I/O Multiplexing

Computer Network Programming

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*

A Scenario

- Given the echo client introduced earlier, what if it is blocked in a call to fgets(), and the echo server is terminated?
  - The server TCP correctly sends a FIN to the client TCP, but the client process never sees it until the client process reads from the socket later
- What we need here is the capability to handle multiple I/O descriptors at the same time
  - I/O multiplexing!

Introduction

- I/O multiplexing: to be notified, by kernel, if one or more I/O conditions are ready.
- I/O multiplexing is typically used in networking applications in the following,
  - When a client is handling multiple descriptors (normally interactive input and a network socket)
  - When a client to handle multiple sockets at the same time (this is possible, but rare)
  - If a TCP server handles both a listening socket and its connected sockets
  - If a server handles both TCP and UDP
  - If a server handles multiple services and perhaps multiple protocols

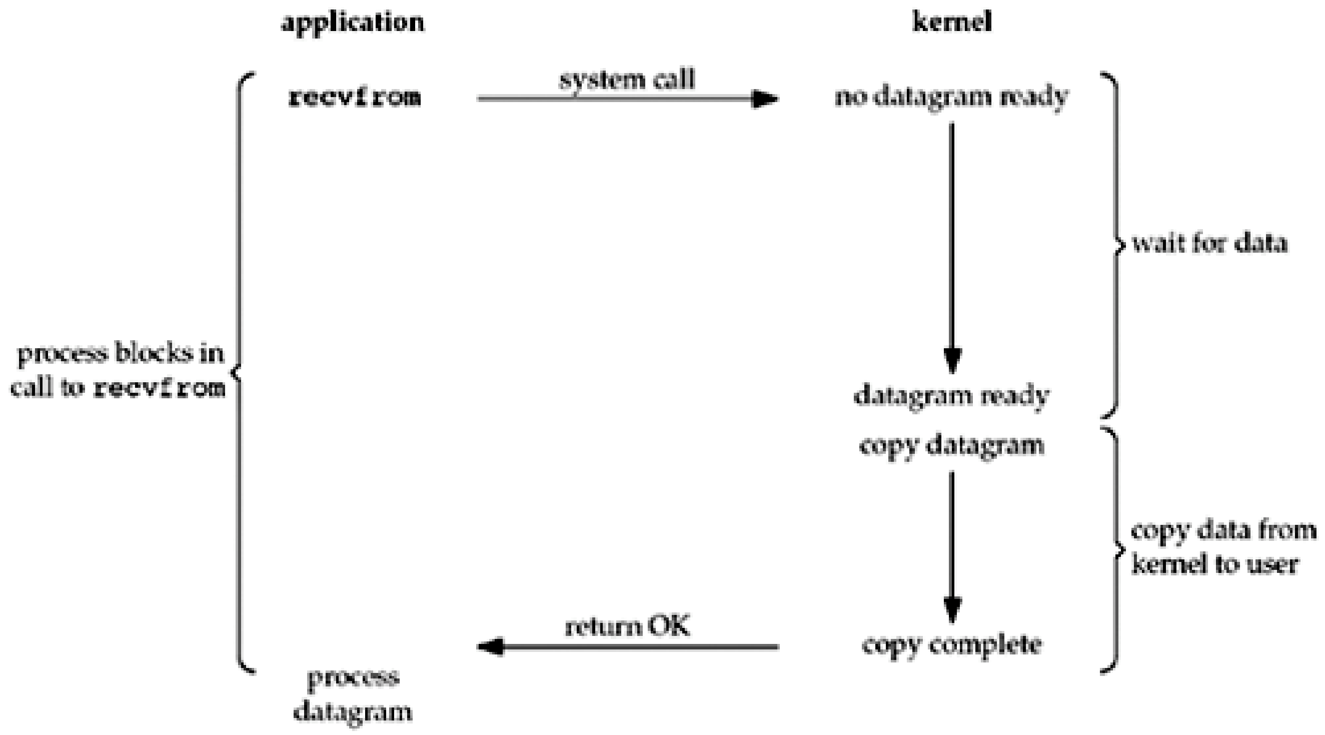
I/O Models

- Five I/O models:
  - blocking I/O: blocked all the way
  - nonblocking I/O: if no data, 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)

The phases for an input operation

- There are normally two distinct phases for an input operation:
  - Waiting for the data to be ready. This involves waiting for data to arrive on the network. When the packet arrives, it is copied into a buffer within the kernel.
  - Copying the data from the kernel to the process. This means copying the (ready) data from the kernel's buffer into our application buffer

Blocking I/O Model



- A process that performs an I/O operation will wait (block) until the operation is completed
- By default, **all sockets I/Os are blocking I/Os**

Blocking I/O Model

- We use UDP for this example instead of TCP because with UDP, the concept of data being "ready" to read is simple:
  - either an entire datagram has been received or it has not.
  - With TCP it gets more complicated, as additional variables such as the socket's low-water mark come into play.

- We also refer to recvfrom as a system call to differentiate between our application and the kernel, regardless of how recvfrom is implemented.
- There is normally a switch from running in the application to running in the kernel, followed at some time later by a return to the application.

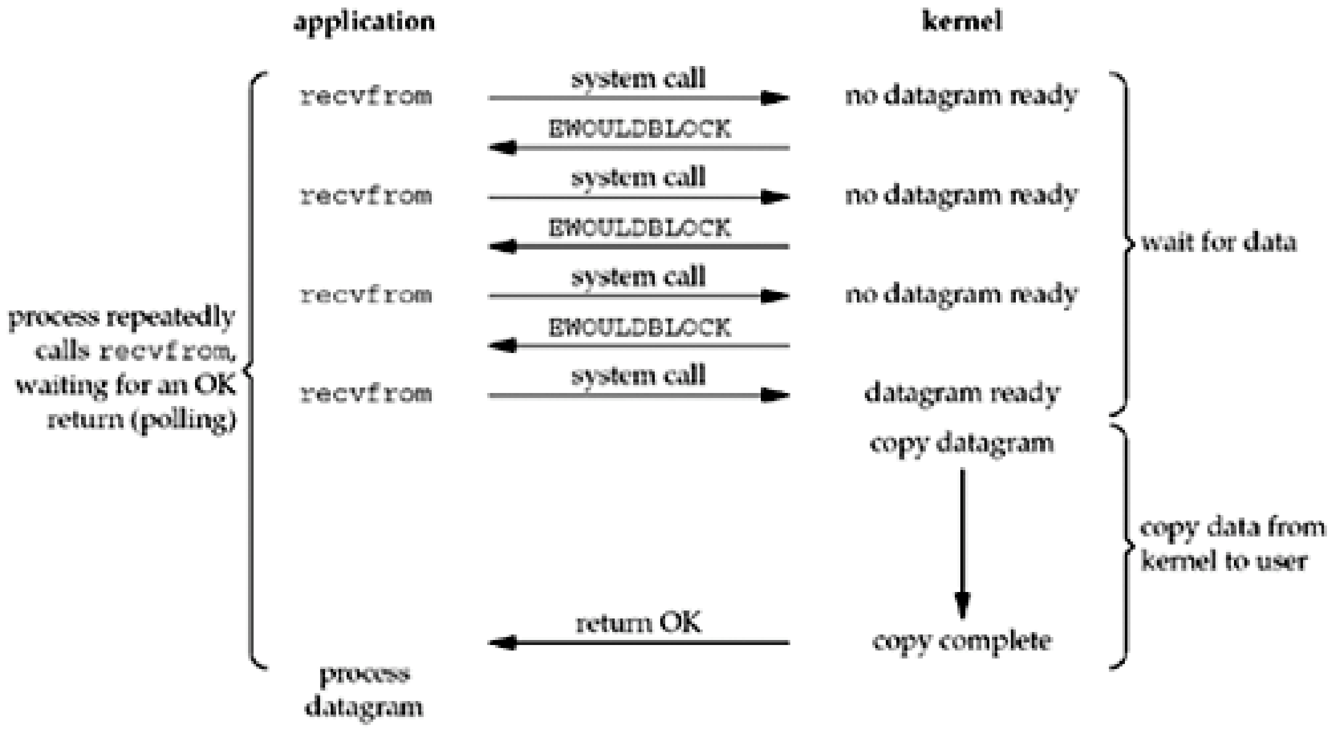
Blocking I/O Model

- In the figure above, the process calls recvfrom and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs.
- The most common error is the system call being interrupted by a signal.
- We say that the process is blocked the entire time from when it calls recvfrom until it returns. When recvfrom returns successfully, our application processes the datagram.

