IO Multiplexing

Sanjaya Kumar Jena

ITER, Bhubanewar

Text Book(s)

W. Richard Stevens, Bill Fenner, & Andrew M. Rudoff

Unix Network Programming

The Sockets Networking API

Volume-1, Third Edition

Introduction

- **I/O multiplexing**- The capability to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e., input is ready to be read, or the descriptor is capable of taking more output).
- I/O multiplexing is provided by the **`select`** and **`poll`** functions. Other variants are **`pselect`** and **`ppoll`** functions

I/O multiplexing is typically used in networking applications in the following scenarios:

- When a client is handling multiple descriptors (normally interactive input and a network socket).
- It is possible, but rare, for a client to handle multiple sockets at the same time.
- 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 (e.g., the `inetd` daemon).

I/O multiplexing is not limited to network programming. Many nontrivial applications find a need for these techniques.

I/O Models

The five I/O models that are available to us under Unix:

- ✍ blocking I/O
- ✍ nonblocking I/O
- ✍ I/O multiplexing (select and poll)
- ✍ signal driven I/O (SIGIO)
- ✍ asynchronous I/O (the POSIX aio_ functions)

`read` System Call

- UNIX provides sequential access to files and other devices through the **`read`** and **`write`** functions.
- The **`read`** function attempts to retrieve **`nbyte`** bytes from the file or device represented by **`fildes`** into the user variable **`buf`**.
- A large enough buffer must be provided to hold **`nbyte`** bytes of data.
- SYNOPSIS:

```
#include <unistd.h>

ssize_t read(int fildes, void *buf, size_t nbyte);


(1)  If successful, read returns the number of bytes
     actually read.
(2)  If unsuccessful, read returns -1 and sets errno.
```

- The **`ssize_t`** data type is a signed integer data type used for the number of bytes read, or -1 if an error occurs.
- The **`size_t`** is an unsigned integer data type for the number of bytes to read.

Note: `Read`

- A **`read`** operation for a regular file may return fewer bytes than requested if, for example, it reached end-of-file before completely satisfying the request.
- A **`read`** operation for a regular file returns 0 to indicate end-of-file.
- When reading from a terminal, read returns 0 when the user enters an end-of-file character(CTRL+D).

write System Call

- The **write** function attempts to output **nbyte** bytes from the user buffer **buf** to the file represented by file descriptor **fildes**.
- SYNOPSIS:

```
#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);


 (1)  If successful, write returns the number of bytes
      actually written.
 (2)  If unsuccessful, write returns -1 and sets errno.
```

open System Call

- The open function associates a file descriptor with a file or physical device.
- SYNOPSIS:

```
#include <fcntl.h>
#include <sys/stat.h>

int open(const char *path, int oflag, ...);

(1) If successful, open returns a nonnegative integer
    representing the open file descriptor.

(2) If unsuccessful, open returns -1 and sets errno.
```

- The `path` parameter of open points to the pathname of the file or device.
- The `oflag` parameter specifies status flags and access modes for the opened file.
- A third parameter must be included to specify access permissions if a file is created.

File Descriptor

- Files are designated with in C program either by **file pointers** or by **file descriptor**.
- A file **descriptor** is an integer value that represents a file or device that is open.
- It is an index into the process file descriptor table.
- The file descriptor table is in the process user area and provides access to the system information for the associated file or device.
- File pointers and file descriptors provide logical designations called *handles* for performing device-independent input and output.

File Descriptor

- ✍ The symbolic names for the **file pointers** that represent standard input, standard output and standard error are **`stdin`**, **`stdout`** and **`stderr`**, respectively. These symbolic names are defined in **`stdio.h`**.

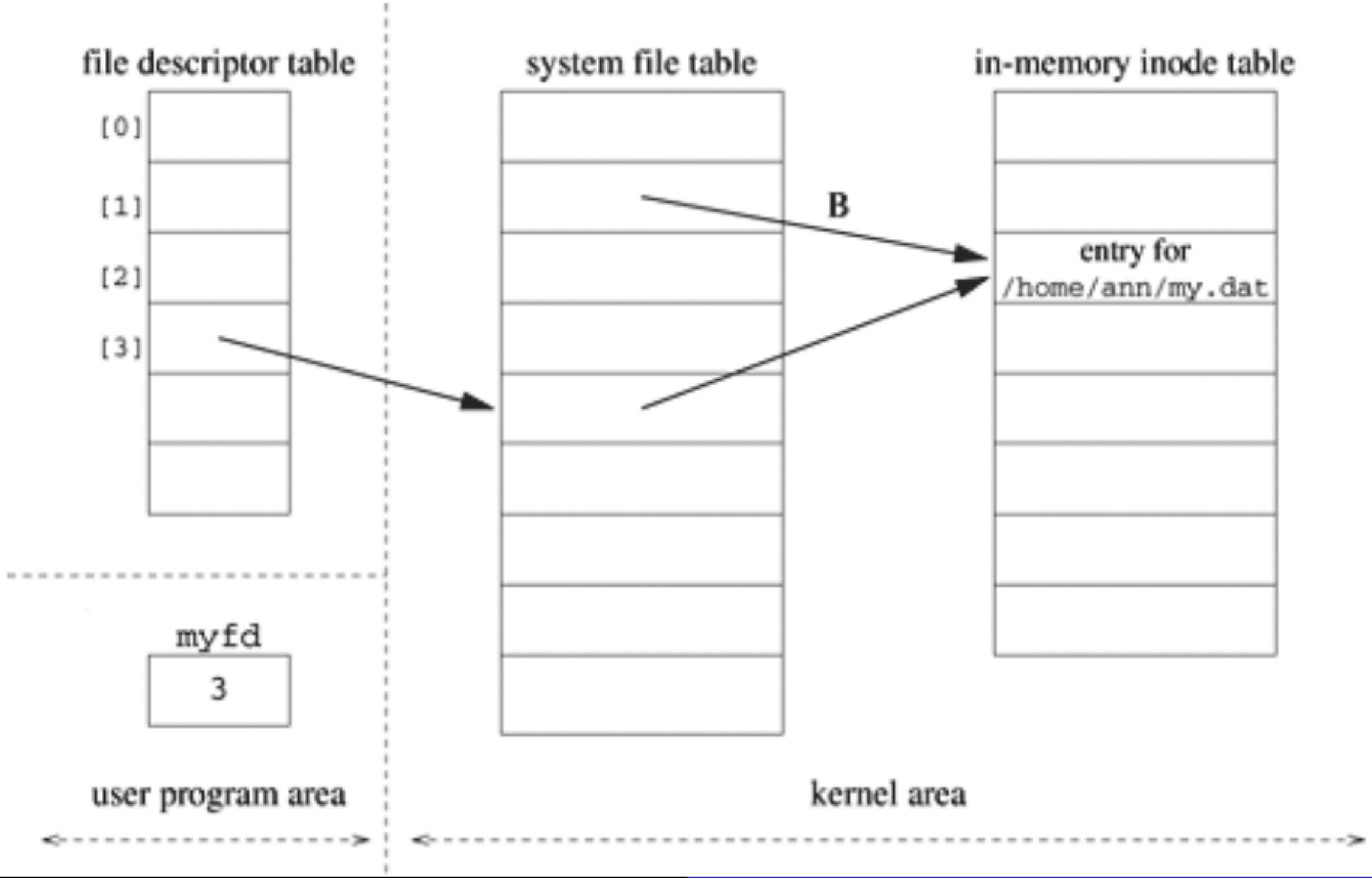
- ✍ The symbolic names for the **file descriptors** that represent standard input, standard output and standard error are **`STDIN_FILENO`**, **`STDOUT_FILENO`** and **`STDERR_FILENO`**, respectively. These symbolic names are defined in **`unistd.h`**.

