Network Programming: I/O Multiplexing

Li-Hsing Yen
NYCU

Ver. 1.0.0

I/O Multiplexing: *select* and *poll*

- Introduction
- I/O models
- *select* function
- Rewrite *str_cli* function
- Supporting batch input with *shutdown* function
- Rewrite concurrent TCP echo server with *select*
- *pselect* function: avoiding signal loss in race condition
- *poll* function: polling more specific conditions than *select*
- Rewrite concurrent TCP echo server with *poll*

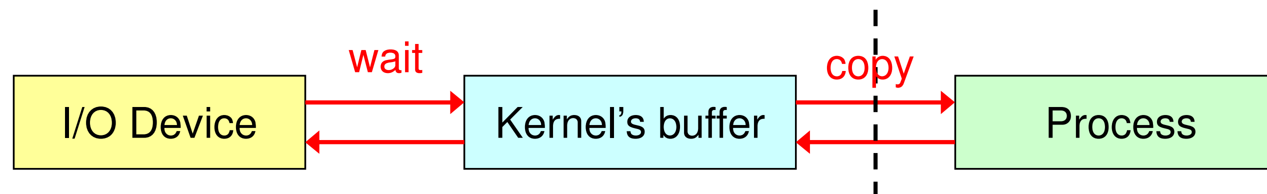
Some materials in these slides are taken from Prof. Ying-Dar Lin with his permission of usage

Introduction

- I/O multiplexing: to be notified, by kernel, if one or more I/O conditions are ready.
- Scenarios in networking applications:
  - a client handling multiple descriptors (stdio/socket)
  - a client handling multiple sockets
  - a TCP server handling a listening socket and its connected sockets (兩件事都自己來)
  - a server handling both TCP and UDP
  - a server handling multiple services and protocols

I/O動作如何進行?

- 一般process無法直接對I/O裝置下命令，必須透過system call請求kernel幫忙進行I/O動作
  - 如果動作無法立即完成，則process會被block
- kernel會對每個I/O裝置維護一個buffer

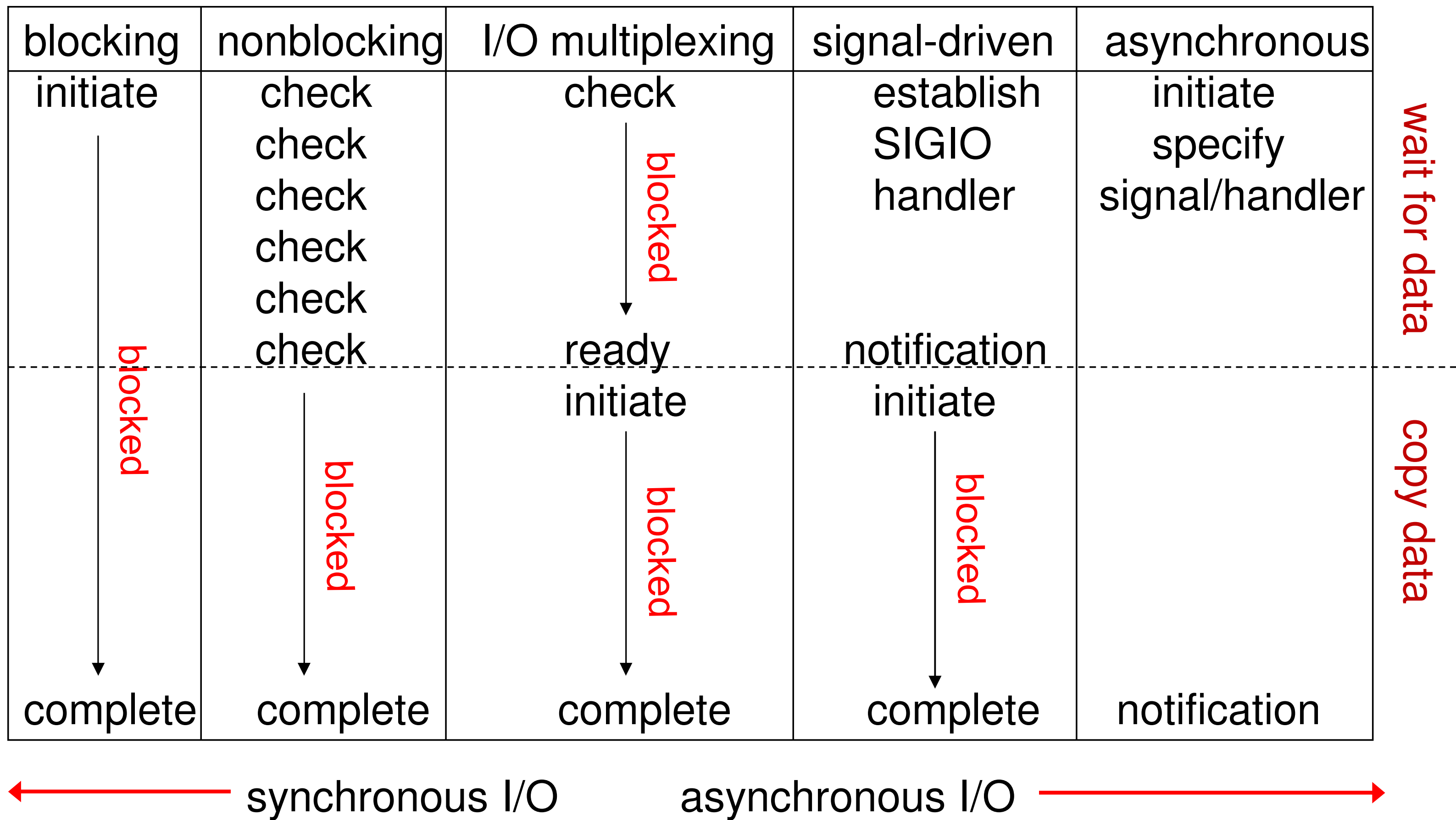


- 對輸入而言，等待(wait)資料輸入至buffer需要時間，從buffer搬移(copy)資料也需要時間。
- 根據等待模式不同，I/O動作可分為五種模式

Five I/O Models

- blocking I/O: blocked all the way
- nonblocking I/O: if no data in buffer, immediate returns EWOULDBLOCK
- I/O multiplexing (*select* and *poll*): blocked separately in wait and copy
- signal driven I/O (SIGIO): nonblocked in wait but blocked in copy (signaled when I/O can be initiated)
- asynchronous I/O (*aio_*): nonblocked all the way (signaled when I/O is complete)

