Alternative I / O Models

Conventional I / O Model

Perform I / O on a single file descriptor. Block if there is no data on the stream pointed by the descriptor.

Blocking can happen on reading as well as writing.

Some cases, we need to check for availability without blocking or need to monitor multiple streams at the same time.

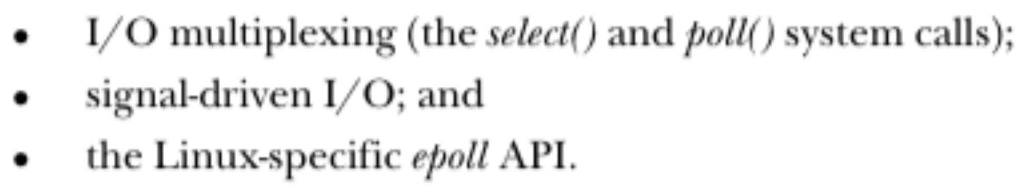
Most blocking I/O calls can be marked as non blocking.

Two problems in using I/O calls in non-blocking mode: unresponsive if the polling time is high and wasted CPU cycles if polling is tight

Another alternative is to use multiple processes or threads. Multiple processes is high overhead and IPC is expensive.

Multiple threads are efficient but can introduce complexity to the programs.

Alternative I / O Models



- I/O multiplexing (the *select()* and *poll()* system calls);
- signal-driven I/O; and
- the Linux-specific *epoll* API.

I/O multiplexing allow a process to monitor large number of descriptions and check whether IO is possible in any one of them.

Signal driven IO is where kernel send a signal when input data is available or data could be written. Gives much better performance with large number of descriptors.

Spill is Linux specific. Provides high performance and allows multiplexing data across large number of descriptors.

Descriptors become available for different reasons: arrival of input data, space becoming available on socket buffers, completion of a socket connection.

These techniques DO NOT perform actual IO, they just notify the state changes.

Level triggered versus edge triggered

- *Level-triggered notification*: A file descriptor is considered to be ready if it is possible to perform an I/O system call without blocking.
- *Edge-triggered notification*: Notification is provided if there is I/O activity (e.g., new input) on a file descriptor since it was last monitored.

With level triggered notification, a file descriptor can be read as long as it has data. Edge triggered notification only comes when state changes. So all data needs to be read right away. Otherwise, data will remain unread until next event.

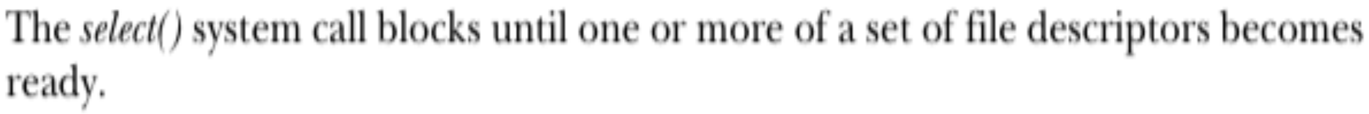
Level trigger read / write can be small enough to be responsive to serve other descriptors

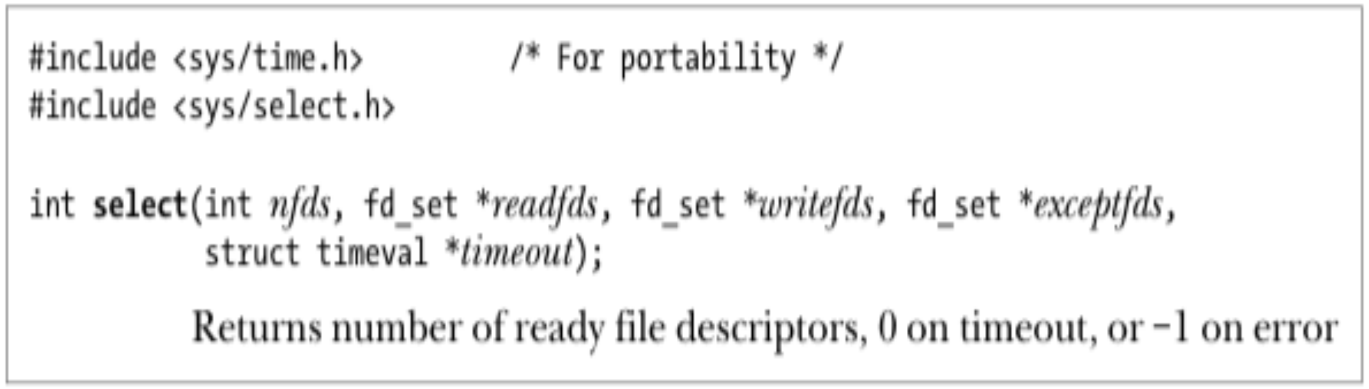
Edge triggered programs need to read as much as possible. Actual IO is still non blocking and the last read will return error.