- ✍ The **numeric values** that represent standard input, standard output and standard error are 0, 1, and 2 respectively.

File Descriptor

A schematic of the file descriptor table after a program executes the following.

```
myfd = open("/home/ann/my.dat", O_RDONLY);
```

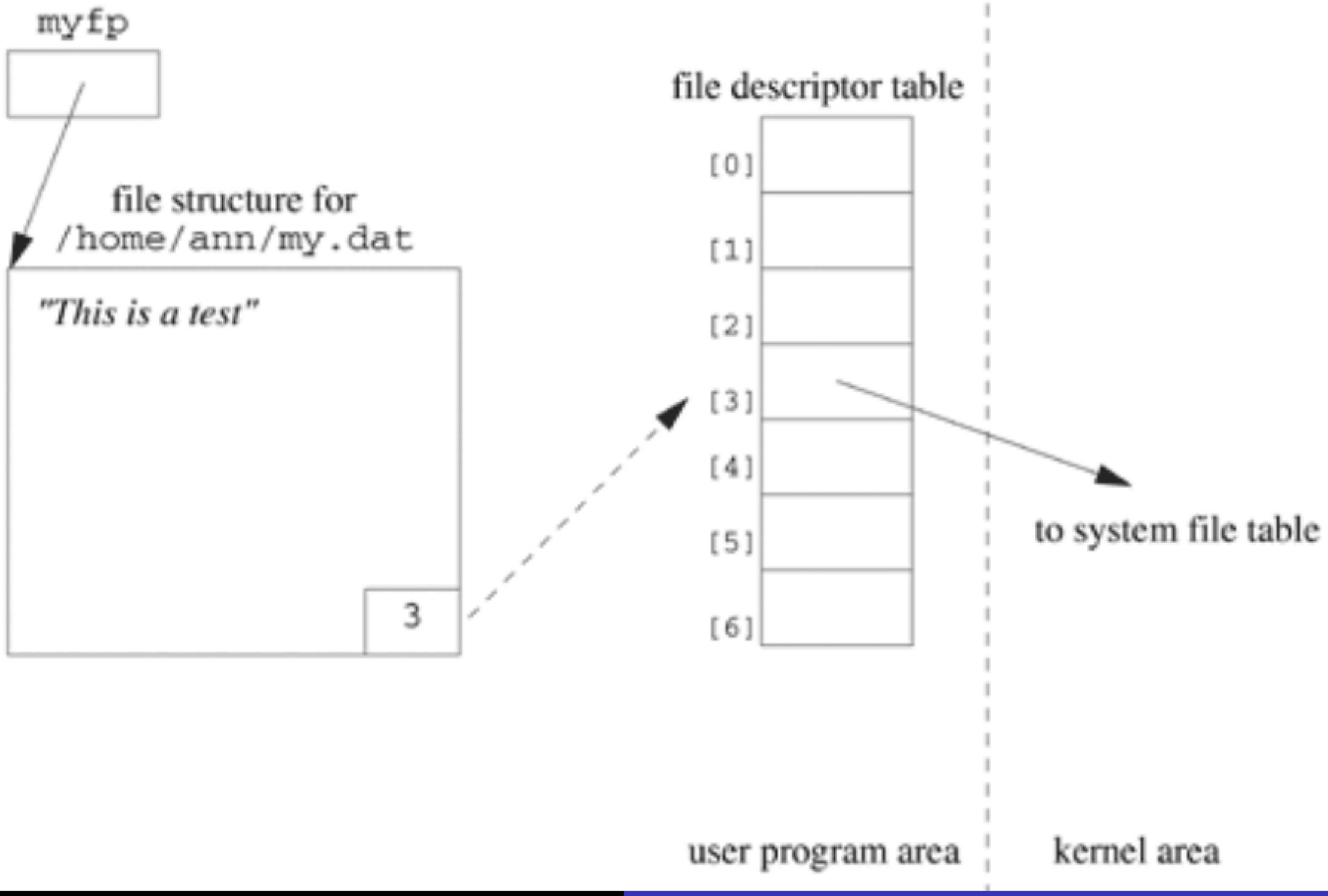


File Descriptor

- The **`open`** function creates an entry in the file descriptor table that points to an entry in the system file table.
- The **`open`** function returns the value 3, specifying that the file descriptor entry is in position three of the process file descriptor table.
- The system file table, which is shared by all the processes in the system, has an entry for each active **`open`**.