Comparison of Five I/O Models



I/O Multiplexing: 使用select

- 本章要做的是I/O Multiplexing(第三種I/O model): 使用select system call
- select要求kernel測試某些裝置是否滿足我們設定的條件。若滿足則return，否則等待至此條件滿足時為止(select可被block, maybe forever)
- select呼叫可以指定等待的時間上限
- select呼叫可以指定測試多個I/O裝置

對輸入裝置使用select

- 可以使用select要求kernel測試某個輸入裝置是否ready for reading
- 當kernel中對應此裝置的buffer中已有相當數量(可設定)的輸入資料時，此裝置即是ready for reading
- select return後，由我們的程式自行呼叫其它的system call將buffer中的資料搬回來(也是blocking call，但所耗時間有限)。

對輸出裝置使用select

- 可以使用**select**要求kernel測試某個輸出裝置是否ready for writing
- 當kernel中對應此裝置的buffer中已有相當空間(可設定門檻值)可放置輸出資料時，此裝置即是ready for writing
- **select** return後，由我們的程式自行呼叫其它的system call將欲輸出資料搬入buffer中(也是blocking call，但所耗時間有限) 。

select Function

```
#include <sys/select.h>
#include <sys/time.h>
int select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
        const struct timeval *timeout);
    returns: positive count of ready descriptors, 0 on timeout, -1 on error
```

```
struct timeval {        (null: wait forever; 0: do not wait)
    long tv_sec; /*second */
    long tv_usec; /* microsecond */
};
```

readset, *writeset*, and *exceptset* specify the descriptors that we want the kernel to test for reading, writing, and exception conditions, respectively.

maxfdp1 is the maximum descriptor to be tested plus one.

select Fucntion

- We can call select and tell the kernel to return only when
  - any descriptor in {1, 4, 5} is ready for reading (buffer中已有資料)
  - any descriptor in {2, 7} is ready for writing
  - any descriptor in {1, 4} has an exception condition pending
  - after 10.2 seconds have elapsed

OR

Buffer中已有空間

裝置有例外狀況

Specifying Descriptor Values

- We need to declare variables of data type fd_set and use macros to manipulate these variables.

```
fd_set --- implementation dependent
    four macros:  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_set rset;

FD_ZERO(&rset);
FD_SET(1, &rset);
FD_SET(4, &rset);
FD_SET(5, &rset);
```

Turn on bits for descriptors 1, 4, and 5

$maxfdp1 = 6$

Socket Ready Conditions for `select`

Condition	readable?	writeable?	Exception?
enough data to read read-half closed new connection ready	x x x		
writing space available write-half closed		x x	
pending error	x	x	
TCP out-of-band data			x

Low-water mark (enough data/space to read/write in socket receive/send buffer): default is 1/2048, may be set by SO_RCVLOWAT/SO_SNDLOWAT socket option

Maximum number of descriptors for select?

Redefine FD_SETSIZE and recompile kernel

Ready For Reading 的解讀

- 對connected socket而言
  - 此socket is ready for read代表對方已有資料送過來，目前存放在kernel buffer
  - 可以呼叫read將資料從kernel buffer處取回
- 對listening socket而言
  - 此socket is ready for read代表已有新的client連線建立，目前在complete connection queue
  - 可以呼叫accept取回此新socket的file descriptor

Low-Water Mark (低水位)

- 對socket receive buffer而言
  - 如收到data量不足low-water mark, socket is not ready for reading
  - Default = 1 byte
- 對socket send buffer而言
  - 如可用空間(available space)不足low-water mark, socket is not ready for writing
  - Default = 2048 byte

用select改寫str_cli

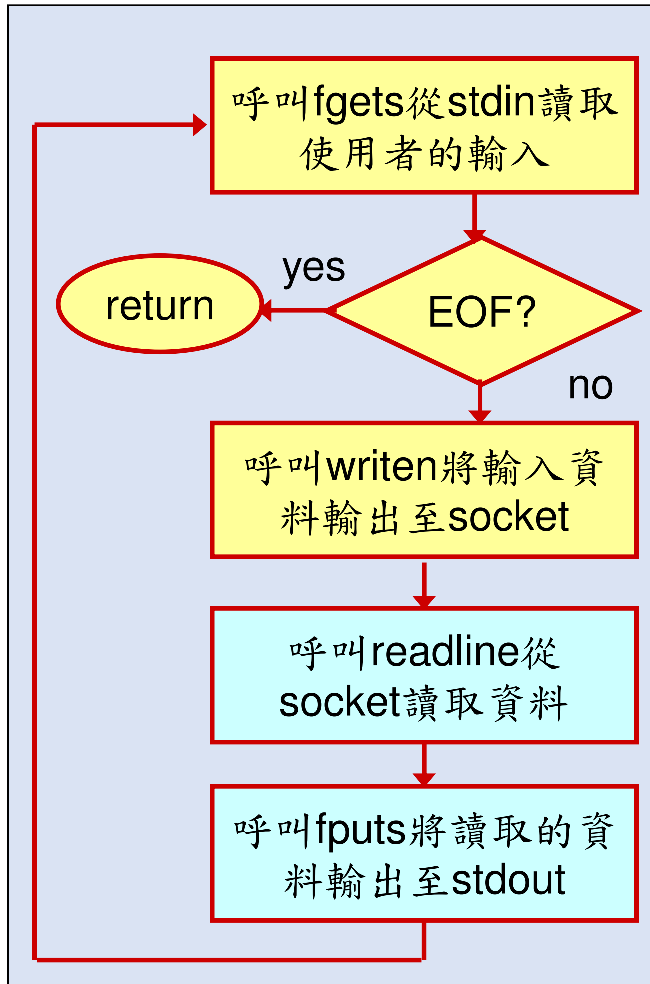
- 上一章的str_cli用兩個system calls分別取得input
  - fgets用於stdin的user input
  - readline用於socket input

