

# Thao tác file trong Linux

## Mở file

Để mở một file, ta dùng system call `open()` có prototype như sau (theo man page `man7.org`):

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

#include <sys/stat.h>

#include <fcntl.h>

int open (const char *pathname, int flags);

int open (const char *pathname, int flags, mode_t mode);
```

System call `open()` mở file có tên với đường dẫn đầy đủ "`pathname`" hoặc tạo ra và mở file đó nếu nó chưa tồn tại .

Nếu system call `open()` gọi thành công, nó sẽ trả về một số nguyên là số mô tả file của file đó, số này được sử dụng để tham chiếu đến file đó cho các system call sau này. Nếu system call thất bại, nó sẽ trả về -1 và ghi giá trị lỗi vào biến toàn cục `errno`.

Đối số "`flags`" là một bitmask dùng để chỉ chế độ truy cập vào file. Đối số "`mode`" cũng là một bitmask để chỉ định quyền truy cập (permission) vào file nếu `open()` được dùng để tạo ra một file mới (với cờ truyền vào là `O_CREAT`). Nếu `open()` chỉ mở một file có sẵn thì có thể bỏ qua đối số "`mode`".

Chúng ta sẽ xem nhanh một ví dụ dưới đây để hiểu rõ hơn về cách sử dụng system call `open()` và các đối số truyền vào của nó:

```
/* Mở một file đã tồn tại tên là hello.txt để đọc */
fd = open("file1.txt", O_RDONLY);
if (fd == -1)
    perror("Open fail");

/* Mở một file hoặc tạo mới nếu nó chưa tồn tại (O_CREAT) để đọc và ghi (O_RDWR),
sau khi mở xóa hết nội dung cũ của nó (O_TRUNC), quyền truy cập (đối số mode) là
đọc và ghi cho owner (S_IRUSR | S_IWUSR) */
fd = open("file2.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
if (fd == -1)
    perror("Open fail");

/* Mở một file hoặc tạo file mới để ghi vào (O_WRONLY) tiếp từ cuối file (O_APPEND) */
fd = open("file3.txt", O_WRONLY | O_CREAT | O_TRUNC | O_APPEND | S_IRUSR | S_IWUSR);
if (fd == -1)
    perror("Open fail");
```

### Cờ (flags) của system call `open()`

Đối số `flags` truyền vào `open()` là một bitwise OR của nhiều cờ với nhau. Như ví dụ trên, cờ truyền vào là bitwise OR của 3 đối số `O_RDWR | O_CREAT | O_TRUNC`. Cờ truyền vào phải chứa một trong 3 giá trị `O_RDONLY` (chỉ đọc), `O_WRONLY` (chỉ ghi), và `O_RDWR` (cả đọc và ghi).

Ngoài 3 chế độ truy cập trên, flags có thể cộng thêm các giá trị cờ khác phục vụ cho việc điều khiển file. Bảng dưới đây liệt kê các cờ có thể truyền vào khi gọi system call `open()`:

Flag	Mục đích	SUS
<code>O_RDONLY</code>	Mở file để chỉ đọc	v3
<code>O_WRONLY</code>	Mở file để chỉ ghi	v3
<code>O_RDWR</code>	Mở file để đọc và ghi	v3
<code>O_CLOEXEC</code>	Thiết lập cờ close-on-exec	v4
<code>O_CREAT</code>	Tạo file nếu nó chưa tồn tại	v3
<code>O_DIRECT</code>	Trao đổi dữ liệu trực tiếp giữa user space và file trên ổ cứng, không qua kernel buffer cache (buffer cache sẽ nói ở bài sau)	
<code>O_DIRECTORY</code>	Trả về fail nếu đối số “pathname” không phải đường dẫn	v4
<code>O_EXCL</code>	Dùng với <code>O_CREAT</code> : chỉ tạo một file mới	v3
<code>O_LARGEFILE</code>	Hỗ trợ mở một file lớn	
<code>O_NOATIME</code>	Không cập nhật lần mở file trước khi đọc file	
<code>O_NOCTTY</code>	Không để file “pathname” trở thành một terminal điều khiển	v3
<code>O_NOFOLLOW</code>	Không tham chiếu ngược các liên kết mềm	v4
<code>O_TRUNC</code>	Xóa nội dung file hiện tại nếu có của file để có độ dài là 0	v3
<code>O_APPEND</code>	Ghi tiếp nối từ địa chỉ cuối cùng của file	v3
<code>O_ASYNC</code>	Tạo ra một signal khi file sẵn sàng đọc hoặc ghi	
<code>O_DSYNC</code>	Cung cấp việc bảo toàn dữ liệu đã được đồng bộ	v3
<code>O_NONBLOCK</code>	Mở file ở chế độ nonblock, nghĩa là <code>read()</code> hoặc <code>write()</code> sẽ return ngay nếu file chưa sẵn sàng.	v3
<code>O_SYNC</code>	Đồng bộ dữ liệu được ghi vào file	v3

## System call `creat()`

Quay lại thời điểm Unix mới được triển khai, system call `open()` chỉ có 2 đối số (pathname và flag) và chỉ có chức năng mở file đã tồn tại mà không thể tạo được một file mới. Thay vào đó, system call `creat()` được dùng để tạo ra một file mới, có prototype như sau:

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

System call `creat()` tạo ra và mở một file với tên gọi là “pathname”, hoặc nếu nó đã tồn tại thì mở file đó và cắt hết nội dung file để có độ dài là 0. Giống như `open()`, `creat()` cũng trả về một mô tả file nếu thành công. Thực tế, việc gọi `creat()` cũng tương tự như gọi `open()` với các đối số cụ thể như sau:

```
fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Do `open()` hỗ trợ nhiều chức năng điều khiển hơn là chỉ mở file, `creat()` hiện nay ít được sử dụng, thường thì chúng chỉ tồn tại trên các chương trình ngày xưa.

## Đóng file

Để đóng một mô tả file, chúng ta dùng system call `close()`. Mô tả file này cũng sẽ được giải phóng và có thể cấp phát lại sau này bởi tiến trình. Khi một tiến trình bị kết thúc, tất cả các mô tả file đang mở sẽ tự động được kernel thu hồi lại.

System call `close()` có prototype như sau:

```
#include <unistd.h>

int close(int fd);
```

System call `close()` trả về giá trị 0 nếu thành công, 1 nếu xảy ra lỗi. Các lỗi của `close()` có thể là đóng một mô tả file chưa được mở, hoặc đóng một mô tả file hai lần.

Khi lập trình Linux, chúng ta nên tạo thành thói quen luôn đóng các mô tả file sau khi sử dụng. Việc này sẽ làm cho code của chúng ta dễ đọc hơn. Đồng thời giải phóng mô tả file cho tiến trình sử dụng về sau vì số lượng mô tả file của tiến trình có giới hạn, việc này đặc biệt quan trọng với những daemon hoặc tiến trình tồn tại mãi trong hệ thống ví dụ như một máy server.

## Đọc file

Sau khi mở file bằng `open()`, chúng ta dùng system call `read()` để đọc nội dung của file. Prototype của `read()` như sau:

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
```

System call `read()` đọc một số lượng lớn nhất là “count” byte từ một file có mô tả file là `fd` và lưu vào một vùng nhớ có địa chỉ `buffer`. Buffer này phải có độ dài ít nhất là “count” byte.

Nếu thành công, `read()` trả về số byte lớn nhất được đọc, hoặc 0 nếu đọc được ký tự end-of-file (EOF), hoặc -1 nếu thất bại. Số byte đọc được từ `read()` có thể nhỏ hơn đối số `count`, trường hợp này có thể xảy ra khi `read()` gặp ký tự end-of-file (nghĩa là file có độ dài nhỏ hơn `count`) với file text hoặc ký tự new line nếu đọc từ terminal (ví dụ `stdin`).

System call `read()` được tạo ra để đọc bất kỳ loại dữ liệu như text, binary hay struct ở dạng binary từ bất kỳ file `fd` nào như regular file, socket, pipe.... Đó là lý do buffer có kiểu dữ liệu void. Vì vậy, nếu bạn muốn đọc dữ liệu vào một string, `read()` sẽ không tự động thêm ký tự `\null` vào cuối buffer để tạo thành string hoàn chỉnh, do đó bạn phải tự làm việc này sau khi sử dụng `read()`.

Xét ví dụ minh họa sau, chương trình đọc dữ liệu từ bàn phím (`STDIN_FILENO`) và in kết quả đọc ra màn hình:

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

#define MAX_READ    16
int main(void)
{
    char buf[MAX_READ] = "";
    int numRead;
    int i;

    numRead = read(STDIN_FILENO, buf, MAX_READ);
    if (numRead == -1)
```