File Pointer

```
FILE *myfp;
if ((myfp = fopen("/home/ann/my.dat", "w")) == NULL)
perror("Failed to open /home/ann/my.dat");
else
fprintf(myfp, "This is a test");
```



`oflag` Argument Setting

✍ The POSIX values for the **access mode flags** in **`oflag`** argument:

1. **`O_RDONLY`** : read-only access
2. **`O_WRONLY`** : write-only access
3. **`O_RDWR`** : read-write-only access

Note: specify exactly one of above designating read-only, write-only or read-write access.

✍ The **`oflag`** argument is also constructed by taking the bitwise OR (|) of the desired combination of the access mode and the **additional flags**.

✍ The additional flags:

1. **`O_APPEND`**
2. **`O_CREAT`**
3. **`O_EXCL`**
4. **`O_NOCTTY`**
5. **`O_NONBLOCK`**
6. **`O_TRUNC`**

Additional Flags

`O_APPEND`: The O_APPEND flag causes the file offset to be moved to the end of the file before a write, allowing you to add to an existing file.

`O_CREAT`: The O_CREAT flag causes a file to be created if it doesn't already exist. If O_CREAT flag is included, a third argument to **`open()`** must be passed to designate the permissions.

`O_EXCL`: If you want to avoid writing over an existing file, use the combination O_CREAT | O_EXCL. This combination returns an error if the file already exists.

`O_NOCTTY`: The O_NOCTTY flag prevents an opened device from becoming a controlling terminal.

`O_NONBLOCK`: The O_NONBLOCK flag controls whether the **`open()`** returns immediately or blocks until the device is ready.

`O_TRUNC`: O_TRUNC truncates the length of a regular file opened for writing to 0.

open() Examples

The following code segment opens the file /home/students/my.dat for reading

```
#include <fcntl.h>
#include <sys/stat.h>

int myfd;
myfd = open("/home/students/my.dat", O_RDONLY);
```

How would you modify above Example to open /home/ann/my.dat for nonblocking read?

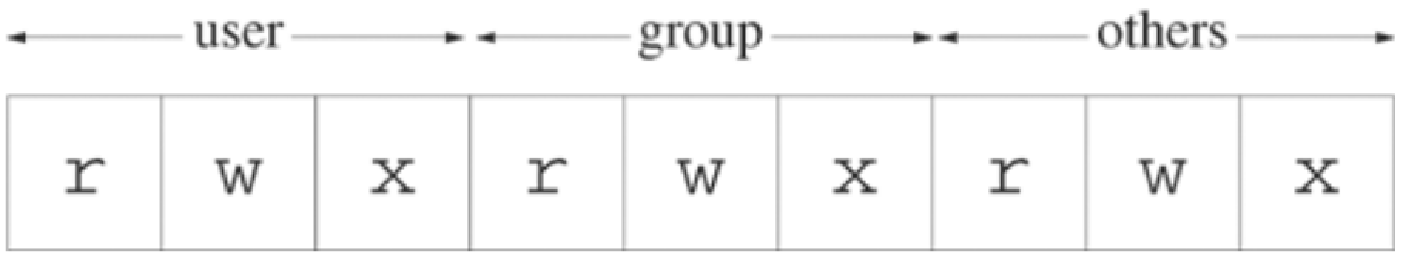
```
Perform OR the O_RDONLY and the O_NONBLOCK flags.

myfd = open("/home/students/my.dat", O_RDONLY |
    O_NONBLOCK);
```

Third argument to `open()`

- ☞ Each file has three classes associated with it: a user (or owner), a group and everybody else (others).
- ☞ The possible permissions or privileges are read(r), write(w) and execute(x). These privileges are specified separately for the user, the group and others.
- ☞ When you open a file with the O_CREAT flag, you must specify the permissions as the third argument to open in a mask of type mode_t.
- ☞ POSIX defines symbolic names for masks corresponding to the permission bits. These names are defined in **sys/ stat.h**

POSIX symbolic names for file permissions



Symbol	meaning
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute by owner
S_IRWXU	read, write, execute by owner
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXG	read, write, execute by group
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute by others
S_IRWXO	read, write, execute by others
S_ISUID	set user ID on execution
S_ISGID	set group ID on execution

Example

The following code segment creates a file, info.dat, in the current directory. If the info.dat file already exists, it is overwritten. The new file can be read or written by the user and only read by everyone else.

```
int fd;

mode_t fdmode = (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

if ((fd = open("info.dat", O_RDWR | O_CREAT, fdmode)) ==
    -1)
    perror("Failed to open info.dat");
```

`close()` System Call

The close function has a single parameter, **`fildes`**, representing the open file whose resources are to be released.

```
#include <unistd.h>

int close(int fildes);

(1) If successful, close returns 0.
(2) If unsuccessful, close returns -1 and sets errno.
```

Monitoring Multiple file Descriptor

A case: when a program expects input from two different sources, but it doesn't know which input will be available first. If the program tries to read from source A, and in fact, input was only available from source B, the program blocks. To solve this problem, we need to block until input from either source becomes available. Blocking until at least one member of a set of conditions becomes true is called *OR synchronization*. The condition for the case described is "input available" on a descriptor.

Monitoring Method

1. To monitor multiple file descriptors is to use a separate process for each one.
2. Using system calls: ***select***, and `poll`

Monitoring Through Separate Process

The parent process opens both files before creating the child process. The parent monitors the first file descriptor, and the child monitors the second.