均為Blocking I/O

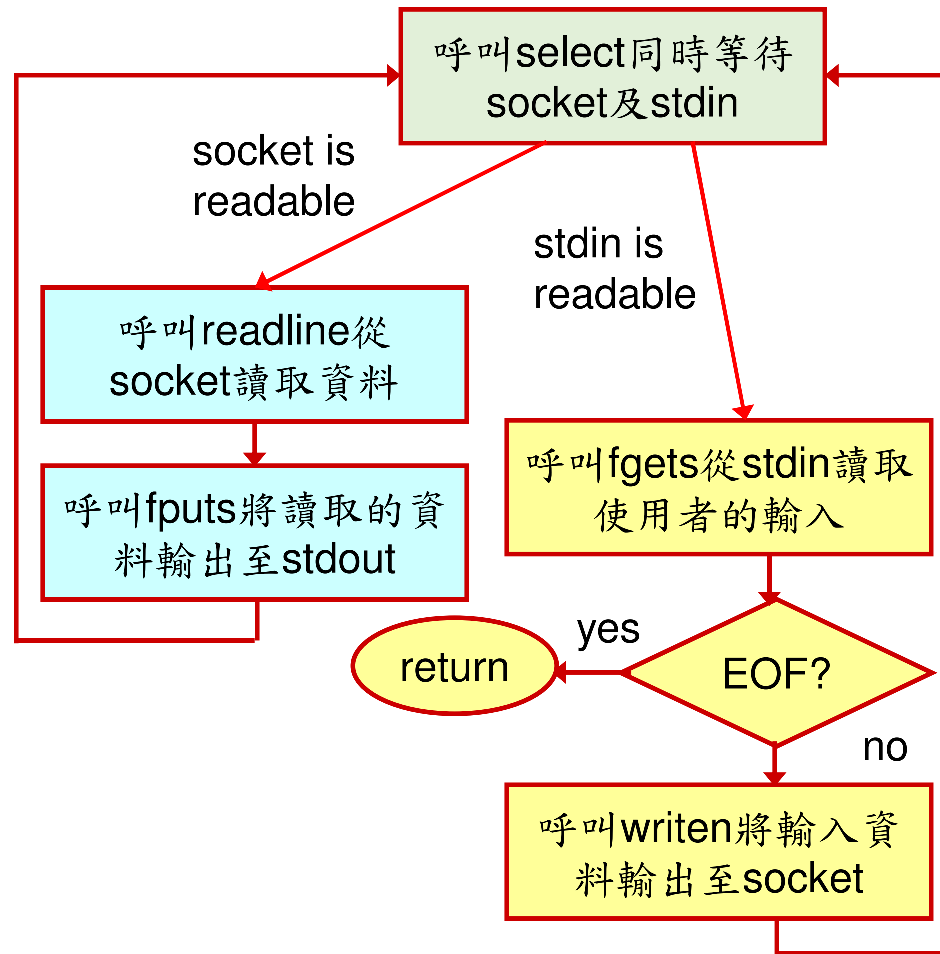
- 這樣沒有辦法同時等待兩個inputs。因此當client等在fgets時，無法同時取得socket input進來的資料
- 本章改用select來同時等待兩個inputs

兩個版本的比較

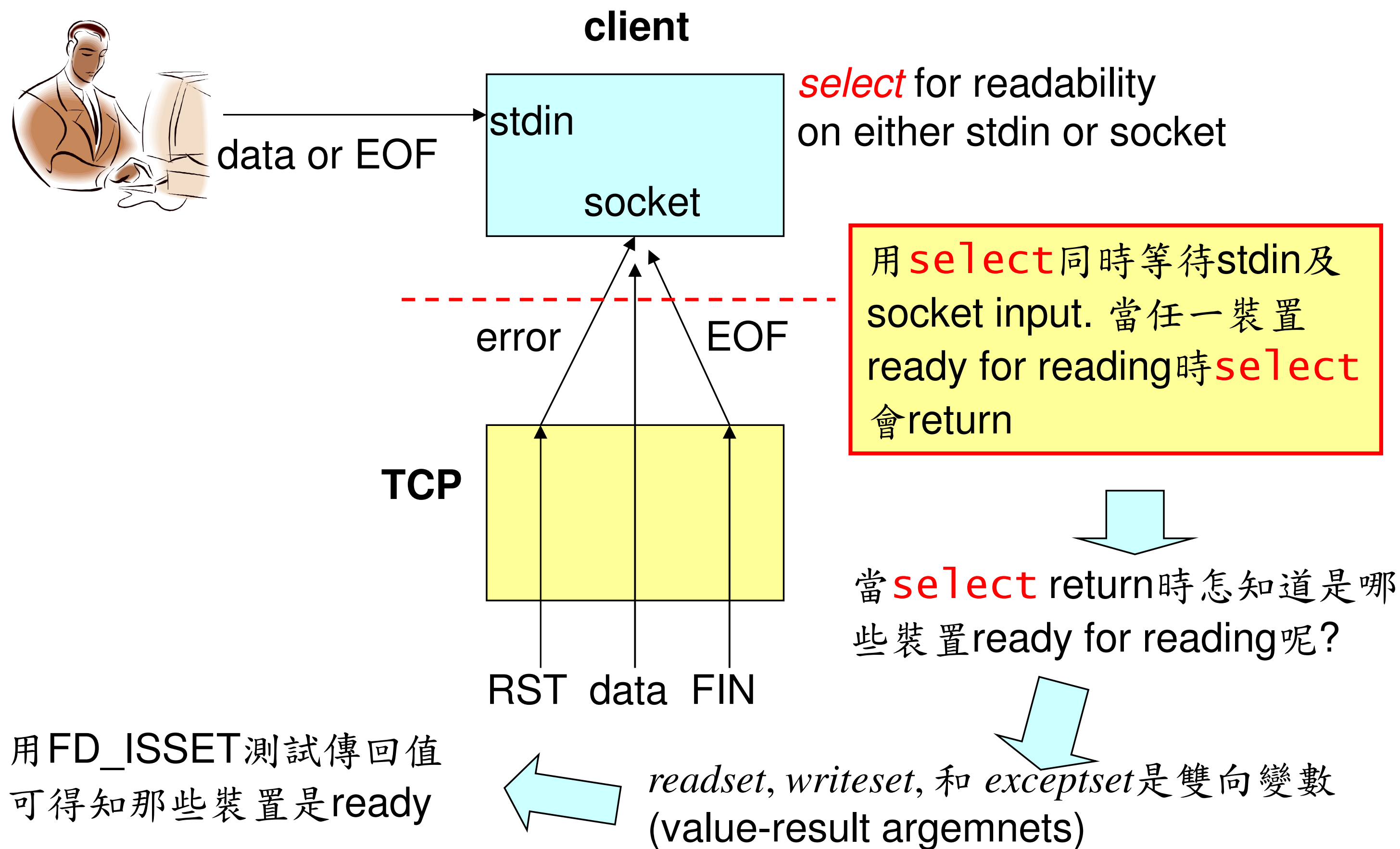
Ch. 5



Ch. 6



Rewrite *str_cli* Function with `select`



Rewrite *str_cli* Function with *select*

select/strcliselect01.c

```
#include       "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
        int             maxfdp1;
        fd_set          rset;
        char            sendline[MAXLINE], recvline[MAXLINE];

        FD_ZERO(&rset);
        for ( ; ; ) {
            FD_SET(fileno(fp), &rset);
            FD_SET(sockfd, &rset);
            maxfdp1 = max(fileno(fp), sockfd) + 1;
            Select(maxfdp1, &rset, NULL, NULL, NULL);
```

