# I/O Multiplexing - select()

Mô hình universal file I/O đề cập ở các bài trước đều chỉ thao tác với một mô tả file. Về nguyên lý hoạt động, mỗi system call file I/O khi được gọi sẽ block cho đến khi dữ liệu được gửi. Ví dụ, khi ta muốn đọc một file từ một pipe bằng system call read(), read() có thể block chương trình nếu tại thời điểm đó không có dữ liệu ở trong pipe; và write() sẽ block chương trình nếu như không có đủ bộ nhớ của pipe để lưu trữ dữ liệu được ghi vào.

Nếu ứng dụng của chúng ta chỉ cần làm việc với một mô tả file, và thời gian block khi đọc/ghi file không lớn thì chỉ cần dùng các file I/O system call read() và write() cơ bản. Tuy nhiên, trong thực tế chương trình có thể phải theo dõi nhiều mô tả file (ví dụ, một server cần phải phục vụ nhiều mô tả file từ các client socket). Nếu read() đang chờ đọc dữ liệu từ một mô tả file của một client chưa có dữ liệu, chương trình sẽ tiếp tục block trong khi các client khác đã gửi dữ liệu vào mô tả file và đang chờ được đọc.

Về góc độ lập trình, ta có thể đề xuất hai cách giải quyết vấn đề này như sau:

- **Nonblocking I/O**: Có thể thiết lập một mô tả file ở chế độ nonblocking bằng cách bật cờ O_NONBLOCK của open() khi mở file. Khi đó các system call file I/O nếu thấy file đó chưa ở trạng thái sẵn sàng, nó sẽ return ngay lập tức và trả giá trị lỗi vào biến *errno*. Khi đó chúng ta sẽ biết là mô tả file đó chưa sẵn sàng để đọc/ghi, và sẽ làm các việc khác rồi quay lại thăm dò (polling) mô tả file đó sau. Tuy nhiên, phương pháp này cũng có hạn chế: nếu tần suất thăm dò file quá thưa thì thời gian trễ để chương trình thao tác file sẽ dài; còn nếu tần suất thăm dò quá dày sẽ gây lãng phí tài nguyên CPU của hệ thống.

- **Tạo một thread mới để thăm dò mô tả file** Tiến trình cha tạo ra một thread chỉ làm nhiệm vụ thăm dò mô tả file cần theo dõi, nó sẽ block cho đến khi mô tả file đó sẵn sàng. Phương pháp này khá chân phương và dễ làm, nhưng nếu chúng ta làm việc trên nhiều mô tả file thì sẽ phải tạo ra số thread bằng số mô tả file. Việc này sẽ gây tiêu tốn tài nguyên và làm chương trình trở nên phức tạp.

Vì sự hạn chế của hai phương pháp trên, chúng ta cần một giải pháp khác để kiểm tra tính sẵn sàng của nhiều mô tả file cùng một lúc. Và một trong số các phương pháp đơn giản và được dùng phổ biến trong các hệ thống nhúng Linux là I/O multiplexing.

Nguyên lý hoạt động của multiplexing I/O là: chương trình sẽ theo dõi nhiều mô tả file cùng một lúc và block cho đến khi một trong các mô tả file cần theo dõi sẵn sàng hoặc hết thời gian timeout thiết lập. Trong bài này, chúng ta sẽ phân tích và sử dụng hai system call I/O multiplexing có khả năng tương đương được dùng phổ biến là select() và poll().