Example

```
int main(int argc, char *argv[]) {
   int bytesread, childpid, fd, fd1, fd2;
   if (argc != 3) {
      fprintf(stderr, "Usage: %s file1 file2\n", argv[0]);
      return 1;
   }
   if ((fd1 = open(argv[1], O_RDONLY)) == -1) {
      fprintf(stderr, "Failed to open file %s:%s\n", argv[1], strerror(errno));
      return 1;
   }
   if ((fd2 = open(argv[2], O_RDONLY)) == -1) {
      fprintf(stderr, "Failed to open file %s:%s\n", argv[2], strerror(errno));
      return 1;
   }
   if ((childpid = fork()) == -1) {
      perror("Failed to create child process");
      return 1;
   }
   if (childpid > 0)    /* parent code */
      fd = fd1;
   else
      fd = fd2;
   bytesread = copyfile(fd, STDOUT_FILENO);
   fprintf(stderr, "Bytes read: %d\n", bytesread);
   return 0;
}
```

`select()` System Call

- ✍ The **`select`** call provides a method of monitoring file descriptors from a single process.
- ✍ **`select`** allow a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become ready for some class of IO operation.
- ✍ A file descriptor is considered ready if it is possible to perform the corresponding IO operation(i.e. read without blocking)
- ✍ **`select`** allows the process to instruct the kernel to 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`** can monitor for three possible conditions - a read can be done without blocking, a write can be done without blocking, or a file descriptor has error conditions pending.
- ✍ **`select`** watches three independent sets of file descriptors such as (1) readfds (2) writefds (3) exceptfds
- ✍ Four macros **`FD_SET`**, **`FD_CLR`**, **`FD_ISSET`** and **`FD_ZERO`** are used to manipulate the descriptor sets in an implementation-independent way.

Prototype of `select()` and macros

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

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

- On successful return, **`select`** clears all the descriptors in each of **`readfds`**, **`writefds`** and **`exceptfds`** except those descriptors that are ready.
- If successful, the **`select`** function returns the number of file descriptors that are ready.
- If unsuccessful, **`select`** returns -1 and sets `errno`.

Macros

```
void FD_CLR(int fd, fd_set *fdset);

int FD_ISSET(int fd, fd_set *fdset);

void FD_SET(int fd, fd_set *fdset);

void FD_ZERO(fd_set *fdset);
```

How To Use `select` System call?

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

- ✍ The **`nfds`** parameter of select gives the range of file descriptors to be monitored. The value of **`nfds`** must be at least one greater than the largest file descriptor to be checked.
- ✍ The **`readfds`** parameter specifies the set of descriptors to be monitored for reading.
- ✍ **`writefds`** specifies the set of descriptors to be monitored for writing,
- ✍ **`exceptfds`** specifies the file descriptors to be monitored for error conditions.
- ✍ The descriptor sets are of type **`fd_set`**.
- ✍ Any of these parameters may be **`NULL`**, in which case select does not monitor the descriptor for the corresponding event.
- ✍ The last parameter is a **`timeout`** value that forces a return from **`select`** after a certain period of time has elapsed, even if no descriptors are ready. When timeout is **`NULL`**, **`select`** may block indefinitely.

Example to use `select()`

Let us consider that **`select()`** to monitor readfds, and writefds file descriptors.

call to `select`::

```
fd_set readfds; /* reading fds et*/
fd_set writefds; /* writing fd set */
int nfds;
...
...
select(nfds, &readfds, &writefds, NULL, NULL);
```

`select` prototype::

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

Setting The Four Macros

A design problem is how to specify one or more descriptor values for each of these three arguments.

- **`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.
- 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 ? */
```

Macros Description

- ✍ The **`FD_SET`** macro sets the bit in **`fdset`** corresponding to the **`fd`** file descriptor.
- ✍ The **`FD_CLR`** macro clears the corresponding bit.
- ✍ The **`FD_ZERO`** macro clears all the bits in **`fdset`**.
- ✍ Use these three macros to set up descriptor(s) before calling **`select`**.
- ✍ Use the **`FD_ISSET`** macro after select returns, to test whether the bit corresponding to the file descriptor **`fd`** is set or not.

`fd_set` data type

- ✍ The file descriptor sets for the **`select`** function are specified as **`fd_set`** objects.
- ✍ The **`fd_set`** data type represents file descriptor sets for the **`select`** function. It is actually a bit array.
- ✍ So, allocate a descriptor set of the **`fd_set`** data type.
- ✍ **Example:** To define a variable of type **`fd_set`** is **`fd_set rdset;`**.
- ✍ The four macros are used to set and test the bits in the set.
- ✍ Like ordinary variable assignment, we can also assign it to another descriptor set across an equals sign (=).

Example: Define a variable of the type **`fd_set`** and turn on the bits for descriptors 1, 4, 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 */
```

`select`: The Timeout Argument

- The timeout argument (the last argument in select) tells the kernel how long to wait for one of the specified descriptors to become ready.
- A timeval structure defined in `<time.h>` specifies the number of seconds and microseconds.
- The predefined **`timeval`** structure:

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

- There are three possibilities:
  - **Wait forever:** set the timeout argument as **`NULL`**.
  - **Wait up to a fixed amount of time:**

```
struct timeval tv
  tv.tv_sec=5;      /* seconds */
  tv. tv_usec=0;     /* microseconds */
  ...
select(----, ----, ----, ----, &tv);
```

  - **Do not wait at all:** set tv_sec and tv_usec to Zero.

```
select(---, ---, ---, ---, &tv);
```

`select`: The Descriptor Sets Argument

- The three middle arguments, **`readfds`**, **`writefds`**, and **`exceptfds`**, specify the descriptors that we want the kernel to test for reading, writing, and exception conditions.
- Let **`readfds`** to be tested;
```
fd_set readfds;
select(---, &readfds, ---, ---, ---);
```
- Let **`writefds`** to be tested;
```
fd_set writefds;
select(---, ---, &writefds, ---, ---);
```
- Let both **`readfds`** and **`writefds`** to be tested;
```
fd_set writefds;
select(---, &readfds, &writefds, ---, ---);
```
- Let **`exceptfds`** not to be tested;
```
select(---, ---,---, NULL, ---);
```
- Let **`readfds`** to be tested and right to others not
```
fd_set readfds; int maxfdp1;
select(maxfdp1, &readfds, NULL, NULL, NULL);
```
- Simillarly set the descriptor sets on requirement.