傳回檔案指標fp 的descriptor no

放在loop內

Why?

只測試read; may block forever

用FD_ISSET測試
裝置是否ready

select/strcliselect01.c

```
            if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
                if (Readline(sockfd, recvline, MAXLINE) == 0)
                    err_quit("str_cli: server terminated prematurely");
                Fputs(recvline, stdout);
            }
```

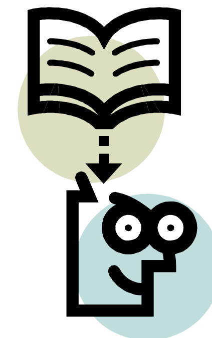
用FD_ISSET測試
裝置是否ready

```
            if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
                if (Fgets(sendline, MAXLINE, fp) == NULL)
                    return;        /* all done */
                Writen(sockfd, sendline, strlen(sendline));
            }
        }
}
```

User按了
Ctrl+D

使用select函數常犯的兩個錯誤

- 忘記*maxfdp1*是descriptor的最大值加1
- 忘記*readset*, *writeset*, 和 *exceptset*是雙向變數
  - select return時會改變它們的值
  - 因此再次呼叫select時別忘了要重新設定這些變數的內容



Redirecting Input on UNIX Systems

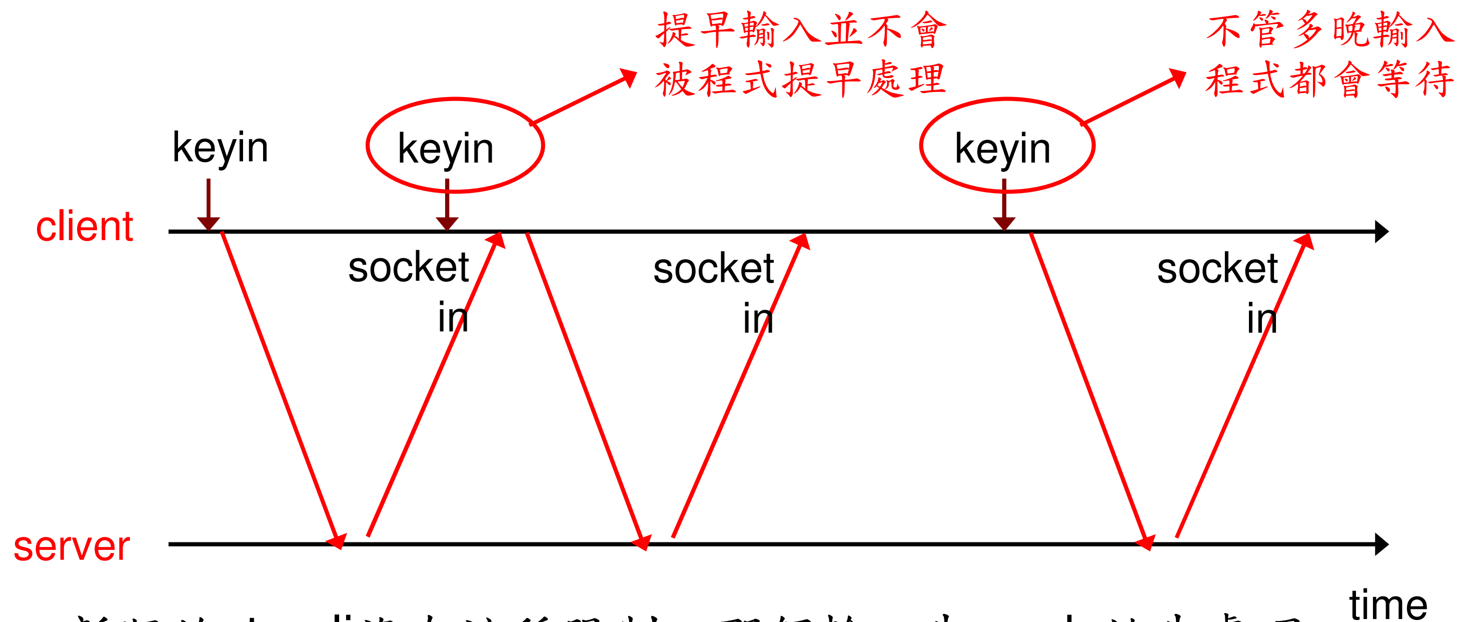
- Standard I/O - keyboard and screen
- Input Redirection symbol (<)
  - for UNIX and DOS
  - Example:

```
sum < input
```