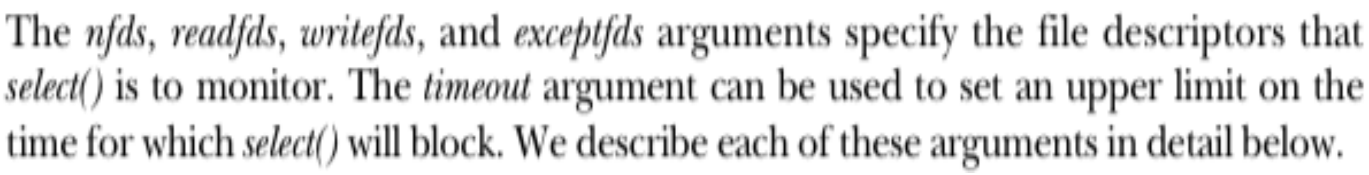
I/O model	Level-triggered?	Edge-triggered?
select(), *poll()*	•	
Signal-driven I/O		•
epoll	•	•

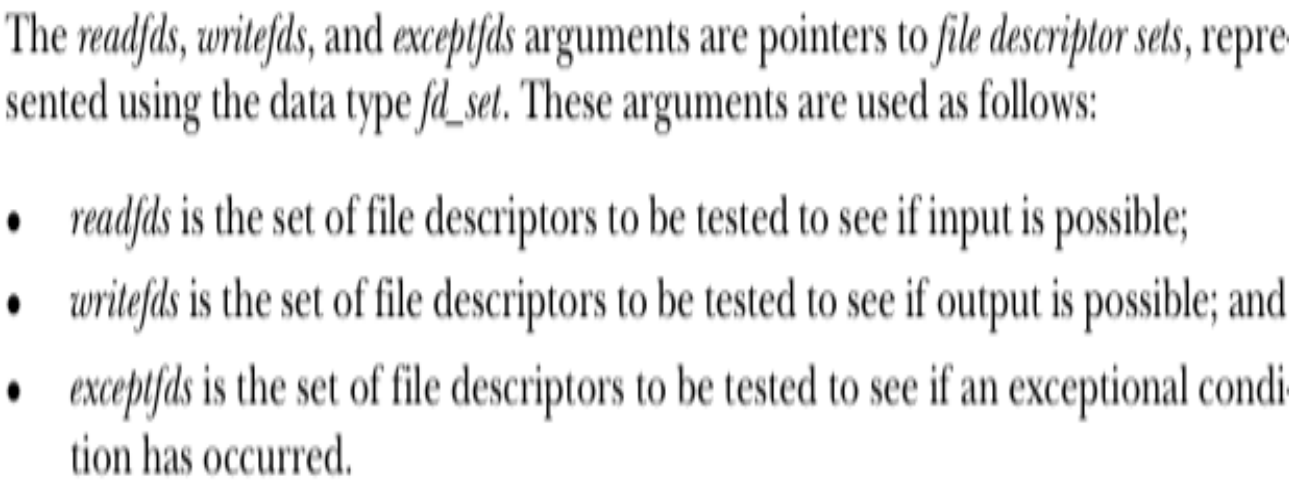
With either level or edge triggering it is still necessary to do non blocking IO to avoid blocking. This is due to many reasons: multiple processes reading or writing on the descriptors, kernel bugs, large enough data in a write that fills the buffers.

I / O Multiplexing


The *select()* system call blocks until one or more of a set of file descriptors becomes ready.