Nonblocking I/O Model

- When a socket is set to be nonblocking, we are telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead".

Nonblocking I/O Model



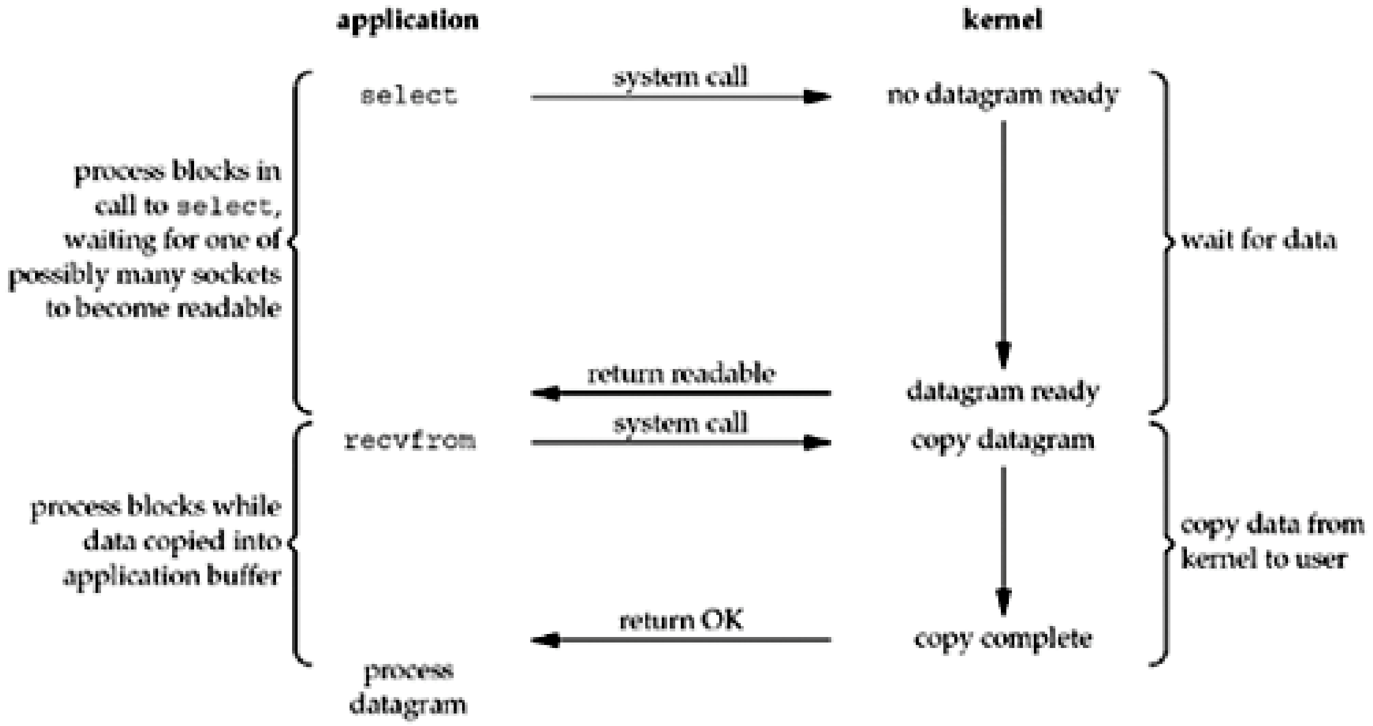
- For the first three recvfrom, there is no data to return and the kernel immediately returns an error of EWOULDBLOCK.
- For the fourth time we call recvfrom, a datagram is ready, it is copied into our application buffer, and recvfrom returns successfully. We then process the data.

Nonblocking I/O Model

- When an application sits in a loop calling recvfrom on a nonblocking descriptor, it is called polling.
- The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

I/O Multiplexing Model

- With I/O multiplexing, we call **select** or **poll** and block in one of these two system calls, instead of blocking in the actual I/O system call.



application
kernel
select
system call
no datagram ready
process blocks in call to select, waiting for one of possibly many sockets to become readable
wait for data
return readable
datagram ready
recvfrom
system call
copy datagram
process blocks while data copied into application buffer
copy data from kernel to user
return OK
copy complete
process datagram

I/O Multiplexing Model

- We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.

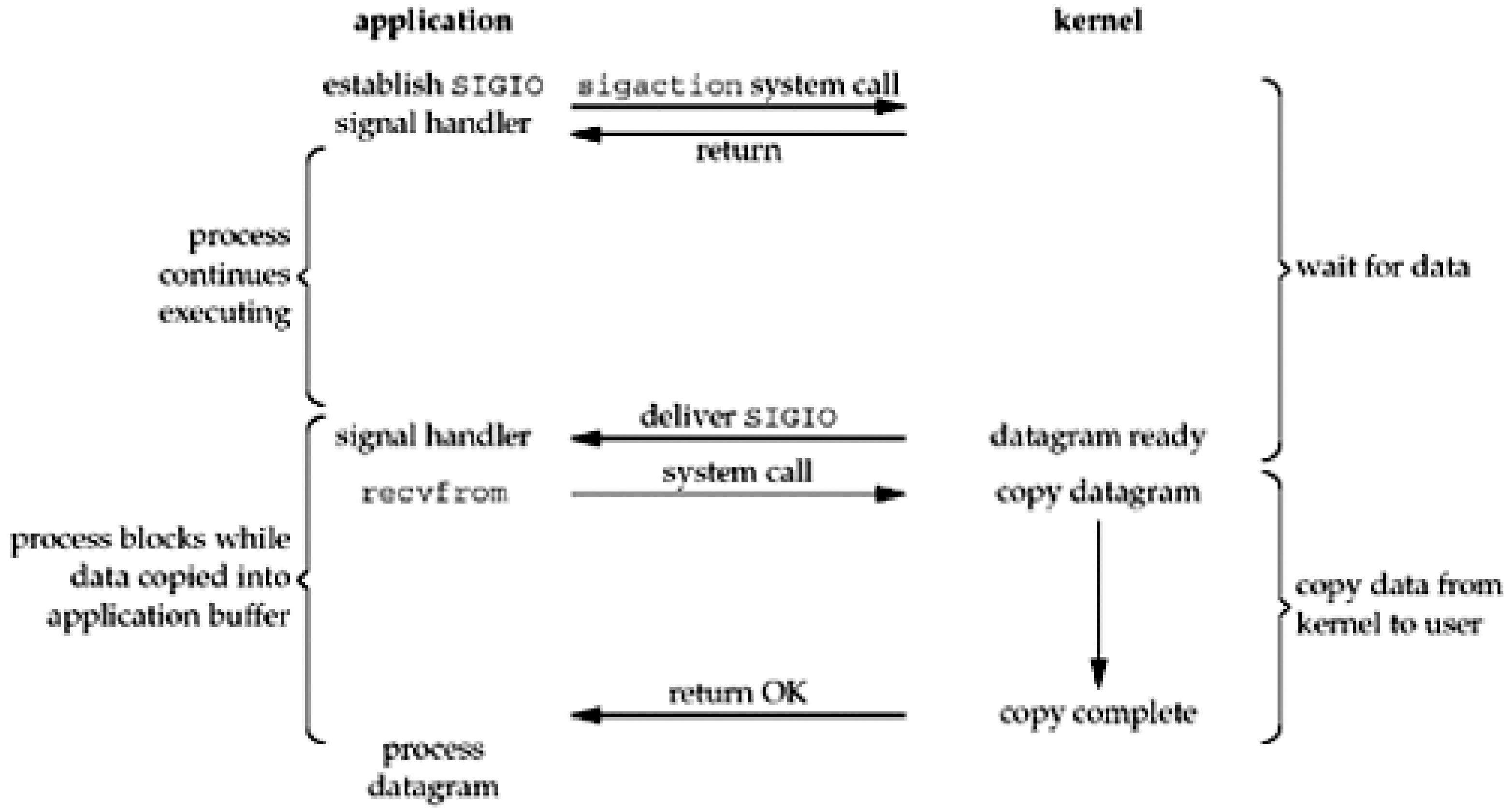
- **Comparing I/O Multiplexing Model, Nonblocking I/O Model and Blocking I/O Model**:
  - Disadvantage: using select requires two system calls (select and recvfrom) instead of one
  - Advantage: we can wait for more than one descriptor to be ready (see the select function later in this chapter)

