application là người dùng của Device driver.

Một điểm khác biệt giữa Application và Device driver là vấn đề an toàn khi thực thi tác vụ. Nếu một Application chỉ đơn giản thực thi và kết thúc, thì công việc của Device driver phức tạp hơn nhiều. Bên cạnh việc thực thi những lệnh được lập trình nó còn phải đảm bảo an toàn cho hệ thống khi không còn hoạt động. Nói cách khác, trước khi kết thúc, Device driver phải khôi phục trạng thái trước đó của hệ thống trả lại tài nguyên cho các Device driver khác sử dụng khi cần, tránh tình trạng xung đột phần cứng.

Một Application có thể thực thi những lệnh mà không cần định nghĩa trước đó, các lệnh này chứa trong thư viện liên kết động của hệ điều hành. Khi viết chương trình cho Application, chúng ta sẽ tiếc kiệm được thời gian, cho ra sản phẩm nhan hơn. Trong khi đó, Device driver muốn sử dụng lệnh nào thì đòi hỏi phải định nghĩa trước đó. Việc định nghĩa này được thực hiện khi chúng ta dùng khai báo #include <linux/library.h>, những thư viện này phải thực sự tồn tại, nghĩa là còn ở dạng mã lệnh C chưa biên dịch. Các thư viện này chứa trong hệ thống mã nguồn của hệ điều hành trước khi được biên dịch.

Một chương trình Application dang thực thi nếu phát sinh một lỗi thì không còn hoạt động được nữa. Trong khi đó, khi một tác vụ trong module bị lỗi, nó chỉ ảnh hưởng đến câu lệnh gọi mà thôi (nghĩa là kết quả truy xuất sẽ không đúng) các lệnh tiếp theo sau vẫn có thể tiếp tục thực thi. Thông thường lúc này chúng ta sẽ thoát khỏi chương trình bằng lệnh exit(n), để đảm bảo dữ liệu xử lý là chính xác.

Chúng ta có hai thuật ngữ mới, user space (không gian người dùng) và kernel space (không gian kernel). Không gian ở đây chúng ta nên hiểu là không gian bộ nhớ ảo, do hệ thống Linux định nghĩa và quản lý. Các chương trình ứng dụng Application được thực thi trong user space, còn những Device driver khi được biên dịch thành tập tin .ko sẽ được lưu trữ trong kernel space. Kernel space và User space liên hệ nhau thông qua hệ điều hành (operating system).

Trong khi hầu hết các lệnh trong từng tiến trình và tuyến được thực hiện tuần tự nhau, kết thúc lệnh này rồi mới thực hiện lệnh tiếp theo, trong user space; Thì các module trong device driver có thể cùng một lúc phục vụ đồng thời nhiều tiến trình, tuyến. Do đó Device driver khi lập trình phải đảm bảo giải quyết được vấn đề này tránh tình trạng xung đột vùng nhớ, phần cứng trong quá trình thực thi.

IV.Kết luận:

Chúng ta đã tìm hiểu sơ lược về cấu trúc tổng quát trong hệ thống nhúng, hiểu được vai trò chức năng của từng thành phần. Bên cạnh đó chúng ta cũng đã phân biệt được những đặc điểm khác nhau giữa chương trình trong user space và Device Driver trong kernel space. Những kiến thức này rất quan trọng khi bước vào lập trình driver cho thiết bị. Chúng ta phải biết phân công nhiệm vụ giữa user application và kernel driver sao cho đạt hiệu quả cao nhất.

Bài tiếp theo chúng ta sẽ đi vào tìm hiểu các loại driver trong hệ thống Linux, cách nhận dạng từng loại, cũng như các thao tác cần thiết khi làm việc với driver.

BÀI 2

PHÂN LOẠI VÀ NHẬN DẠNG DRIVER TRONG LINUX

I. Tổng quan về Device Driver:

Một trong những mục đích quan trọng nhất khi sử dụng hệ điều hành trong hệ thống nhúng là làm sao cho người sử dụng không nhận biết được sự khác nhau giữa các loại phần cứng trong quá trình điều khiển. Nghĩa là hệ điều hành sẽ quy những thao tác điều khiển khác nhau của nhiều loại phần cứng khác nhau thành một thao tác điều khiển chung duy nhất. Ví dụ như, hệ điều hành quy định tất cả những ổ đĩa, thiết bị vào ra, thiết bị mạng điều dưới dạng tập tin và thư mục. Việc khởi động hay tắt thiết bị chỉ đơn giản là đóng hay mở tập tin (thư mục) đó còn sau khi thao tác đóng hay mở hệ điều hành làm gì đó là công việc của device driver.

Trong một hệ thống nhúng, không phải chỉ có CPU mới có thể xử lý thông tin mà tất cả những thiết bị phần cứng điều có một cơ cấu điều khiển được lập trình sẵn, đặc trưng cho từng thiết bị. Mỗi một thẻ nhớ, USB, chuột, USB Camera, … điều là những hệ thống nhúng độc lập, chúng có từng nhiệm vụ riêng, đảm trách một công việc xử lý thu thập thông tin cụ thể. Mỗi bộ điều khiển của các thiết bị đó điều chứa những thanh ghi lệnh và thanh ghi trạng thái. Và để điều khiển được thì chúng ta phải cung cấp những số nhị phân cần thiết vào thanh ghi lệnh, đọc thanh ghi trạng thái cho biết trạng thái thực hiện. Tương tự khi muốn thu thập dữ liệu, chúng ta phải cung cấp những mã cần thiết, theo những bước cần thiết do nhà sản xuất quy định. Thay vì phải làm những công việc nhàm chán đó, chúng ta sẽ giao cho device driver đảm trách. Device driver thực chất là những hàm được lập trình sẵn, nạp vào hệ điều hành. Có ngõ vào là những giao diện chung, ngõ ra là những thao tác riêng biệt điều khiển từng thiết bị của device driver đó.

Linux cung cấp cho chúng ta 3 loại device driver: Character driver, block driver, và network driver. Character driver hoạt động theo nguyên tắc truy xuất dữ liệu không có vùng nhớ đệm, nghĩa là thông tin sẽ di chuyển lập tức từ nơi gửi đến nơi nhận theo từng byte. Block driver thì khác, hoạt động theo cơ chế truy xuất dữ liệu theo vùng nhớ đệm. Có hai vùng nhớ đệm, đệm ngõ vào và đệm ngõ ra. Dữ liệu trước khi di chuyển vào (ra) hệ thống phải chứa trong vùng nhớ đệm, cho đến khi vùng nhớ đệm đầy thì mới được phép xuất (nhập). Nghĩa là dữ liệu di chuyển theo từng khối. Network driver hoạt động

theo một cách riêng dạng socket mạng, chủ yếu dùng trong truyền nhận dữ liệu từ xa giữa các máy với nhau trong mạng cục bộ hay internet bằng các giao thức mạng phổ biến.

**Trong suốt phần này chúng ta chủ yếu nghiên cứu về character driver. Mục tiêu là có thể tự mỉnh thiết kế một character driver đơn giản;

II. Các đặc điểm của device driver trong hệ điều hành Linux:

Chúng ta đã biết như thế nào là device driver, đặc điểm của từng loại device driver. Thế nhưng các loại driver này được hệ điều hành Linux quản lý như thế nào?

Bất kỳ một thiết bị nào trong hệ điều hành Linux cũng được quản lý thông qua tập tin và thư mực hệ thống. Chúng được gọi là các tập tin thiết bị hay là các tập tin hệ thống. Những tập tin này điều chứa trong thư mục /dev. Trong thư mục /dev chúng ta thực hiện lệnh ls –l, hệ thống sẽ cho ra kết quả sau:

```
crw-rw-rw-    1 root     root       1,   3 Apr 11  2002 null
crw-------    1 root     root      10,   1 Apr 11  2002 psaux
crw-------    1 root     root       4,   1 Oct 28 03:04 tty1
crw-rw-rw-    1 root     tty        4,  64 Apr 11  2002 ttys0
crw-rw----    1 root     uucp       4,  65 Apr 11  2002 ttyS1
crw--w----    1 vcsa     tty        7,   1 Apr 11  2002 vcs1
crw--w----    1 vcsa     tty        7, 129 Apr 11  2002 vcsa1
crw-rw-rw-    1 root     root       1,   5 Apr 11  2002 zero
…
```

Cột thứ nhất cho chúng ta thông tin về loại device driver. Theo thông tin trên thì tất cả điều là character driver vì những ký tự đầu tiên điều là “c”, tương tự nếu là block driver thì ký tự đầu là “b”. Chúng ta chú ý đến cột thứ 4 và 5, tại đây có hai thông tin cách nhau bằng dấu “,” hai số này được gọi là Major và Minor. Mỗi thiết bị trong hệ điều hành điều có một số 32 bits riêng biệt để quản lý. Số này được chia thành hai thông tin, thông tin thứ nhất là Major number. Major number là số có 12 bit, dùng để phân biệt từng nhóm thiết bị với nhau, hay nói cách khác những thiết bị cùng loại sẽ có chung một số Major. Các thiết bị cùng loại có cùng số Major được phân biệt nhau thông qua thông tin thứ hai là số Minor. Số Minor là số có chiều dài 20 bit. Với hai số Major và Minor, tổng cộng hệ điều hành có thể quản lý số thiết bị tối đa là $2^{12}*2^{20}$ tương đương với 2^{32}.

Trong lập trình driver, đôi khi chúng ta muốn thao tác với hai thông tin Major và Minor numbers. Kernel cung cấp cho chúng ta những hàm rất hữu ích để thực hiện những công việc này. Sau đây là một số hàm tiêu biểu:

```
#include <linux/types.h>
#include <linux/kdev_t.h>
int MAJOR (dev_t dev);
int MINOR (dev_t dev);
dev_t MKDEV (int major, int minor);
```

Trước khi sử dụng những hàm này, chúng ta phải khai báo thư viện phù hợp cho chúng. thư viện <linux/types.h> chứa định nghĩa kiểu dữ liệu dev_t, biến kiểu này dùng để chứa số định danh cho thiết bị. Thư viện <linux/kdev_t.h> chứa định nghĩa cho những hàm MAJOR(), MINOR(), MKDEV, …

Hàm MAJOR (dev_t dev) dùng để tách số Major của thiết bị dev_t dev và lưu vào một biến kiểu int;

Hàm MINOR (dev_t dev) dùng để tách số Minor của thiết bị dev_t dev và lưu vào một biến kiểu int;

Hàm MKDEV (int major, int minor) dùng để tạo thành một số định danh thiết bị kiểu dev_t từ hai số int major, và int minor;

Đối với kernel 2.6 trở đi thì số device driver dev_t có 32 bit. Nhưng đối với những kernel đời sau đó thì dev_t có 16 bit.

III. Kết luận:

Trong bài này chúng ta đã đi vào tìm hiểu một cách khái quát vai trò ý nghĩa của từng loại device driver trong hệ thống Linux, mỗi device driver điều có những ưu và nhược điểm riêng và đóng góp một phần để làm cho hệ thống chạy ổn định. Chúng ta cũng đã biết cách thức quản lý thông tin thiết bị của Linux thông qua thư mục /dev, mỗi thiết bị trong Linux điều có một số định danh, tùy vào từng hệ thống mà số này có bao nhiêu bit, số định danh có thể được tạo thành từ hai số riêng biệt Major và Minor bằng hàm MKDEV() hoặc có thể tách riêng một số định danh dev_t thành hai số Major và Minor bằng hai hàm MAJOR() và MINOR (). Những hàm này rất quan trong trong lập trình driver.

Trong giới hạn về thời gian, quyển sách này chỉ trình bày cho các bạn cách lập trình một character driver. Trên cơ sở đó các bạn sẽ tự mình tìm hiểu cách lập trình cho các loại driver khác. Bài sau chúng ta sẽ tìm hiểu sâu hơn về character driver, cấu trúc dữ liệu bên trong, các hàm thao tác khởi tạo character device, …

BÀI 3

CHARACTER DEVICE DRIVER

I. Tổng quan character device driver:

Character device driver là một trong 3 loại driver trong hệ thống Linux. Đây là driver dễ và phổ biến nhất trong các ứng dụng giao tiếp vừa và nhỏ đối với lập trình nhúng. Character driver và các loại driver khác điều được hệ điều hành quản lý dưới dạng tập tin và thư mục. Hệ điều hành sử dụng các hàm truy xuất tập tin chuẩn để giao tiếp trao đổi thông tin giữa người lập trình và thiết bị do driver điều khiển. Chẳng hạn những hàm như `read, write, open, release, close, …` được dùng chung cho tất cả các character driver, những hàm này còn được gọi là giao diện điều khiển giữa hệ điều hành (được người lập trình ra lệnh) và device driver (được hệ điều hành ra lệnh). Hoạt động bên trong giao diện này là những thao tác của từng device driver đặc trưng cho từng thiết bị đó. Công việc lập trình các thao tác này gọi là lập trình driver.

Một character driver muốn cài đặt và hoạt động bình thường thì phải trải qua nhiều bước lập trình. Đầu tiên là đăng ký số định danh cho driver, số định danh là số mà hệ điều hành linux cung cấp cho mỗi driver để quản lý. Tiếp theo, mô tả tập lệnh mà driver hỗ trợ, chúng ta có thể xem tập lệnh là những thao tác hoạt động bên trong của driver dùng để điều khiển một thiết bị vật lý. Sau khi đã mô tả tập lệnh, chúng ta sẽ liên kết các tập lệnh này với các giao diện chuẩn mà hệ điều hành linux hỗ trợ, nhằm mục đích giao tiếp giữa hệ điều hành và các thiết bị ngoại vị vật lý mà driver điều khiển. Tiếp theo chúng ta định nghĩa liên kết các giao diện này với cấu trúc mô tả tập tin khi thiết bị được mở. Cuối cùng chúng ta thực hiện cài đặt driver thiết bị vào hệ thống thư mục tập tin linux, thông thường nằm trong thư mục /dev.

Trong phần này chúng ta sẽ tìm hiểu một cách chi tiết các bước lập trình driver đã nêu.

II. Số định danh character driver:

Thế nào là số định danh, đặc diểm và vai trò của số định danh của character driver cũng hoàn toàn tương tự như device driver khác mà chúng ta đã nghiên cứu rất kỹ trong bài trước. Chúng ta cũng đã biết những hàm rất quan trọng để thao tác với số định danh này. Ở đây không nhắc lại mà thêm vào đó là làm thế nào để tạo lập một số định danh cho thiết bị nào đó mà không sinh ra lỗi.

1. Xác định số định danh hợp lệ cho thiết bị mới theo cách thông thường:

Một trong những công việc quan trong đầu tiên cần phải làm trong driver là xác định số định danh cho thiết bị. Có hai thông tin cần xác định là số *Major* và số *Minor*.

Trước hết chúng ta phải biết số định danh nào còn trống trong hệ thống chưa được các thiết bị khác sử dụng. Thông tin về số định danh được linux sử dụng chứa trong tập tin *Documentation/devices.txt*. Tập tin này chứa số định danh, tên thiết bị, thời gian tạo lập, loại thiết bị, … đã được linux sử dụng hay sẽ dùng cho những mục đích đặc biệt nào đó. Đọc nội dung trong tập tin này, chúng ta sẽ tìm được số định danh phù hợp, và công việc tiếp theo là đăng ký số định danh đó vào linux.

Linux kernel cung cấp cho chúng ta một hàm dùng để đăng ký số định danh cho thiết bị, hàm đó là:

```
#include <linux/fs.h>
int register_chrdev_region (dev_t first, unsigned int count, char
*name);
```

Để sử dụng được hàm, chúng ta phải khai báo thư viện `<linux/fs.h>`. Tham số thứ nhất `dev_t first` là số định danh thiết bị đầu tiên muốn đăng ký với số Major là số hợp lệ chưa được sử dụng, Minor thông thường cho bằng 0. Tham số thứ hai `unsigned int count` là số thiết bị muốn đăng ký, chẳng hạn muốn đăng ký 1 thiết bị thì ta nhập 1, lúc này chỉ có một thiết bị mang số định danh là `dev_t first` được đăng ký. Tham số thứ ba `char *name` là tên thiết bị muốn đăng ký.

Hàm `register_chrdev_region ()` trả về giá trị kiểu int là 0 nếu quá trình đăng ký thành công. Và trả về số mã lỗi âm khi quá trình đăng ký không thành công.

Tất cả những thông tin khi đăng ký thành công sẽ được hệ điều hành chứa trong tập tin `/proc/devices` và `sysfs` khi quá trình cài dặt thiết bị kết thúc.

Cách đăng ký số định danh trên có một nhược điểm lớn là chỉ áp dụng khi người đăng ký đồng thời là người lập trình nên driver đó vì thế họ sẽ biết rõ số định danh nào là còn trống. Khi driver được sử dụng trên những máy tính khác, thì số định danh được chọn có thể bị trùng với các driver khác. Vì thế việc lựa chọn một số định danh động là cần thiết. Vì số định danh động sẽ không trùng với bất kỳ số định danh nào tồn đang tồn tại trong hệ thống.

Ví dụ, nếu muốn đăng ký một character driver có tên là `"lcd_dev"`, số lượng là 1, số Major đầu tiên là 2, chúng ta tiến hành khai báo hàm như sau:

```
/*Khai báo biến lưu trữ mã lỗi trả về của hàm*/
int res;
/*Thực hiện đăng ký thiết bị cho hệ thống*/
res = register_chrdev_region (2, 1, "lcd_dev");
if (res < 0) {
   printk ("Register device error!");
   exit (1);
}
```

2. Xác định số định danh cho thiết bị theo cách ngẫu nhiên:

Linux cung cấp cho chúng ta một hàm đăng ký số định danh động cho driver thiết bị mới.

```
#include <linux/fs.h>
int alloc_chrdev_region (dev_t *dev, unsigned int firstminor, unsigned int count, char *name);
```

Cũng tương tự như hàm `register_chrdev_region ()`, hàm `alloc_chrdev_region ()` cũng làm nhiệm vụ đăng ký định danh cho một thiết bị mới. Nhưng có một điểm khác biệt là số Major trong định danh không còn cố định nữa, số này do hệ thống linux tự động cấp vì thế sẽ không trùng với bất kỳ số định danh nào khác đã tồn tại.

Tham số thứ nhất của hàm, `dev_t *dev`, là con trỏ kiểu `dev_t` dùng để lưu trữ số định danh đầu tiên trong khoảng định danh được cấp nếu hàm thực hiện thành công;

Tham số thứ hai, `unsigned int first minor`, là số Minor đầu tiên của khoảng định danh muốn cấp;

Tham số thứ ba, `unsigned int count`, là số lượng định danh muốn cấp, tính từ số Major được cấp động và số Minor `unsigned int first minor`;

Tham số thứ tư, `char *name`, là tên của driver thiết bị muốn đăng ký.

Ví dụ khi muốn đăng ký thiết bị tên `"lcd_dev"`, số Minor đầu tiên là 0, số thiết bị muốn đăng ký là 1, số định danh khi tạo ra được lưu vào biến dev_t dev_id. Lúc này hàm `alloc_chrdev_region ()` khai báo như sau:

```
/*Khai báo biến dev_t để lưu giá trị định danh đầu tiên trả về của hàm*/
dev_t dev_id;
/*Khai báo biến lưu mã lỗi trả về của hệ thống*/
int res;
```

```
/*Thực hiện đăng ký thiết bị với định danh động*/
res = alloc_chrdev_region (&dev_id, 0, 1, "lcd_dev");
/*Kiểm tra mã lỗi trả về*/
if ( res < 0) {
   printk ("Allocate devices error!");
   return res;
}
```

Tuy nhiên viêc dăng ký số định danh động cho thiết bị đôi khi cũng có nhiều bất lợi. Giả sử số định danh của thiết bị cần được sử dụng cho những mục đích khác, vì thế số định danh luôn thay đổi khi mỗi lần cài đặt driver sẽ sinh ra lỗi trong quá trình thực thi lệnh. Để kết hợp ưu điểm của 2 phương pháp, chúng ta sẽ đăng ký driver thiết bị theo cách sau:

```
/*Khai báo các biến cần thiết*/
int lcd_major; //Biến lưu trữ số Major
int lcd_minor; //Biến lưu trữ số Minor
dev_t dev_id; //Biến lưu trữ số định danh thiết bị
int result;   //Biến lưu mã lỗi
/*Nếu số Major hợp lệ, đã tồn tại*/
if (lcd_major) {
        dev = MKDEV(lcd_major, lcd_minor); //Tạo số định danh
/*Đăng ký thiết bị với số định danh cố định*/
        result   =   register_chrdev_region   (dev,   lcd_nr_devs,
        "lcd_dev");
} else {
/*Nếu số Major chưa tồn tại, thực hiện tìm kiếm số Major động*/
result   =   alloc_chrdev_region(&dev_id,   lcd_minor,   lcd_nr_devs,
"lcd_dev");
/*Cập nhật lại số Major động cần sử dụng trong những lần sau*/
lcd_major = MAJOR (dev_id);
}
/*Kiểm tra kết quả thực thi của hai lệnh trên*/
if (result < 0) {
    printk(KERN_WARNING "lcd: can't get major %d\n", lcd_major);
    return result;
```

```
}
```

Như vậy ta có thể cập nhật lại số định danh động khi vừa tạo ra để sử dụng cho những chương trình liên quan bằng kỹ thuật như trong đoạn mã lệnh trên.