# Giải pháp select()

Select() system call có prototype như sau:

```
#include <sys/time.h> /* For portability */
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

Select() hoạt động bằng cách gom các mô tả file mà bạn muốn theo dõi vào một tập hợp (fd_set) và sẽ block chương trình cho đến khi một hoặc nhiều file trong tập hợp sẵn sàng. Select() trả về một số nguyên là số mô tả file đã sẵn sàng, hoặc 0 nếu xảy ra timeout, hoặc -1 nếu xảy ra lỗi.

Trong prototype trên, các đối số nfds, readfds, writefds và exceptfds được dùng để chỉ các mô tả file cần giám sát; timeout là thời gian thiết lập giới hạn mà select() sẽ block chương trình.

# Tập hợp mô tả file (file descriptor sets)

Các mô tả file cần giám sát được chia thành 3 loại và truyền vào select() bằng cách lưu vào các tập hợp readfds, writefds và exceptfds:

- **Readfds**: tập hợp mô tả file được kiểm tra trạng thái sẵn sàng đọc
- **Writefds**: tập hợp mô tả file được kiểm tra tra trạng thái sẵn sàng ghi
- **Exceptfds**: tập hợp mô tả file được kiểm tra nếu có điều kiện ngoại lệ (exceptional condition) xảy ra.

Đối số đầu tiên của select() là nfds được tính bằng số mô tả file cao nhất cần giám sát trong cả 3 tập hợp cộng thêm 1. Giá trị nfds được thêm vào nhằm giúp tăng hiệu năng cho select() hơn do kernel sẽ không cần kiểm tra các mô tả file lớn hơn nfds.

Thuật ngữ exceptional condition thường bị hiểu nhầm là có lỗi xảy ra với mô tả file. Thực ra ngoại lệ ở đây thường là sự thay đổi trạng thái của pseudoterminal hoặc out-of-band data ở socket. Trong giới hạn bài học về multiplexing I/O, chúng ta sẽ không đi quá sâu vào việc giải thích rõ hai thuật ngữ này, vì trong thực tế exceptfd không được sử dụng nhiều.

Ví dụ, nếu bạn muốn kiểm tra xem một hoặc nhiều mô tả file sẵn sàng đọc mà không cần block chương trình không thì chỉ cần truyền các mô tả file đó vào tập hợp readfds. Nếu select() return 0, nghĩa là hết thời gian timeout mà không có file nào sẵn sàng đọc. Nếu select() return lớn hơn 0 thì tại thời điểm đó có file sẵn sàng đọ và các file đó sẽ được lưu trong tập hợp readfds.

Các tập hợp readfds, writefds và exceptfds có kiểu dữ liệu fd_set là một kiểu dữ liệu trong Linux triển khai ở dạng bit mask. Tuy nhiên trong lập trình, chúng ta sẽ không thay đổi trực tiếp các tập hợp mô tả file này mà sẽ dùng các macro sau:

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

void FD_ZERO(fd_set *fdset);

void FD_SET(int fd, fd_set *fdset);

void FD_CLR(int fd, fd_set *fdset);

int FD_ISSET(int fd, fd_set *fdset);
```

- FD_ZERO: khởi tạo tập hợp mô tả file có địa chỉ fdset về trạng thái rỗng. Trước khi truyền các file cần theo dõi vào các tập hợp readfds, writefds, exeptfdsd, chúng ta phải dùng macro này để khởi tạo các tập hợp mô tả file này.
- FD_SET: thêm mô tả file fd vào tập hợp mô tả file có địa chỉ fdset
- FD_CLR: gỡ bỏ mô tả file fd khỏi tập hợp mô tả file có địa chỉ fdset
- FD_ISSET: Kiểm tra xem mô tả file cần theo dõi có nằm trong tập hợp fdset không.

# timeout

Đối số timeout là thời gian tối đa select() sẽ block chương trình nếu chưa có mô tả file nào sẵn sàng. Nếu timeout được thiết lập NULL, select() sẽ block vô hạn (trừ khi có signal xảy ra).

Timeout có kiểu dữ liệu cấu trúc như sau:

```
struct timeval {
   long tv_sec; /* Seconds */
   long tv_usec; /* Microseconds */
};
```

Nếu cả 2 trường của timeout được thiết lập 0, select() sẽ không block chương trình; nó sẽ xem mô tả file nào trong các tập hợp sẵn sàng và return luôn. Timeout được thiết lập 0 trong trường hợp ta muốn thăm dò các mô tả file sẵn sàng chưa ngay tại thời điểm tức thời đó.