`select`: The First Argument

- ✍ The first argument to **`select`** specifies the number of descriptors to be tested.
- ✍ Its value is the maximum descriptor to be tested plus one.
- ✍ The descriptors 0, 1, 2, up through and including last one are to be tested.
- ✍ For example, If indicators for descriptors 1, 4, and 5 are turn on the , then first argument value of **`select`** will be value 6.
- ✍ The reason it is 6 and not 5 is that we are specifying the number of descriptors, not the largest value, and descriptors start at 0.

  so, **`select(6, ---,---,--, ---);`**

A Sample Program Using `Select`

```
int main(void)
{
   fd_set rfds;
   struct timeval tv;
   int retval;
   /* Watch stdin (fd 0) to see when it has input. */
   FD_ZERO(&rfds);
   FD_SET(0, &rfds);
   /* Wait up to five seconds. */
   tv.tv_sec = 5;
   tv.tv_usec = 0;
   retval = select(1, &rfds, NULL, NULL, &tv);
   /* Don't rely on the value of tv now! */
   if (retval == -1)
      perror("select()");
   else if (retval)
      printf("Data is available now.\n");
      /* FD_ISSET(0, &rfds) will be true. */
   else
      printf("No data within five seconds.\n");
return 0;
}
```

Sample Program on `select`

A function that blocks until one of two file descriptors is ready.

```
int whichisready(int fd1, int fd2) {
    int maxfd, nfds;
    fd_set readset;
    maxfd = (fd1 > fd2) ? fd1 : fd2;
    FD_ZERO(&readset);
    FD_SET(fd1, &readset);
    FD_SET(fd2, &readset);
    nfds = select(maxfd+1, &readset, NULL, NULL, NULL);
    if (nfds == -1)
        return -1;
    if (FD_ISSET(fd1, &readset))
        return fd1;
    if (FD_ISSET(fd2, &readset))
        return fd2;
    return -1;
}
```

The function **`whichisready`** blocks until at least one of the two file descriptors passed as parameters is ready for reading and returns that file descriptor. If both are ready, it returns the first file descriptor. If unsuccessful, **`whichisready`** returns -1.

Monitoring standard input and Socket

Rewrite **`str_cli`** function using **select** to notify as soon as the server process terminates.

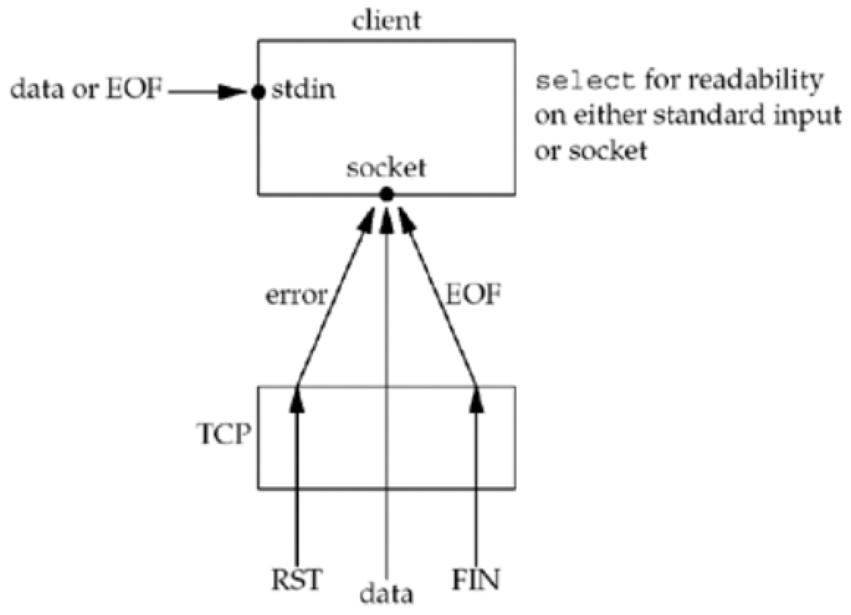


Figure: Conditions handled by **`select`** in **`str_cli`**

Three conditions are handled with the socket:

- ✍ If the peer TCP sends data, the socket becomes readable and read returns greater than 0 (i.e., the number of bytes of data).
- ✍ If the peer TCP sends a FIN (the peer process terminates), the socket becomes readable and read returns 0 (EOF).
- ✍ If the peer TCP sends an RST (the peer host has crashed and rebooted), the socket becomes readable, read returns 1, and errno contains the specific error code.

Implementation of `str_cli` function using `select`

```
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);
  if (FD_ISSET(sockfd, &rset)) { /* socket is readable */
     if (read(sockfd, recvline, MAXLINE) == 0){
        perror("str_cli: server terminated prematurely");
        return;
     }
    fputs(recvline, stdout);
 }
 if (FD_ISSET(fileno(fp), &rset)) { /* input is readable */
     if (fgets(sendline, MAXLINE, fp) == NULL)
       return;    /* all done */
    write(sockfd, sendline, strlen(sendline));
 }
}}
```

Batch Input and Buffering

- our **`str_cli`** function operates in a stop-and-wait mode, which is fine for interactive use: It sends a line to the server and then waits for the reply.
- This amount of time is one RTT plus the server's processing time (which is close to 0 for a simple echo server).
- We can therefore estimate how long it will take for a given number of lines to be echoed if we know the RTT between the client and server.(The ping program is an easy way to measure RTTs.)
- 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 Figure below shows our stop-and-wait mode.