  - Rather than inputting values by hand, read them from a file

Stop-and-Wait (Interactive) Mode

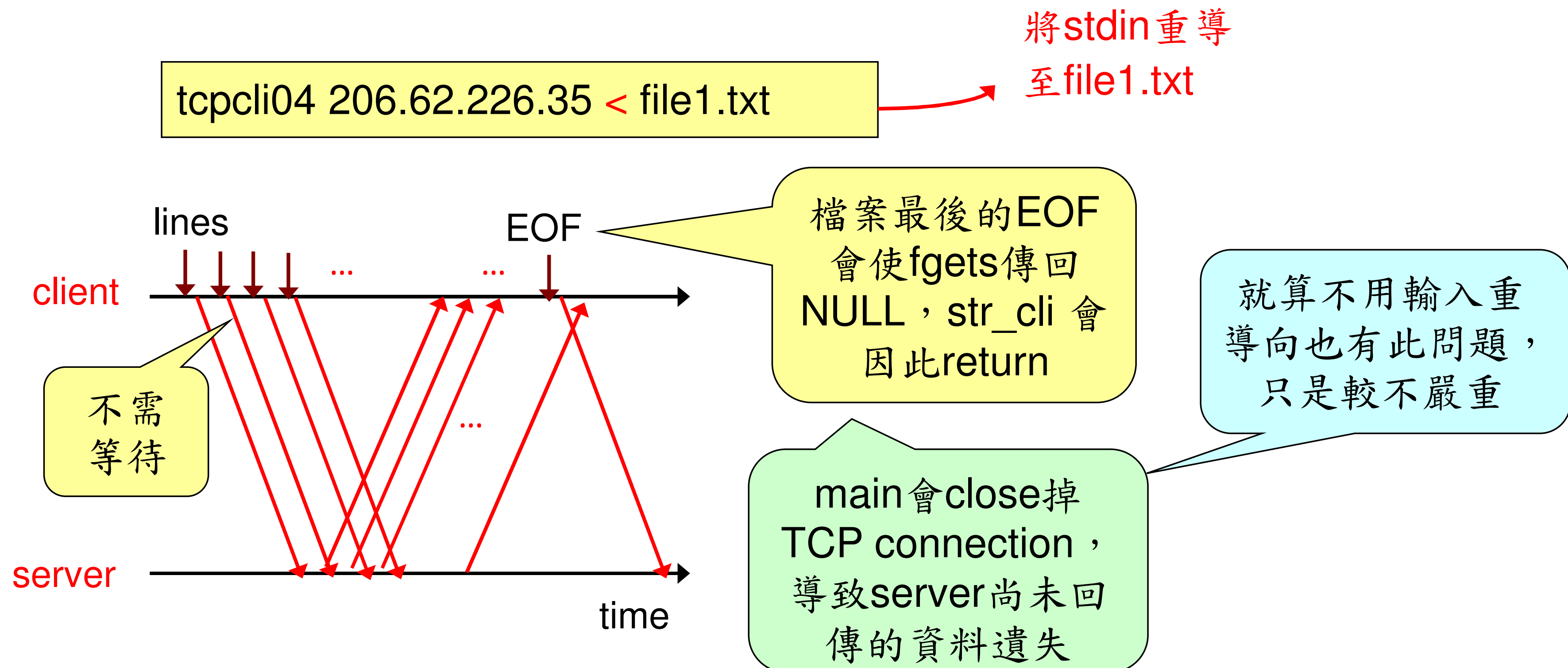
- 原來版本的str_cli強制輪流處理stdin輸入與socket輸入



新版的str_cli沒有這種限制，那個輸入先ready就先處理

Batch Mode (in new version)

- 如果在新版str_cli使用input redirection



Solution: Use Shutdown Instead of Close

- In *str_cli*, close write-half of TCP connection, by *shutdown*, while leaving read-half open.

Write-half: 我至對方
Read-half 對方至我

先關client至server方向的connection
server至client方向的connection暫時不要關

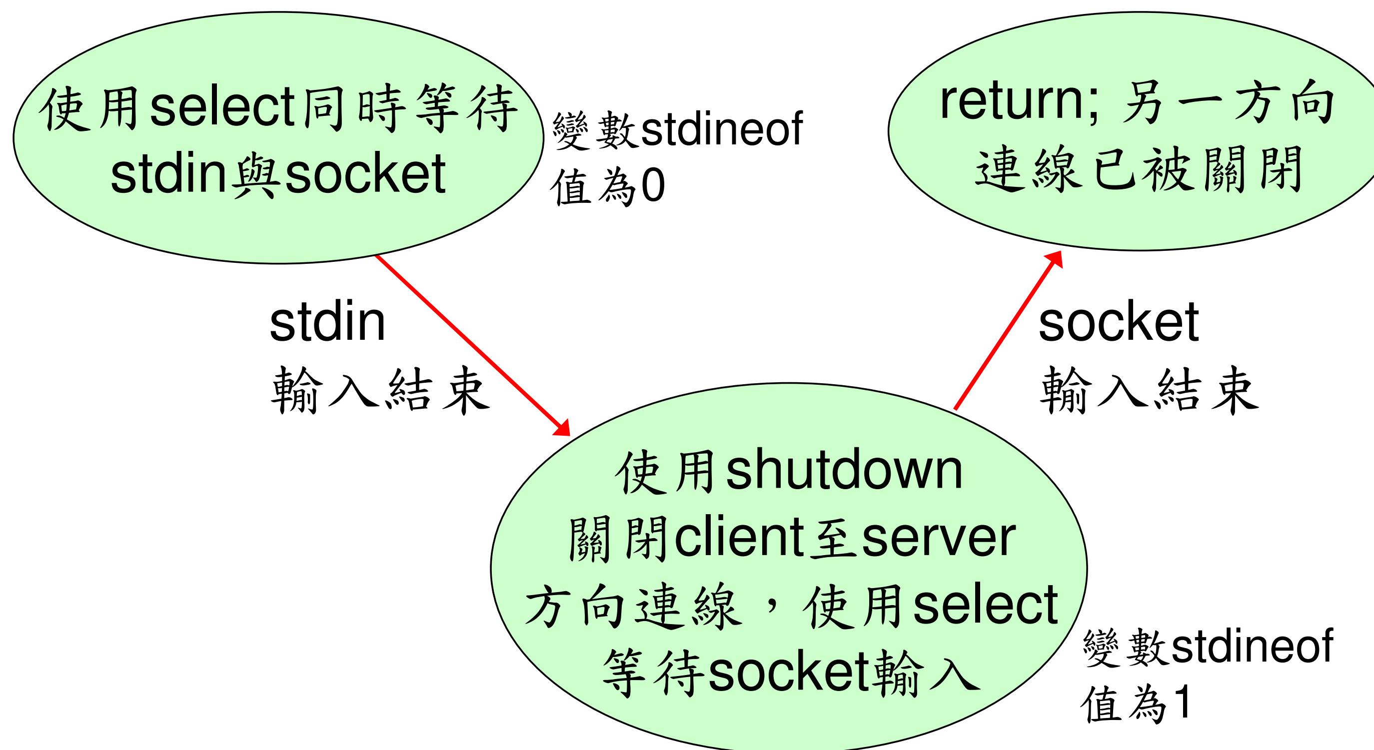
等到server的資料全部送回來後再關掉server至client方向的connection