Multithreading with blocking I/O

- Another closely related I/O model is to use multithreading with blocking I/O.
- That model very closely resembles the model described above, except that instead of using select to block on multiple file descriptors, the program uses multiple threads (one per file descriptor), and each thread is then free to call blocking system calls like recvfrom.

Signal Driven I/O Model

- The signal-driven I/O model uses signals, telling the kernel to notify us with the SIGIO signal when the descriptor is ready.



Signal Driven I/O Model

- We first enable the socket for signal-driven I/O and install a signal handler using the sigaction system call.
- The return from this system call is immediate and our process continues; it is not blocked.
- When the datagram is ready to be read, the SIGIO signal is generated for our process. We can either:
  - read the datagram from the signal handler by calling recvfrom and then notify the main loop that the data is ready to be processed
  - notify the main loop and let it read the datagram.
- **The advantage to this model**
  - we are not blocked while waiting for the datagram to arrive.
  - The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

Asynchronous I/O Model

- **Asynchronous I/O** is defined by the POSIX specification, and various differences in the *real-time* functions that appeared in the various standards which came together to form the current POSIX specification have been reconciled.
- These functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete.
- **The main difference between this model and the signal-driven I/O mode**l
  – signal-driven I/O, the kernel tells us when an I/O operation can be initiated
  – asynchronous I/O, the kernel tells us when an I/O operation is complete.

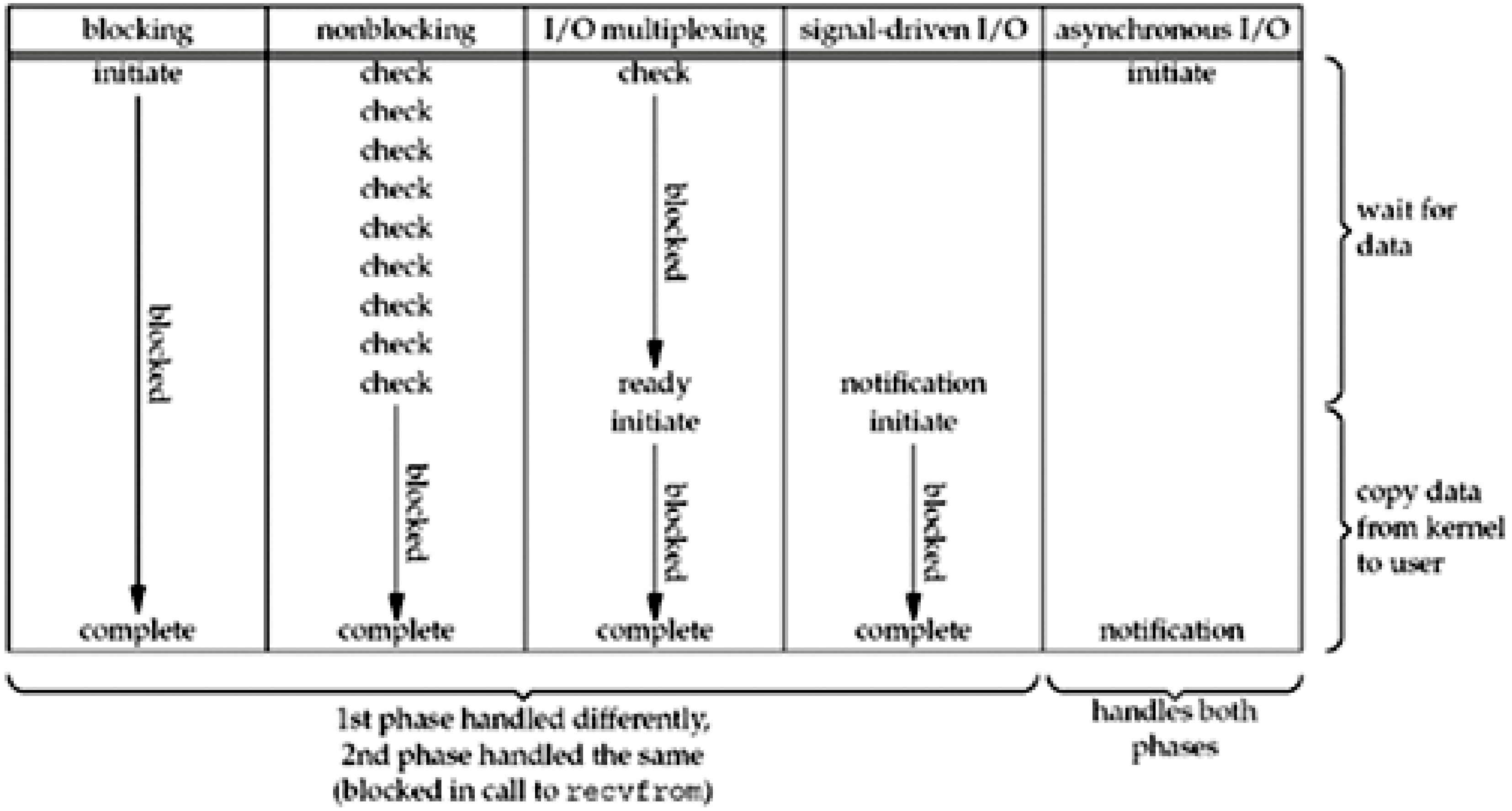
Asynchronous I/O Model

- We call aio_read (the POSIX asynchronous I/O functions begin with aio_ or lio_) and pass the kernel the following:
  - descriptor, buffer pointer, buffer size (the same three arguments for read),
  - file offset (similar to lseek),
  - and how to notify us when the entire operation is complete.
- This system call returns immediately and our process is not blocked while waiting for the I/O to complete.
- We assume in this example
  - we ask the kernel to generate some signal when the operation is complete.
  - This signal is not generated until the data has been copied into our application buffer, which is different from the signal-driven I/O model.

Comparison of Five I/O Models

- The main difference between the first four models is the first phase
- The second phase in the first four models is the same: the process is blocked in a call to recvfrom while the data is copied from the kernel to the caller's buffer.
- Asynchronous I/O, however, handles both phases and is different from the first four.

Comparison of Five I/O Models



Synchronous I/O versus Asynchronous I/O

- POSIX defines these two terms as follows:
  - A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes.
  - An asynchronous I/O operation does not cause the requesting process to be blocked.
- Using these definitions, the first four I/O models (blocking, nonblocking, I/O multiplexing, and signal-driven I/O) are all synchronous because the actual I/O operation (recvfrom) blocks the process.
- Only the asynchronous I/O model matches the asynchronous I/O definition.

select Function

- The select function allows the process to instruct the kernel to either:
  - Wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs, or
  - When a specified amount of time has passed.

select Function

- This means that we tell the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait.
- The descriptors in which we are interested are not restricted to sockets; any descriptor can be tested using select.

```
#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 */
```

The timeout argument

- The timeout argument
  - The timeout argument tells the kernel how long to wait for one of the specified descriptors to become ready.
  - A timeval structure specifies the number of seconds and microseconds.