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

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```
Returns number of ready file descriptors, 0 on timeout, or –1 on error

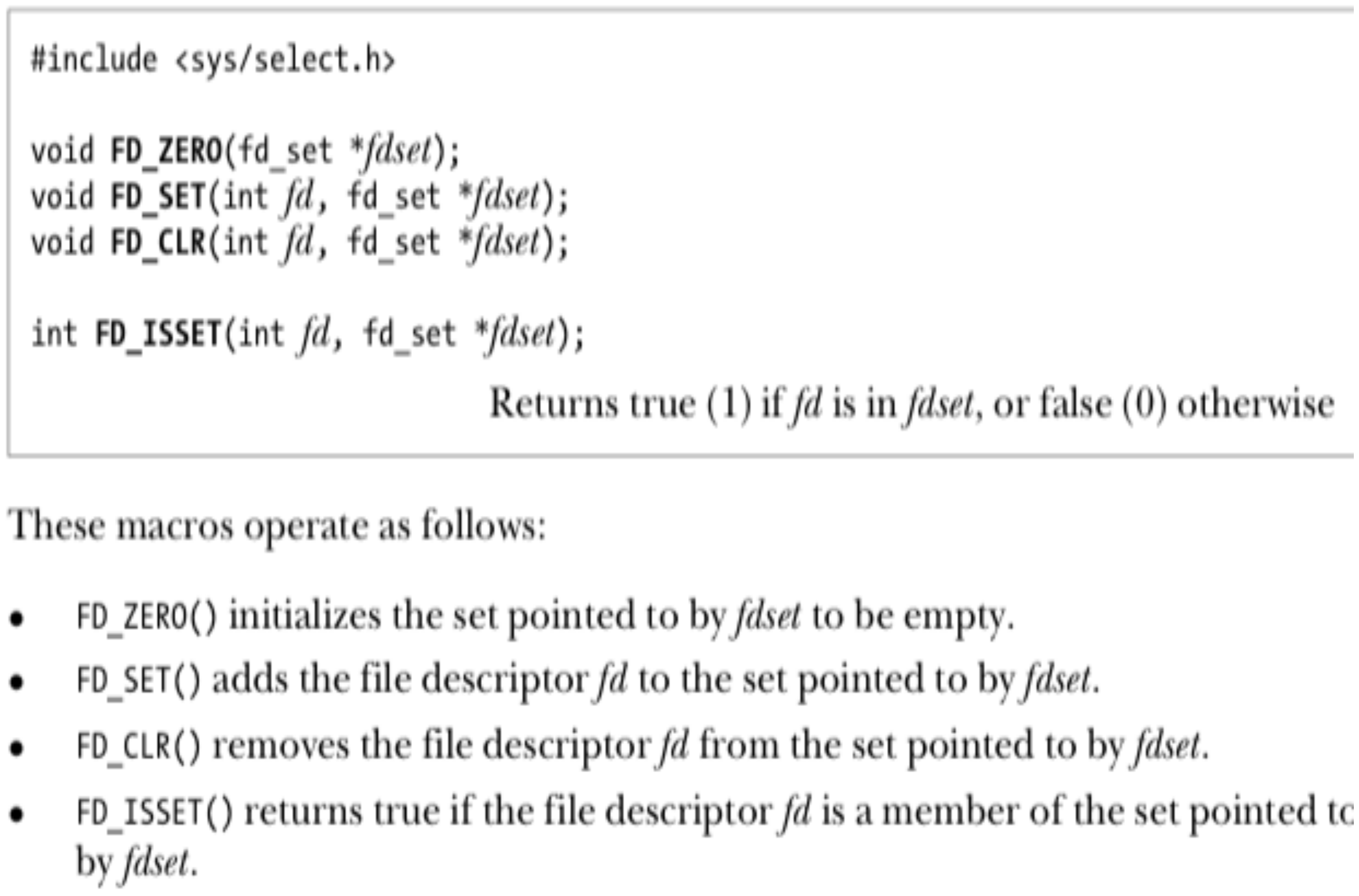

The *nfds*, *readfds*, *writefds*, and *exceptfds* arguments specify the file descriptors that *select()* is to monitor. The *timeout* argument can be used to set an upper limit on the time for which *select()* will block. We describe each of these arguments in detail below.


The *readfds*, *writefds*, and *exceptfds* arguments are pointers to *file descriptor sets*, repre
sented using the data type *fd_set*. These arguments are used as follows:

- *readfds* is the set of file descriptors to be tested to see if input is possible;
- *writefds* is the set of file descriptors to be tested to see if output is possible; and
- *exceptfds* is the set of file descriptors to be tested to see if an exceptional condi
  tion has occurred.