Function shutdown

```
#include <sys/socket.h>
int shutdown(int sockfd, int howto);   returns: 0 if OK, -1 on error
howto: SHUT_RD, SHUT_WR, SHUT_RDWR
```

- initiates TCP normal termination regardless of descriptor’s reference count
- selectively closes one direction of the connection (SHUT_RD or SHUT_WR)

改寫後str_cli的三個狀態



Rewrite *str_cli* with select and shutdown

select/strcliselect02.c

```
#include       "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
        int             maxfdp1, stdineof;
        fd_set          rset;
        char            sendline[MAXLINE], recvline[MAXLINE];

        stdineof = 0;
        FD_ZERO(&rset);
        for ( ; ; ) {
                if (stdineof == 0)
                        FD_SET(fileno(fp), &rset);
                FD_SET(sockfd, &rset);
                maxfdp1 = max(fileno(fp), sockfd) + 1;
                Select(maxfdp1, &rset, NULL, NULL, NULL);
```

判斷client至server方向連結是否已斷的旗標變數

當client至server方向連結未斷時才要test stdin

select/strcliselect02.c

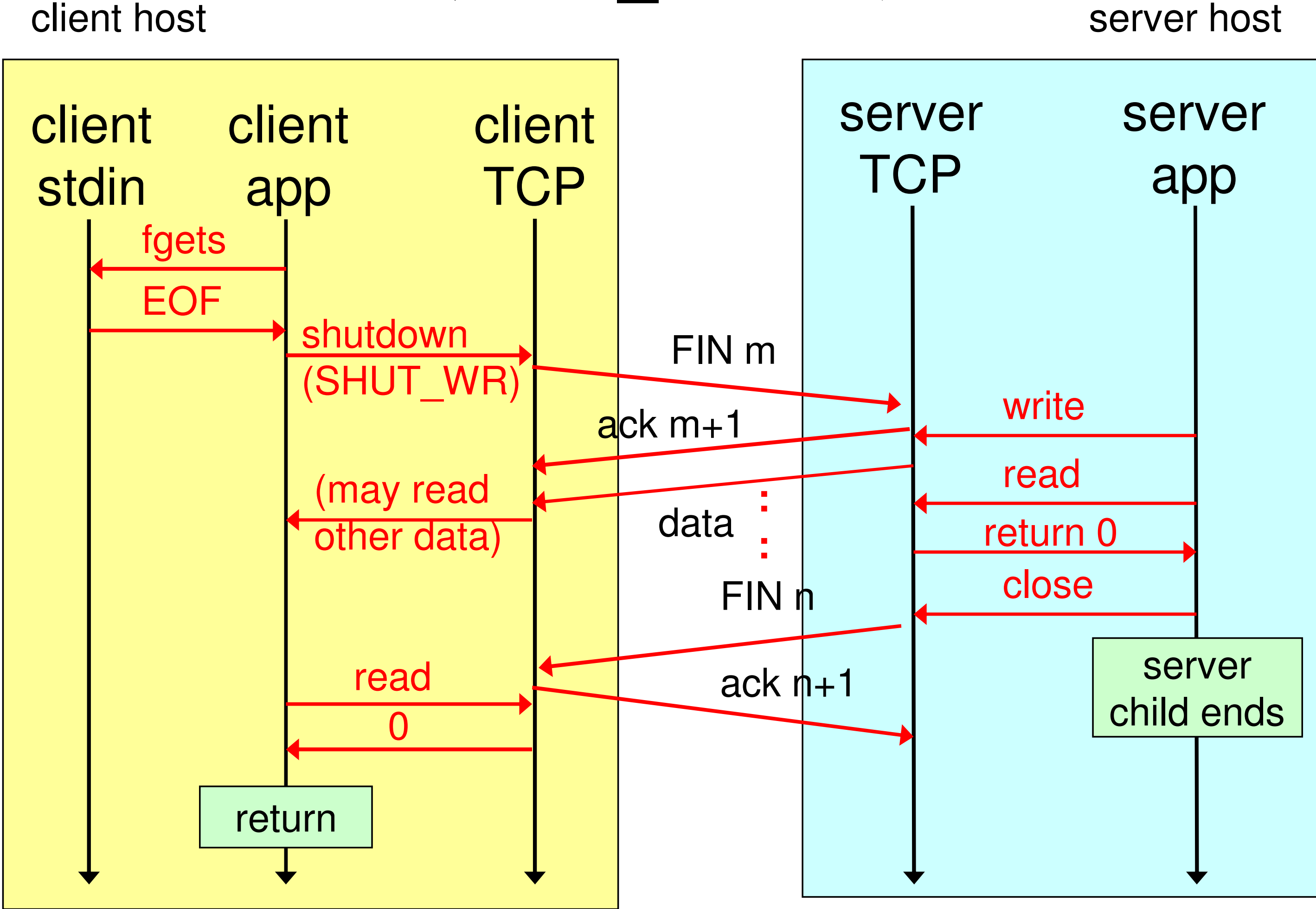
```
        if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
            if (Readline(sockfd, recvline, MAXLINE) == 0) {
                if (stdineof == 1)
                    return;         /* normal termination */
                else
                    err_quit("str_cli: server terminated prematurely");
            }
            Fputs(recvline, stdout);
        }
        if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
            if (Fgets(sendline, MAXLINE, fp) == NULL) {
                stdineof = 1;
                Shutdown(sockfd, SHUT_WR);      /* send FIN */
                FD_CLR(fileno(fp), &rset);
                continue;
            }
            Writen(sockfd, sendline, strlen(sendline));
        }
    }
}
```