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

The timeout argument

- There are three possibilities for the timeout:
  - **Wait forever** (timeout is specified as a null pointer). Return only when one of the specified descriptors is ready for I/O.
  - **Wait up to a fixed amount of time** (timeout points to a timeval structure). Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the timeval structure.
  - **Do not wait at all** (timeout points to a timeval structure and the timer value is 0, i.e. the number of seconds and microseconds specified by the structure are 0). Return immediately after checking the descriptors. This is called polling.

select Function

- Is used to tell the kernel to notify the calling process when some event(s) of interest occurs
  - For example,
    - Any descriptor in the set { 1, 2, 4} is ready for reading
    - Any descriptor in the set { 3, 5} is ready for writing
    - Any descriptor in the set { 2, 3, 6} has an exception pending
    - After waiting for 5 seconds and 40 milliseconds

The descriptor sets arguments

- The three middle arguments, **readset**, **writeset**, and **exceptset**, specify the descriptors that we want the kernel to test for reading, writing, and exception conditions.
- select uses descriptor sets, typically an array of integers, with each bit in each integer corresponding to a descriptor.
- For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63, and so on.

The descriptor sets arguments

- All the implementation details are irrelevant to the application and are hidden in the fd_set datatype and the following four macros:

```
void FD_ZERO(fd_set *fdset);          /* clear all bits in fdset */
void FD_SET(int fd, fd_set *fdset);  /* turn on the bit for fd in fdset */
void FD_CLR(int fd, fd_set *fdset);  /* turn off the bit for fd in fdset */
int FD_ISSET(int fd, fd_set *fdset); /* is the bit for fd on in fdset ? */
```

The descriptor sets arguments

- We allocate a descriptor set of the fd_set datatype, we set and test the bits in the set using these macros, and we can also assign it to another descriptor set across an equals sign “=“ in C.
- An array of integers using one bit per descriptor, is just one possible way to implement select.
- Nevertheless, it is common to refer to the individual descriptors within a descriptor set as bits, as in "turn on the bit for the listening descriptor in the read set."

The descriptor sets arguments

- The following example defines a variable of type fd_set and then turn on the bits for descriptors 1, 4, and 5:

```
fd_set rset;

FD_ZERO(&rset);          /* initialize the set: all bits off */
FD_SET(1, &rset);        /* turn on bit for fd 1 */
FD_SET(4, &rset);        /* turn on bit for fd 4 */
FD_SET(5, &rset);        /* turn on bit for fd 5 */
```

- It is important to initialize the set, since unpredictable results can occur if the set is allocated as an automatic variable and not initialized.

The descriptor sets arguments

- Any of the middle three arguments to select, **readset**, **writeset**, or **exceptset**, can be specified as a null pointer if we are not interested in that condition.
- Indeed, if all three pointers are null, then we have a higher precision timer than the normal Unix sleep function. The poll function provides similar functionality.

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 */
};
```

- Three uses for select() :
  - –Wait forever if the structure pointer is NULL
  - –Wait up to tv_sec and tv_usec
  - –No wait if both tv_sec and tv_usec set to 0
- *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

Specifying Descriptor Values

- We need to declare variables of data type fd_set and use macros to manipulate these variables
- 中間三個參數readset, writeset和exceptset的類型都是fd_set*.其實fd_set* 就是int*(整形陣列). 所有的descriptor號碼都用整形陣列的一個bit表述

```
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);
```

maxfdp1 = 6

```
fd_set rset;