Character driver bao gồm có nhiều thành phần, đăng ký số định danh chỉ là một trong những thành phần đó. Bên cạnh số định danh character driver còn có những bộ phận như: Cấu trúc dữ liệu (data structure) được gọi là file_operation, cấu trúc này chứa những tập lệnh được người lập trình driver định nghĩa; Cấu trúc mô tả tập tin (file) chứa những thông tin cơ bản của tập tin thiết bị; Cấu trúc tập tin chứa thông tin quản lý tập tin thiết bị trong hệ thống linux.

Phần tiếp theo chúng ta sẽ tìm hiểu cách gán các hành vi cho character device driver thông qua việc thao tác với file_operations.

III. Cấu trúc lệnh của character driver:

Cấu trúc lệnh của character driver (file_operations) là một cấu trúc dùng để liên kết những hàm chứa các thao tác của driver điều khiển thiết bị với những hàm chuẩn trong hệ điều hành giúp giao tiếp giữa người lập trình ứng dụng với thiết bị vật lý. Cấu trúc *file_operation* được định nghĩa trong thư viện `<linux/fs.h>`. Mỗi một tập tin thiết bị được mở trong hệ điều hành linux điều được hệ điều hành dành cho một vùng nhớ mô tả cấu trúc tập tin, trong cấu trúc tập tin có rất nhiều thông tin liên quan phục vụ cho việc thao tác với tập tin đó (chúng ta sẽ nghiên cứu kỹ trong phần sau). Một trong những thông tin này là *file_operations*, dùng mô tả những hàm mà driver thiết bị đang được mở hỗ trợ. Có thể nói một cách khác mỗi tập tin thiết bị trong hệ thống linux tương tự như một vật thể và *file_operation* là những công dụng của vật thể đó.

Cấu trúc *file_operations* là một thành phần trong cấu trúc *file_structure* khi tập tin thiết bị được mở. Mỗi thành phần trong file_*operations* bao gồm những lệnh căn bản theo chuẩn do hệ điều hành định nghĩa, nhưng những lệnh này chưa được định nghĩa thao tác cụ thể, đây là nhiệm vụ của người lập trình driver. Chúng ta phải liên kết những thao tác muốn lập trình với những dạng hàm chuẩn này.

Sau đây chúng ta sẽ tìm hiểu một số những thành phần quan trong trong cấu trúc *file_structure*:

```
struct module *owner
```

Đây không phải là một lệnh trong driver mà chỉ con trỏ cho biết tên driver nào quản lý những lệnh được liên kết. Thông tin này được thiết lập thông qua macro THIS_MODULE định nghĩa trong thư viện <linux/module.h>.

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

Hàm chuẩn này dùng để yêu cầu nhận dữ liệu từ thiết bị vật lý. Nhận dữ liệu như thế nào sẽ do người lập trình quyết định, phù hợp với quy định của từng thiết bị. Tham số thứ nhất, `struct file *`, là con trỏ đến cấu trúc tập tin đang mở trong hệ điều hành, dùng để phân biệt thiết bị này với thiết bị khác. Tham số thứ hai, `char __user *`, là con trỏ được khai báo trong user space, chứa thông tin đọc được từ thiết bị. Tham số thứ ba, size_t là kích thước dữ liệu muốn đọc (tính bằng byte). Tham số thứ tư, `loff_t *`, là con trỏ chỉ vị trí dữ liệu trong thiết bị cần đọc về, nếu để trống thì mặc định là vị trí đầu tiên. Hàm có giá trị trả về là kích thước dữ liệu đọc về thành công.

```
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t
*);
```

Hàm này dùng để ghi thông tin của người dùng vào thiết bị vật lý. Các thao tác ghi cụ thể sẽ do người lập trình quyết định tùy theo từng thiết bị phần cứng. Tham số thứ nhất, `struct file *`, là con trỏ đến cấu trúc tập tin đang mở trong hệ điều hành, dùng để phân biệt thiết bị này với thiết bị khác khi sử dụng nhiều thiết bị. Tham số thứ hai, `char __user *`, là con trỏ được khai báo trong user space, chứa thông tin muốn ghi từ người sử dụng. Tham số thứ ba, `size_t` là kích thước dữ liệu muốn ghi (tính bằng byte). Tham số thứ tư, `loff_t *`, là con trỏ chỉ địa dữ liệu trong thiết bị cần ghi thông tin, nếu để trống thì mặc định là vị trí đầu tiên. Hàm có giá trị trả về là kích thước dữ liệu ghi thành công.

```
int (*open) (struct inode *, struct file *);
```

Đây là hàm luôn được thực thi khi thao tác với driver. Hàm được gọi khi ta sử dụng lệnh mở tập tin driver thiết bị sử dụng. Chúng ta không cần thiết phải lập trình thao tác cho lệnh này. Trong cấu trúc lệnh có thể đặt giá trị `NULL`, như vậy khi đó driver sẽ không được cảnh báo khi thiết bị được mở. Mặc dù không quan trọng nhưng chúng ta nên khai báo lệnh open_device trong chương trình để sử dụng mã lỗi trả về khi cần thiết.

```
int (*release) (struct inode *, struct file *);
```

Hàm chuẩn này dược thực thi khi driver thiết bị không còn sử dụng, thoát khỏi hệ thống linux. Cũng tương tự như hàm open, hàm release có thể không cần khai báo trong

cấu trúc tập lệnh file_operation. Tuy nhiên để thuận lợi trong quá trình lập trình, chúng ta nên khai báo hàm release_device trong driver để có thể trả về mã lỗi nếu cần thiết.

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

ioctl là một hàm rất mạnh trong cấu trúc tập lệnh `file_operations`. Hàm này có thể tích hợp nhiều hàm khác do người lập trình driver định nghĩa. Những hàm khác nhau được phân biệt thông qua các tham số của hàm ioctl. Tham số thứ nhất, `struct inode *`, là cấu trúc tập tin trong hệ thống thư mục linux (chúng ta sẽ nghiên cứu trong phần sau). Tham số thứ hai, `struct file *`, là cấu trúc tập tin đang mở trong hệ thống linux. Tham số thứ ba, `unsigned int`, là số `unsigned int` phân biệt những lệnh khác nhau, có thể gọi đây là số định danh lệnh. tham số thứ ba, dạng `unsigned long`, là tham số của hàm tương ứng với số định danh lệnh. Chúng ta sẽ nghiên cứu sâu các sử dụng hàm trong những bài sau.

Sau đây là một ví dụ cho thấy cách gán chức năng cho các hàm sử dụng trong tập lệnh của character device driver.

```
/*Khai báo cấu trúc lệnh cho driver*/
struct file_operations lcd_fops = {
/*Tên của module sở hữu tập lệnh này*/
.owner =    THIS_MODULE,
/*Gán lệnh đọc lcd_read vào hàm chuẩn read*/
.read =     lcd _read,
/*Gán hàm ghi dữ liệu vào hàm chuẩn write*/
.write =    lcd _write,
/*Gán hàm lcd_ioctl vào hàm chuẩn ioctl*/
.ioctl =    lcd _ioctl,
/*Gán hàm khởi động thiết bị vào hàm chuẩn, có thể đặt giá trị NULL*/
.open =     lcd _open,
/*Gán hàm thoát thiết bị vào hàm chuẩn, có thể đặt giá trị  NULL*/
.release =  lcd _release,
};
```

Tiếp theo chúng ta sẽ nghiên cứu cấu trúc khác lớn hơn trong character device driver. Cấu trúc này chứa những thông tin thao tác tập tin cần thiết khi tập tin đang mở trong đó có cấu trúc tập lệnh *file_operations*.

IV. Cấu trúc mô tả tập tin của character driver:

Cấu trúc mô tả tập tin (file_structure), định nghĩa trong thư viện `<linux/fs.h>` là cấu trúc quan trọng thứ hai trong *character device driver*. Cấu trúc này không xuất hiện trong hệ thống thư mục tập tin của Linux. Mà chỉ xuất hiện khi tập tin được mở, sử dụng trong hệ thống. Khi một tập tin được mở, linux sẽ cung cấp một không gian vùng nhớ lưu trữ những thông tin quan trọng phục vụ cho quá trình lập trình sử dụng tập tin. Những thông tin đó là:

```
mode_t f_mode;
```

Thông tin này quy định chế độ truy xuất tập tin thiết bị. Một tập tin khi được mở trong hệ thống sẽ có thể chỉ được phép đọc, chỉ được phép ghi, hay cả hai bằng cách sử dụng các bit cờ `FMODE_READ` và `FMODE_WRITE`. Chúng ta nên kiểm tra chế độ truy xuất của tập tin thiết bị khi sử dụng hàm ioclt hay open. Nhưng khi sử dụng hàm read và write thi không cần thiết. Vì trước khi thực thi các hàm này hệ thống sẽ tự động kiểm tra các cờ hợp lệ hay không, nếu không hệ thống sẽ bỏ qua không thực thi.

```
loff_t f_pos;
```

Là thông tin lưu vị trí truy cập tập tin, phục vụ cho thao tác `read` và `write`. Đây là số có 64 bits, khả năng truy xuất rất rộng. Người lập trình driver có thể tham khảo thông tin này để biết vị trí hiện tại của con trỏ truy cập tập tin. Tuy nhiên nên hạn chế thay đổi thông tin này. Để thay đổi thông tin này, chúng ta có thể thay đổi trực tiếp bằng cách thay đổi tham số `filp -> f_pos` hoặc có thể sử dụng những hàm chuẩn trong linux.

```
unsigned int f_flags;
```

Đây là những cờ thể hiện chế độ truy cập tập tin, bao gồm những giá trị có thể như, `O_RDONLY`, `O_NONBLOCK`, `O_SYNC` trong những thông tin này thì `O_NONBLOCK` được sử dụng nhiều nhất để kiểm tra lệnh thực hiện có phải là lệnh truy xuất theo block hay không. Còn những thông tin truy xuất khác thông thường được kiểm tra thông qua `f_mode`. Các định nghĩa cho giá trị bit cờ chứa trong thư viện `<linux/fcnt.h>`

```
struct file_operations *f_op;
```

Thông tin này chứa định nghĩa các tập lệnh tương ứng của từng tập tin thiết bị. Thông tin này đã được giải thích rõ trong phần trên.

```
void *private_data;
```

Đây là con trỏ đến vùng nhớ dành riêng cho người sử dụng driver. Vùng nhớ này được xóa khi tập tin được mở, nhưng vẫn tồn tại khi tập tin được đóng, vì thế chúng ta phải tiến hành giải phóng vùng nhớ này trước khi thoát.

```
struct dentry *f_dentry;
```

Chứa thông tin về tập tin nguồn được mở, mỗi tập tin được mở trong hệ thống linux điều bắt nguồn từ một tập tin nào đó lưu trong bộ nhớ. Người viết driver thường dùng thông tin này hơn là thông tin về i_node của thiết bị để quản lý vị trí tập tin thiết bị được mở trong hệ thống.

Trong thực tế một tập tin tổng quát được mở trong hệ thống có thể có nhiều hơn những thông tin nêu trên. Nhưng đối với tập tin driver thì những thông tin đó không cần thiết. Tất cả những driver điều thao tác trên cơ sở những cấu trúc tập tin được xây dựng sẵn.

V. Cấu trúc tập tin của character driver:

Cấu trúc tập tin (*inode structure*) được kernel sử dụng để đặc trưng cho một tập tin driver thiết bị. Cấu trúc này hoàn toàn khác với cấu trúc file structure được giải thích trong phần trước, điều này có nghĩa là có thể có nhiều *file structure* biểu thị cấu trúc tập tin đang mở nhưng tất cả những *file structure* này điều có nguồn gốc từ một *inode structure* duy nhất.

Kernel dùng cấu trúc file structure này để biểu diễn một tập tin thiết bị trong cấu trúc hệ thống của mình (hay nói cụ thể hơn là cấu trúc cây thư mục). Chúng ta có thể mở tập tin này với nhiều chế độ truy xuất khác nhau, mỗi chế độ truy xuất sẽ tương đương với một cấu trúc *file structure*. Cấu trúc *inode structure* chứa rất nhiều thông tin về tập tin thiết bị, trong công việc lập trình driver chúng ta chỉ quan tâm đến những thông tin sau đây:

```
dev_t i_rdev;
```

Mỗi một cấu trúc inode structure đại diện cho một tập tin thiết bị, thông tin này trong inode structure chứa số định danh thiết bị mà chúng ta đã tạo trong phần trước.

```
struct cdev *i_cdev;
```

`struct cdev` là kiểu cấu trúc lưu trữ thông tin của một tập tin lưu trữ trong kernel. Và thông tin `i_cdev` là con trỏ đến cấu trúc này.

Linux cung cấp cho chúng ta hai hàm chuẩn để tìm số định danh Major và Minor của thiết bị biểu thị bằng inode structure. Hai hàm đó là:

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

Chúng ta nên dùng hàm này để lấy thông tin về số định danh thiết bị bên cạnh việc truy cập trực tiếp thông tin `dev_t i_rdev` trong cấu trúc inode structure.

VI. Cài đặt character device driver vào hệ thống Linux:

Sau khi đăng ký số định danh thiết bị, thiết lập liên kết với *file_operations*, gán *file_operations* với *file_structure*, và định nghĩa một *inode_structure*, chúng ta đã hoàn thành cơ bản một *character device driver*. Công việc cuối cùng là tiến hành cài đặt *character driver* này vào hệ điều hành và sử dụng. Phần này sẽ trình bày cho chúng ta các bước để cài đặt những cấu trúc trên vào *kernel* trở thành một *character driver* hoàn chỉnh.

Những hàm được giới thiệu sau đây được định nghĩa trong thư viện `<linux/cdev.h>`.

Trước khi cài đặt thông tin vào kernel chúng ta phải khai báo cho kernel dành ra một không gian vùng nhớ riêng, chuẩn bị cho quá trình cài đặt. Có hai cách thực hiện công việc này.

- ***Cách 1:*** Khai báo cấu trúc trước khi định nghĩa thông tin.

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
...
(Tương tự cho những trường khác)
```

Đầu tiên chúng ta khai báo con trỏ cấu trúc dạng `struc cdev` để chứa con trỏ struct cdev trống do hàm `cdev_alloc()` trả về. Tiếp theo, định nghĩa từng thông tin liên quan cho cấu trúc vừa tạo ra. Chẳng hạn, trong câu lệnh trên, chúng ta cập nhật con trỏ cấu trúc lệnh cho cdev này bằng lệnh gán cơ bản `my_cdev->ops = &my_fops;` trong đó `&my_fops` là cấu trúc lệnh đã được tạo thành từ trước được gán vào trường ops của cấu trúc `my_cdev`.

- ***Cách 2:*** Thực hiện định nghĩa thông tin trước khi khai báo cho kernel.

```
struct cdev *my_cdev;
my_cdev->ops = &my_ops;
```

```
...
(Tiếp theo cho những trường khác)
cdev_init ( my_cdev, my_ops);
```

Như vậy, sau khi định nghĩa xong các trường cần thiết cho cấu trúc struct cdev, chúng ta tiến hành gọi hàm `cdev_init ();` để thông báo cho kernel dành ra một vùng nhớ riêng lưu trữ cdev mới vừa tạo ra.

Cấu trúc của hàm `cdev_init ()` như sau:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Với `struct cdev *cdev` là con trỏ cấu trúc lưu thông tin driver đã được khai báo, `struct file_operations *fops` là cấu trúc tập lệnh của driver.

Sau khi khai báo cho kernel dành một vùng nhớ lưu trữ cấu trúc driver, công việc cuối cùng là triệu gọi hàm `cdev_add ()` để cài đặt cấu trúc này vào kernel. Cấu trúc của hàm cdev_add như sau:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Trong đó, `struct cdev *dev` là con trỏ cấu trúc driver cần cài đặt. `dev_t num` là số định danh thiết bị đầu tiên muốn cài đặt vào hệ thống, số định danh này được xác định phù hợp với hệ thống thông qua các hàm đặc trưng. Tham số cuối cùng, `unsigned int count`, là số thiết bị muốn cài đặt vào kernel.

Hàm này sẽ trả về giá trị 0 nếu quá trình cài đặt driver vào kernel thành công. Ngược lại sẽ trả về mã lỗi âm. Kernel chỉ có thể gọi được những hàm trong driver khi nó được cài đặt thành công vào hệ thống.

Đôi khi chúng ta muốn tháo bỏ cấu trúc device driver ra khỏi hệ thống, linux cung cấp cho chúng ta hàm `cdev_del()` để thực hiện công việc này. Cấu trúc của hàm như sau:

```
void cdev_del(struct cdev *dev);
```

Với `cdev *dev` lả cấu trúc driver muốn tháo bỏ, khi driver được tháo bỏ, thì kernel không thể sử dụng những hàm định nghĩa trong kernel được nữa.

VII. Tổng kết:

Trên đây chúng ta đã tìm hiểu rất kỹ những thành phần cấu tạo nên một character driver do hệ thống linux định nghĩa và quản lý. Chúng ta cũng đã biết cách kết nối những trường có liên quan trong từng cấu trúc với những hàm được định nghĩa và cách cài đặt những cấu trúc đó vào kernel.

Để tỉm hiểu hơn về nguyên tắc giao tiếp giữa driver và user, trong bài sau chúng ta sẽ bàn về các giao diện chuẩn trong cấu trúc lệnh của character device driver, đó là các hàm `read`(), `write`() và `ioctl`();

BÀI 4

CÁC GIAO DIỆN HÀM TRONG DRIVER

I. Tổng quan về giao diện trong cấu trúc lệnh file_operations:

Cấu trúc lệnh *file_operations* là một trong 3 cấu trúc quan trọng của *character device driver*, bao gồm một số những hàm chuẩn giúp giao tiếp giữa chương trình ứng dụng trong *user* và *driver* trong kernel. Chương trình ứng dụng chứa những yêu cầu thực thi từ người dùng chạy trong môi trường *user(user space). Driver* chứa những hàm chức năng điều khiển thiết bị vật lý được lập trình sẵn chạy trong môi trường *kernel(kernel spcae).* Để những yêu cầu từ *user* được nhận ra và điều khiển được phần thiết bị vật lý thì cần phải có một giao diện điều khiển. Những giao diện điều khiển này thực chất là những hàm `read()`, `write()`, và `ioctl()`, ... được linux định nghĩa sẵn, bản thân nó không có chức năng cụ thể, chức năng cụ thể được định nghĩa bởi người lập trình *driver*. Chẳng hạn, hàm `read()` có chức năng tổng quát là đọc thông tin từ driver đến một vùng nhớ đệm trong user (của một chương trình ứng dụng đang chạy), nhưng *driver* muốn lấy được thông tin cung cấp cho *user* thì phải qua nhiều thao tác điều khiển các thanh ghi lệnh thanh ghi dữ liệu của thiết bị vật lý được viết bởi người lập trình *driver,* dữ liệu thu được không thể truyền qua *user* một cách trực tiếp mà phải thông qua các hàm hỗ trợ trong giao diện khác như `copy_to_user()`.

Trong phần này, chúng ta sẽ tìm hiểu nguyên lý hoạt động của những giao diện này, làm cơ sở để viết hoàn chỉnh một *character driver* trong những bài sau.

II. Giao diện `read()` & `write()`:

Do hàm `read()` và `write()` nói chung điều có cùng một nhiệm vụ là trao đổi thông tin qua lại giữa user (*application*) và kernel (*driver*). Nếu hàm `read()` dùng để chép dữ liệu từ *driver* qua *applcation*, thì ngược lại hàm `write()` dùng để chép dữ liệu từ *application* sang *driver*. Hơn nữa hai hàm này điều có những tham số tương tự nhau. Vì thế chúng ta sẽ nghiên cứu hai hàm này cùng lúc.

1. Cấu trúc lệnh:

```
ssize_t read(struct file *filp, char __user *buff, size_t count,
loff_t *offp);
ssize_t write(struct file *filp, const char __user *buff, size_t
count, loff_t *offp);
```

2. Giải thích:

Trong cả hai hàm trên, `filp` là con trỏ tập tin thiết bị muốn truy xuất dữ liệu; `count` là kích thước muốn truy xuất tính bằng đơn vị byte. `buff` là con trỏ đến ô nhớ khai báo trong *application* để lưu dữ liệu đọc về trong trường hợp là hàm `read()`, trong trường hợp là hàm `write()` thì chính là con trỏ đến vùng nhớ rỗng cần ghi dữ liệu đến. Cuối cùng `offp` là con trỏ dạng *long offset* lưu vị trí con trỏ hiện tại của tập tin đang được truy cập. Sau đây chúng ta sẽ tìm hiểu kỹ hơn từng tham số trong hàm.

Con trỏ `const char __user *buff` trong hàm `read(...)` và `write(...)` điều là những con trỏ đến một vùng nhớ trong user space do người lập trình khai báo để lưu dữ liệu truy xuất từ *driver* thiết bị. Hơn nữa *driver* hoạt động trong môi trường *kernel*. Giữa *user space* và *kernel space* có những nguyên tắc quản lý bộ nhớ khác nhau. Do đó, *driver*, hoạt động trong môi trường *kernel*, không thể giao tiếp với con trỏ này, hoạt động trong môi trường *user*, một cách trực tiếp mà phải thông qua một số hàm đặc biệt.

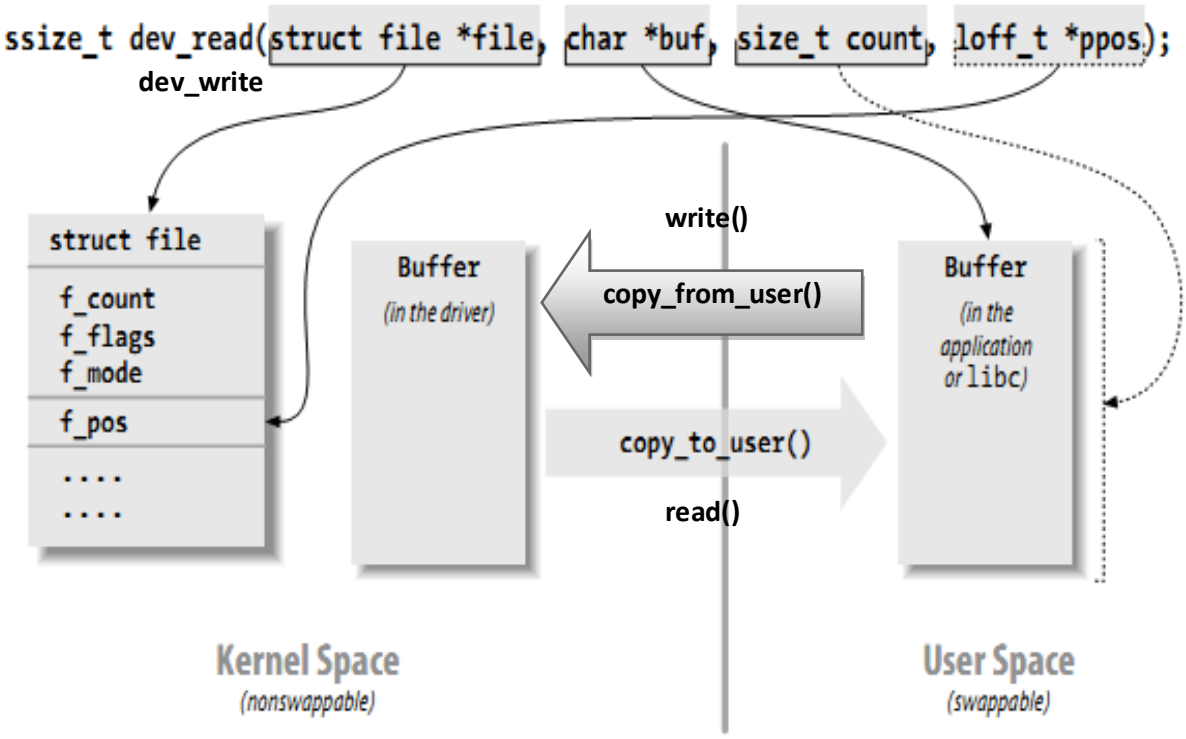
Những hàm giao tiếp này được linux định nghĩa trong thư viện `<asm/uaccess.h>`. Các hàm này là:

- `unsigned long copy_to_user (void __user *to, const void *from, unsigned long count);` Công dụng của hàm là chép dữ liệu từ *kernel space* sang *user space*; Trong đó, `void __user *to` là con trỏ đến vùng nhớ trong *user space* lưu dữ liệu chép từ *kernel space*; `const void *from` là con trỏ chứa dữ liệu trong *kernel space* muốn chép qua *user space*; `unsigned long count` là số bytes dữ liệu muốn chép.
- `unsigned long copy_from_user (void *to, const void __user *from, unsigned long count);` Công dụng của hàm là chép dữ liệu từ *user space* sang *kernel space*; Trong đó, `void *to` là con trỏ đến vùng nhớ trong *kernel space* lưu dữ liệu chép từ *user space*; `const void __user *from` là con trỏ chứa dữ liệu trong *user space* muốn chép qua *kernel space*; `unsigned long count` là số bytes dữ liệu muốn chép.

Hai hàm điều có giá trị trả về kiểu `unsigned long`. Trước khi truyền dữ liệu chúng kiểm tra giá trị con trỏ vùng nhớ có hợp lệ hay không. Nếu không hợp lệ, thì quá trình truyền dữ liệu không thể thực hiện và hàm sẽ trả về mã lỗi `-EFAULT`. Nếu trong quá trình truyền dữ liệu, giá trị con trỏ cập nhật bị lỗi, thì quá trình truyền dừng ngay tại thời điểm đó, và như thế chỉ một phần dữ liệu được chép. Phần dữ liệu truyền thành công sẽ được

thông báo trong giá trị trả về dưới dạng số byte. Chúng ta căn cứ vào giá trị này để truyền lại thông tin nếu cần thiết. Nếu không có lỗi xảy ra, hàm sẽ trả về giá trị 0.

Chúng ta sẽ minh họa nguyên tắc hoạt động của hàm `read()` và `write()` thông qua sơ đồ hình 3-3:



Hình 3-3- Nguyên tắc hoạt động của giao diện hàm read() và write()

Chúng ta thấy trong sơ đồ trên có hai không gian vùng nhớ đệm, vùng nhớ đệm bên *user space* và vùng nhớ đệm bên *kernel space*. Chương trình trong *user space* gọi hàm `read()` hay hàm `write()` thì driver chạy trong *kernel space* sẽ tiến hành thực hiện những thao tác chép dữ liệu bằng cách sử dụng hai hàm `copy_to_user()` hay `copy_from_user()` để giao tiếp trao đổi thông tin giữa hai vùng đệm này. Đồng thời cập nhật những thông tin cần thiết phục vụ theo dõi quá trình truyền dữ liệu. Công việc này được thực hiện bởi người lập trình driver.

III. Giao diện ioctl():

Trên đây chúng ta đã tìm hiểu hai giao diện `read()` và `write()` dùng trao đổi dữ liệu qua lại giữa *user space* và *kernel space*, hay nói đúng hơn là giữa *driver* thiết bị vật lý và chương trình ứng dụng. Thế nhưng trong thực tế, ngoài việc trao đổi dữ liệu, một *driver* thiết bị còn phải thực hiện nhiều công việc khác ví dụ: cập nhật các thông số giao tiếp

(tốc độ *baund*, số bits dữ liệu truyền nhận, ...) của thiết bị vật lý, điều khiển trực tiếp các thanh ghi lệnh phần cứng, đóng mở cổng giao tiếp, ... Linux cung cấp cho một giao diện chuẩn khác phục vụ cho tất cả các thao tác điều khiển trên, đó là giao diện `ioctl()`.

Như vậy, `ioclt()` có thể thực hiện những chức năng của hàm `read()` và `write()`, thế nhưng chỉ hiệu quả trong trường hợp số lượng dữ liệu cần truyền không cao, mang tính chất rời rạc. Trong trường hợp dữ liệu truyền theo từng khối, liên tục thì sử dụng hàm `read()` và `write()` là một lựa chọn khôn ngoan.

1. Cấu trúc lệnh:

Hàm `ioctl()` có cấu trúc như sau:

```
#include <linux/ioctl.h>
int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
```

2. Giải thích:

Như trên đã phân tích, tất cả những giao diện trong *character device driver* đều không có những chức năng cụ thể. Những tham số lệnh chỉ mang tính tổng quát, người lập trình *driver* phải hiểu rõ ý nghĩa từng tham số sau đó sẽ định nghĩa cụ thể chức năng từng tham số này này để đưa vào sử dụng tùy theo yêu cầu trong thực tế. Sau đây chúng ta sẽ tìm hiểu các tham số trong hàm ioctl.

- `struct inode *inode` và `struct file *filp` là con trỏ inode structure và *file structure* tương ứng với từng thiết bị được mở trong chương trình. Hai tham số này điều chứa trong số mô tả tập tin (*file descriptor*) gọi tắt là *fd*, được trả về khi thực hiện mở thiết bị thành công bởi hàm `open()`. *(**Theo nguyên tắc, trước khi thực hiện giao tiếp với các thiết bị trong hệ thống linux, chúng ta phải truy cập đến cấu trúc inode của tập tin thiết bị trên bộ nhớ, kích hoạt khởi động thiết bị bằng hàm `open()`, hàm này sẽ quy định chế độ truy xuất tập tin thiết bị này, sau khi kích hoạt thành công hàm `open()` sẽ trả về số mô tả tập tin fd, mang tất cả thông tin của thiết bị đang được mở trong hệ thống).*
- `unsigned int cmd`, là số unsigned int do người lập trình driver quy định. Mỗi lệnh trong ioctl được phân biệt nhau thông qua số cmd này.

Số `unsigned int cmd` có 16 bits được chia thành 2 phần. Phần đầu tiên có 8 bits MSBs được gọi là số "*magic*" dùng phân biệt lệnh của thiết bị này với thiết bị khác. Phần thứ hai có 8 bit LSBs dùng để phân biệt những lệnh khác nhau của cùng một thiết bị. Số

định danh lệnh này tương tự như số định danh thiết bị, phải được định nghĩa duy nhất trong hệ thống linux. Để biết được số định danh lệnh nào còn trống, chúng ta sẽ tìm trong tập tin *Documentation/ioctl-number.txt*. Tập tin này chứa thông tin về cách sử dụng hàm `ioctl()` và thông tin về số định danh lệnh đã được sử dụng trong hệ thống linux. Nhiệm vụ của chúng ta là đối chiếu so sánh để loại trừ những số định danh này, tìm ra số định danh trống cho thiết bị của mình. Sau khi tìm ra khoảng số định danh lệnh cho mình, chúng ta nên ghi rõ khoảng định danh đó vào tập tin *ioctl-number.txt* để sau này thuận tiện cho viêc lập trình driver mới.

Do hàm `ioctl()` tồn tại trong cả hai môi trường, giao diện trong *user space* và những thao tác lệnh hoạt động trong *kernel space* nên các số định danh lệnh này phải quy định chung trong cả hai môi trường. Đối với các chương trình *application* và *driver* có sử dụng giao diện `ioctl`, việc đầu tiên trong đoạn mã chương trình là định nghĩa số định danh lệnh, mỗi số định danh lệnh có một chế độ truy xuất dữ liệu khác nhau. Những chế độ này được giải thích trong tập tin tài liệu *Documentation/ioctl-number.txt*.

Số định danh thiết bị được định nghĩa theo dạng thức sau:

*/*Định nghĩa số magic cho số định danh lệnh là 'B' trong bảng mã ascii là 66 thập phân*/*

*/*Phần định nghĩa số định danh thiết bị được đặt đầu mỗi chương trình hoặc bên trong một <tập tin.h> thư viện*/*

```
#define IOC_GPIODEV_MAGIC  'B'
```

*/*Định nghĩa lệnh thứ nhất tên GPIO_GET, với mã số lệnh trong thiết bị là 10, chế độ truy xuất là _IO*/*

```
#define GPIO_GET        _IO(IOC_GPIODEV_MAGIC, 10)
#define GPIO_SET        _IO(IOC_GPIODEV_MAGIC, 11)
#define GPIO_CLEAR      _IO(IOC_GPIODEV_MAGIC, 12)
#define GPIO_DIR_IN     _IO(IOC_GPIODEV_MAGIC, 13)
#define GPIO_DIR_OUT    _IO(IOC_GPIODEV_MAGIC, 14)
```

Cấu trúc định nghĩa số định danh lệnh có dạng:

`<chế độ truy xuất lệnh>(<số magic thiết bị>, <số mã lệnh thiết bị>)`

- `<chế độ truy xuất lệnh>`: `ioctl` có 4 trạng thái lệnh: (Quy định bởi 2 bits trạng thái)

`_IO` lệnh `ioctl` không có tham số;

_IOW lệnh ioctl chứa tham số dùng để nhận dữ liệu từ *user* (tương dương chức năng của hàm copy_from_user());

_ IOR lệnh ioctl chứa tham số dùng để gửi dữ liệu sang *user* (tương đương chức năng của hàm copy_to_user());

_IOWR lệnh ioctl chứa tham số dùng cho hai nhiệm vụ, đọc và ghi dữ liệu của *user*.

- <số magic thiết bị>: Là số đặc trưng cho từng thiết bị do người dùng định nghĩa.
- <số mã lệnh thiết bị>: Là số đặc trưng cho từng lệnh trong mỗi thiết bị.

Căn cứ vào số định danh lệnh này, người lập trình *driver* sẽ lựa chọn lệnh nào sẽ được thực thi khi *user* gọi hàm ioctl. Công việc lựa chọn lệnh được thực hiện bằng cấu trúc switch...case... như sau:

Đây là đoạn chương trình mẫu cho hàm ioctl() được khai báo sử dụng trong driver gpio_dev mang tên gpio_ioctl;

```
static int
gpio_ioctl(struct inode * inode, struct file * file, unsigned int
cmd, unsigned long arg)
{
   int retval = 0;
/*Thực hiện lựa chọn lệnh thực thi bằng lệnh switch với tham số  lựa chọn là số định
danh lệnh cmd*/
   switch (cmd)
   {
   case GPIO_GET:
   /*Hàm thao tác cho lệnh GPIO_GET*/
      break;
   case GPIO_SET:
   /*Hàm thao tác cho lệnh GPIO_SET*/
      break;
   case GPIO_CLEAR:
   /*Hàm thao tác cho lệnh GPIO_CLEAR*/
      break;
   case GPIO_DIR_IN:
   /*Hàm thao tác cho lệnh GPIO_DIR_IN*/
```

```
            break;
        case GPIO_DIR_OUT:
        /*Hàm thao tác cho lệnh GPIO_DIR_OUT*/
            break;
        default:
        /*Trả về mã lỗi trong trường hợp không có lệnh thực thi phù  hợp*/
            retval = -EINVAL;
            break;
        }
        return retval;
    }
```

- `unsigned long arg`, là tham số lệnh tương ứng với số định danh lệnh, tham số này có vai trò là bộ nhớ đệm chứa thông tin từ user hay thông tin đến user tùy theo chế độ của lệnh được định nghĩa ở đầu chương trình driver. Trong trường hợp lệnh không có tham số, chúng ta không cần thiết phải sử dụng tham số này trong user, chương trình trong ioctl vẫn thực thi mà không có lỗi. Trong trường hợp hàm cần nhiều tham số, chúng ta khai báo sử dụng tham số này như là một con trỏ dữ liệu.

IV.Tổng kết:

Như vậy với những giao diện chính trong character device như `read()`, `write()` và `ioctl()`, `open()`, ... chúng ta có thể giao tiếp giữa *user* và *kernel* rất thuận lợi. Điều này cũng có nghĩa rằng: Giữa người sử dụng và thiết bị vật lý có thể giao tiếp trao đổi thông tin với nhau dễ dàng thông qua hoạt động của driver với sự hỗ trợ của giao diện. Bên cạnh những giao diện trên, *linux* còn hỗ trợ rất nhiều giao diện khác, ứng dụng của những giao diện này không nằm trong các driver được nghiên cứu trong giáo trình cho nên chúng sẽ không được đề cập đến. Đối với những driver lớn, sẽ có rất nhiều giao diện khác nhau xuất hiện, nhiệm vụ của chúng ta là tìm hiểu nguyên lý hoạt động của giao diện đó dựa vào những kiến thức đã học. Đây cũng chính là mục đích cuối cùng mà nhóm biên tập muốn thực hiện trong cuốn giáo trình này.

Trước khi bước vào viết một driver hoàn chỉnh, chúng ta sẽ tìm hiểu các cấu trúc để viết một driver đơn giản trong bài sau.

BÀI 5

TRÌNH TỰ VIẾT CHARACTER DEVICE DRIVER

Lập trình driver là một trong những công việc quan trọng mà một người lập trình hệ thống nhúng cần phải nắm vững. Đề một chương trình ứng dụng hoạt động tối ưu, người lập trình phải biết phân công nhiệm vụ giữa *driver* và *application* sao cho hợp lý. Trong một ứng dụng điều khiển, nếu *driver* đảm trách nhiều chức năng thì chương trình trong *application* sẽ trở nên đơn giản, nhưng bù lại tài nguyên phần cứng sẽ dễ bị xung đột, không thuận lợi cho các *driver* khác dùng chung tài nguyên. Ngược lại nếu *driver* chỉ thực hiện những công việc rất đơn giản, thì chương trình trong *application* trở nên phức tạp, việc chuyển qua lại giữa các tiến trình trở nên khó khăn. Việc phân chia nhiệm vụ này đòi hỏi phải có nhiều kinh nghiệm lập trình thực tế với hệ thống nhúng mới có thể đem lại hiệu quả tối ưu cho hệ thống.

Với những kiến thức tổng quát đã trình bày trong những phần trước về *driver* mà chủ yếu là *character device driver*, chúng ta đã có những khái niệm ban đầu về vai trò của *driver* trong hệ thống nhúng, cách thức điều khiển thiết bị vật lý, các giao diện giao tiếp thông tin giữa *user space* và *kernel space*, ... trong phần này chúng ta sẽ đi vào các bước cụ thể để viết hoàn chỉnh một *character device driver*.

Công việc phát triển một ứng dụng điều khiển mới thường tiến hành theo những bước sau:

- Tìm hiểu nhu cầu điều khiển ngoài thực tế: Từ hoạt động thực tiễn hay đơn đặt hàng xác định yêu cầu, thời gian thực hiện, giá cả chi phí, ...
- Phân tích yêu cầu điều khiển: Từ những yêu cầu giới hạn về thời gian, chi phí, chất lượng, người lập trình tìm ra phương pháp tối ưu hóa hệ thống, lựa chọn công nghệ phù hợp, ...; Lập danh sách các yêu cầu cần thực hiện trong hệ thống điều khiển;
- Phân công nhiệm vụ điều khiển giữa *application* và *driver* để hệ thống hoạt động tối ưu;
- Lập trình *driver* theo những yêu cầu được lập;
- Lập trình *application* sử dụng *driver* hoàn tất chương trình điều khiển;
- Chạy kiểm tra độ tin cậy hệ thống;

- Giao cho khách hành sử dụng, bào trì sử chữa khắc phục lỗi khi thực thi;

Chúng ta đang tập trung nghiên cứu bước lập trình *driver*, khi đã có những yêu cầu cụ thể. Trong bước này, có rất nhiều cách thực hiện, tuy nhiên nhìn chung điều có những thao tác căn bản sau:

1. Viết lưu đồ hoặc máy trạng thái thực thi cho dirver;
2. Lập trình mã lệnh;
3. Biên dịch *driver*;
4. Cài đặt vào hệ thống linux;
5. Viết chương trình ứng dụng kiểm tra hoạt động driver;
6. Nếu đạt yêu cầu gán cố định vào cấu trúc *kernel* linux, biên dịch lại nhân để *driver* hoạt động lâu dài trong hệ thống; Nếu không tiến hành chỉnh sửa, kiểm tra cho đến khi hoàn thiện;

Trong đó các bước 1, 5, 6 chúng ta đã có dịp nghiên cứu trong phần I lập trình nhúng căn bản hoặc trong các tài liệu chuyên ngành khác. Phần này chúng ta sẽ tìm hiểu chi tiết các bước 2, 3, và 4;

I. Lập trình mã lệnh trong *character driver*:

Có rất nhiều cách viết *character driver* nhưng cũng giống như chương trình ứng dụng *application*, cũng có những cấu trúc chung, chuẩn, từ đó sẽ tùy biến theo từng yêu cầu cụ thể. Ở đây, chúng ta sẽ tìm hiểu hai dạng cấu trúc: Cấu trúc dạng 1, và cấu trúc dạng 2;

Cấu trúc dạng 1: Bao gồm những thao tác cơ bản nhất, nhưng đa số những thao tác kiểm tra lỗi, lấy số định danh lệnh, số định danh thiết bị, ... người lập trình driver phải tự mình thực hiện. Có một ưu điểm là người lập trình có thể tùy biến theo yêu cầu của ứng dụng, có khả năng phát triển driver. Nhưng bù lại, thời gian thực hiện hoàn chỉnh *driver* để có thể chạy trong linux lâu hơn.

Cấu trúc dạng 2: Những thao tác cơ bản trong việc thành lập các thông số cho driver như: Lấy số định danh lệnh, thiết bị, khai báo cài đặt driver vào hệ thống, ... được gói gọn trong một câu lệnh duy nhất. Ưu điểm là thời gian thực hiện hoàn chỉnh một *driver* nhanh hơn, đơn giản hơn, các dịch vụ kiểm tra lỗi được hỗ trợ sẵn, giúp tránh xảy ra lỗi trong quá trình sử dụng. Thế nhưng việc tùy biến của người dùng nằm trong một giới hạn cho phép.

Thông thường chúng ta nên dùng cấu trúc dạng 2 cho những *driver* đơn giản, và những khả năng trong cấu trúc này hầu hết cũng tương tự như trong cấu trúc dạng 1, cũng đủ cho chúng ta thực hiện tất cả những thao tác điều khiển khác nhau.

Sau đây chúng ta sẽ tìm hiểu chi tiết từng cấu trúc để có cái nhìn cụ thể hơn về những ưu và nhược điểm nêu trên. Các chương trình *driver* cũng có cấu trúc tương tự như các chương trình trong C, chỉ khác là chúng có thêm những hàm chuẩn định nghĩa riêng cho việc giao tiếp điều khiển giữa *user space* và *kernel space* và còn nhiều đặc điểm khác nữa, sẽ được chúng ta đề cập trong mỗi phần cấu trúc;

1. Cấu trúc chương trình character device driver dạng 1:

*/*Bước 1: Khai báo thư viện cho các hàm dùng trong chương trình, thư viện dùng trong kernel space khác với thư viện dùng trong user space. Thư viện trong driver là những hàm, biến, hằng số, ... được định nghĩa sẵn và hầu hết được lưu trong thư mục linux/ trong cấu trúc mã nguồn của linux. Do đó khi khai báo, chúng ta phải chỉ rõ đường dẫn đến thư mục này */*