One example exceptional condition is the arrival of out-of-band data. File descriptor sets are manipulated by the following macros.

IO Multiplexing Continued.



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

void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);

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

Returns true (1) if *fd* is in *fdset*, or false (0) otherwise

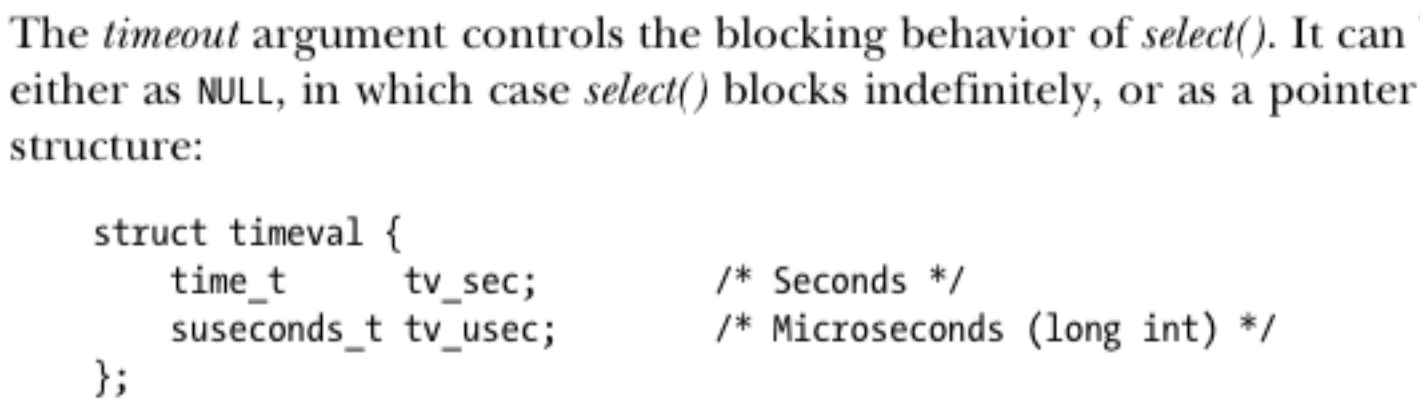
These macros operate as follows:

- `FD_ZERO()` initializes the set pointed to by *fdset* to be empty.
- `FD_SET()` adds the file descriptor *fd* to the set pointed to by *fdset*.
- `FD_CLR()` removes the file descriptor *fd* from the set pointed to by *fdset*.
- `FD_ISSET()` returns true if the file descriptor *fd* is a member of the set pointed tc by *fdset*.

Before calling select() the fd_set must be initialized and set. If select() is performed in loop then initialization should be done inside the loop.

On return, select() call modifies the set so that it includes only those ready descriptors.

If certain events are not necessary, then those file descriptors should be NULL.



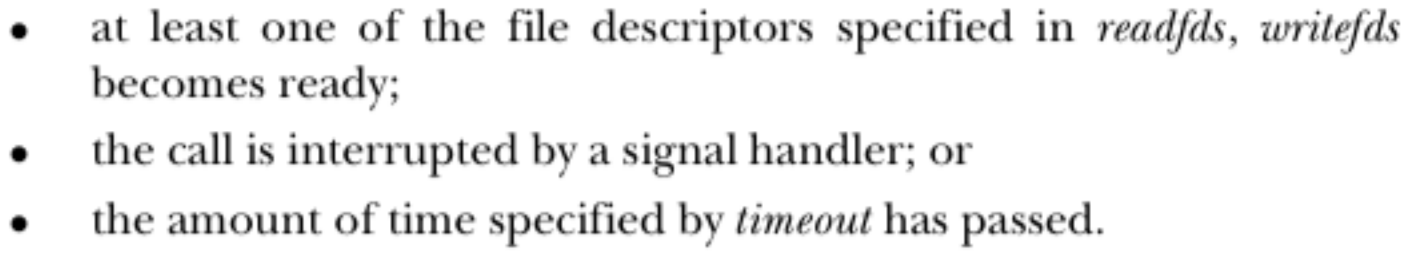
The *timeout* argument controls the blocking behavior of *select()*. It can
either as NULL, in which case *select()* blocks indefinitely, or as a pointer
structure:

```
struct timeval {
    time_t      tv_sec;         /* Seconds */
    suseconds_t tv_usec;        /* Microseconds (long int) */
};
```

If the timeout parameter is NULL, select blocks. If timeout parameters are zero, select returns immediately.