FD_ZERO(&rset);        initialize the set, and bit off
FD_SET(1, &rset);      Turn on bits for
FD_SET(4, &rset);      descriptors 1, 4, and 5
FD_SET(5, &rset);
```

Return value of select

- The return value from this function indicates the total number of bits that are ready across all the descriptor sets. If the timer value expires before any of the descriptors are ready, a value of 0 is returned. A return value of –1 indicates an error (which can happen, for example, if the function is interrupted by a caught signal).

When Is A Descriptor Ready?

- **Ready for read**
  - A read operation will not block
- **Ready for write**
  - A write operation will not block
- **Presence of exception data**
  - E.g., out-of-band data received, some control status information from a master pseudo-terminal, etc

Ready for Read

- Data received is >= read buffer low-water mark
  - Can be set using SO_RCVLOWAT
  - Default is 1 for TCP and UDP sockets
- The read-half of the connection is closed
  - A read operation will return 0 (EOF) without blocking
- A listening socket with an established connection
  - An accept operation on the socket will not block
- A socket error is pending
  - A read operation on the socket will return an error (-1) without blocking

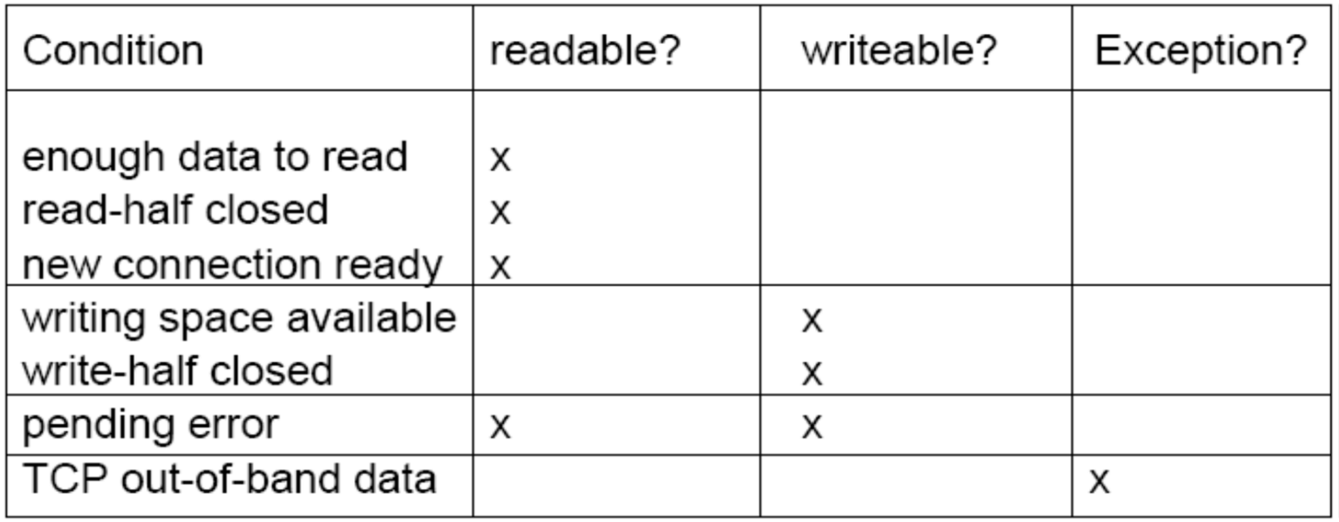
Ready for Write

- Available space in send buffer is >= low-water mark
  - Can be set using SO_SNDLOWAT
  - Default is 2048 for TCP and UDP sockets
- The write-half of the connection is closed
  - A write operation will generate SIGPIPE without blocking
- A socket using a non-blocking connect() has completed the connection, or the connect() call has failed
  - A socket error is pending
- A write operation on the socket will return an error (-1) without blocking

Low-Water Mark

- The purpose of read/write low-water marks is to give the application control over how much data must be available for reading/writing before select() returns readable or writable
  - Note : when a descriptor is writable, SO_SNDLOWAT indicates the minimum available write buffer size. One may not know how much of the buffer is actually available to be filled
  - The same holds for a read descriptor

Socket Ready Conditions for select



Condition	readable?	writeable?	Exception?
enough data to read	x		
read-half closed	x		
new connection ready	x		
writing space available		x	
write-half closed		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

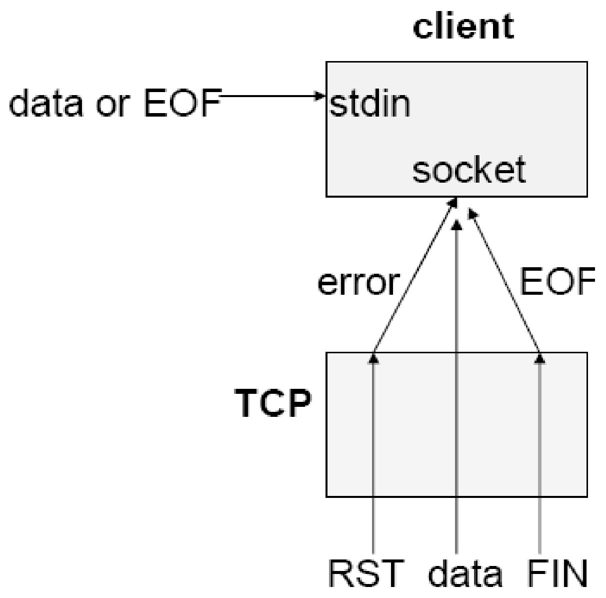
Low-Water Mark (低水位)

- •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

Process Alternative

- The use of blocking I/O and select() can achieve the similar behavior/effect of non-blocking I/O. However, there is another alternative — using processes
  - One may fork() processes and have each process handle only one direction of I/O
    - E.g., Process 1 reads from stdin (blocking) to network, and Process 2 reads from network (blocking) to stdout
    - Beware of process synchronization issues
- 我們前面用兩個迴圈設計過一次str_cli,那個版本的主要問題是client被困在等待兩 個descriptor裡面,在server已經關閉的情況下,卻完全不知情,一定要再次輸入 standard input才能發現socket descriptor的關閉.
- 下面我們使用select來構建新的版本, select設置standard input和socket兩個 descriptor哪個有可讀的I/O就可返回資料.

Rewrite *str_cli* Function with select



select for readability
on either stdin or socket

用select同時等待stdio及socket input. 當任一裝置ready for reading時select會return

當select return時怎知道是哪些裝置ready for reading呢?

用FD_ISSET測試傳回值可得知那些裝置是ready

readset, *writeset*, 和*exceptset*是雙向變數 (vale-result arguments)

- If the peer TCP sends data, the socket becomes readable, and read() returns greater than 0 (# of bytes read)
- If the peer TCP sends a FIN, the socket becomes readable and read returns 0 (EOF)
- If the peer TCP sends an RST, the socket becomes readable, read() returns -1, and error contains the specific error code

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?

Rewrite *str_cli* Function with *select*

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

只測試read

User按了
Ctrl+D

Call select

- 8–13
- We only need one descriptor set—to check for readability. This set is initialized by FD_ZERO and then two bits are turned on using FD_SET: the bit corresponding to the standard I/O file pointer, fp, and the bit corresponding to the socket, sockfd. The function fileno converts a standard I/O file pointer into its corresponding descriptor. select (and poll) work only with descriptors.
- select is called after calculating the maximum of the two descriptors. In the call, the write-set pointer and the exception-set pointer are both null pointers. The final argument (the time limit) is also a null pointer since we want the call to block until something is ready.

Handle readable socket

- 14–18
- If, on return from select, the socket is readable, the echoed line is read with readline and output by fputs.

Handle readable input

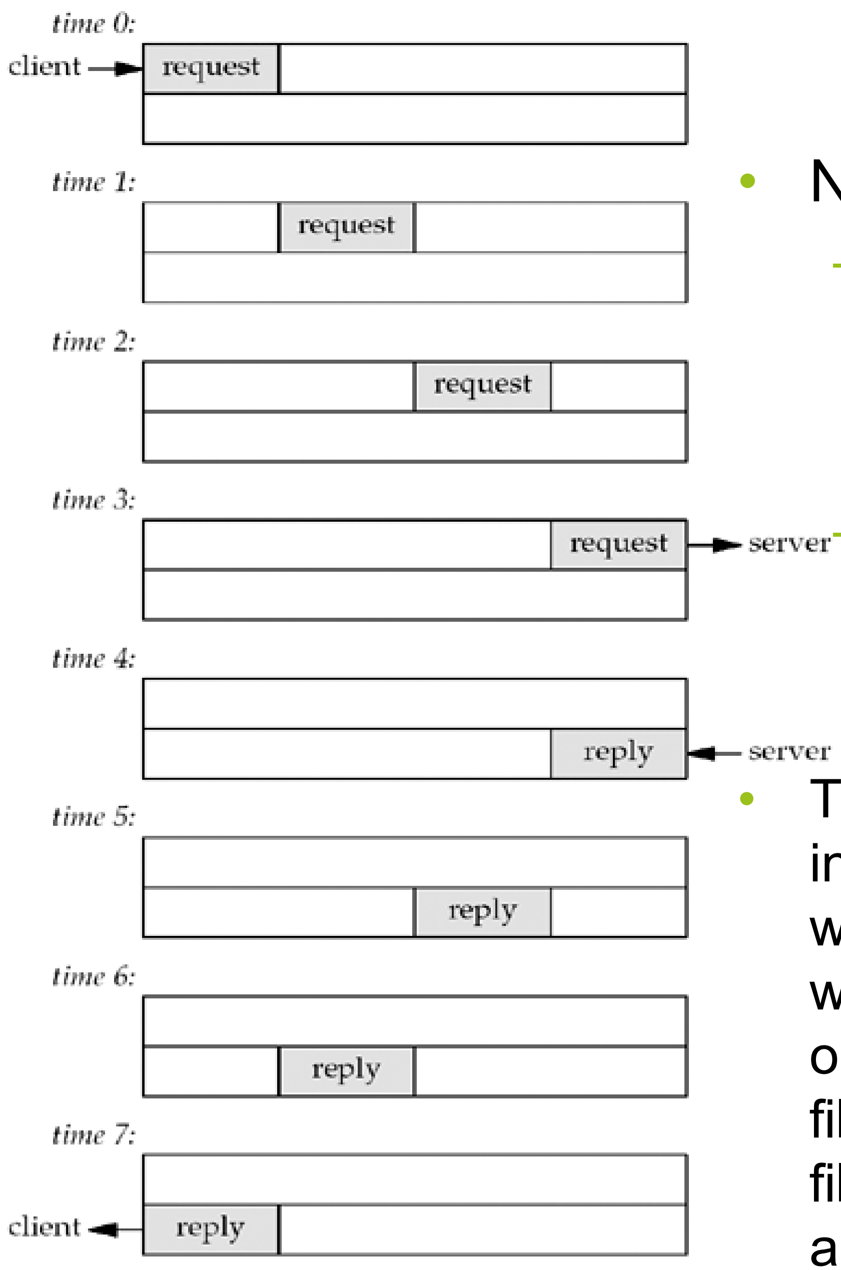
- 19–23
- If the standard input is readable, a line is read by fgets and written to the socket using writen.

Batch Input and Buffering

- Up to this moment, all versions of str_cli() functions operate in a stop-and-wait mode
  - A client sends a line to the server and wait for the reply
  - Time needed for one single request/reply is one RTT plus server's processing time (close to zero for our simple echo server model)
    - One may use the system *ping* program to measure RTTs
  - It is fine for interactive use, but not a good use of available network bandwidth
- Use batch-mode operations to better utilize the available high-speed network connections

Stop-and-wait mode

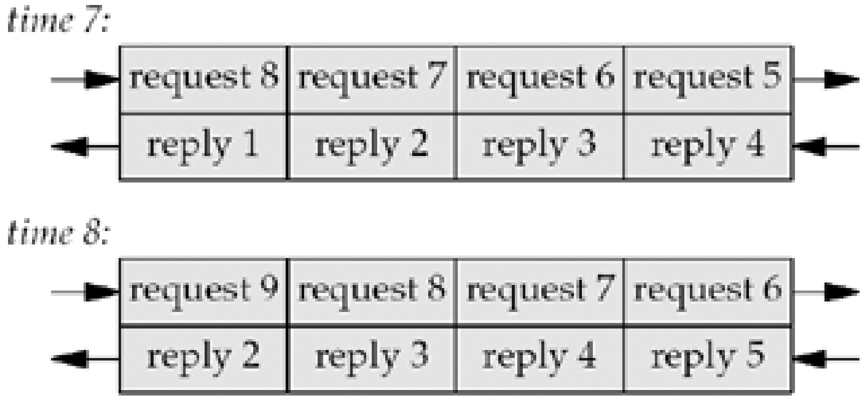
- If we consider the network between the client and server as a full-duplex pipe, with requests going from the client to the server and replies in the reverse direction, then the following figure shows our stop-and-wait mode:
  - If we consider the network between the client and server as a full-duplex pipe, with requests going from the client to the server and replies in the reverse direction, then the following figure shows our stop-and-wait mode:
- Assumptions for illustration purposes :
  - RTT = 8 units of time
  - No server process time (0)
  - Size of request = size of reply
  - Full duplex data transfers



- Note that this figure:
  - Assumes that there is no server processing time and that the size of the request is the same as the reply
  - Shows show only the data packets, ignoring the TCP acknowledgments that are also going across the network
- This stop-and-wait mode is fine for interactive input. The problem is: if we run our client in a batch mode, when we redirect the input and output, however, the resulting output file is always smaller than the input file (and they should be identical for an echo server).

Illustration of Batch Mode

- To see what's happening, realize that in a batch mode, we can keep sending requests as fast as the network can accept them. The server processes them and sends back the replies at the same rate. This leads to the full pipe at time 7, as shown below:



- We assume:
  - After sending the first request, we immediately send another, and then another
  - We can keep sending requests as fast as the network can accept them, along with processing replies as fast as the network supplies them.

Supporting Batch Input with *shutdown*

- Stop-and-wait mode (interactive) vs batch mode (redirected stdin/stdout)