No more data from server

client至server方向連結已斷

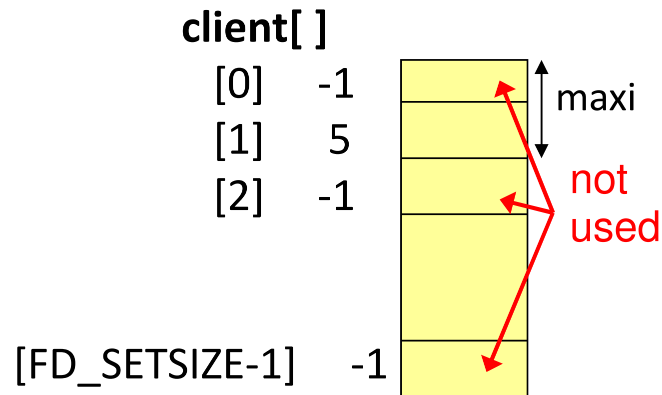
只斷client至server方向連結

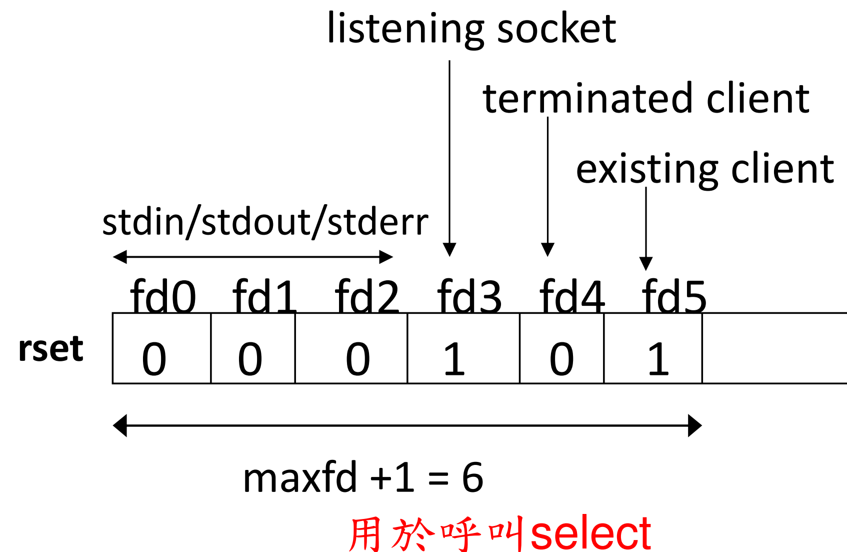
新版str_cli的流程



Concurrent TCP Echo Server with select

- A single server process using select to handle any number of clients (不fork child process)
- Need to keep track of the clients by *client*[ ] (client descriptor array) and *rset* (read descriptor set)





儲存connected socket descriptor

用於呼叫select

Rewrite Concurrent TCP Echo Server with *select*

Initialization

tcpcliserv/tcpservselect01.c

```
#include	"unp.h"
int main(int argc, char **argv)
{
	int					i, maxi, maxfd, listenfd, connfd, sockfd;
	int					nready, client[FD_SETSIZE];
	ssize_t				n;
	fd_set				rset, allset;
	char				line[MAXLINE];
	socklen_t			clilen;
	struct sockaddr_in	cliaddr, servaddr;

	listenfd = Socket(AF_INET, SOCK_STREAM, 0);

	bzero(&servaddr, sizeof(servaddr));
	servaddr.sin_family      = AF_INET;
	servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
	servaddr.sin_port        = htons(SERV_PORT);
```

tcpcliserv/tcpservselect01.c

Initialization (cont.)

```
Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

Listen(listenfd, LISTENQ);

maxfd = listenfd;/* initialize */
maxi = -1;/* index into client[] array */
for (i = 0; i < FD_SETSIZE; i++)
      client[i] = -1;/* -1 indicates available entry */
FD_ZERO(&allset);
FD_SET(listenfd, &allset);
```

測試listening socket是否 ready for reading

Loop

select的 ready的

tcpcliserv/tcpservselect01.c

```
for ( ; ; ) {
        rset = allset;      /* structure assignment */
        nready = Select(maxfd+1, &rset, NULL, NULL, NULL);
        if (FD_ISSET(listenfd, &rset)) {      /* new client connection */
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
            for (i = 0; i < FD_SETSIZE; i++)
                if (client[i] < 0) {
                    client[i] = connfd;     /* save descriptor */
                    break;
                }
            if (i == FD_SETSIZE)
                err_quit("too many clients");
            FD_SET(connfd, &allset);      /* add new descriptor to set */
            if (connfd > maxfd)
                maxfd = connfd;           /* for select */
            if (i > maxi)
                maxi = i;                 /* max index in client[] array */
            if (--nready <= 0)
                continue;                 /* no more readable descriptors */
        }
```

新連結

block

放入第一個找到的空格

加進select要test的descriptor set

如果沒有其它ready的descriptor

Loop (cont.)

tcpcliserv/tcpservselect01.c

```
for (i = 0; i <= maxi; i++) {   /* check all clients for data */
        if ( (sockfd = client[i]) < 0)
                continue;
        if (FD_ISSET(sockfd, &rset)) {
                if ( (n = Readline(sockfd, line, MAXLINE)) == 0) {
                                /* connection closed by client */
                        Close(sockfd);
                        FD_CLR(sockfd, &allset);
                        client[i] = -1;
                } else
                        Writen(sockfd, line, n);
                if (--nready <= 0)
                        break;                          /* no more readable descriptors */
        }
    }
  }
}
```

跳過沒有放descriptor的空格

已連結

從要test的descriptor set中去除

變空格

如果沒有其它ready的descriptor,可以提早離開loop

前頁程式的潛在問題

- **select** 在 socket 的 input buffer 有資料可讀時即會 return 此 socket 已 ready for reading
- **Readline** 被設計成要讀到以換行字元結束的一列文字 (或是socket被結束掉)才會return
- 有心人士可能利用這項弱點進行攻擊

Denial of Service Attacks

- 問題出在Server程式可能block在**`Readline`**中
- Attack scenario:
  - a malicious client sends 1 byte of data (other than a newline) and sleep
  - server hangs until the malicious client either sends a newline or terminates
  - (server 要讀到 newline, **`Readline`** 才能 return)

Solutions to the Attack

- When a server is handling multiple clients, the server can never block in a function call related to a single client

- Possible solutions:
  - Use nonblocking I/O (Ch. 15)
  - separate thread/process for each client
  - timeout on I/O operations (Sec. 13.2)

poll Function: polling more specific conditions than select

結構陣列儲存位址　陣列中的資料數目　微秒

```
#include <poll.h>
int poll (struct pollfd *fdarray, unsigned long ndfs, int timeout);
    returns: count of ready descriptors, 0 on timeout, -1 on error
```

每個descriptor要test的條件是用pollfd結構來表示

這些pollfd結構集合成一個陣列

要測的file descriptor number; -1表此結構無效

```
struct pollfd {
    int fd;            /* a descriptor to poll */
    short events;      /* events of interested fd, value argument */
    short revents;     /* events that occurred on fd, result argument */
};
```