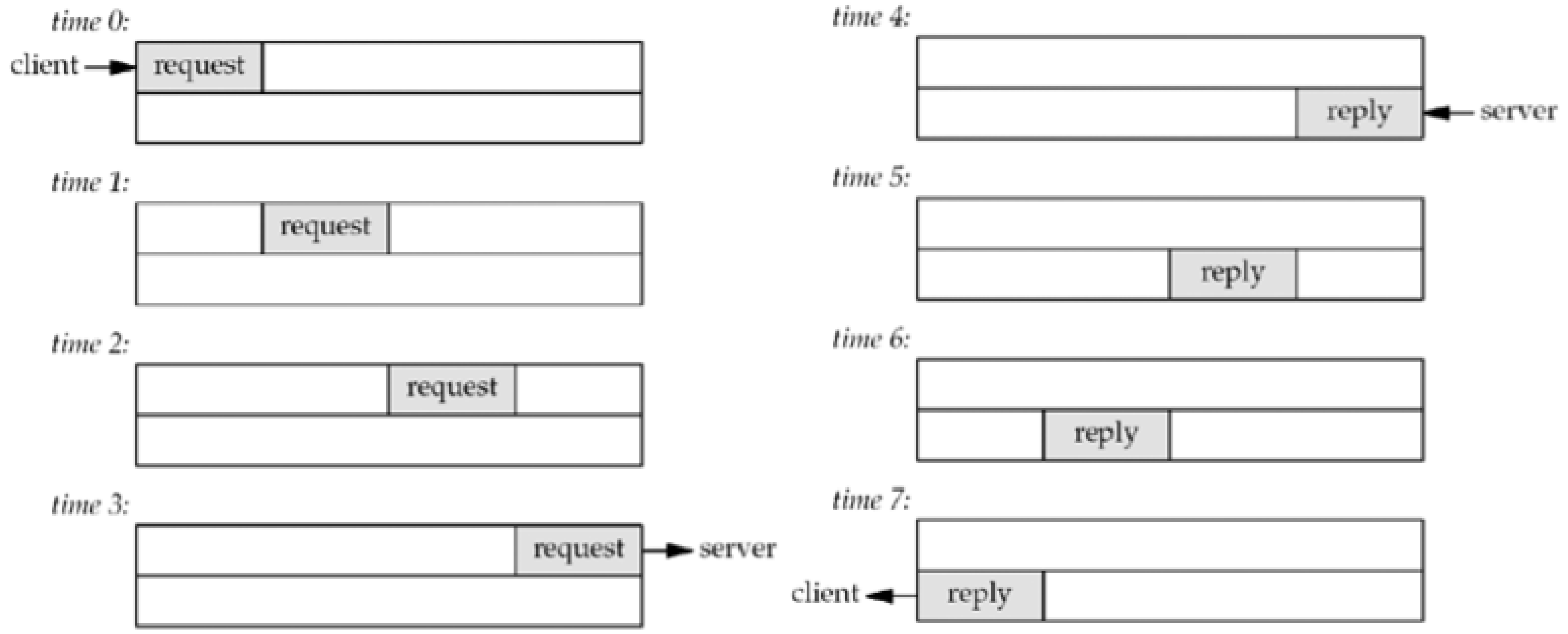
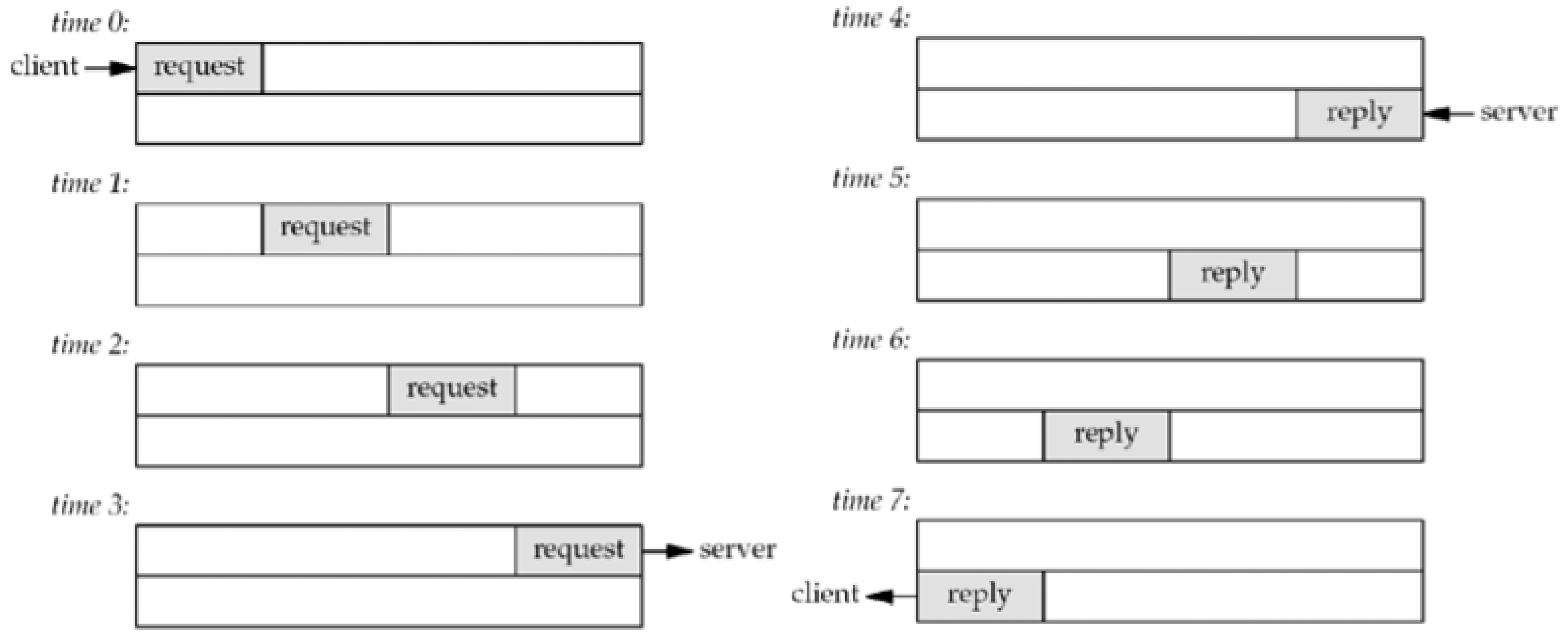