```
tcpcli04 206.62.226.35 < file1.txt
```

將stdin重導至file1.txt

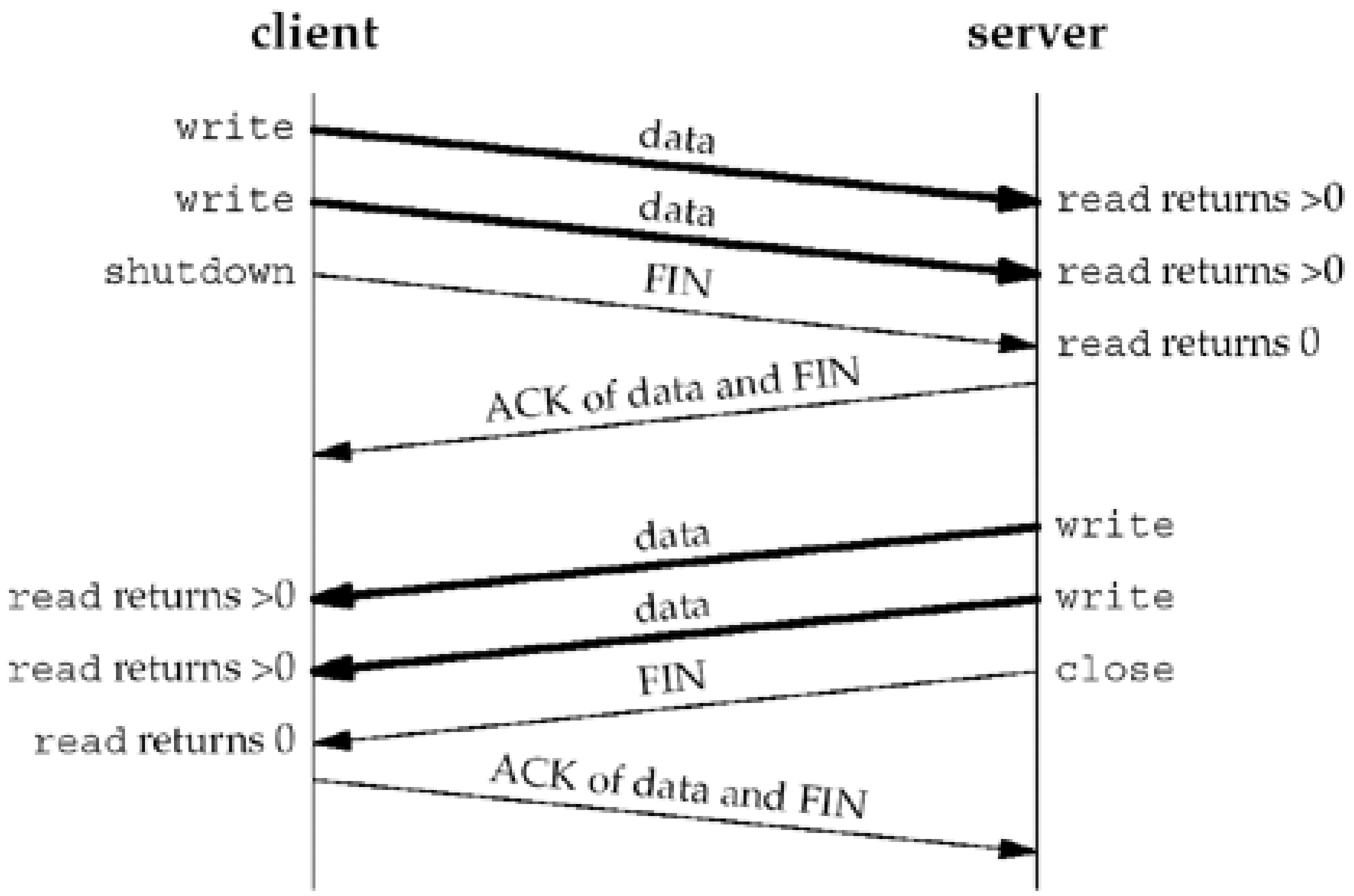
- In batch mode, *str_cli* returns right after EOF on input and *main* returns immediately, while leaving server replies unprocessed.
  - 導致server最後傳回來的部份資料遺失
- Solution: In *str_cli*, close write-half of TCP connection, by *shutdown*, while leaving readhalf open

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

close() vs. shutdown()

- The normal way to terminate a network connection is to call the close function. But, there are two limitations with close that can be avoided with shutdown:
  - close decrements the descriptor's reference count and closes the socket only if the count reaches 0 (Section 4.8). With shutdown, we can initiate TCP's normal connection termination sequence (the four segments beginning with a FIN in Figure 2.5), regardless of the reference count.
  - close terminates both directions of data transfer, reading and writing. Since a TCP connection is full-duplex, there are times when we want to tell the other end that we have finished sending, even though that end might have more data to send us. This is the scenario we encountered in the previous section with batch input to our str_cli function. The figure below shows the typical function calls in this scenario.

Closing Half a TCP Connection



The *shutdown()* Function

- Syntax

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

int  shutdown(int sockfd, int howto);

                                Returns: 0 if OK, -1 on error
```

where howto has 3 values:

- SHUT_RD
  - The read-half is closed.
- SHUT_WR
  - The write-half is closed.
- SHUT_RDWR
  - Both read and write halves are closed

The *shutdown()* Function

- The action of the function depends on the value of the howto argument:
  - SHUT_RD: The read half of the connection is closed. No more data can be received on the socket and any data currently in the socket receive buffer is discarded. The process can no longer issue any of the read functions on the socket. Any data received after this call for a TCP socket is acknowledged and then silently discarded.
  - SHUT_WR: The write half of the connection is closed. In the case of TCP, this is called a half-close. Any data currently in the socket send buffer will be sent, followed by TCP's normal connection termination sequence. As we mentioned earlier, this closing of the write half is done regardless of whether or not the socket descriptor's reference count is currently greater than 0. The process can no longer issue any of the write functions on the socket.
  - SHUT_RDWR: The read half and the write half of the connection are both closed. This is equivalent to calling shutdown twice: first with SHUT_RD and then with SHUT_WR.