IO Multiplexing and Timeout

Select returns under three conditions when timeout is NULL or has non zero parameters



- at least one of the file descriptors specified in *readfds*, *writefds*
  becomes ready;
- the call is interrupted by a signal handler; or
- the amount of time specified by *timeout* has passed.

Select returns -1, 0, N -1 indicates and error situation 0 indicates a timeout N indicates the number of file descriptors that have data

Example Select Program

```
int
main(int argc, char *argv[])
{
    fd_set readfds, writefds;
    int ready, nfds, fd, numRead, j;
    struct timeval timeout;
    struct timeval *pto;
    char buf[10];                          /* Large enough to hold "rw\0" */

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageError(argv[0]);

    /* Timeout for select() is specified in argv[1] */

    if (strcmp(argv[1], "-") == 0) {
        pto = NULL;                        /* Infinite timeout */
    } else {
        pto = &timeout;
        timeout.tv_sec = getLong(argv[1], 0, "timeout");
        timeout.tv_usec = 0;               /* No microseconds */
    }

    /* Process remaining arguments to build file descriptor sets */

    nfds = 0;
    FD_ZERO(&readfds);
    FD_ZERO(&writefds);
```

```
for (j = 2; j < argc; j++) {
    numRead = sscanf(argv[j], "%d%2[rw]", &fd, buf);
    if (numRead != 2)
        usageError(argv[0]);
    if (fd >= FD_SETSIZE)
        cmdLineErr("file descriptor exceeds limit (%d)\n", FD_SETSIZE);

    if (fd >= nfds)
        nfds = fd + 1;              /* Record maximum fd + 1 */
    if (strchr(buf, 'r') != NULL)
        FD_SET(fd, &readfds);
    if (strchr(buf, 'w') != NULL)
        FD_SET(fd, &writefds);
}

/* We've built all of the arguments; now call select() */

ready = select(nfds, &readfds, &writefds, NULL, pto);
                                    /* Ignore exceptional events */
if (ready == -1)
    errExit("select");

/* Display results of select() */

printf("ready = %d\n", ready);
```

```
    for (fd = 0; fd < nfds; fd++)
        printf("%d: %s%s\n", fd, FD_ISSET(fd, &readfds) ? "r" : "",
                FD_ISSET(fd, &writefds) ? "w" : "");

    if (pto != NULL)
        printf("timeout after select(): %ld.%03ld\n",
               (long) timeout.tv_sec, (long) timeout.tv_usec / 10000);
    exit(EXIT_SUCCESS);
}
```

When is File Descriptor Ready?

A file descriptor is READY if IO on it would not block -- does not mean it would successfully transfer data!

Regular Files

- A *read()* will always immediately return data, end-of-file, or an erro file was not opened for reading).
- A *write()* will always immediately transfer data or fail with some erro

Sockets

Condition or event	***select()***
Input available	r
Output possible	w
Incoming connection established on listening socket	r
Out-of-band data received (TCP only)	x
Stream socket peer closed connection or executed *shutdown(SHUT_WR)*	rw

Problems with Select()

Select() is widely implemented. Performance can suffer if large number of descriptors are watched using select().

Kernel is asked to watch a list of descriptors without giving it a persistent set that identifies the watch list.

By giving the watch list to kernel, we could let the kernel generate the notification when the event arrives -- complexity is reduced to the number of events from number of descriptors.

IO Multiplexing using Signals

Select() checks whether IO possible on given file descriptors. Signal based approach asks the kernel to notify the process when IO is possible. Process can perform other activities in the meantime.