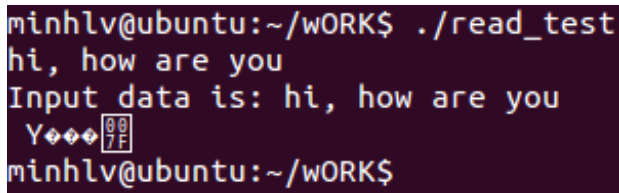
```

    perror("read error");
    printf("Input data is: %s\n", buf);

    return 0;
}

```

Bạn chạy chương trình và nhập vào một string, ví dụ như sau:



```

minhlv@ubuntu:~/WORK$ ./read_test
hi, how are you
Input data is: hi, how are you
Y♦♦♦
minhlv@ubuntu:~/WORK$

```

Tại sao chuỗi buf đọc ra lại như vậy? Nguyên nhân là read() chỉ đọc các ký tự từ bàn phím cho đến khi chúng ta nhấn Enter (16 ký tự), nó không tự động chừa ra và thêm ký tự \null vào cuối string. Để khắc phục lỗi này, chúng ta sẽ thêm ký tự \null vào cuối string sau khi đọc xong:

```

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

#define MAX_READ    16
int main(void)
{
    char buf[MAX_READ + 1] = ""; /*Length MAX_READ+1 để chừa cho ký tự \null*/
    int numRead;
    int i;

    numRead = read(STDIN_FILENO, buf, MAX_READ);
    if (numRead == -1)
        perror("read error");

    buf[numRead] = '\0' /*Thêm ký tự \null vào cuối để tạo thành string*/
    printf("Input data is: %s, strlen:%d\n", buf, (int)strlen(buf));

    return 0;
}

```

## Ghi file

Để ghi vào một file, ta dùng system call write() có prototype như sau:

```

#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);

```

Đối số truyền vào của write() tương tự như read(): buffer là địa chỉ một vùng nhớ lưu dữ liệu được ghi vào file; count là số byte được ghi vào file từ buffer và fd là một số mô tả file trỏ đến file mà chúng ta muốn ghi vào.

Giá trị trả về của write() là số byte thực tế được ghi vào file (ssize\_t là một kiểu dữ liệu trong Linux được khai báo kiểu số nguyên) hoặc -1 nếu việc ghi bị lỗi. Số byte dữ liệu được ghi vào thực tế có thể nhỏ hơn số count, gọi là ghi một phần (partial write). Với các file thông thường (regular file), write() đảm bảo ghi toàn bộ vào file, trừ khi có lỗi xảy ra. Nhưng với các file đặc biệt như socket, bạn nên dùng một vòng lặp để đảm bảo việc ghi một nội dung vào file được hoàn tất. Ví dụ như đoạn code dưới đây:

```
ssize_t ret;

/*Ghi một nội dung độ dài len có địa chỉ bắt đầu từ vùng nhớ buf vào file fd*/

while (len != 0 && (ret = write (fd, buf, len)) != 0)
{
    if (ret == -1)
    {
        if (errno == EINTR)
            continue;

        perror ("write" );
        break;
    }
    len -= ret;
    buf += ret;
}
```

## Kết luận

System call `open()` không chỉ có khả năng mở một file có sẵn mà còn có thể tạo ra một file mới. Ngoài ra, bạn cũng nên nhớ các cờ hữu ích của `open()` phục vụ cho việc đọc, ghi vào file mà chúng ta sẽ học ở bài sau. Bạn hãy tạo thành thói quen luôn sử dụng cặp system call `open()` và `close()` để tiết kiệm tài nguyên mô tả file cho tiến trình của mình.

Bài học cũng đã giới thiệu 2 system call cơ bản là `read()` và `write()` để đọc, ghi nội dung một file. Bài sau sẽ giới thiệu về đồng bộ file trong hệ thống Linux, qua đó hiểu rõ hơn bản chất của việc đọc/ghi một file Linux từ ứng dụng user space.