```
#include <linux/module.h> //Thư viện thứ nhất trong thư mục linux
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
......
/*Bước 2: Định nghĩa những hằng số cần dùng trong chương trình driver*/
#define ...
...
/*Bước 3: Khai báo số định danh thiết bị (Device number); Cấu trúc (file structure); Cấu trúc (file operations); */
int                 device_devno; //Khai báo biến lưu số định danh thiết bị
struct cdev         device_cdev; //Khai báo biến lưu cấu trúc tập tin
struct file_operations  device_fops; //Khai báo biến lưu cấu trúc lệnh
/*Bước 4: Khai báo những biến, cấu trúc cần dùng trong chương trình driver*/
unsigned int variable_0; //Những biến này được định nghĩa tương tự trong
unsigned long variable_1;//mã lệnh C, và C++
struct clk *device_clock_0;//
....
```

```
/*Bước 5: Khai báo, định nghĩa những hàm con, chương trình con được sử dụng trong driver*/
    /*Chương trình con thứ nhất, có hai tham số parameter_0 và parameter_1 cùng kiểu int*/
void sub_program_0 (int parameter_0, int parameter_1) {
    /*Những lệnh thao tác cho chương trình con*/
    lệnh_0;
    lệnh_1;
    lệnh_2;
}
    /*Hàm thứ nhất, trả về kiểu int, hai tham số con trỏ parameter_0 và parameter_1 kiểu int*/
int func_0 (int *parameter_0, int *parameter_1) {
    /*Các thao tác lệnh cho hàm*/
    lệnh_0;
    lệnh_1;
    lệnh_2;
}
/*Bước 6: Định nghĩa hàm init cho driver, hàm init là hàm được thực thi đầu tiên khi thực hiện cài đặt driver vào hệ thống linux bằng lệnh shell insmod driver_name.ko*/
    /*Hàm init có một vai trò quan trọng trong lập trình driver. Ban đầu hàm sẽ gọi thực thi các hàm xác định số định danh thiết bị; Cập nhật các thông số của cấu trúc tập tin, cấu trúc lệnh; Đăng ký các cấu trúc vào hệ thống linux; Khởi tạo các thông số điều khiển ban đầu cho các thiết bị ngoại vi mà driver điều khiển*/
    /*Hàm init có kiểu dữ liệu trả về dạng int, static trong trường hợp này nghĩa là hàm init chỉ dùng riêng cho driver sở hữu*/
static int __init at91adc_init (void) {
    /*Khai báo biến lưu giá trị trả về của hàm*/
    int ret;
```

```
/*Xác định số định danh động cho thiết bị, tránh trường hợp bị trùng khi cài đặt
với các driver thiết bị khác đã có trong hệ thống*/
//Chỉ đến địa chỉ biến lưu số định danh đầu tiên tìm được;
ret = alloc_chrdev_region ( &device_devno
                            0,    //Số Minor đầu tiên yêu cầu;
                            1,    //Số thiết bị yêu cầu;
                            "device_name" );//Tên thiết bị muốn đăng ký;
/*Kiểm tra mã lỗi khi thực thi hàm alloc_chrdev_region()*/
if (ret < 0) {
      //In thông báo lỗi khi thực hiện xác định số định danh thiết bị;
      printk (KERN_INFO  "Device_name: Device number allocate
      fail");
      ret = -ENODEV; //Trả về mã lỗi cho biến ret;
      return ret; //Trả về mã lỗi cho hàm init;
}
/*Khởi tạo thiết bị với số định danh đã xác định, yêu cầu hệ thống dành một vùng
nhớ lưu driver thiết bị sắp được tạo ra;*/
cdev_init ( &device_cdev,   //Trỏ đến cấu trúc tập tin đã được định nghĩa;
            device_devno,   //Số định danh thiết bị đã được xác định;
            1 );            //Số thiết bị muốn đăng ký vào hệ thống;
/*Chập nhật những thông tin cần thiết cho cấu trúc tập tin vừa được hệ thống
dành ra*/
device_cdev.owner = THIS_MODULE; /*Cập nhật người sở hữu cấu trúc tập
tin*/
device_cdev.ops = &device_fops; /*Cập nhật cấu trúc lệnh đã được định
nghĩa*/
/*Đăng ký thiết bị vào hệ thống */
ret = cdev_add ( &device_cdev, //Trỏ đến cấu trúc tập tin đã được khởi tạo;
                 device_devno, //Số định danh thiết bị đã được xác định;
                 1 ); //Số thiết bị muốn đăng ký;
/*Kiểm tra lỗi trong quá trình đăng ký thiết bị vào hệ thống*/
if (ret < 0)
```

```
        {
        //In thông báo khi lỗi xuất hiện
        printk(KERN_INFO "Device: Device number allocation
        failed\n");
        ret = -ENODEV; //Trả về mã lỗi cho biến;
        return ret;//Trả về mã lỗi cho hàm;
        }
    /*Thực hiện các công việc khởi tạo thiết bị vật lý do driver điều khiển*/
    ...
    //Trả về mã lỗi 0 khi quá trình thực thi hàm init không có lỗi;
    return 0;
}
```

*/*Bước 7: Khai báo và định nghĩa hàm exit, hàm exit là hàm được thực hiện ngay trước khi driver được tháo gỡ khỏi hệ thống bằng lệnh shell rmmod device.ko; Những tác vụ bên trong hàm exit được lập trình nhằm khôi phục lại trạng thái hệ thống trước khi cài đặt driver. Chẳng hạn như giải phóng vùng nhớ, timer, vô hiệu hóa các nguồn phát sinh ngắt, ...*/*

```
static void __exit device_exit (void) {
    /*Các lệnh giải phóng vùng nhớ sử dụng trong driver*/
    ...
    /*Các lệnh giải phóng nguồn ngắt, timer, ...*/
    ...
    /*Tháo thiết bị ra khỏi hệ thống bằng hàm*/
    unregister_chrdev_region(  device_devno, /*Số định danh đầu tiên nhóm
    thiết bị;*/
                                1); //Số thiết bị cần tháo gỡ;
    /*In thông báo driver thiết bị đã được tháo ra khỏi hệ thống*/
    printk(KERN_INFO "device: Unloaded module\n");
}
```

*/*Bước 8: Khai báo hàm open, đây là hàm được thực thi ngay sau khi lệnh open trong user space thực thi. Những lệnh chứa trong hàm này thông thường dùng cập nhật dữ liệu ban đầu cho chương trình driver, khởi tạo môi trường hoạt động ban đầu để hệ thống làm việc ổn định*/*

```
static int device_open (struct inode *inode, struct file *filp)
{
   /*Nơi lập trình các lệnh khởi tạo hệ thống*/
   return 0;
}
```

*/*Bước 9: Khai báo và định nghĩa hàm release, đây là hàm được thực thi ngay trước khi driver bị đóng bởi hàm close trong user space*/*

```
static int device_release (struct inode *inode, struct file *filp)
{
   /*Nơi thực thi những lệnh trước khi driver bị đóng, không còn sử dụng*/
   return 0;
}
```

*/*Bước 10: Khai báo các hàm read(), write(), ioctl() trong hệ thống khi cần sử dụng*/*

*/*Hàm read() đọc thông tin từ driver qua chương trình trong user*/*

```
static ssize_t device_read (struct file *filp, char __iomem *buf,
size_t bufsize, loff_t *f_pos) {
```

*/*Định nghĩa các tác vụ hoạt động trong hàm read, để truy cập lấy thông tin từ thiết bị vật lý*/*

```
...
}
```

*/*Hàm write() ghi thông tin từ user qua driver*/*

```
static ssize_t device_write (struct file *filp, char __iomem *buf,
size_t bufsize, loff_t *f_pos) {
```

*/*Định nghĩa các tác vụ hoạt động trong hàm write(), để truy cập lấy thông tin từ chương trình ứng dụng trong user*/*

```
...
}
```

*/*Hàm ioctl định nghĩa các trạng thái lệnh thực thi theo số định danh lệnh từ chương trình ứng dụng bên user*/*

```
static int
gpio_ioctl (struct inode * inode, struct file * file, unsigned int
cmd, unsigned long arg) {
```

*/*Khai báo biến lưu mã lỗi trả về cho hàm giao diện ioctl()*/*

```
    int retval = 0;
    /*Chia trường hợp thực thi lệnh*/
    switch (cmd) {
          case LENH_0:
    /*Lệnh 0 được thực thi*/
        break;
          case LENH_1:
    /*Lệnh 1 được thực thi*/
        break;
          default:
    /*Có thể thông báo, không có lệnh nào để thực thi*/
        retval = -EINVAL;
        break;
          }
          return retval;
}
```

*/*Bước 11: Gán các con trỏ hàm giao diện chuẩn vào cấu trúc lệnh file structure*/*

```
struct file_operations gpio_fops = {
        .read      = device_read,
        .write     = device_write,
        .ioctl     = device_ioctl,
        .open      = device_open,
        .release   = device_release,
};
```

*/*Bước 12: Gán các hàm exit và init vào hệ thống driver*/*

`module_init (device_init);` */*Hàm device_init() thực hiện khi driver được cài đặt*/*

`module_exit (device_exit);` */*Hàm device_exit() thực hiện khi driver được gỡ bỏ*/*

*/*Bước 13: Khai báo các thông tin liên quan đến driver*/*

```
MODULE_AUTHOR("Tên người viết driver");
MODULE_DESCRIPTION("Mô tả nhiệm vụ của driver");
```

```
MODULE_LICENSE("GPL"); //Bản quyền của driver là GPL
```

2. Cấu trúc chương trình character device driver dạng 2:

Cấu trúc chương trình *character device driver* dạng 2 cũng tương tự như dạng 1, nhưng điểm khác biệt là kỹ thuật tạo và đăng ký thiết bị mới vào hệ thống. Để thuận tiện cho việc tham khảo chúng tôi không ngại nhắc lại những bước tương tự và giải thích những điều mới khi tiếp xúc với câu lệnh.

```
/*Bước 1: Khai báo những thư viện cần thiết cho các lệnh dùng trong chương trình driver*/
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <asm/uaccess.h>
#include <asm/atomic.h>
#include <linux/miscdevice.h>
/*Bước 2: Định nghĩa những hằng số cần dùng trong chương trình driver*/
#define ...
...
/*Bước 3: Khai báo số định danh thiết bị, các biến phục vụ đồng bộ hóa dữ liệu*/
/*Biến lưu số major của thiết bị*/
static int dev_major;
/*Khai báo biến atomic_t dùng đồng bộ hóa tài nguyên driver thiết bị*/
/*Cách đồng bộ hóa tài nguyên của thiết bị:
Trước khi mở thao tác với driver thiết bị, kỹ thuật atomic sẽ kiểm tra xem thiết bị đã được mở hay chưa, nếu thiết bị đã được mở (nhận biết bằng số counter) thì không cho phép truy cập đến thiết bị này, thông báo lỗi cho người sử dụng. Ngược lại, cho phép mở thiết bị, thực thi những câu lệnh tiếp theo trong hệ thống. Cụ thể là:
- Ban đầu cho biến kiểu atomic_t bằng 1, nghĩa là thiết bị chưa được mở;
- Khi thiết bị được mở, chúng ta giảm biến kiểu atomic_t một đơn vị, lúc này giá trị biến kiểu atomic_t bằng 0. Trong quá trình mở, hay đóng điều phải có sự so sánh biến atomic_t.
- Chỉ đóng driver thiết bị khi biến kiểu atomic_t bằng 1. Chỉ mở khi biến kiểu atomic_t bằng 0.
```

- *Đóng thiết bị, thực hiện tăng biến kiểu atomic_t. Mở thiết bị, thực hiện giảm biến kiểu atomic_t.*/*

```
static atomic_t device_open_cnt = ATOMIC_INIT(1);
/*Bước 4: Khai báo các biến dùng trong chương trình driver*/
...
/*Bước 5: Khai báo và định nghĩa các hàm, chương trình con cần sử dụng trong quá trình lập trình driver*/
...
/*Bước 6: Khai báo và định nghĩa hàm open, là hàm được thực thi khi thực hiện lệnh open() trong user application*/
static int device_open (struct inode *inode, struct file *file) {
   /*Khai báo biến lưu mã lỗi trả về*/
   int result = 0;
   /*Khai báo biến lưu số minor của thiết bị, đồng thời cập nhật số minor của thiết bị trong hệ thống*/
   unsigned int device_minor = MINOR (inode->i_rdev);
   /*Thực hiện kiểm tra biến atomic_t trong quá trình mở thiết bị nhiều lần*/
   if (!atomic_dec_and_test(&device_open_cnt)) {
      /*Tăng biến kiểu atomic_t*/
      tomic_inc(&device_open_cnt);
      /*In ra thông báo thiết bị muốn mở đang bận*/
      printk(KERN_ERR DRVNAME ": Device with minor ID %d already in
      use\n", dev_minor);
      /*Trả về mã lỗi bận*/
      return -EBUSY;
   }
/*Thực hiện những công việc tiếp theo nếu mở tập tin thiết bị thành công*/
...
/*Trả về mã lỗi cuối cùng nếu quá trình thực thi không bị lỗi*/
return result;
}
```

```
/*Bước 7: Khai báo và định nghĩa hàm close(), được thực hiện khi thực thi hàm close trong user application*/
static int device_close (struct inode *inode, struct file *file) {
   /*Thực hiện đồng bộ hóa counter trước khi tăng*/
   smp_mb_before_atomic_inc();
   /*Tăng biến đếm atomic khi đóng tập tin thiết bị*/
   atomic_inc(&device_open_cnt);
   /*Trả về mã lỗi 0 cho hàm close()*/
   return 0;
}

/*Bước 8: Khai báo và cập nhật cấu trúc lệnh file_operations cho thiết bị*/
struct file_operations device_fops = {
   /*Gán hàm device_read của thiết bị vào giao diện chuẩn, nếu sử dụng*/
   .read = device_read,
   /*Gán hàm device_write của thiết bị vào giao diện chuẩn, nếu sử dụng*/
   .write = device_write,
   /*Gán hàm device_open của thiết bị vào giao diện chuẩn open*/
   .open = device_open,
   /*Gán hàm device_release của thiết bị vào giao diện chuẩn release*/
   .release = device_release,
};
/*Bước 9: Khai báo và cập nhật cấu trúc tập tin thiết bị theo kiểu miscdevice*/
static struct misdevice device_dev = {
   /*Thực hiện kỹ thuật lấy số minor động cho thiết bị bằng hằng số định nghĩa sẵn trong thư viện linux/miscdevice.h*/
   .minor = MISC_DYNAMIC_MINOR,
   /*Cập nhật tên cho thiết bị*/
   .name = "device",
   /*Cập nhật cấu trúc lệnh đã được định nghĩa*/
   .fops = device_fops,
};
```

*/*Bước 10: Khai báo định nghĩa hàm init thực hiện khi cài đặt driver vào hệ thống*/*

*/*Tương tự như trong dạng 1, tuy nhiên với kỹ thuật mới này, trong hàm init chúng ta không cần phải lấy số định danh thiết bị, khởi tạo vùng nhớ cho driver, cài đặt driver vào hệ thống. Tất cả những công việc này được thực hiện bởi một hàm duy nhất đó là misc_register()*/*

```
static int __init device_mod_init(void) {
   /*Những lệnh cần thực hiện khi cài đặt driver vào hệ thống*/
   ...
   /*Thực hiện đăng ký driver vào hệ thống, kết hợp trả về mã lỗi*/
   return     misc_register(&devive_dev);
}
```

*/*Bước 11: Khai báo và định nghĩa hàm exit()*/*

```
static void __exit device_mod_exit (void) {
   /*Thực hiện những công việc khôi phục hệ thống trước khi driver bị tháo gỡ*/
   ...
   /*Tháo gỡ thiết bị mang tên là device_dev*/
   misc_deregister (&device_dev);
}
```

*/*Bước 12: Gắn các hàm exit và init vào hệ thống driver*/*

```
module_init (device_mod_init);
module_exit (device_mod_exit);
```

*/*Bước 13: Cập nhật các thông tin cá nhân cho driver*/*

```
MODULE_LICENSE("Bản quyền driver thông thường là GPL");
MODULE_AUTHOR("Tên người viết driver");
MODULE_DESCRIPTION("Mô tả khái quát thông tin về driver");
```

II. Biên dịch driver:

Sau khi lập trình *dirver*, công việc tiếp theo là biên dịch *driver* biến tập tin mã nguồn C thành tập tin ngôn ngữ máy *driver* trước khi cài đặt vào hệ điều hành. Sau đây là các bước biên dịch driver.

Biên dịch driver có 2 dạng, đầu tiên biên dịch driver khi tập tin mã nguồn driver thuộc một bộ phận trong cấu trúc mã nguồn mở của *kernel*, khi đó *driver* được biên dịch cùng lúc với *kernel*, cách này chỉ áp dụng khi *driver* đã hoạt động ổn định, hơn nữa chúng ta

không cần cài đặt lại *driver* khi khởi động lại hệ thống. Phương pháp thứ hai là biên dịch driver khi nó nằm ngoài cấu trúc mã nguồn mở của *kernel*, *driver* có thể được biên dịch trong khi *kernel* đang chạy, ưu điểm của phương pháp này là thời gian thực hiện nhanh, thích hợp cho việc thử nghiệm *driver* mới, thế nhưng mỗi lần hệ thống khởi động lại, driver sẽ bị mất do đó phải cài đặt lại khi khởi động. Khi *driver* đã hoạt động ổn định chúng ta mới biên dịch *driver* theo cách 1.

Biên dịch *driver* hoàn toàn khác với biên dịch chương trình ứng dụng. Chương trình ứng dụng có những thư viện chuẩn trong hệ thống *linux*, nên khi biên dịch ta chỉ việc chép tập tin chương trình vào trong một thư mục bất kỳ trong cấu trúc *root file system* và gọi lệnh biên dịch. Nhưng đối với *driver*, những thư viện sử dụng không nằm sẵn trong hệ thống, mà nằm trong cấu trúc mã nguồn mở của *kernel*. Vì thế trước khi biên dịch *driver* chúng ta phải giải nén tập tin mã nguồn mở của *kernel* vào cấu trúc *root file system*. Sau đó tạo tập tin `Makefile` để dùng lệnh `make` trong *shell* biên dịch *driver*. Cấu trúc Makefile đã được hướng dẫn kỹ trong phần lập trình hệ thống nhúng căn bản. Trong phần này chúng ta chỉ tạo ra `Makefile` với nội dung cần thiết để có thể biên dịch được *driver*.

Biên dịch *driver* được tiến hành theo các bước sau:

1. Chép tập tin *driver* mã nguồn C vào thư mục nào đó trong cấu trúc *root file system*.
2. Tạo tập tin có tên `Makefile` nằm trong cùng thư mục với tập tin *driver* mã nguồn C. Tập tin `Makefile` có nội dung như sau:

```
/*Thông báo cho trình biên dịch biết loại chip mà driver sẽ cài đặt*/
export ARCH=arm
/*Khai báo chương trình biên chéo là những tập tin có tên đầu tiên là arm-none-linux-gnueabi-...*/
export CROSS_COMPILE=arm-none-linux-gnueabi-
/*Tại đây tập tin mã nguồn driver sẽ được biên dịch thành tập tin .ko, có thể cài đặt vào linux*/
obj-m += <tên tập tin driver mã nguồn C>.o
/*Tùy chọn all, thực hiện chuyển đến cấu trúc mã nguồn mở của kernel, tại đây driver sẽ được biên dịch thông qua lệnh modules */
all:
```

```
make –C <đường dẫn đến thư mục chứa cấu trúc mã nguồn mở
của kernel> M=$(PWD) modules
```

*/*Tùy chọn clean, thực hiện chuyển đến cấu trúc mã nguồn mở của kernel, thực hiện xóa những tập tin .o, .ko được tạo thành trong lần biên dịch trước*/*

```
clean:
make -C / đường dẫn đến thư mục chứa cấu trúc mã nguồn mở
của kernel> M=$(PWD) clean
```

3. Tại thư mục chứa mã nguồn *driver*, dùng lệnh *shell*: `make clean all`
   Lúc này hệ thống linux sẽ xóa những tập tin có đuôi .o, .ko, ... tạo thành trong những lần biên dịch trước.Tiếp theo, biên dịch tập tin mã nguồn driver thành tập tin .ko, tập tin này có thể được cài dặt vào hệ thống linux thông qua những thao tác sẽ được hướng dẫn trong phần sau.

III. Cài đặt driver vào hệ thống linux:

Trong phần I, chúng ta đã trình bày hai cấu trúc chung để viết hoàn chỉnh một *character device driver*, mỗi cấu trúc điều có những ưu và nhược điểm riêng. Những ưu và nhược điểm này còn được biểu hiện rõ trong việc cài đặt *driver* vào hệ thống.

Sau khi đã biên dịch thành công *driver*, xuất hiện tập tin .ko trong thư mục chứa tập tin mã nguồn, chúng ta dùng những câu lệnh trong *shell* để hoàn tất công đoạn cuối cùng đưa driver vào hoạt động trong hệ thống, tiến hành kiểm tra chức năng. Tùy theo kỹ thuật áp dụng lập trình *driver* mà sẽ có những thao tác cài đặt khác nhau:

1. Cài đặt driver khi áp dụng cấu trúc dạng 1:

Tiến hành theo các bước sau:

- Di chuyển đến thư mục chứa tập tin .ko vừa biên dịch xong;
- Tại dòng lệnh shell, thực thi: `insmod <tên driver>.ko`;
- Vào tập tin */proc/devices* tìm tên thiết bị vừa cài đặt vào hệ thống, xác định số *Major* và số *Minor* động của thiết bị;
- Sử dụng câu lệnh trong shell: `mknod /dev/<tên thiết bị> c <Số Major> <Số Minor>`, tạo inode trong thư mục */dev/*, làm tập tin thiết bị sử dụng cho những chương trình ứng dụng;

2. Cài đặt driver khi áp dụng cấu trúc dang 2:

- Di chuyển đến thư mục chứa tập tin .ko vùa biên dịch xong;
- Tại dòng lệnh *shell*, thực thi lệnh: `insmod <tên driver>.ko`;

  Khi đó, cấu trúc inode tự động được tạo ra trong thư mục */dev/* liên kết với số định danh thiết bị lưu trong tập tin */proc/devices* mà không cần phải thông qua những câu lệnh shell khác như trong cách 1. Như vậy, với cấu trúc dạng 2 thời gian cài đặt driver vào hệ thống được rút gọn đáng kể.

IV.Tổng kết:

Từ những kiến thức lý thuyết nền tảng trong những bài trước, chúng ta đã rút ra được những bước tổng quát để lập trình hoàn chỉnh một character *device driver*. Có nhiều cách để viết ra một *character driver*, trong bài này chi đề cập 2 cách căn bản làm nền tảng cho các bạn nghiên cứu thêm những cách khác hiệu quả hơn ngoài thực tế.

Trong bài sau, chúng ta sẽ thực hành viết một *character device driver* mang tên là *helloworld*. *Driver* này sẽ áp dụng tất cả những kỹ thuật đã được học. Nếu cần thiết, các bạn có thể xem lại lý thuyết cũ trước khi qua bài sau để nắm được những vấn đề cốt lõi trong lập trình *driver*.

BÀI 6

HELLO WORLD DRIVER

I. Mở đầu:

Đến đây, chúng ta đã có được những kiến thức gần như đầy đủ để tự mình viết một *character device driver*. Các bạn đã biết thế nào là *character driver*, số định danh lệnh, số định danh thiết bị, cấu trúc *inode*, *file structure*,...cũng như những lệnh khởi tạo và cập nhật chúng. Với những yêu cầu của từng lệnh, chúng ta đã rút ra được hai cấu trúc chung khi muốn viết một *driver* hoàn chỉnh, đã được tìm hiểu trong bài trước. Tùy vào từng cấu trúc mà có các bước cải đặt *driver* khác nhau. Thế nhưng, chúng ta chỉ mới dừng lại các thao tác trên giấy, chưa thực sự lập trình ra được một *driver* nào. Trong bài này, chúng ta sẽ viết một dự án có tên *helloworld* nhằm mục đích thực tế hóa những thao tác lệnh đã được trình bày trong phần driver.

Với mục đích là đem những thao tác lệnh đã được học vào thực tế chương trình, dự án này bao gồm có hai thành phần cần hoàn thành đó là driver và. Driver trong dự án sẽ áp dụng 3 giao diện chính dùng để trao đổi thông tin dữ liệu và điều khiển qua lại giữa hai lớp user và kernel, đó là các giao diện hàm read(), write() và ioctl() cùng với các hàm đóng mở driver như open() hay close(). Chương trình ứng dụng (Application) sẽ áp dụng những hàm về truy cập tập tin, ... để truy xuất những thông tin trong *driver*, hiển thị cho người dùng. Ngoài ra, *driver* và *application* điều có nhiệm vụ xuất thông tin có liên quan để người lập trình biết môi trường nào đang thực thi.

Theo những yêu cầu trên, đầu tiên chúng ta sẽ phân công tác vụ cho *driver* và *application* trong mục II sau đó lập trình theo những tác vụ đã phân công trong mục III. Người lập trình sẽ biên dịch, cài đặt chương trình vào hệ thống linux, thực thi và quan sát kết quả. Từ đó giải thích rút ra nhận xét.

II. Phân công tác vụ giữa *driver* và *application*:

helloworld là một dự án bao quát tất cả những kỹ thuật lập trình *driver* được nghiên cứu trong những nội dung trước. Nhiệm vụ chính là dùng hàm `printf()` và `printk()` xuất thông tin ra màn hình hiển thị theo một quy luật phù hợp, từ đó người lập trình có thể hiểu nguyên lý hoạt động của từng hàm sử dụng để áp dụng vào trường hợp khác. *Driver* và *Application* điều có nhiệm vụ riêng, tùy vào từng giao diện sử dụng mà yêu cầu

của từng ví dụ sẽ khác nhau, sao cho toát lên được ý nghĩa cốt lõi của giao diện hàm. Sau đây là chi tiết từng yêu cầu của dự án *helloworld*.

1. Driver:

a. Giao diện read():

Giao diện `read()` được sử dụng để chép thông tin từ *kernel space* sang *user space*. Thông tin được lưu trong *driver* là số khởi tạo ban đầu khi *driver* được mở chứa trong một biến cục bộ. Tại thời điểm gọi giao diện `read()`, thông tin này sẽ được chép qua *user space*, chứa trong biến khai báo trong *user application* thông qua hàm `copy_to_user()`.

Sau khi chép thông tin cho *user, driver* thông báo cho người lập trình biết quá trình chép thành công hay không.

b. Giao diện write():

Giao diện `write()` dùng để chép thông tin từ *user space* qua *kernel space*. Thông tin từ *user* là dữ liệu kiểu số do người dùng nhập vào chuyển qua *kernel* thông qua giao diện `write()` lưu vào một biến trong *kernel*, biến này cũng là nơi lưu thông tin được chuyển sang *user* khi giao diện `read()` được gọi.

Sau khi nhận dữ liệu từ *user, driver* thông báo cho người lập trình quá trình chép thành công. Ngược lại sẽ trả về mã lỗi.

c. Giao diện ioctl():

`ioctl()` là một giao diện hàm đa chức năng, nghĩa là chỉ cần một dạng câu lệnh mà có thể thực hiện được tất cả những chức năng khác. Để thể hiện được những chức năng này của hàm, chúng ta lập trình một ví dụ sau:

Xây dựng giao diện `ioctl()` thành một hàm toán học, thực hiện các phép toán cộng, trừ, nhân, chia. Các tham số được truyền từ *user*, sau khi tính toán xong, kết quả được gửi ngược lại *user*. Các phép toán được lựa chọn thông qua các số định danh lệnh.

2. Application:

Chương trình trong *user*, sử dụng kỹ thuật tùy chọn trong hàm `main()` để kiểm tra tất cả những chức năng do chương trình trong *driver* hỗ trợ. Mỗi chức năng sẽ được thực hiện do người sử dụng lựa chọn.

Trước khi đi vào viết chương trình cụ thể, chúng ta sẽ tìm hiểu hệ điều hành linux thực hiện tùy chọn trong hàm `main()` như thế nào:

Hàm `main()` là hàm được thực hiện đầu tiên khi chương trình ứng dụng được gọi thực thi. Từ hàm này, người lập trình sẽ tiến hành tất cả những chức năng như tạo lập tiến trình, tuyến, gọi các chương trình con, ... Thông thường hàm `main()` được khai báo như sau:

```
void main(void) {
/*Các lệnh do người lập trình định nghĩa*/
}
```

Với cách khai báo này thì hàm `main()` không có tham số cũng như không có dữ liệu trả về.

Ngoài ra linux còn hỗ trợ cho người lập trình cách khai báo hàm `main()` khác có dạng như sau:

```
int main (int argc, char **argv) {
   /*Các lệnh do người lập trình định nghĩa*/
}
```

Với cách lập trình này hàm `main()` có thể được người sử dụng cung cấp các tham số trong khi gọi thực thi. Cú pháp cung cấp tham số trong câu lệnh `shell` như sau:

```
./<tên chương trình> <tham số 1> <tham số 2> <...> <tham số n>
```

Hàm `main()` có hai tham số:

- Tham số thứ nhất `int argc` là số `int` lưu số lượng tham số khai báo trong câu lệnh *shell* trên, bao gồm cả tham số đầu tiên là tên chương trình chứa hàm `main()`.
- Tham số thứ hai `char **argv` là mảng con trỏ lưu nội dung từng tham số nhập trong câu lệnh gọi chương trình thực thi trong *shell*;

Như vậy tên chương trình chứa hàm `main()` sẽ thuộc về tham số đầu tiên có nội dung lưu trong `argv[0]`. Tương tự `<tham số 1>` là tham số thứ hai chứa trong `argv[1]`, ...Tương tự cho các tham số khác. Tổng quát nếu có n tham số trong câu lệnh shell thì trong hàm main có: `argc = n+1; argv[0], argv[1], argv[2], ..., argv[n]` lưu nội dung của từng tham số.

Trong chương trình ứng dụng *user application* của dự án *helloworld*, hàm `main()` được khai báo có dạng tùy chọn như trên. Khi người dùng nhập: `"add", "sub", "mul", "div", "read", "write"`, thì câu lệnh tương ứng sẽ thực hiện gọi các giao diện hàm cần thiết thực hiện chức năng cộng, trừ , nhân, chia, đọc và ghi được lập trình

trong *dirver*. Thao tác cụ thể sẽ được chúng tôi chú thích trong từng dòng lệnh của *dirver* và *application*.

III. Chương trình driver và application:

1. *Chương trình driver:*

```
/*Chương trình driver mang tên helloworld_dev.c*/
/*Khai báo các thư viện cần thiết*/
#include <linux/module.h> /*dùng cho các cấu trúc module*/
#include <linux/errno.h> /*dùng truy xuất mã lỗi*/
#include <linux/init.h> /*dùng cho các macro module_init(), ...*/
#include <asm/uaccess.h> /*dùng cho các hàm copy_to(from)_user*/
#include <asm/atomic.h> /*dùng cho kiểm tra atomic*/
#include <linux/genhd.h> /*dùng cho các giao diện hàm*/
#include <linux/miscdevice.h> /*dùng cho kỹ thuật tạo device tự động*/
/*Khai báo tên cho device và driver*/
#define DRVNAME      "helloworld_dev"
#define DEVNAME      "helloworld"
/*Khai báo số type cho số định danh lệnh dùng trong ioctl() */
#define HELLOWORLD_DEV_MAGIC  'B' /*Số type là số 66 trong mã ascii*/
/*Số định danh lệnh cho hàm add*/
#define HELLOWORLD_ADD _IOWR(HELLOWORLD_DEV_MAGIC, 10,unsigned long)
/*Số định danh lệnh cho hàm sub*/
#define HELLOWORLD_SUB _IOWR(HELLOWORLD_DEV_MAGIC, 11,unsigned long)
/*Số định danh lệnh cho hàm mul*/
#define HELLOWORLD_MUL _IOWR(HELLOWORLD_DEV_MAGIC, 12,unsigned long)
/*Số định danh lệnh cho hàm div*/
#define HELLOWORLD_DIV _IOWR(HELLOWORLD_DEV_MAGIC, 13,unsigned long)

/*Biến lưu giá trị truyền sang user space khi gọi hàm read() từ user;  hay lưu giá trị nhận được từ user khi hàm write được gọi*/
static char helloworld_dev_number = 34;
/*Khai báo biến đếm atomic và khởi tạo*/
static atomic_t helloworld_open_cnt = ATOMIC_INIT(1);
/*Cấu trúc ioctl() thực hiện các phép toán add, sub, mul và div*/
```

```
static int
helloworld_ioctl(struct inode * inode, struct file * file, unsigned
int cmd, unsigned long arg[])
{     /*Biến lưu mã lỗi trả về của ioctl()*/
      int retval;
      /*Phân biệt các trường hợp lệnh khác nhau*/
      switch (cmd)
   {
   case HELLOWORLD_ADD:
            /*Thông báo quá trình tính toán bắt đầu*/
            printk ("Driver: Start ...\n");
            /*Thực hiện tính toán, chép qua user qua biến arg[2]*/
            arg[2] = arg[0] + arg [1];
            /*In thông báo tính toán kết thúc*/
            printk ("Driver: Calculating is complete\n");
            break;

   case HELLOWORLD_SUB:
            /*Thông báo quá trình tính toán bắt đầu*/
            printk ("Driver: Start ...\n");
            /*Thực hiện tính toán, chép qua user qua biến arg[2]*/
            arg[2] = arg[0] - arg [1];
            /*In thông báo tính toán kết thúc*/
            printk ("Driver: Calculating is complete\n");
            break;

   case HELLOWORLD_MUL:
            /*Thông báo quá trình tính toán bắt đầu*/
            printk ("Driver: Start ...\n");
            /*Thực hiện tính toán, chép qua user qua biến arg[2]*/
            arg[2] = arg[0] * arg [1];
            /*In thông báo tính toán kết thúc*/
            printk ("Driver: Calculating is complete\n");
            break;
```

```
   case HELLOWORLD_DIV:
            /*Thông báo quá trình tính toán bắt đầu*/
            printk ("Driver: Start ...\n");
            /*Thực hiện tính toán, chép qua user qua biến arg[2]*/
            arg[2] = arg[0] / arg [1];
            /*In thông báo tính toán kết thúc*/
            printk ("Driver: Calculating is complete\n");
            break;
   default:
            /*Thông báo lỗi cho người sử dụng, không có lệnh hỗ trợ*/
            printk ("Driver: I don't have this operation!\n");
            retval = -EINVAL;
            break;
   }
  /*Trả về mã lỗi cho hàm ioctl() */
   return retval;
}
/*Khai báo và định nghĩa hàm giao diện read*/
static ssize_t helloworld_read (struct file *filp, char __iomem buf[],
size_t bufsize, loff_t *f_pos)
{
      /*Khai báo con trỏ đệm dữ liệu cần truyền qua user */
      int *read_buf;
      /*Biến lưu kích thước đã truyền thành công*/
      int read_size = 0;
      /*Cập nhật con trỏ đệm phát*/
      read_buf = &helloworld_dev_number;
      /*Gọi hàm truyền dữ liệu từ con trỏ đệm phát sang user, kiểm tra mã lỗi nếu
      đúng trả về kích thước phát thành công*/
      if (copy_to_user (buf, read_buf, bufsize) != 0 ) return -EFAULT;
      else read_size = bufsize;
      /*Thông báo số bytes truyền thành công qua user*/
      printk ("Driver: Has sent %d byte(s) to user.\n", read_size);
      /*Trả về số byte dữ liệu truyền thành công qua user*/
      return read_size;
```

```
}
/*Định nghĩa giao diện hàm write()*/
static ssize_t helloworld_write (struct file *filp, char __iomem
buf[], size_t bufsize, loff_t *f_pos)
{
      /*Khai báo biến đệm nhận dữ liệu từ user*/
      char write_buf[1];
      /*Khai báo biến lưu kích thước nhận thành công*/
      int write_size = 0;
      /*Gọi hàm lấy dữ liệu từ user, có kiểm tra lỗi, đúng sẽ trả về số byte dữ liệu
      nhận thành công*/
      if (copy_from_user (write_buf, buf, bufsize) != 0) return -
      EFAULT; else write_size = bufsize;
      /*Cập nhật thông tin cho lần đọc tiếp theo*/
      helloworld_dev_number = write_buf[0];
      /*In ra thông báo số byte dữ liệu nhận thành công từ user*/
      printk ("Driver: Has received %d byte(s) from user.\n",
      write_size);
      /*Trả về số byte nhận thành công cho hàm write()*/
      return write_size;
}


/*Khai báo giao diện open()*/
static int
helloworld_open(struct inode *inode, struct file *file) {
      int result = 0;
      unsigned int dev_minor = MINOR(inode->i_rdev);
      if (!atomic_dec_and_test(&helloworld_open_cnt)) {
      atomic_inc(&helloworld_open_cnt);
      printk(KERN_ERR DRVNAME ": Device with minor ID %d already in
      use\n", dev_minor);
      result = -EBUSY;
      goto out;
   }
out:
```

```
    return result;
}
```

*/*Khai báo giao diện hàm close*/*

```
static int
helloworld_close(struct inode * inode, struct file * file)
{
    smp_mb__before_atomic_inc();
    atomic_inc(&helloworld_open_cnt);
    return 0;
}
```

*/*Gán các giao diện hàm vào cấu trúc file operation*/*

```
struct file_operations helloworld_fops = {
    .ioctl        = helloworld_ioctl,
    .read         = helloworld_read,
    .write        = helloworld_write,
    .open         = helloworld_open,
    .release      = helloworld_close,
};
```

*/*Gán cấu trúc file operation vào inode, cập nhật tên thiết bị, hiện thị trong inode*/*

```
static struct miscdevice helloworld_dev = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name          = "helloworld",
        .fops          = &helloworld_fops,
};
```

*/*Khai báo và định nghĩa hàm thực hiện lúc cai đặt driver*/*

```
static int __init
helloworld_mod_init(void)
{
    return misc_register(&helloworld_dev);
}
```

*/*Khai báo định nghĩa hàm thực thi lúc tháo gỡ driver khỏi hệ thống*/*

```
static void __exit
helloworld_mod_exit(void)
{
    misc_deregister(&helloworld_dev);
}
```

```
/*Gán hàm định nghĩa vào macro init và exit module*/
module_init (helloworld_mod_init);
module_exit (helloworld_mod_exit);
/*Cập nhật thông tin của driver, quyền sở hữu, tên người viết chương  trình*/
MODULE_LICENSE("GPL");
MODULE_AUTHOR("coolwarmboy");
MODULE_DESCRIPTION("Character device for helloworld");
```

2. Chương trình application:

```
/*Chương trình application mang tên helloworld_app.c*/
/*Khai báo các thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/ioctl.h>
/*Định nghĩa các số định danh lệnh cho giao diện ioctl, các số định  nghĩa này giống như trong driver*/
#define HELLOWORLD_DEV_MAGIC  'B'
/*Số định danh lệnh dùng cho lệnh add*/
#define HELLOWORLD_ADD _IOWR(HELLOWORLD_DEV_MAGIC, 10, unsigned long)
/*Số định danh lệnh dùng cho lệnh sub*/
#define HELLOWORLD_SUB _IOWR(HELLOWORLD_DEV_MAGIC, 11, unsigned long)
/*Số định danh lệnh dùng cho lệnh mul*/
#define HELLOWORLD_MUL _IOWR(HELLOWORLD_DEV_MAGIC, 12, unsigned long)
/*Số định danh lệnh dùng cho lệnh div*/
#define HELLOWORLD_DIV _IOWR(HELLOWORLD_DEV_MAGIC, 13, unsigned long)
/*Chương trình con in ra hướng dẫn cho người dùng*/
void
print_usage()
{
   printf("helloworld_app       add|sub|mul|div|read|write       arg_1
   (arg_2)\n");
   exit(0);
}
/*Hàm main thực thi khi khi chương trình được gọi từ shell, main khai  báo theo dạng tùy chọn tham số*/
int
main(int argc, char **argv)
{     /*Biến lưu số file descripition của driver khi được mở*/
      int fd;
      /*Biến lưu mã lỗi trả về cho hàm main*/
```

```
int ret = 0;
/*Bộ đệm dữ liệu nhận được từ driver*/
char read_buf[1];
/*Bộ đệm dữ liệu cần truyền sang driver*/
char write_buf[1];
/*Bộ đệm dữ liệu 2 chiều dùng trong hàm giao diện ioctl()*/
unsigned long arg[3];
/*Đầu tiên mở driver cần tương tác, quy định chế độ truy xuất driver là đọc
và ghi*/
if ((fd = open("/dev/helloworld", O_RDWR)) < 0)
{
    /*Nếu có lỗi trong quá trình mở thiết bị, in ra thông báo cho người dùng*/
    printf("Error whilst opening /dev/helloworld\n");
    /*Trả về mã lỗi cho hàm main*/
    return fd;
}
/*Liệt kê các trường hợp có thể xảy ra khi giao tiếp với người dùng*/
if (argc == 4) {
        /*Trong trường hợp là các hàm toán học, dùng ioctl, kiểm tra các phép
        toán và thực hiện tính toán*/
        /*Trong trường hợp là phép toán cộng*/
        if (!strcmp(argv[1],"add")) {
            arg[0] = atoi(argv[2]);
            arg[1] = atoi(argv[3]);
            ioctl (fd, HELLOWORLD_ADD, arg);
            printf ("User: %ld + %ld = %ld\n", arg[0], arg[1],
            arg[2]);
        }
        /*Trong trường hợp là phép trừ*/
        else if (!strcmp(argv[1],"sub")) {
            arg[0] = atoi(argv[2]);
            arg[1] = atoi(argv[3]);
            ioctl (fd, HELLOWORLD_SUB, arg);
            printf ("User: %ld - %ld = %ld\n", arg[0], arg[1],
            arg[2]);
```

```
        }
        /*Trong trường hợp là phép nhân*/
        else if (!strcmp(argv[1],"mul")) {
                arg[0] = atoi(argv[2]);
                arg[1] = atoi(argv[3]);
                ioctl (fd, HELLOWORLD_MUL, arg);
                printf ("User: %ld x %ld = %ld\n", arg[0], arg[1],
                arg[2]);
        }
        /*Trong trường hợp là phép chia*/
        else if (!strcmp(argv[1],"div")) {
                arg[0] = atoi(argv[2]);
                arg[1] = atoi(argv[3]);
                ioctl (fd, HELLOWORLD_DIV, arg);
                printf ("User: %ld / %ld = %ld\n", arg[0], arg[1],
                arg[2]);
        }
        /*Trong trường hợp không có lệnh hỗ trợ, in ra hướng dẫn cho người
        dùng*/
        else {
                print_usage();
        }
}
/*Trong trường hợp là lệnh chép thông tin sang driver*/
else if (argc == 3) {
        if (!strcmp(argv[1],"write")) {
                write_buf[0] = atoi(argv[2]);
                printf ("User: has just sent number: %d\n",
                write_buf[0]);
                write(fd, write_buf, 1);
        } else {
                print_usage();
        }
}
/*Trong trường hợp lệnh đọc thông tin từ driver*/
else if (argc == 2) {
```

```
            if (!strcmp(argv[1],"read")) {
                  read(fd, read_buf, 1);
                  printf ("User: has just received number: %d\n",
                  read_buf[0]);
            } else {
                  print_usage();
            }
      }
      /*In ra hướng dẫn cho người dùng trong trường hợp không có lệnh  nào hỗ trợ*/
      else {
            print_usage();
      }
}
```

3. Thực thi chương trình.

Biên dịch chương trình driver:

- Khởi tạo biến môi trường thêm vào đường dẫn đến thư mục chứa công cụ hỗ trợ biên dịch;
- Tạo tập tin có tên Makefile trong thư mục chứa chương trình driver như sau:

```
export ARCH=arm
export CROSS_COMPILE=arm-none-linux-gnueabi-
obj-m += helloworld_dev.o
all:
      #Tùy thuộc vào nơi chứa cấu trúc mã nguồn kernel mà chọn đường
      #dẫn phù hợp cho lệnh make all
      make -C /home/arm/project/kernel/linux-2.6.30 M=$(PWD) modules
clean:
      #Tùy thuộc vào vị trí chứa cấu trúc mã nguồn kernel mà chọn đường
      #dẫn cho phù hợp với lệnh make clean
      make -C /home/arm/project/kernel/linux-2.6.30 M=$(PWD) clean
```

- Trỏ đến thư mục chứa driver thực thi lệnh shell: `make clean all` lúc này tập tin `helloworld_dev.c` sẽ được biên dịch thành `helloworld_dev.ko`;
- Chép tập tin `helloworld.ko` vào kit nhúng cần cài đặt;
- Cài đặt driver vào hệ thống linux bằng lệnh shell: `insmod helloworld_dev.ko`;

Biên dịch chương trình application:

Thực hiện theo các bước sau:

- Cài đặt biến môi trường trỏ vào nơi chứa công cụ hỗ trợ biên dịch, (Nếu đã làm trong khi biên dịch driver thì không cần làm bước này);
- Trong thư mục chứa mã nguồn chương trình helloworld_app.c thực thi lệnh shell:

  `arm-none-linux-gnueabi-gcc helloworld_app.c –o helloworld_app`

  Lúc này tập tin `helloworld_app` sẽ được tạo thành nếu chương trình `helloworld_app.c` không có lỗi.
- Chép tập tin `helloworld_app` đã biên dịch vào kit;
- Thực thi chương trình kiểm tra hoạt động của driver đã lập trình;

  Sau đây là kết quả thực thi chương trình trong từng trường hợp:

  ✓ *Trường hợp 1*: Kiểm tra hoạt động của phép toán add trong driver;

```
./helloworld_app add 100 50
Driver: Start ...
Driver: Calculating is complete
User: 100 + 50 = 150
```

  Chúng ta thấy driver đã hoạt động đúng chức năng của mình. Khi chương trình được gọi thực thi, nó gọi hàm ioctl thực hiện truyền hai tham số, kết quả được tính toán trong driver khi tính toán xong, nó gửi dữ liệu qua user. Cuối cùng, user xuất dữ liệu đã tính toán nhận được từ driver ra ngoài màn hình. Tương tự cho các phép toán khác như sau:

  ✓ *Trường hợp 2*: Kiểm tra hoạt động của phép toán sub trong driver;

```
./helloworld_app sub 100 50
Driver: Start ...
Driver: Calculating is complete
User: 100 - 50 = 50
```

  ✓ *Trường hợp 3*: Kiểm tra hoạt động của phép toán mul trong driver;

```
./helloworld_app mul 100 50
Driver: Start ...
Driver: Calculating is complete
User: 100 x 50 = 5000
```

  ✓ *Trường hợp 3*: Kiểm tra hoạt động của phép toán div trong driver;

```
./helloworld_app div 100 50
Driver: Start ...
```

```
Driver: Calculating is complete
User: 100 / 50 = 2
```

- ✓ Trường hợp 5: Kiểm tra đọc dữ liệu từ driver;

```
./helloworld_app read
Driver: Has sent 1 byte(s) to user.
User: has just receive number: 34
```

Khi chương trình application được gọi thực thi, với trường hợp lệnh read, hàm read được gọi thực thi yêu cầu dữ liệu từ driver, driver truyền một số lưu trong vùng nhớ của mình cho user application. User application nhận được thông tin, thông báo cho người dùng.

- ✓ Trường hợp 6: Kiểm tra ghi dữ liệu vào driver thông qua hàm write:

```
./helloworld_app write 100
User:Driver: Has received 1 byte(s) from user
 has just sent number: 100
```

Có một vấn đề nảy sinh là, mặc dù thông tin được chuyển qua driver thành công nhưng kết quả in ra xảy ra một lỗi nhỏ về thứ tự in của những dòng thông tin. Nguyên nhân là trong khi user in thông báo, driver đã nhận được dữ liệu và xuất ra thông báo của mình. Vì thế sau khi driver in xong thông báo, user mới tiếp tục thực hiện nhiệm vụ của mình. Như thế chúng ta cần phải chú ý đến vấn đề đồng bộ dữ liệu giữa driver và application để tránh trường hợp này xảy ra.

IV.Tổng kết:

Như vậy chúng ta đã hoàn thành công việc viết hoàn chỉnh một *driver* đơn giản dựa vào các bước mẫu trong bài học trước. Các bạn cũng đã hiểu nguyên lý hoạt động của hàm *main* có tham số, cách lựa chọn tham số hoạt động trong từng trường hợp cụ thể theo yêu cầu.

Với những thao tác trên, các bạn có thể tự mình viết những *dirver* đơn giản trong xử lý tính toán, xuất nhập thông báo, ... Thế nhưng, trong thực tế, *driver* không phải chỉ dùng trong việc truy xuất những ký tự thông báo. Nhiệm vụ chính của nó là điều khiển các thiết bị phần cứng thông qua các hàm giao tiếp với các cổng vào ra, truy xuất thanh ghi lệnh, dữ liệu, ...của thiết bị, thu thập thông tin lưu vào vùng nhớ đệm trong *driver* chờ chương trình trong *user* truy xuất. Để có thể giao tiếp với các thiết bị phần cứng thông qua các cổng vào ra, trong bài sau chúng ta sẽ nghiên cứu các hàm giao tiếp gpio do *linux* hỗ trợ sẵn.

BÀI 7

CÁC HÀM HỖ TRỢ GPIO

I. Tổng quan về GPIO:

GPIO, viết tắt của cụm từ General Purpose *Input/Output*, là một thư viện phần mềm điều khiển các cổng vào ra tích hợp trên vi điều khiển hay các ngoại vi IO liên kết với vi điều khiển đó. Hầu hết các vi điều khiển điều hỗ trợ thư viện này, giúp cho việc lập trình các cổng vào ra trở nên thuận tiện hơn. Các tập lệnh vào ra và điều khiển, cách quy định số chân, ... hầu hết tương tự nhau so với các loại vi khiển khác nhau. Điều này làm tăng tính linh hoạt, giảm thời gian xây dựng hệ thống.

Theo như quy định chuẩn, mỗi một chân IO trên vi điều khiển sẽ tương ứng với một số GPIO của thư viện này. Số GPIO được quy định như sau: Đối với vi điều khiển ARM9260, số cổng vào ra là 3x32 cổng, tương ứng với 3 ports, đó là các Port A, Port B, và Port C. Mỗi chân quy định trong GPIO theo quy luật sau:

BASEx32+PIN;

- Trong đó BASE là số cơ sở của Port. Port A có cơ sở là 1, Port B là 2, Port C là 3. PIN là số thứ tự của từng chân trong Port. Chân 0 có giá trị PIN là 32, 1 là 33, ... Ví dụ, chân thứ 2 của Port A có số GPIO là 33; Chân thứ 2 của Port B có số GPIO là 65, .. tương tự cho các chân còn lại trên vi điều khiển. Đối với các vi điều khiển khác có số Port lớn hơn ta chỉ việc tuân theo quy luật trên để tìm số GPIO phù hợp.

GPIO cho các loại vi điều khiển khác nhau điều có chung những tính chất:

- Mỗi chân trong GPIO điều có thể có hai chế độ *input* và *output*, tùy vào loại vi điều khiển mà GPIO đang sử dụng.
- Trong chế độ *input*, các chân GPIO có thể lập trình để trở thành nguồn ngắt hệ thống.
- Và nhiều chức năng khác nữa, trong quyển sách này chúng ta chỉ tìm hiểu những chức năng phổ biến nhất phục vụ giao tiếp với các chân IO trong các chương trình ứng dụng.

Một trong những thao tác đầu tiên để đưa GPIO hoạt động trong hệ thống là xác định GPIO cần dùng bằng hàm:

```
gpio_request(); //Yêu cầu truy xuất chân GPIO
```

Tiếp đến, chúng ta cấu hình chân GPIO là ngõ vào hay ngõ ra bằng hai hàm:

```
int gpio_direction_input(unsigned gpio);
int gpio_direction_output(unsigned gpio, int value);
```

Công việc cuối cùng là đưa dữ liệu đến chân GPIO, nếu là ngõ ra; hoặc đọc dữ liệu từ chân GPIO, nếu là ngõ vào; ta sử dụng hai hàm sau:

```
int gpio_get_value(unsigned gpio); //Đọc dữ liệu;
void gpio_set_value(unsigned gpio, int value); //Xuất dữ liệu;
```

Ngoài ra còn có nhiều hàm chức năng khác sẽ được trình bày trong mục sau.

***Có nhiều cách thao tác với gpio. Hoặc thao tác với giao diện thiết bị trong cấu trúc root file system (đây là những dirver đã được lập trình sẵn), việc điều khiển sẽ là thao tác với tập tin và thư mục (lập trình trong user application). Hoặc dùng trực tiếp những lệnh trong mã nguồn kernel, nghĩa là người sử dụng tạo riêng cho mình một driver sử dụng trực tiếp các hàm giao tiếp với gpio sau đó mới viết chương trình ứng dụng điều khiển IO theo những giao diện trong driver hỗ trợ. Nhằm mục đích thuận tiện cho viêc điều khiển IO, phần lập trình nhúng nâng cao chỉ trình bay cách thứ hai, điều khiển trực tiếp IO thông qua thư viện gpio.h trong kernel.*

II. Các hàm chính trong GPIO:

1. Hàm gpio_is_valid():

Cú pháp hàm như sau:

```
#include <linux/gpio.h>
int gpio_is_valid (int number);
```

Do thư viện *gpio* dùng chung cho nhiều loại vi điều khiển khác nhau, nên số lượng chân IO của từng loại cũng khác nhau. Cần thiết phải cung cấp một hàm kiểm tra sự tồn tại của chân trong hệ thống. Hàm `gpio_is_valid()` làm nhiệm vụ này. Hàm có tham số là `int number` là số chân *gpio* muốn kiểm tra. Hàm trả về giá trị 0 nếu số chân *gpio* cung cấp là hợp lệ, nghĩa là chân *gpio* này tồn tại trong hệ thống mà *gpio* đang hỗ trợ. Ngược lại, nếu chân *gpio* không hợp lệ, hàm trả về mã lỗi âm "`-EINVAL`".

Ví dụ, nếu muốn kiểm tra chân gpio 32 hợp lệ hay không, ta dùng đoạn mã lệnh sau:

```
/*Khai báo biến lưu mã lỗi trả về*/
int ret;
/*Gọi hàm kiểm tra gpio*/
ret = gpio_is_valid(32);
/*Kiểm tra mã lỗi trả về hệ thống*/
if (ret < 0) {
```

```
    /*Nếu xảy ra lỗi, in thông báo cho người dùng*/
    printk ("This gpio pin is not valid\n");
    /*Trả về mã lỗi cho hàm gọi*/
    return ret;
}
```

Công việc kiểm tra được thực hiện đầu tiên khi muốn thao tác với một chân nào đó trong gpio. Nếu hợp lệ các lệnh tiếp theo sẽ được thực hiện. Ngược lại, in ra thông báo cho người dùng và thoát ra khỏi driver.

2. Hàm gpio_request():

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
int gpio_request ( unsigned gpio, const char *lable);
```

Sau khi kiểm tra tính hợp lệ của chân *gpio*, công việc tiếp theo là khai báo sử dụng chân gpio đó. *Linux kernel* cung cấp cho chúng ta hàm `gpio_request()` để thực hiện công việc này. Hàm `gpio_request` có hai tham số. Tham số thứ nhất `unsigned gpio` là chân gpio muốn khai báo sử dụng; Tham số thứ hai `const char *lable` là tên muốn đặt cho *gpio*, phục vụ cho quá trình thao tác với chân gpio dễ dàng hơn. Tham số này có thể để trống bằng cách dùng hằng số rỗng NULL.

Sau đây là đoạn chương trình mẫu minh họa cách sử dụng hàm `gpio_request()`:

```
/*Những đoạn lệnh trước*/
...
/*Khai báo biến lưu về mã lỗi cho hàm gọi*/
int ret;
/*Gọi hàm gpio_request (), yêu cầu sử dụng chân gpio 32 với tên là "EXPL"*/
ret = gpio_request (32, "EXPL");
/*Kiểm tra mã lỗi trả về*/
if (ret) {
    /*Nếu xảy ra lỗi thì in ra thông bào cho người sử dụng*/
printk (KERN_WARNING "EXPL: unable to request gpio 32");
    /*Trả về mã lỗi cho hàm gọi*/
return ret;
}
/*Nếu không có lỗi, thực thi những đoạn lệnh khác*/
```

```
...
```

3. Hàm gpio_free():

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
void gpio_free (unsigned gpio);
```

Hàm `gpio_free()` có chức năng ngược lại với hàm `gpio_request()`. Hàm `gpio_free()` dùng giải phóng chân *gpio* nào đó không cần sử dụng cho hệ thống. Hàm này chỉ có một tham số trả về `unsigned gpio` là số *gpio* của chân muốn giải phóng. Hàm không có dữ liệu trả về.

Sau dây là đoạn chương trình ví dụ cách sử dụng hàm gpio_free:

```
/*Đoạn chương trình giải phóng chân gpio 32 ra khỏi driver*/
gpio_free(32);
```

4. Hàm gpio_direction_input():

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
int gpio_direction_input (unsigned gpio);
```

Hàm `gpio_direction_input` dùng để cài đặt chế độ giao tiếp cho chân *gpio* là *input*. Nghĩa là chân gpio này sẽ nhận dữ liệu từ bên ngoài lưu vào vùng nhớ đệm bên trong. Hàm `gpio_direction_input` có tham số `unsigned gpio` là chân *gpio* muốn cài đặt chế độ *input*. Hàm trả về giá trị 0 nếu quá trình cài đặt thành công. Ngược lại, sẽ trả về mã lỗi âm.

Sau đây là một ví dụ cho hàm `gpio_direction_input()`:

```
/*Hàm cài đặt chân gpio 32 ở chế độ ngõ vào*/
/*Khai báo biến lưu giá trị mã lỗi trả về cho hàm gọi*/
int ret;
/*Cài đặt chân 32 chế độ ngõ vào*/
ret = gpio_direction_input (32);
/*Kiểm tra lỗi thực thi */
if (ret) {
   /*Nếu có lỗi in ra thông báo cho người dùng*/
   printk (KERN_WARNING "Unable to set input mode");
   /*Trả về mã lỗi cho hàm gọi*/
   return ret;
```

```
}
...
```

5. Hàm gpio_direction_output():

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
int gpio_direction_output (unsigned gpio, int value);
```

Hàm `gpio_direction_output` dùng để cài đặt chế độ giao tiếp cho chân *gpio* là *output*. Chân *gpio* sẽ làm nhiệm vụ xuất dữ liệu bên trong chương trình ra ngoài. Hàm `gpio_direction_output()` có hai tham số. Tham số thứ nhất, `unsigned gpio`, là chân gpio muốn cài đặt. Tham số thứ hai, `int value`, là giá trị ban đầu của *gpio* khi nó là ngõ ra. Hàm trả về giá trị 0 nếu quá trình cài đặt thành công. Ngược lại, sẽ trả về mã lỗi âm.

Sau đây là một ví dụ cho hàm `gpio_direction_output()`:

```
/*Hàm cài đặt chân gpio 32 ở chế độ ngõ ra, giá trị ban đầu là 1*/
/*Khai báo biến lưu giá trị mã lỗi trả về cho hàm gọi*/
int ret;
/*Cài đặt chân 32 chế độ ngõ vào*/
ret = gpio_direction_output (32, 1);
/*Kiểm tra lỗi thực thi */
if (ret) {
   /*Nếu có lỗi in ra thông báo cho người dùng*/
   printk (KERN_WARNING "Unable to set gpio 32 into output mode");
   /*Trả về mã lỗi cho hàm gọi*/
   return ret;
}
...
```

6. Hàm gpio_get_value():

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
int gpio_get_value (unsigned gpio);
```

Khi chân `unsigned gpio` là ngõ vào, hàm `gpio_get_value()` sẽ lấy giá trị tín hiệu của chân này. Hàm có giá trị trả về dạng *int*, bằng 0 nếu ngõ vào mức thấp, khác 0 nếu ngõ vào mức cao.