# Ví dụ

Sau khi nắm được cách hoạt động và prototype của select(), chúng ta sẽ xét một ví dụ dưới đây sử dụng select() dưới đây để hiểu rõ hơn.

Trong ví dụ này, bạn sẽ viết một chương trình dùng select() để theo dõi mô tả file stdin. Dùng sysem call select() để block cho đến khi stdin có dữ liệu để đọc (nghĩa là cho đến khi bạn nhập từ bàn phím). Thời gian timeout là 5 giây, nếu sau 5 giây bạn không gõ vào bàn phím, select() sẽ return 0 và in lỗi ra màn hình. Nếu có dữ liệu nhập từ bàn phím thì chương trình sẽ in ra dữ liệu mà bạn đã nhập. Chương trình này chỉ theo dõi 1 mô tả file nên chưa phát huy được hết khả năng của select(), vì mục đích của nó là viết một chương trình đơn giản để hướng dẫn bạn cách sử dụng select().

Source code của chương trình như sau:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define TIMEOUT 5         /* Timeout cho select() là 5s */

#define BUF_LEN 1024     /* buffer length */

int main (void)
{
 struct timeval tv;
 fd_set readfds;
 int ret = -1;

 /* Khởi tạo tập hợp readfds và thêm mô tả file stdin vào readfds*/
 FD_ZERO(&readfds);

 FD_SET(STDIN_FILENO, &readfds);

 /* Thiết lập timeout */
 tv.tv_sec = TIMEOUT;
 tv.tv_usec = 0;

 /*Block stdin đến khi stdin sẵn sàng đọc*/

 /*Tập hợp mô tả file writefds và exceptfds truyền vào NULL*/
 ret = select (STDIN_FILENO + 1, &readfds, NULL, NULL, &tv);

 if (-1 == ret)
 {
  perror("Select error.\n" );
  return 1;
 }
```

```
 else if (0 == ret)
 {
  printf("Timeout after %d seconds.\n" , TIMEOUT);
  return 0;
 }

 /*
 Kiểm tra xem stdin có nằm trong readfds không
 Nếu FD_ISSET trả về 1, stdin nằm trong readfds và stdin sẵn sàng đọc
 */

 if (FD_ISSET(STDIN_FILENO, &readfds))
 {
  char buf[BUF_LEN+1];
  int len = -1;

  /* Đọc dữ liệu từ mô tả file của stdin */

  len = read(STDIN_FILENO, buf, BUF_LEN);

  if (-1 == len)
  {
   perror("Read fd error.\n" );
   return 1;
  }

  if(len)
  {
   buf[len] = '\0' ; /*manual vì read() không thêm ký tự null vào cuối string*/
   printf ("read: %s\n" , buf);
  }
  return 0;
 }
 return 1;
}
```

Bây giờ bạn compile và chạy chương trình nhé.

Chạy chương trình, sau đó không gõ gì cả, chương trình sẽ thoát sau 5 giây timeout:

```
minhlv@thevngeek:~/work/code$ gcc -o select select.c
minhlv@thevngeek:~/work/code$ ./select
Timeout after 5 seconds.
minhlv@thevngeek:~/work/code$
```

Bây giờ bạn chạy chương trình, và gõ đoạn text bất kỳ vào bàn phím:

```
minhlv@thevngeek:~/work/code$ ./select
hello
read: hello

minhlv@thevngeek:~/work/code$
```

# Kết luận

Multiplexing I/O nói chung và system call select() được sử dụng rất rất nhiều trong lập trình Linux. Vậy nên việc hiểu và biết cách sử dụng multiplexing I/O là điều kiện bắt buộc của tất cả các kỹ sư Linux. Trong bài sau, chúng ta sẽ tìm hiểu một kỹ thuật multiplexing I/O khác là poll().