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 stdio

Rewrite *str_cli* with select and shutdown

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));
                }
        }
}
```

client至server
方向連結已斷

只斷client至
server方向連結

Rewrite *str_cli* with select and shutdown

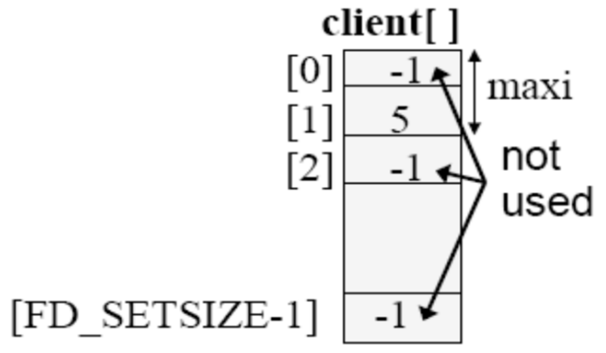
- 5–8
  - stdineof is a new flag that is initialized to 0. As long as this flag is 0, each time around the main loop, we select on standard input for readability.
- 17–25
  - When we read the EOF on the socket, if we have already encountered an EOF on standard input, this is normal termination and the function returns. But if we have not yet encountered an EOF on standard input, the server process has prematurely terminated. We now call read and write to operate on buffers instead of lines and allow select to work for us as expected.

Rewrite *str_cli* with select and shutdown

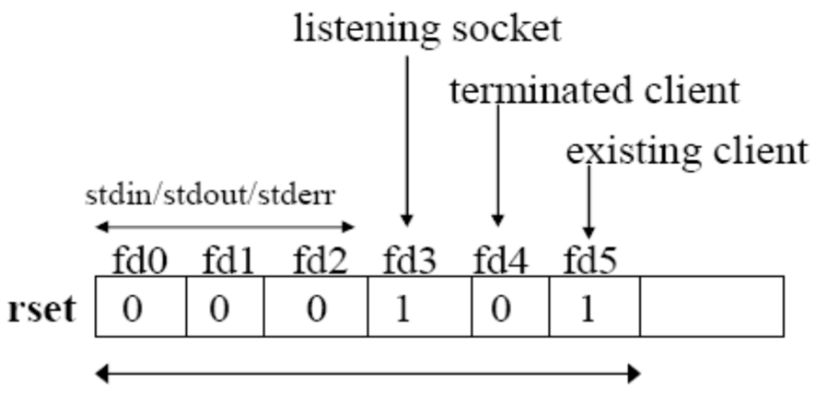
- 26–34
  - When we encounter the EOF on standard input, our new flag, stdineof, is set and we call shutdown with a second argument of SHUT_WR to send the FIN. Here also, we've changed to operating on buffers instead of lines, using read and writen.

Concurrent TCP Echo Server with select

- A single server process using select to handle any number of clients (It is not using fork child process)
- 引入了select, 就可以在server端使用select來管理所有的socket(因 為socket本質上是file). 使用了select的話,多進程(fork)就變得不必要了
- Need to keep track of the clients by *client*[ ] (client descriptor array) and *rset* (read descriptor set)



儲存已open socket的descriptor



maxfd +1 = 6 用於呼叫select

Rewrite Concurrent TCP Echo Server with *select*

tcpcliserv/tcpservselect01.c

Initialization

```
#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);
```

Rewrite Concurrent TCP Echo Server with *select*

tcpcliserv/tcpservselect01.c

The steps to create the listening socket are the same as seen earlier: socket, bind, and listen. We initialize our data structures assuming that the only descriptor that we will select on initially is the listening socket.

```
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);
```

select waits for something to happen: either the establishment of a new client connection or the arrival of data, a FIN, or an RST on an existing connection.

tcpcliserv/tcpservselect01.c

新連結

放入第一個找到的空格

If the listening socket is readable, a new connection has been established. We call accept and update our data structures accordingly.

```
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 */

}
```

Loop (cont.)

tcpcliserv/tcpservselect01.c

```
for (i = 0; i <= maxi; i++) { /* check all clients for data */
        if ( (sockfd = client[i]) < 0)
                continue;                    跳過沒有放descriptor的空格
                if (FD_ISSET(sockfd, &rset)) {
                        if ( (n = Readline(sockfd, line, MAXLINE)) == 0) {
                                /* connection closed by client */
                        Close(sockfd);
                        FD_CLR(sockfd, &allset);    從要test的
                        client[i] = -1;             descriptor set中
                        } else              變空格   去除
                                Writen(sockfd, line, n);
                        if (--nready <= 0)
                                break; /* no more readable descriptors */
                        }       如果沒有其它ready的
                                descriptor,可以提早離開loop
                }
        }
}
```

A test is made for each existing client connection as to whether or not its descriptor is in the descriptor set returned by select. If so, a line is read from the client and echoed back to the client. If the client closes the connection, read returns 0 and we update our data structures accordingly.

Loop (cont.)

- **Block in select**
  - 26–27 select waits for something to happen: either the establishment of a new client connection or the arrival of data, a FIN, or an RST on an existing connection.
- **accept new connections**
  - 28–45 If the listening socket is readable, a new connection has been established. We call accept and update our data structures accordingly. We use the first unused entry in the client array to record the connected socket. The number of ready descriptors is decremented, and if it is 0, we can avoid the next for loop. This lets us use the return value from select to avoid checking descriptors that are not ready.

Loop (cont.)

- **Check existing connections**
  - 46–60 A test is made for each existing client connection as to whether or not its descriptor is in the descriptor set returned by select.
  - If so, a line is read from the client and echoed back to the client. If the client closes the connection, read returns 0 and we update our data structures accordingly.
  - We never decrement the value of maxi, but we could check for this possibility each time a client closes its connection.

Denial of Service Attacks

- Weakness of the TCP echo server — vulnerable to denial-of-service (DOS) attacks
- Attack scenario :
  - A malicious client sends 1 byte of data (without a newline)
  - Server hangs until the client either sends a newline or terminates
- Possible solutions :
  - Use nonblocking I/O for the server listening socket
  - Have each client served by a separate process/thread
  - Place a timeout on the I/O operations

pselect Function

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

/* Returns: count of ready descriptor, 0 on timeout, -1 on error */
int pselect(int maxfdp1, fd_set* readset, fd_set* writeset, fd_set*
exceptset,
            const struct timespec* timeout, const sigset_t *sgmask);
```

- 和普通的select相比pselect有如下變化:
  - 使用了timespec,而不是timeval, tv_usec將更加精確
  - 增加了第六個參數:指向signal mask的指針:因為select有block的可能,在select被 block的時候,可能會有signal丟失,或者signal會打擾select的block:如要不丟失signal,又不中斷select,就要用到sigprocmask:
  - 從某種意義上來講,pselect就是收到sigprocmask保護的select

pselect Function: Avoiding Signal Loss in Race Condition

```
#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 */ };
```

SIGINT occurs in here

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

signal lost if the signal occurs between the test of *intr_flag* and the call to *select*

```
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( );
          }
....}
```

More on Servers

- A server may be designed/developed in several different ways :
  - Iterative
  - Concurrent
  - Preforked
  - Threaded
  - Prethreaded

Iterative Servers

- Loop around to serve only one client at a time
  - Other clients block while one is being serviced
- It is simple to develop one, but limited usefulness
  - Only useful for very simple services where the time to serve a client request is very short (e.g., a daytime server)

Concurrent Servers

- Use fork() to serve each client
  - The server will not wait for a client to finish before it starts serving another client
  - There will be some operating system overhead for fork()
- Good for “medium load” servers
  - A lot of servers are programmed this way