Sau đây là một ví dụ đọc vào giá trị chân *gpio* 32, kiểm tra và in thông tin ra màn hình:

```
...
if (gpio_get_value(32)) {
   printk ("The input is high value\n");
} else {
   printk ("The input is low value\n");
}
...
```

7. Hàm gpio_set_value():

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
void gpio_set_value (unsigned gpio, int value);
```

Ngược lại với hàm `gpio_get_value()`, hàm `gpio_set_value()` có chức năng xuất dữ liệu cho một chân `unsigned gpio` ngõ ra. Với dữ liệu chứa trong tham số `int value`. Bằng 0 nếu muốn xuất ra mức thấp, bằng 1 nếu muốn xuất ra mức cao. Hàm `gpio_set_value()` không có dữ liệu trả về.

Sau đây là một ví dụ xuất mức cao ra chân gpio 32:

```
/*Các lệnh khởi tạo gpio 32 là ngõ ra*/
...
/*Xuất mức cao ra chân gpio 32 */
gpio_set_value (32, 1);
/*Các lệnh xử lý tiếp theo*/
...
```

8. Hàm gpio_to_irq():

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
int gpio_to_irq (unsigned gpio);
```

Đôi khi chúng ta muốn cài đặt chế độ ngắt cho một chân nào đó trong *gpio* dùng để thu nhận thông tin đồng bộ từ phần cứng. Hàm `gpio_to_irq()` thực hiện chức năng này. Hàm có tham số `unsigned gpio` là chân *gpio* muốn cài đặt chế độ ngắt. Hàm có giá trị trả về là số IRQ nếu quá trình cài đặt chế độ ngắt thành công. Ngược lại sẽ trả về mã lỗi âm.

Số IRQ được sử dụng cho hàm `request_irq()` khởi tạo ngắt cho hệ thống, hàm sẽ gán số IRQ với hàm xử lý ngắt. Hàm này sẽ thực hiện những thao tác do người lập trình quy định khi xuất hiện ngắt từ tín hiệu ngắt có số định danh ngắt là IRQ. Bên cạnh đó hàm `request_irq()` còn quy định chế độ ngắt cho chân *gpio* là ngắt theo cạnh hay ngắt theo mức, ...

Sau đây là đoạn chương trình ví dụ khởi tạo ngắt từ chân gpio.

```
/*Đoạn chương trình khởi tạo ngắt cho chân gpio 70, PC6*/
/*Khai báo biến lưu mã lỗi trả về hàm gọi*/
int ret;
/*Yêu cầu chân gpio, với định danh là IRQ*/
 ret = gpio_request (70, "IRQ");
/*Kiểm tra mã lỗi trả về*/
   if (ret) {
/*Thông báo cho người lập trình có lỗi xảy ra*/
        printk (KERN_ALERT "Unable to request PC6\n");
   }
/*Cài đặt chế độ kéo lên cho chân gpio*/
   at91_set_GPIO_periph (70, 1);
/*Cài đặt chân gpio là input*/
   at91_set_gpio_input  (70, 1);
 /*Cài đặt chân gpio có chế độ chống lỗi*/
   at91_set_deglitch    (70, 1);
/*Cài đặt gpio thành nguồn ngắt, lưu về số định danh ngắt irq*/
 irq = gpio_to_irq (70);   /* Get IRQ number */
```

```
/*Thông báo chân gpio đã khởi tạo ngắt */
   printk (KERN_ALERT "IRQ = %d\n", irq);
/*Khai báo ngắt chân gpio cho hệ thống*/
   ret = request_irq (irq, gpio_irq_handler,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "MyIRQ", NULL);
/*Kiểm tra mã lỗi trả về khi khai báo ngắt*/
   if (ret)  {
/*Thông báo cho người sử dụng số định danh ngắt không còn trống*/
    printk (KERN_ALERT "IRQ %d is not free\n", irq);
/*Trả về mã lỗi cho hàm gọi */
        return ret;
   }
```

Như vậy để tránh lỗi trong quá trình khởi tạo ngắt cho chân gpio, trước tiên chúng ta phải cài đặt những thông số cần thiết cho gpio đó, chẳng hạn như chân gpio phải hợp lệ và ở chế độ ngõ vào.

III. Kết luận:

Trên đây, chúng ta đã sử dụng được những hàm điều khiển các chân gpio ngoại vi để thực hiện chức năng cụ thể của yêu cầu ứng dụng. Kết hợp với những kiến thức trong những bài trước: Giao diện điều khiển liên kết *user space* với *kernel space*, các hàm trì hoãn thời gian trong *user space*, các kỹ thuật lập trình C, ... chúng ta có thể viết được hầu hết tất cả những ứng dụng có liên quan đến các cổng vào ra: chẳng hạn như điều khiển LED, điều khiển LCD, ...

gpio không chỉ điều khiển các cổng vào ra trên vi điều khiển, theo yêu cầu trong thực tế, đa số các kit nhúng, số cổng vào ra rất hạn chế, đòi hỏi phải có các chân io mở rộng từ những linh kiện phụ trợ khác, vì thế *gpio* còn hỗ trợ thêm các hàm điều khiển các chân io mở rộng. Vấn đề này sẽ được đề cập trong những tài liệu khác không thuộc phạm vi của giáo trình này.

Trước khi di vào ứng dụng các hàm thao tác với *gpio* trong điều khiển LED đơn, LCD, ... bài tiếp theo sẽ tìm hiểu thêm về cách trì hoãn thời gian trong *kernel*, với cách trì hoãn thời gian này, chúng ta không cần dùng đến các hàm trì hoãn khác trong user space vì lý do yêu cầu đồng bộ hóa phần cứng hệ thống, hay một số trường hợp chúng ta muốn sử dụng delay trong driver gắn liền với những thao tác điều khiển.

BÀI 8

THAO TÁC THỜI GIAN TRONG KERNEL

I. Sơ lược về thời gian trong kernel:

Thời gian trong hệ điều hành linux nói riêng và các hệ điều hành khác nói chung điều rất quan trọng. Trong hệ điều hành, những hoạt động điều dựa theo sự tác động mang tính chất chu kỳ. Chẳng hạn như hoạt động chia khe thời gian thực thi, chuyển qua lại giữa các tiến trình với nhau, đồng bộ hóa hoạt động giữa các thiết bị phần cứng có thời gian truy xuất không giống nhau, và nhiều hoạt động quan trọng khác nữa.

Khi làm việc với hệ thống linux, chúng ta cần phân biệt hai khái niệm thời gian: Thời gian tuyệt đối và thời gian tương đối. Thời gian tuyệt đối được hiểu như thời gian thực của hệ thống, là các thông tin như ngày-tháng-năm-giờ-phút-giây và các đơn vị thời gian khác nhỏ hơn, phục vụ cho người sử dụng. Thời gian tương đối được hiểu như những khoảng thời gian được cập nhật không cố định, mang tính chất chu kỳ, mốc thời gian không cố định và thông thường không biết trước. Ví dụ thời gian tuyệt đối là thời điểm trong một ngày, có mốc thời gian tính từ ngày 1 tháng 1 năm 1970 quy định trong các thiết bị thời gian thực. Thời gian tương đối là khoảng thời gian tính từ thời điểm xảy ra một sự kiện nào đó, chẳng hạn định thời tác động một khoảng thời gian sau khi hệ thống có lỗi hoặc cập nhật thông số hiện tại của hệ thống sau mỗi một thời khoảng cố định.

Để có thể kiểm tra, quản lý và thao tác với thời gian một cách chính xác, hệ điều hành phải dựa vào thiết bị thời gian được tích hợp trên hầu hết các vi xử lý đó là *timer*. *Timer* được sử dụng bởi hệ điều hành được gọi là *timer* hệ thống, hệ điều hành cài đặt các thông số thích hợp cho *timer*, quy định khoảng thời gian sinh ra ngắt, khoảng thời gian giữa hai lần xảy ra ngắt liên tiếp được gọi bởi thuật ngữ *tick*, giá trị của *tick* do người sử dụng *driver* quy định. Khi xảy ra ngắt, hệ điều hành linux sẽ cập nhật giá trị của biến *jiffies*, chuyển tiến trình, cập nhật thời gian trong ngày, ...

Trong phần lập trình *user application*, chúng ta đã tìm hiểu những hàm thao tác với thời gian *user space*. Đây là những hàm được hỗ trợ sẵn bởi hệ điều hành, nghĩa là chúng được lập trình để hoạt động ổn định không gây ảnh hưởng đến các tiến trình khác chạy đồng thời. Trong *kernel*, những hàm thao tác với thời gian hoạt động bên ngoài tầm kiểm soát của hệ điều hành, sử dụng trực tiếp tài nguyên phần cứng của hệ thống, cụ thể là thời gian hoạt động của vi xử lý trung tâm trong vi điều khiển. Vì thế nếu sử dụng không phù

hợp thì hệ điều hành sẽ hoạt động không ổn định, xảy ra những lỗi về thời gian thực trong khi thực hiện tác vụ. Nhưng thời gian thực hiện của các hàm này sẽ nhanh hơn, vì không qua trung gian là hệ điều hành, phù hợp với yêu cầu điều khiển của *driver* là nhanh chóng và chính xác.

Trong bài này, đầu tiên chúng ta sẽ tìm hiểu nguyên tắc quản lý thời gian trong *kernel*, sau đó là các hàm tương tác, xử lý thời gian thực, cách sử dụng định thời trong *timer*, và cuối cùng là một số kỹ thuật trì hoãn thời gian trong *kernel*.

II. Đơn vị thời gian trong *kernel*:

Để quản lý chính xác thời gian, hệ điều hành sử dụng bộ định thởi *timer* tích hợp sẵn trên vi điều khiển mà nó hoạt động. Bộ định thời (*timer*) được đặt cho một giá trị cố định sao cho có thể sinh ra ngắt theo chu kỳ cho trước. Khoảng thời gian của một chu kỳ ngắt được gọi là *tick*. Trong một giây có khoảng *N ticks* được hoàn thành. Hay nói cách khác, tốc độ tick là *N Hz*. Giá trị *N* được định nghĩa bởi người sử dụng trước khi biên dịch *kernel*. Thông thường một số hệ điều hành có giá trị mặc định là *N = 100* và đây cũng là giá trị mặc định của hệ điều hành chạy trên kit *KM9260*.

Giá trị của HZ được định nghĩa như sau:

```
# define USER_HZ 100 /* User interfaces are in "ticks" */
```

và

```
# define HZ 100 /*Internal kernel timer frequency*/
```

trong tập tin `\arch\arm\include\asm\param.h`. Chúng ta thấy, giá trị mặc định của HZ là 100. Có nghĩa là *timer* hệ thống được cài đặt sao cho trong một giây có khoảng 100 lần ngắt xảy ra. Hay nói cách khác, chu kỳ ngắt là 10 ms. Chúng ta cũng chú ý, trong tập tin có hai tham số cần sữa chữa một lúc đó là *USER_HZ* và *HZ*. Để thuận tiện cho việc chuyển đổi qua lại thời gian giữa *kernel* và *user* thì giá trị của chúng phải giống nhau.

Giá trị *HZ* thay đổi phải phù hợp với ứng dụng mà hệ điều hành đang phục vụ. Làm thế nào để làm được điều này.

Chẳng hạn, giá trị của *HZ* ảnh hưởng rất nhiều đến độ phân giải của những hàm định thời ngắt. Với giá trị *HZ = 100*, thời gian lập trình định thời ngắt tối thiểu là *10ms*. Với giá trị *HZ=1000*, thời gian lập trình định thời ngắt tối thiểu là *1ms*. Điều nàycó nghĩa là giá trị của *HZ* càng lớn càng tốt. Liệu thực sự có phải như thế?

Khi tăng giá trị của *HZ*, điều có những ưu và nhược điểm. Về ưu điểm, tăng giá trị *HZ* làm độ phân giải của thời gian *kernel* tăng lên, như thế một số ứng dụng có liên quan đến

thời gian cũng chính xác hơn ,...Về nhược điểm, khi tăng giá trị của *HZ*, thì vi điều khiển thực hiện ngắt nhiều hơn, tiêu tốn thời gian cho việc lưu lưu ngăn xếp, khởi tạo ngắt, ... nhiều hơn, ... do vậy tùy từng ứng dụng cụ thể mà chúng ta thay đổi giá trị *HZ* cho phù hợp để hệ thống hoạt động tối ưu.

III. jiffies:

jiffies là một biến toàn cục được định nghĩa trong thư viện `linux/jiffies.h` để lưu số lượng *ticks* đạt được kể từ khi hệ thống bắt đầu khởi động. Khi xảy ra ngắt timer hệ thống, kernel tiến hành tăng giá trị của jiffies lên 1 đơn vị. Như vậy nếu như có *HZ* ticks trong một giây thì *jiffies* sẽ tăng trong một giây là *HZ* đơn vị. Nếu như chúng ta đọc được giá trị *jiffies* hiện tại là *N*, thì thời gian kể từ khi hệ thống khởi động là *N* ticks hay *N/HZ* giây. Đôi khi chúng ta muốn đổi giá trị *jiffies* sang giây và ngược lại ta chỉ việc chia hay nhân giá trị *jiffies* cho(với) *HZ.*

Trong *kernel, jiffies* được lưu trữ dưới dạng số nhị phân 32 bits. Là 32 bits có trong số thấp trong tổng số 64 bits của biến *jiffies_64*. Với chu kỳ *tick* là 10ms thì sau một khoảng thời gian 5.85 tỷ năm đối với biến *jiffies_64* và *1.36* năm đối với biến *jiffes* mới có thể bị tràn. Xác suất để biến *jiffies_64* bị tràn là cực kỳ nhỏ và *jiffies* là rất nhỏ. Thế nhưng vẫn có thể xảy ra đối với những ứng dụng đòi hỏi độ tin cậy rất cao. Nếu cần có thể kiểm tra nếu yêu cầu chính xác cao.

Khi thao tac với *jiffies, kernel* hỗ trợ cho chúng ta các hàm so sánh thời gian sau, tất cả các hàm này điều được định nghĩa trong thư viện `linux/jiffies.h`:

Đầu tiên là hàm *get_jiffies_64(),* với giá trị jiffies thì chúng ta có thể đọc trực tiếp theo tên của nó, thế nhưng *jiffies_64* không thể đọc trực tiếp mà phải thông qua hàm riêng vì giá trị của *jiffies_64* được chứa trong số 64 bits. Hàm không có tham số, giá trị trả về của hàm là số có 64 bits.

Cuối cùng là các hàm so sánh thời gian theo giá trị của *jiffies*. Các hàm này được định nghĩa trong thư viện `linux/jiffies.h` như sau:

```
#define time_after(unknown,known)(long)(known)-(long)(unknown)<0)
#define time_before(unknown,known)((long)(unknown)-(long)(known)<0)
#define time_after_eq(unknown,known)((long)(unknown)-(long)(known)>= 0)
#define time_before_eq(unknown,known)((long)(known)-(long)(unknown)>= 0)
```

các hàm này trả về giá trị kiểu `boolean`, tùy theo tên hàm và tham số của hàm. Hàm `time_after(unknown, known)` trả về giá trị đúng nếu `unknown > known`, tương tự cho

các hàm khác. Ứng dụng của các hàm này khi chúng ta muốn so sánh hai khoảng thời gian với nhau để thực thi một tác vụ nào đó, chẳng hạn ứng dụng trong trì hoãn thời gian như trong đoạn chương trình sau:

```
/*Đoạn chương trình trì hoãn thời gian 1s dùng jiffies*/
/*Khai báo biến lưu thời điểm cuối cùng  muốn so sánh*/
unsigned long timeout = jiffies + HZ; //Trì hoãn 1s
/*Kiểm tra xem giá trị timeout có bị tràn hay không*/
if (time_after(jiffies, timeout)) {
   printk("This timeout is overflow\n");
   return -1;
}
/*Thực hiện trì hoãn thời gian nếu không bị tràn*/
if (time_before(jiffies, timeout)) {
   /*Do nothing loop to delay*/
}
```

IV. Thời gian thực trong kernel:

Trong phần lập trình ứng dụng *user*, chúng ta đã tìm hiểu kỹ về các lệnh xử lý thời gian thực trong thư viện `<time.h>`. Trong *kernel* cũng có những hàm được xây dựng sẵn thao tác với thời gian thực nằm trong thư viện `<linux/time.h>`. Sự khác biệt giữa hai thư viện này là vai trò của chúng trong hệ thống. `<time.h>` chứa trong lớp *user*, giao tiếp với *kernel*, tương tự như các hàm giao diện trung gian giao tiếp giữa người dùng với thời gian thực trong *kernel*, vì thế thư viện chứa những hàm chủ yếu phục vụ cho người dùng; Đối với thư viện `<linux/time.h>` hoạt động trong lớp *kernel*, chứa những hàm được lập trình sẵn phục vụ chủ yếu cho hệ thống *kernel*, giúp người lập trình *driver* quản lý thời gian thực tiện lợi hơn, ví dụ như các hàm xử lý thời gian ngắt, định thời, so sánh thời gian, ... điểm đặc biệt là thời gian thực trong *kernel* chủ yếu quản lý dưới dạng số giây tuyệt đối, không theo dạng ngày tháng năm như trong *user*.

Các kiểu cấu trúc thời gian, ý nghĩa các tham số trong những hàm thời gian đa phần tương tự như trong thư viện `<time.h>` trong *user application* nên chúng sẽ được trình bày sơ lược trong khi giải thích.

1. Các kiểu, cấu trúc thời gian:

a. ***Cấu trúc timespec:*** cấu trúc này được định nghĩa như sau:

```
struct timespec {
   __kernel_time_t  tv_sec;          /* seconds */
   long             tv_nsec;         /* nanoseconds */
};
```

Trong đó, `tv_sec` dùng lưu thông tin của giây; `tv_nsec` dùng lưu thông tin nano giây của giây hiện tại.

b. Cấu trúc timeval: Cấu trúc này được định nghĩa như sau:

```
struct timeval {
   __kernel_time_t       tv_sec;          /* seconds */
   __kernel_suseconds_t  tv_usec;   /* microseconds */
};
```

Trong đó, `tv_sec` dùng lưu thông tin về số giây; `tv_usec` dùng lưu thông tin về micro giây của giây hiện tại.

c. Cấu trúc timezone: Cấu trúc này được định nghĩa như sau:

```
struct timezone {
   int   tz_minuteswest; /* minutes west of Greenwich */
   int   tz_dsttime;     /* type of dst correction */
};
```

Trong đó, `tz_minuteswst` là số phút chênh lệch múi giờ về phía Tây của Greenwich; `tz_dsttime` là tham số điều chỉnh chênh lệch thời gian theo mùa;

2. Các hàm so sánh thời gian:

a. timespec_equal: Hàm được định nghĩa như sau:

```
static inline int timespec_equal(const struct timespec *a,
                                 const struct timespec *b)
{
   return (a->tv_sec == b->tv_sec) && (a->tv_nsec == b->tv_nsec);
}
```

Nhiệm vụ của hàm là so sánh 2 con trỏ cấu trúc thời gian kiểu `timespec`, `const struct timespec *a` và `const struct timespec *b`. So sánh từng thành phần trong cấu trúc, `tv_sec` và `tv_nsec`, nếu hai thành phần này bằng nhau thì hai cấu trúc thời gian này bằng nhau. Khi đó hàm trả về giá trị *true*, ngược lại trả về giá trị *false*;

b. timespec_compare: Hàm được định nghĩa như sau:

```
static inline int timespec_compare(const struct timespec *lhs, const struct timespec *rhs)
{
   if (lhs->tv_sec < rhs->tv_sec)
        return -1;
   if (lhs->tv_sec > rhs->tv_sec)
        return 1;
   return lhs->tv_nsec - rhs->tv_nsec;
}
```

Nhiệm vụ của hàm là so sánh hai cấu trúc thời gian kiểu `timespec`, `const struc timespec *lhs` và `const struct timespec *rhs`. Kết quả là một trong 3 trường hợp sau:

- Nếu `lhs < rhs` thì trả về giá trị nhỏ hơn 0;
- Nếu `lhs = rhs` thì trả về giá trị bằng 0;
- Nếu `lhs > rhs` thì trả về giá trị lớn hơn 0;

c. timeval_compare: Hàm được định nghĩa như sau:

```
static inline int timeval_compare(const struct timeval *lhs, const struct timeval *rhs)
{
   if (lhs->tv_sec < rhs->tv_sec)
        return -1;
   if (lhs->tv_sec > rhs->tv_sec)
        return 1;
   return lhs->tv_usec - rhs->tv_usec;
}
```

Nhiệm vụ của hàm là so sánh hai cấu trúc thời gian kiểu `timeval`, `const struct timeval *lhs` và `const struc timeval *rhs`. Kết quả trả về là một trong 3 trường hợp:

- Nếu `lhs < rhs` thì trả về giá trị nhỏ hơn 0;
- Nếu `lhs = rhs` thì trả về giá trị bằng 0;
- Nếu `lhs > rhs` thì trả về giá trị lớn hơn 0;

3. Các phép toán thao tác trên thời gian:

a. mktime: Hàm được định nghĩa như sau:

```
extern unsigned long mktime(
const unsigned int year, const unsigned int mon,
const unsigned int day, const unsigned int hour,
const unsigned int min, const unsigned int sec);
```

Nhiệm vụ của hàm là chuyển các thông tin thời gian dạng ngày tháng năm ngày giờ phút giây thành thông tin thời gian dạng giây tính từ thời điểm epoch.

b. set_normalized_timespec: Hàm được định nghĩa như sau:

```
extern void set_normalized_timespec(struct timespec *ts, time_t sec,
long nsec);
```

Sau khi chuyển các thông tin ngày tháng năm ... thành số giây tính từ thời điểm epoch bằng cách sử dụng hàm `mktime`, thông tin trả về chưa phải là một cấu trúc thời gian chuẩn có thể xử lý được trong `kernel`. Đề biến thành cấu trúc thời gian chuẩn ta sử dụng hàm `set_normalize_timespec` để chuyển số giây dang unsigned long thành cấu trúc thời gian `timespec`.