要測試的條件　真正測到的事件

poll函數中的事件設定

- events與revents由下列bit flag 組成

Constant	events	revents	Description
POLLIN	x	x	normal or priority band to read
POLLRDNORM	x	x	normal data to read
POLLRDBAND	x	x	priority band data to read
POLLPRI	x	x	high-priority data to read
POLLOUT	x	x	normal data can be written
POLLWRNORM	x	x	normal data can be written
POLLWRBAND	x	x	priority band data can be written
POLLERR	x		an error has occurred
POLLHUP	x		hang up has occurred
POLLNVAL	x		descriptor is not an open file

設定events與測試revents

- 設定events: 使用bitwise OR

```
struct pollfd            client[OPEN_MAX];
…
client[0].events = POLLRDNORM | POLLRDBAND;
```

- 測試revents: 使用bitwise AND

```
struct pollfd            client[OPEN_MAX];
…
if (client[0].revents & POLLRDNORM)
     …
```

Three Classes of Data Identified by `poll`

- *normal, priority band,* and *high priority*

	normal	priority band	high priority
All regular TCP data	x		
All UDP data	x		
TCP's out-of-band data		x	
Half-closed TCP	x		
Error for TCP	x (or POLLERR)		
New connection	x	x	

Concurrent TCP Echo Server with `poll`

- When using *select*, the server maintains array *client*[ ] and descriptor set *rset.* When using poll, the server maintains array *client* of *pollfd* structure.
- Program flow:
  - allocate array of `pollfd` structures
  - initialize (listening socket: first entry in *client*) (set POLLRDNORM in *events*)
  - call poll; check for new connection (check, in *revents*, and set, in *events*, POLLRDNORM)
  - check for data on an existing connection (check POLLRDNORM or POLLERR in *revents*)

Rewrite Concurrent TCP Echo Server with *poll*

Initialization

tcpcliserv/tcpservpoll01.c

```
#include       "unp.h"
#include       <limits.h>            /* for OPEN_MAX */
int main(int argc, char **argv)
{
       int                          i, maxi, listenfd, connfd, sockfd;
       int                          nready;
       ssize_t                      n;
       char                         line[MAXLINE];
       socklen_t                    clilen;
       struct pollfd                client[OPEN_MAX];
       struct sockaddr_in           cliaddr, servaddr;
       listenfd = Socket(AF_INET, SOCK_STREAM, 0);
       bzero(&servaddr, sizeof(servaddr));
       servaddr.sin_family      = AF_INET;
       servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
       servaddr.sin_port        = htons(SERV_PORT);
```

結構陣列

Initialization (cont.)

tcpcliserv/tcpservpoll01.c

```
Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

Listen(listenfd, LISTENQ);

client[0].fd = listenfd;
client[0].events = POLLRDNORM;
for (i = 1; i < OPEN_MAX; i++)
    client[i].fd = -1;          /* -1 indicates available entry */
maxi = 0;                       /* max index into client[] array */
```

第0個元素放 listening socket

Loop

tcpcliserv/tcpservpoll01.c

```
for ( ; ; ) {
        nready = Poll(client, maxi+1, INFTIM);

        if (client[0].revents & POLLRDNORM) {      /* new client connection */
                clilen = sizeof(cliaddr);
                connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
                for (i = 1; i < OPEN_MAX; i++)
                        if (client[i].fd < 0) {
                                client[i].fd = connfd;       /* save descriptor */
                                break;
                        }
                if (i == OPEN_MAX)
                        err_quit("too many clients");
                client[i].events = POLLRDNORM;
                if (i > maxi)
                        maxi = i;                    /* max index in client[] array */
                if (--nready <= 0)
                        continue;                    /* no more readable descriptors */
        }
```

wait forever

bitwise AND

不會block

放入第一個找到的空格

for data socket

tcpcliserv/tcpservpoll01.c

```
for (i = 1; i <= maxi; i++) {   /* check all clients for data */
        if ( (sockfd = client[i].fd) < 0)
            continue;
        if (client[i].revents & (POLLRDNORM | POLLERR)) {
            if ( (n = readline(sockfd, line, MAXLINE)) < 0) {
                if (errno == ECONNRESET) {
                        /* connection reset by client */
                    Close(sockfd);
                    client[i].fd = -1;
                } else
                    err_sys("readline error");
            } else if (n == 0) {
                    /* connection closed by client */
                Close(sockfd);
                client[i].fd = -1;
            } else
                Writen(sockfd, line, n);
            if (--nready <= 0)
                break;              /* no more readable descriptors */
} } } }
```

bitwise AND

變無效資料

變無效資料

Conclusion

- 本章介紹如何用I/O Multiplexing來取代blocking I/O，以同時等待兩個以上的I/O裝置
- I/O Multiplexing可用`select`或`poll`來達成
- 為何不使用nonblocking I/O呢? 就算Buffer中沒資料也可立即return耶…

pselect Function

```
#include <sys/select.h>
#include <signal.h>
#include <time.h>
int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
            const struct timespec *timeout, const sigset_t *sigmask);
      returns: count of ready descriptors, 0 on timeout, -1 on error
struct timespec {
      time_t tv_sec;  /* seconds */
      long tv_nsec;  /* nanosecond */ };
```

- Invented by Posix.1g
- Two changes from the normal `select`
  - uses `timespec` instead of `timeval`
  - adds a 6th argument: a pointer to a signal mask

The Signal Mask

- Allows a program to
  - ①Disable the delivery of certain signals
  - ②Test some global variables that are set by the handler for these now-disabled signals
  - ③Call `pselect`, telling it to reset the signal mask
- This is to avoid signal loss in race condition

pselect Function: Avoiding Signal Loss in Race Condition

```
if (intr_flag)
      handle_intr( );      /* handle signal */
if ((nready = select ( ... ))< 0) {
      if (errno == EINTR) {
            if (intr_flag)
                  handle_intr( );
       }
       ....
}
```

signal lost if *select* blocks forever

```
sigemptyset (&zeromask);
sigemptyset (&newmask);
sigaddset (&newmask, SIGINT);
sigprocmask (SIG_BLOCK, &newmask, &oldmask);
if (intr_flag)
      handle_intr( );
if ( (nready = pselect ( ... , &zeromask)) < 0) {
      if (errno == EINTR) {
            if (intr_flag)
                  handle_intr( );
       }
 ...}
```