Preforked Servers

- On startup, a server fork()s a configured number of worker processes.
  - When a client connection request arrives, it will be served by an already fork()ed process right away
- Good for a “heavy load” server
- The apache Web server is general configured this way

Threaded Servers

- Another type of concurrent servers
  - Create a thread, instead of using fork(), to handle a client connection request
- Threads have much lower overhead than processes
  - Threads are also called light weight processes (LWP)
- Threaded servers may not be portable
  - Not all systems support threads

Prethreaded Servers

- Similar concept as preforked servers
  - Precreate a configured number of worker threads, instead of processes, upon startup
    - Prethreading amortize the overhead of thread creations all at once, similar to preforking
- Threaded servers have lower overhead in creation time, but are more complex to deal with than forked servers
  - Thread synchronization is a major concern in design

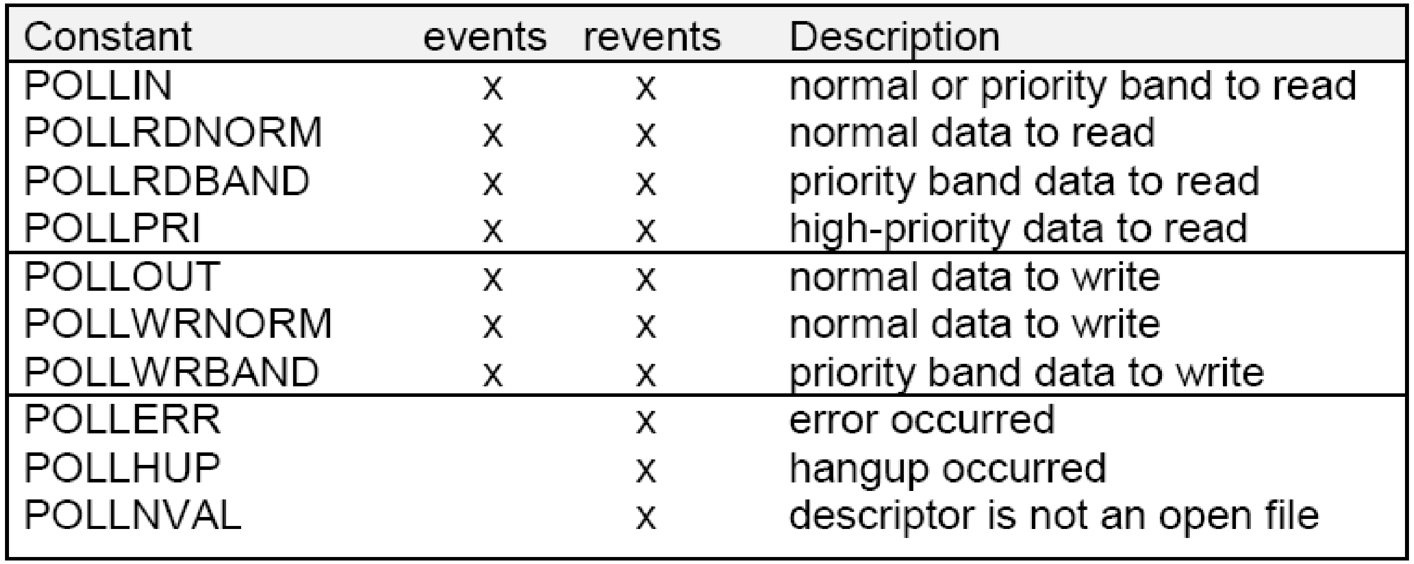
poll Function: polling more specific conditions than select

```
#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

```
struct pollfd {               -1表此結構無效
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



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 to write
POLLWRNORM	x	x	normal data to write
POLLWRBAND	x	x	priority band data to write
POLLERR		x	error occurred
POLLHUP		x	hangup occurred
POLLNVAL		x	descriptor is not an open file

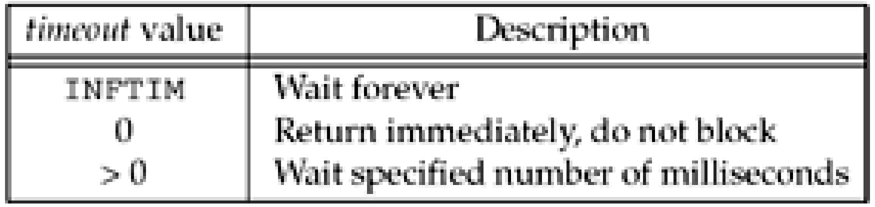
- We have divided this figure into three sections:
  - The first four constants deal with input,
  - the next three deal with output, and
  - the final three deal with errors.
- Notice that the final three cannot be set in events, but are always returned in revents when the corresponding condition exists.

Three classes of data identified by poll

- 需要測試的condition設置在events, poll返回的時候,會把fd的status資訊寫入到 revents成員裡面.(使用兩個成員變數的方法,防止了“調用的時候設置,返回的時候複 寫記憶體”的方法),下面是這兩個condition可能的值:分成了三部分:讀,寫,錯誤資訊.
  - POLLIN can be defined as the logical OR of POLLRDNORM and POLLRDBAND. The POLLIN constant exists from SVR3 implementations that predated the priority bands in SVR4, so the constant remains for backward compatibility. Similarly,
  - POLLOUT is equivalent to POLLWRNORM, with the former predating the latter.

Three Classes of Data Identified by poll

The constant INFTIM is defined to be a negative value. If the system does not provide a timer with millisecond accuracy, the value is rounded up to the nearest supported value.
The POSIX specification requires that INFTIM be defined by including <poll.h>, but many systems still define it in <sys/stropts.h>.
As with select, any timeout set for poll is limited by the implementation's clock resolution (often 10 ms).



timeout value	Description
INFTIM	Wait forever
0	Return immediately, do not block
> 0	Wait specified number of milliseconds

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 structured.
- 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

```
#include "unp.h"                                        tcpcliserv/tcpservpoll01.c
#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];     pullfd結構陣列
        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);
```

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
listen socket, 以POLLRDNPRM事件listen

Call poll, check for new connection

call poll to wait for either a new connection or data on existing connection. When a new connection is accepted, we find the first available entry in the client array by looking for the first one with a negative descriptor.

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 */
}
```

放入第一個
找到的空格

for data socket

Check for data on an existing connection

The two return events that we check for are POLLRDNORM and POLLERR. The second of these we did not set in the events member because it is always returned when the condition is true.

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 */
} } } }
```

Summary

- There are five different models for I/O provided by Unix:
  - Blocking
  - Nonblocking
  - I/O multiplexing
  - Signal-driven I/O
  - Asynchronous I/O
- The default is blocking I/O, which is also the most commonly used.
- We have covered I/O multiplexing in this week.
- True asynchronous I/O is defined by the POSIX specification, but few implementations exist.

Summary

- The most commonly used function for I/O multiplexing is select.
- We tell the select function what descriptors we are interested in (for reading, writing, and exceptions), the maximum amount of time to wait, and the maximum descriptor number (plus one).
- Most calls to select specify readability, and we noted that the only exception condition when dealing with sockets is the arrival of out-of-band data.
- Since select provides a time limit on how long a function blocks.

Summary

- We used our echo client in a batch mode using select and discovered that even though the end of the user input is encountered, data can still be in the pipe to or from the server.
- To handle this scenario requires the shutdown function, and it lets us take advantage of TCP's half-close feature.
- The dangers of mixing stdio buffering (as well as our own readline buffering) with select caused us to produce versions of the echo client and server that operated on buffers instead of lines.

Summary

- POSIX defines the function pselect, which increases the time precision from microseconds to nanoseconds and takes a new argument that is a pointer to a signal set. This lets us avoid race conditions when signals are being caught.
- The poll function from System V provides functionality similar to select and provides additional information on STREAMS devices. POSIX requires both select and poll, but the former is used more often.