Hàm có 3 tham số. Tham số thứ nhất, `struct timespec *ts`, là con trỏ trỏ đến cấu trúc `timespec` được định nghĩa trước đó lưu giá trị thông tin thời gian trả về sau khi chuyển đổi xong; Tham số thứ hai, `time_t sec`, là số giây muốn chuyển đổi sang cấu trúc `timespec` (được lưu vào trường `tv_sec`); Tham số thứ ba, `long nsec`, là số `nano` giây của giây của thời điểm muốn chuyển đổi.

c. timespec_add_safe: Hàm được định nghĩa như sau:

```
extern struct timespec timespec_add_safe(
const struct timespec lhs,
const struct timespec rhs);
```

Nhiệm vụ của hàm là cộng hai cấu trúc thời gian kiểu `timespec`, `const struct timespec lhs` và `const struct timespec rhs`. Kết quả trả về dạng `timespec` là tổng của hai giá trị thời gian nếu không bị tràn, nếu bị tràn thì hàm sẽ trả về giá trị lớn nhất có thể có của kiểu `timespec`.

d. timespec_sub: Hàm được định nghĩa như sau:

```
static inline struct timespec timespec_sub(struct timespec
lhs,struct timespec rhs)
{
   struct timespec ts_delta;
   set_normalized_timespec(&ts_delta, lhs.tv_sec - rhs.tv_sec,
                  lhs.tv_nsec - rhs.tv_nsec);
   return ts_delta;
}
```

Nhiệm vụ của hàm này là trừ hai cấu trúc thời gian kiểu `timespec`, `struct timespec lhs` và `struct timespec rhs`. Hiệu của hai cấu trúc này được trả về dưới dạng `timespec`. Thông thường hàm dùng để tính toán khoảng thời gian giữa hai thời điểm.

e. timespec_add_ns: Hàm được định nghĩa như sau:

```
static __always_inline void timespec_add_ns(struct timespec *a, u64
ns)
{
   a->tv_sec+=__iter_div_u64_rem(a->tv_nsec+ns, NSEC_PER_SEC, &ns);
   a->tv_nsec = ns;
}
```

Nhiệm vụ của hàm này là cộng thêm một khoảng thời gian tính bằng nano giây vào cấu trúc `timespec` được đưa vào hàm. Hàm có hai tham số. Tham số thứ nhất, `struct timespec *a`, là cấu trúc `timespec` muốn cộng thêm. Tham số thứ hai, `u64 ns`, là số nano giây muốn cộng thêm vào.

4. Các hàm truy xuất thời gian:

a. do_gettimeofday: Hàm được định nghĩa như sau:

```
extern void do_gettimeofday(struct timeval *tv);
```

Nhiệm vụ của hàm là lấy về thông tin thời gian hiện tại của hệ thống. Thông tin thời gian hiện tại được trả về cấu trúc dạng `struct timeval *tv` trong tham số của hàm.

b. do_settimeofday: Hàm được định nghĩa như sau:

```
extern int do_settimeofday(struct timespec *tv);
```

Nhiệm vụ của hàm là cài đặt thông tin thời gian hiện tại cho hệ thống dựa vào cấu trúc thời gian dạng `timespec` chứa trong tham số `struct timespec *tv` của hàm.

c. do_sys_settimeofday: Hàm được định nghĩa như sau:

```
extern int do_sys_settimeofday(struct timespec *tv, struct timezone
*tz);
```

Nhiệm vụ của hàm là cài đặt thông tin thời gian hiện tại của hệ thống có tính đến sự chênh lệch thời gian về múi giờ dựa vào hai tham số trong hàm. Tham số thứ nhất, `struct timespec *tv`, là thông tin thời gian muốn cập nhật. Tham số thứ hai, `sruct timezone *tz`, là cấu trúc lưu thông tin chệnh lệch múi giờ muốn cập nhật.

d. getboottime: Hàm được định nghĩa như sau:

```
extern void getboottime (struct timespec *ts);
```

Nhiệm vụ của hàm là lấy về tổng thời gian từ khi hệ thống khởi động đến thời điểm hiện tại. Thông tin thời gian được lưu về cấu trúc kiểu `timespec`, `srtuc timespec`.

5. Các hàm chuyển đổi thời gian:

a. timespec_to_ns: Hàm được định nghĩa như sau:

```
static inline s64 timespec_to_ns(const struct timespec *ts)
{
   return ((s64) ts->tv_sec * NSEC_PER_SEC) + ts->tv_nsec;
}
```

Nhiệm vụ của hàm là chuyển cấu trúc thời gian dạng `timespec`, `const struct timespec *ts`, thành số nano giây lưu vào biến 64 bits làm giá trị trả về cho hàm.

b. timeval_to_ns: Hàm được định nghĩa như sau:

```
static inline s64 timeval_to_ns(const struct timeval *tv)
{
   return ((s64) tv->tv_sec * NSEC_PER_SEC) +
        tv->tv_usec * NSEC_PER_USEC;
}
```

Nhiệm vụ của hàm là chuyển đổi cấu trúc thời gian dạng `timeval`, `const struct timeval *tv`, thành số nano giây lưu vào biến `64 bits` làm giá trị trả về cho hàm.

c. ns_to_timespec: Hàm được dịnh nghĩa như sau

```
extern struct timespec ns_to_timespec(const s64 nsec);
```

Nhiệm vụ của hàm là chuyển thông tin thời gian dưới dạng nano giây, `const s64 nsec`, thành thông tin thời gian dạng `timespec` làm giá trị trả về cho hàm.

d. ns_to_timeval: Hàm được định nghĩa như sau:

```
extern struct timeval ns_to_timeval(const s64 nsec);
```

Nhiệm vụ của hàm là chuyển thông tin thời gian dạng nano giây, `const s64 nsec`, thành thông tin thời gian dạng `timeval` làm giá trị trả về cho hàm.

V. Timer và ngắt dùng timer:

1. Khái quát về timer trong kernel:

Timer là một định nghĩa quen thuộc trong hầu hết tất cả các hệ thống khác nhau. Trong *linux kernel, timer* được hiểu là trì hoãn thời gian động để gọi thực thi một hàm nào đó được lập trình trước. Nghĩa là thời điểm bắt đầu trì hoãn không cố định, thông thường được tính từ lúc cài đặt khởi động *timer* trong hệ thống, khoảng thời gian trì hoãn do người lập trình quy định, khi hết thời gian trì hoãn, *timer* sinh ra một ngắt. *Kernel* tạm hoãn hoạt động hiện tại của mình để thực thi hàm ngắt được lập trình trước đó.

Để đưa một *timer* vào hoạt động, chúng ta cần phải thực hiện nhiều thao tác. Đầu tiên phải khởi tạo *timer* vào hệ thống *linux*. Tiếp theo, cài đặt khoảng thời gian mong muốn trì hoãn. Cài đặt hàm thực thi ngắt khi thời gian trì hoãn kết thúc. Khi xảy ra hoạt động ngắt, *timer* sẽ bị vô hiệu hóa. Vì thế, nếu muốn *timer* hoạt động theo chu kỳ cố định chúng ta phải khởi tạo lại *timer* ngay trong chương trình thực thi ngắt. Số lượng *timer* khởi tạo trong *kernel* là không giới hạn, vì đây là một timer mềm không phải là *timer* vật lý, có giới hạn là dung lượng vùng nhớ cho phép. Chúng ta sẽ trình bày cụ thể những bước trên trong phần sau.

2. Các bước sử dụng timer:

***Lưu ý: Khi sử dụng các hàm trong timer chúng ta phải thêm thư viện `<linux/timer.h>` ở đầu chương trình.*

a. Bước 1: Khai báo biến cấu trúc `timer_list` để lưu *timer* khi khởi tạo

Cấu trúc `timer_list` được *linux kernel* định nghĩa trong thư viện `<linux/timer.h>` như sau:

```
struct timer_list {
struct list_head entry;
unsigned long expires;
void (*function) (unsigned long);
unsigned long data;
struct tvec_t_base_s *base;
```

```
}
```

Ta khai báo timer bằng dòng lệnh sau:

```
/*Khai báo một timer có tên là my_timer*/
struct timer_list my_timer;
```

b. Bước 2: Khai báo và định nghĩa hàm thực thi ngắt

Hàm thực thi ngắt có dạng như sau:

```
void my_timer_function (unsigned long data) {
/*Các thao tác do người lập trình driver quy định*/
...
/*Nếu cần thiết có thể khởi tạo lại timer trong phần cuối chương trình*/
}
```

c. Bước 3: Khởi tạo các tham số cho cấu trúc *timer*

Công việc tiếp theo là yêu cầu *kernel* cung cấp một vùng nhớ cho *timer* hoat động, chúng ta sử dụng câu lệnh:

```
/*Khởi tạo timer vừa khai báo my_timer*/
init_timer (&my_timer);
```

Sau khi *kernel* dành cho *timer* một vùng nhớ, *timer* vẫn còn trống chưa được gán những thông số cần thiết, công việc này của người lập trình *driver*:

```
/*Khởi tạo giá trị khoảng thời gian muốn trì hoãn, đơn vị tính bằng tick*/
my_timer.expires = jiffies  + delay;
/*Gán tham số cho hàm thực thi ngắt, nếu không có tham số ta có thể gán một giá trị bất kỳ*/
my_timer.data = 0;
/*Gán con trỏ hàm xử lý ngắt vào timer*/
my_timer.function = my_function;
```

Trong các tham số trên, chúng ta cần chú ý những tham số sau đây:

- `my_timer.expires` đây là giá trị thời gian tương lai có đơn vị là *tick*, tại mọi thời điểm, *timer* so sánh với số *jiffies*. Nếu số *jiffies* bằng với số `my_timer.expires` thì ngắt xảy ra.

- `my_timer.data` là tham số chúng ta muốn đưa vào hàm thực thi ngắt, đôi khi chúng ta muốn khởi tạo các thông số ban đầu cho hàm thực thi ngắt, kế thừa những thông tin đã xử lý từ trước hay từ người dùng.
- `my_timer.function` là trường chứa con trỏ của hàm phục vụ ngắt đã được khởi tạo và định nghĩa trước đó.

d. Bước 4: Kích hoạt timer hoạt động trong *kernel*:

Chúng ta thực thi lệnh sau:

```
add_timer (&my_timer);
```

Tham số của hàm add_timer là con trỏ của timer được khởi tạo và gán các tham số cần thiết.

Khi *timer* được kích hoạt, hàm thực thi ngắt hoạt động, nó sẽ bị vô hiệu hóa trong chu kỳ tiếp theo. Muốn *timer* tiếp tục hoạt động, chúng ta phải kích hoạt lại bằng câu lệnh sau:

```
/*Kích hoạt timer hoạt động lại cho chu kỳ ngắt tiếp theo*/
mod_timer (&my_timer, jiffies + new_delay);
```

Nếu muốn xóa timer khỏi hệ thống chúng ta sùng lệnh sau:

```
/*Xóa timer khỏi hệ thống*/
del_timer (&my_timer);
```

Tuy nhiên lệnh này chỉ thực hiện thành công khi *timer* không còn hoạt động, nghĩa là khi *timer* còn đang chờ chu kỳ ngắt tiếp theo dùng lệnh `del_timer()` sẽ không hiệu quả. Muốn khắc phục lỗi này, chúng ta phải dùng `lệnh del_timer_sync()`, lệnh này sẽ chờ cho đến khi *timer* hoàn thành chu kỳ ngắt gần nhất mới xóa *timer* đó.

3. Ví dụ:

Đoạn chương trình sau sẽ định thời xuất thông tin ra màn hình hiển thị vời chu kỳ là 1s. Mỗi lần xuất sẽ thay đổi thông tin, thông báo những lần xuất thông tin là khác nhau;

```
/*Khai báo biến cục bộ lưu giá trị muốn xuất ra màn hình, giá trị ban đầu bằng 0*/
int counter = 0;
/*Khai báo biến timer phục vụ ngắt*/
struct time_list my_timer;
/*Khai báo định nghĩa hàm phục vụ ngắt*/
void timer_isr (unsigned long data) {
/*In thông báo cho người dùng*/
```

```
printk ("Driver: Hello, the counter's value is %d\n", counter++);
```

*/*Cài đặt lại giá trị timer cho lần hoạt động tiếp theo, cài đặt chu kỳ 1 giây*/*

```
mod_timer (&my_timer, jiffies + HZ);
}
```

*/*Thực hiện khởi tạo timer trong khi cài đặt driver vào hệ thống, trong hàm init()*/*

```
static int
__init my_driver_init (void) {
```

*/*Các lệnh khởi tạo khác*/*

...

*/*Khởi tạo timer hoạt động ngắt*/*

*/*Khởi tạo timer đã được khai báo*/*

```
init_timer (&my_timer);
```

*/*Cài đặt các thông số cho timer*/*

```
my_timer.expires =  jiffies + HZ; //Khởi tạo trì hoãn ban đầu là 1s;
my_timer.data = 0; //Dữ liệu truyền cho hàm ngắt là 0;
my_timer.function = my_function; //Gán hàm thực thi ngắt cho timer.
```

*/*Kích hoạt timer hoạt động trong hệ thống*/*

```
add_timer (&my_timer);
...
}
```

VI.Trì hoãn thời gian trong kernel:

Trì hoãn thời gian là một trong những vấn đề quan trọng trong lập trình *driver* cũng như trong lập trình *application*. Trong phần trước chúng ta đã tìm hiểu những lệnh trì hoãn thời gian từ khoảng thời gian nhỏ tính theo nano giây đến khoảng thời gian lớn hơn tính bằng giây. Tùy thuộc vào yêu cầu chính xác cao hay thấp mà chúng ta áp dụng kỹ thuật trì hoãn thời gian cho phù hợp. *Kernel* cũng hỗ trợ các kỹ thuật trì hoãn khác nhau tùy theo yêu cầu mà áp dụng kỹ thuật nào phù hợp nhất sao cho không ảnh hưởng đến hoạt động của hệ thống.

1. Vòng lặp vô tận:

Kỹ thuật trì hoãn thời gian đầu tiên là vòng lặp *busy loop*. Đây là cách trì hoãn thời gian cổ điển và đơn giản nhất, áp dụng cho tất cả các hệ thống. Kỹ thuật trì hoãn này có dạng như sau:

```
/*Khai báo thời điểm tương lai  muốn thực hiện trì hoãn*/
unsigned long timeout = jiffies + HZ/10; //Trì hoãn 10ms;
/*Thực hiện vòng lặp vô tận while () trì hoãn thời gian*/
while (time_before(jiffies, timeout))
   ;
```

Với cách trì hoãn thời gian trên, chúng ta dùng biến *jiffies* để so sánh với thời điểm tương lai làm điều kiện cho lệnh `while ()` thực hiện vòng lặp. Như vậy chúng ta chỉ có thể trì hoãn một khoảng thời gian đúng bằng một số nguyên lần của *tick*. Hơn nữa, khác với lớp *user*, vòng lặp trong *kernel* không được chia tiến trình thực hiện. Vì thế vòng lặp vô tận trong kernel sẽ chiếm hết thời gian làm việc của CPU và như thế các hoạt động khác sẽ không được thực thi, hệ thống sẽ bị ngưng lại tạm thời. Điều này rất nguy hiểm cho các ứng dụng đòi hỏi độ tin cậy cao về thời gian thực. Cách trì hoãn thời gian này rất hiếm khi được sử dụng trong những hệ thống lớn.

Để giải quyết vấn đề này, người ta dùng kỹ thuật chia tiến trình trong lúc thực hiện trì hoãn như sau:

```
while (time_before(jiffies, timeout))
    schedule(); //Hàm này chứa trong thư viện <linux/sched.h>
```

Trong khi *jiffies* vẫn thõa mãn điều kiện nhỏ hơn thời điểm đặt trước, *kernel* sẽ chia thời gian thực hiện công việc khác trong hệ thống. Cho đến khi vòng lặp được thoát, những lệnh tiếp theo sẽ tiếp tục thực thi.

Chúng ta cũng có thể áp dụng những lệnh so sánh thời gian thực trong phần trước để thực hiện trì hoãn thời gian với độ chính xác cao hơn.

2. Trì hoãn thời gian bằng những hàm hỗ trợ sẵn:

Linux kernel cũng cung cấp cho chúng ta những hàm thực hiện trì hoãn thời gian với độ chính xác cao, thích hợp cho những khoảng thời gian nhỏ. Đó là những hàm:

- `void udelay(unsigned long usec);` Hàm dùng để trì hoãn thời gian có độ phân giải micrô giây;

- `void ndelay(unsigned long nsec);` Hàm trì hoãn thời gian có độ phân giải nanô giây;
- `void mdelay(unsigned long msec);` Hàm trì hoãn thời gian có độ phân giải mili giây;

***Các hàm trì hoãn thời gian này chỉ thích hợp cho những khoảng thời gian nhỏ hơn 1 tick trong kernel. Nếu lơn hơn sẽ làm ảnh hưởng đến hoạt động của cả hệ thống vì bản chất đây vẫn là những vòng lặp vô tận, chiếm thời gian hoạt động của CPU.*

3. Trì hoãn thời gian bằng hàm schedule_timeout ():

Kỹ thuật này khác với hai kỹ thuật trên, dùng hàm `schedule_timeout()` sẽ làm cho chương trình driver dừng lại tại thời điểm khai báo, rơi vào trạng thái ngủ trong suốt thời gian trì hoãn. Thời gian trì hoãn do người lập trình cài đặt. Để sử dụng hàm này, ta tiến hành các bước sau:

*/*Cài đặt chương trình driver vào trạng thái ngủ*/*

```
set_current_state (TASK_INTERRUPTIBLE);
```

*/*Cài đặt thời gian cho tín hiệu đánh thức chương trình*/*

```
schedule_timeout (unsigned long time_interval);
```

***Trong đó time_interval là khoảng thời gian tính bằng tick muốn cài đặt tín hiệu đánh thức chương trình đang trong trạng thái ngủ.*

VII. Kết luận:

Trong bài này chúng ta đã tìm hiểu rõ về cách quản lý thời gian trong *kernel*, thế nào là *jiffies, HZ,* ... vai trò ý nghĩa của chúng trong duy trì quản thời gian thực cũng như trong trì hoãn thời gian.

Chúng ta cũng đã tìm hiểu các hàm thao tác với thời gian thực trong *kernel*, với những hàm này chúng ta có thể xây dựng các ứng dụng có liên quan đến thời gian thực trong hệ thống.

Có nhiều các khác nhau để thực hiện trì hoãn trong *kernel* tương tự như trong *user*. Nhưng chúng ta phải biết cách chọn phương pháp phù hợp để không làm ảnh hưởng đến hoạt động của toàn hệ thống.

Đến đây chúng ta có thể bước vào các bài thực hành, viết *driver* và ứng dụng cho một số phần cứng cơ bản trong những bài sau. Hoàn thành những bài này sẽ giúp cho chúng ta có được cái nhìn thự tế hơn về lập trình hệ thống nhúng.