Figure: Time line of stop-and-wait mode: interactive input

Observations: Batch Input and Buffering

- ✍ A request is sent by the client at time 0 and assume an RTT of 8 units of time. The reply sent at time 4 is received at time 7.
- ✍ Assume that there is no server processing time and that the size of the request is the same as the reply. It is showed only the data packets between the client and server, ignoring the TCP acknowledgments that are also going across the network.
- ✍ Since there is a delay between sending a packet and that packet arriving at the other end of the pipe, and since the pipe is full-duplex, in this example, we are only using one-eighth of the pipe's capacity.
- ✍ This **`stop-and-wait`** mode is fine for interactive input.



Batch Input

- **In a batch mode :** Keep sending requests as fast as the network can accept them. The server processes them and sends back the replies at the same rate.
- **A case :** Assume that the input file contains only nine lines. The last line is sent at time 8, as shown in Figure. But we cannot close the connection after writing this request because there are still other requests and replies in the pipe. **The cause of the problem is our handling of an EOF on input**: The function returns to the main function, which then terminates.
- But in a batch mode, an EOF on input does not imply that we have finished reading from the socket; there might still be requests on the way to the server, or replies on the way back from the server.
- **Shutdown function :** A way is needed to close one-half of the TCP connection. That is, we want to send a FIN to the server, telling it we have finished sending data, but leave the socket descriptor open for reading. This is done with the shutdown function.

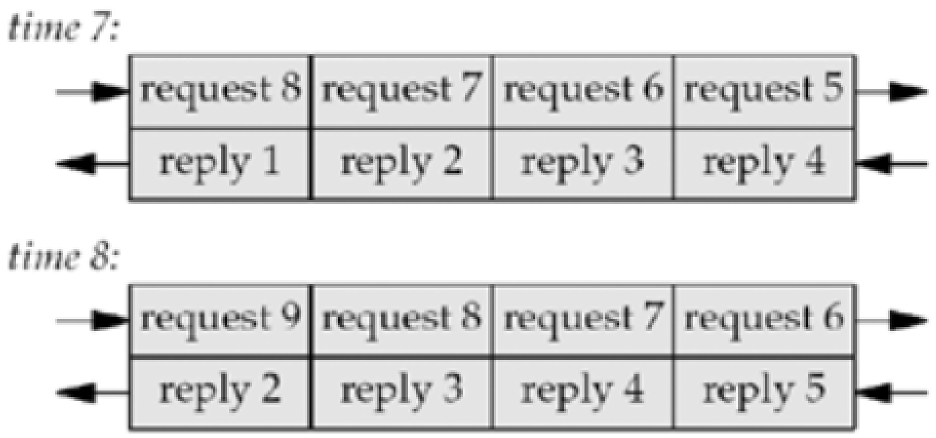


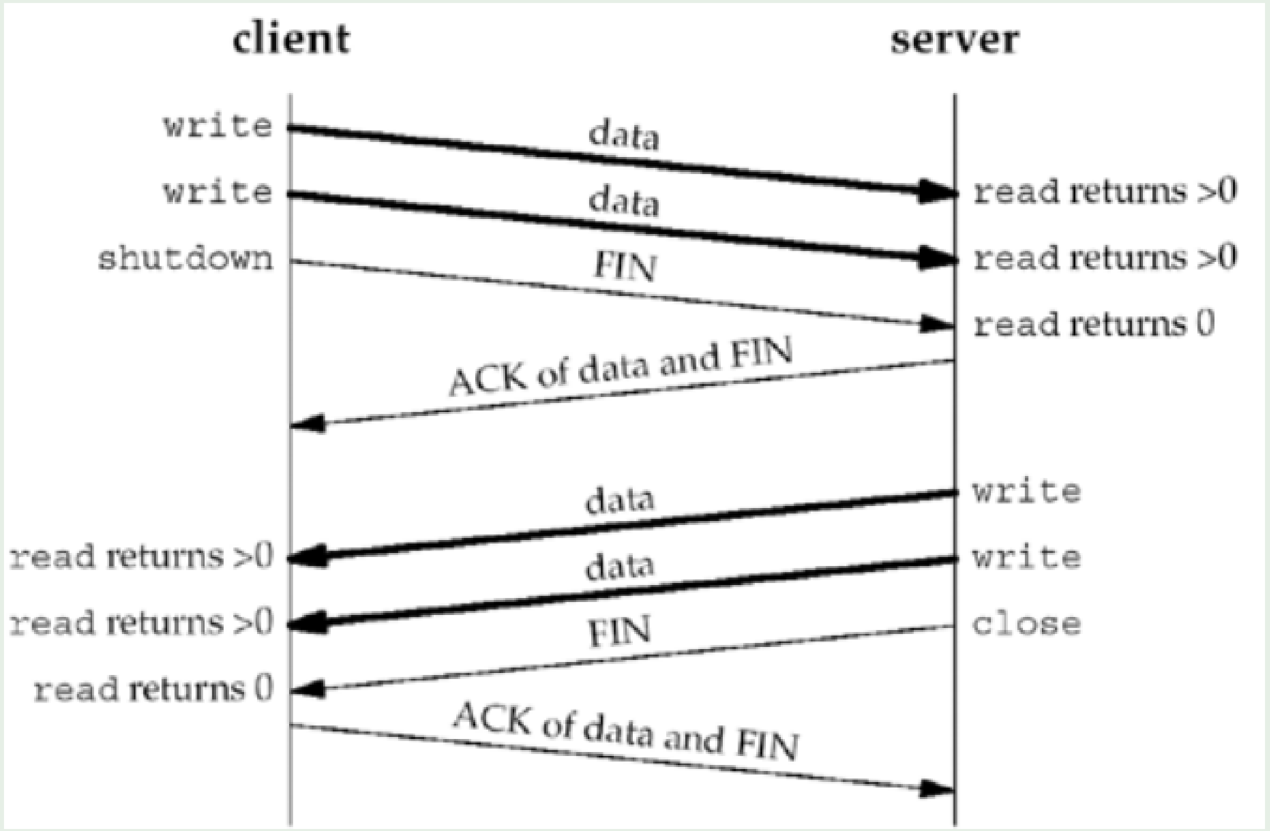
Figure: Filling the pipe between the client and server: batch mode

Shutdown Function

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`**:

1. **`close`** decrements the descriptor's reference count and closes the socket only if the count reaches 0.
2. **`close`** terminates both directions of data transfer, reading and writing.

Shutdown to close half of a TCP connection



Shutdown Function

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

int shutdown(int sockfd, int howto );

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

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.

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.

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.

The `poll()` System Call

- ✍ The **`poll`** function is similar to **`select`**. It waits for one of a set of file descriptors to become ready to perform IO.
- ✍ It organizes the information by file descriptor rather than by type of condition (i.e read, write or error).
- ✍ The possible events for one file descriptor are stored in a **`struct pollfd`**.
- ✍ But **select** organizes information by the type of event and has separate descriptor masks for read, write and error conditions.
- ✍ The **`poll`** function takes three parameters: **`fds`**, **`nfds`** and **`timeout`**.
- ✍ The **`timeout`** value is the time in **milliseconds** that the poll should wait without receiving an event before returning.

The Prototype of `poll()` System Call

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

or

```
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Return value of `poll`:

```
(1) If successful, poll returns the number of
    descriptors that have events.
(2) If unsuccessful, poll returns -1 and sets errno.
(3) The poll function returns 0 if it times out.
```

The `struct pollfd` Structure

The set of file descriptors to be monitored is specified in the **`fds`** argument, which is an array of structures of the following form:

```
struct pollfd {
    int    fd;                /* file descriptor */
    short  events;            /* requested events */
    short  revents;           /* returned events */
};
```

The caller should specify the number of items in the **`fds`** array in **`nfds`**.

The `struct pollfd` Member

- ✍ The **`fd`** is the file descriptor number.
- ✍ The **`events`** and **`revents`** are constructed by taking the logical OR of event flags.

event flags	meaning
`POLLIN`	read other than high priority data without blocking
`POLLPRI`	read high-priority data without blocking
`POLLRDNORM`	read normal data without blocking
`POLLRDBAND`	read priority data without blocking
`POLLOUT`	write normal data without blocking
`POLLWRNORM`	same as POLLOUT
`POLLWRBAND`	priority data can be written without blocking.
`POLLERR`	error occurred on the descriptor
`POLLNVAL`	file descriptor invalid
`POLLHUP`	device has been disconnected

- ✍ Set **`events`** to contain the events to monitor.
- ✍ **`poll`** fills in the **`revents`** with the **`events`** that have occurred.

Sample Code on `poll()`

The sample code is to monitor `stdin` for a timeout value of 6000 milliseconds.

```
int main()
{
  int timeout=6000,retpoll; ssize_t byteread; char buf[20];
  struct pollfd fds[1];
  fds[0].fd=0;
  fds[0].events=POLLIN;
  printf("Enter date within 6000 ms:\n");
  retpoll=poll(fds,1,timeout);
  if(retpoll==0){
    printf("Time out: No data entered\n");
  }
  if(fds[0].revents){
     printf("Data available:\n");
     byteread=read(0,buf,10);
     write(1,buf,byteread);
  }
  return 0;
}
```

The `timeout` parameter of `poll`

The final argument to poll specifies how long we want to wait.

- ✍ **timeout == -1**
  Wait forever. We return when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, poll returns -1 with errno set to EINTR.

- ✍ **timeout == 0**
  Do not wait. All the specified descriptors are tested, and we return immediately. This is a way to poll the system to find out the status of multiple descriptors, without blocking in the call to poll.

- ✍ **timeout > 0**
  Wait timeout milliseconds. We return when one of the specified descriptors is ready or when the timeout expires. If the timeout expires before any of the descriptors is ready, the return value is 0.

The pselect function

Other variant of synchronous I/O multiplexing techniques.

```
#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
```

pselect contains two changes from the normal **select** function:

1. **pselect** uses the **timespec** structure, another POSIX invention, instead of the **timeval** structure as in **select**.
2. **pselect** adds a sixth argument: a pointer to a signal mask.

The `ppoll` function

Other variant of synchronous I/O multiplexing techniques. it waits for one of a set of file descriptors to become ready to perform I/O.

```
#include <signal.h>
#include <poll.h>

int ppoll(struct pollfd *fds, nfds_t nfds, const struct timespec
     *tmo_p, const sigset_t *sigmask);


Returns: count of ready descriptors, 0 on timeout, 1 on error
```

`ppoll` contains the following changes from the normal **`poll`** function:

1. **`ppoll()`** allows an application to safely wait until either a file descriptor becomes ready or until a signal is caught.
2. **`ppoll`** uses the **`timespec`** structure.
3. **`ppoll`** last argument argument: a pointer to a signal mask.

Major Assignment

TCP server using a single process and select

THANK YOU