1. Establish a handler for the signal delivered by the signal-driven I/O mecha-
   nism. By default, this notification signal is SIGIO.
2. Set the *owner* of the file descriptor–that is, the process or process group that i
   to receive signals when I/O is possible on the file descriptor. Typically, we make
   the calling process the owner. The owner is set using an *fcntl()* F_SETOWN opera-
   tion of the following form:

   ```
   fcntl(fd, F_SETOWN, pid);
   ```

3. Enable nonblocking I/O by setting the O_NONBLOCK open file status flag.
4. Enable signal-driven I/O by turning on the O_ASYNC open file status flag. Thi
   can be combined with the previous step, since they both require the use of the
   fcntl() F_SETFL operation (Section 5.3), as in the following example:

   ```
   flags = fcntl(fd, F_GETFL);                    /* Get current flags */
   fcntl(fd, F_SETFL, flags | O_ASYNC | O_NONBLOCK);
   ```

5. The calling process can now perform other tasks. When I/O becomes possible
   the kernel generates a signal for the process and invokes the signal handle
   established in step 1.
6. Signal-driven I/O provides edge-triggered notification (Section 63.1.1). Thi
   means that once the process has been notified that I/O is possible, it shoul
   perform as much I/O (e.g., read as many bytes) as possible. Assuming a non-
   blocking file descriptor, this means executing a loop that performs I/O syste
   calls until a call fails with the error EAGAIN or EWOULDBLOCK.

Example program - Signal based IO Multiplexing

```
/* Establish handler for "I/O possible" signal */

sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
sa.sa_handler = sigioHandler;
if (sigaction(SIGIO, &sa, NULL) == -1)
    errExit("sigaction");

/* Set owner process that is to receive "I/O possible" signal */

if (fcntl(STDIN_FILENO, F_SETOWN, getpid()) == -1)
    errExit("fcntl(F_SETOWN)");

/* Enable "I/O possible" signaling and make I/O nonblocking
   for file descriptor */

flags = fcntl(STDIN_FILENO, F_GETFL);
if (fcntl(STDIN_FILENO, F_SETFL, flags | O_ASYNC | O_NONBLOCK) == -1)
    errExit("fcntl(F_SETFL)");

/* Place terminal in cbreak mode */

if (ttySetCbreak(STDIN_FILENO, &origTermios) == -1)
    errExit("ttySetCbreak");
    for (done = FALSE, cnt = 0; !done ; cnt++) {
        for (j = 0; j < 100000000; j++)
            continue;                   /* Slow main loop down a little */

        if (gotSigio) {                 /* Is input available? */

            /* Read all available input until error (probably EAGAIN)
               or EOF (not actually possible in cbreak mode) or a
               hash (#) character is read */

            while (read(STDIN_FILENO, &ch, 1) > 0 && !done) {
                printf("cnt=%d; read %c\n", cnt, ch);
                done = ch == '#';
            }

            gotSigio = 0;
        }
    }

    /* Restore original terminal settings */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &origTermios) == -1)
        errExit("tcsetattr");
    exit(EXIT_SUCCESS);
}
```

```
#include <signal.h>
#include <ctype.h>
#include <fcntl.h>
#include <termios.h>
#include "tty_functions.h"      /* Declaration of ttySetCbreak() */
#include "tlpi_hdr.h"

static volatile sig_atomic_t gotSigio = 0;
                                /* Set nonzero on receipt of SIGIO */

static void
sigioHandler(int sig)
{
    gotSigio = 1;
}

int
main(int argc, char *argv[])
{
    int flags, j, cnt;
    struct termios origTermios;
    char ch;
    struct sigaction sa;
    Boolean done;
```

When is IO Possible Signaled?

Sockets

Signal-driven I/O works for datagram sockets in both the UNIX and the Internet domains. A signal is generated in the following circumstances:

- An input datagram arrives on the socket (even if there were already unread datagrams waiting to be read).
- An asynchronous error occurs on the socket.

Signal-driven I/O works for stream sockets in both the UNIX and the Internet domains. A signal is generated in the following circumstances:

- A new connection is received on a listening socket.
- A TCP *connect()* request completes; that is, the active end of a TCP connection

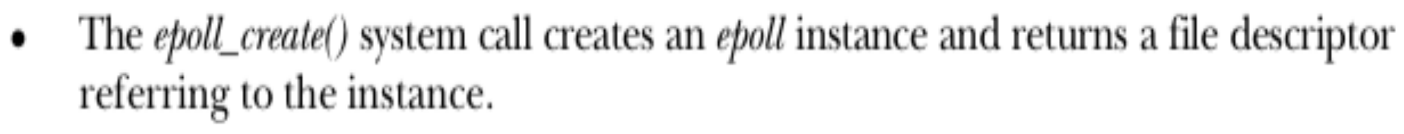
- New input is received on the socket (even if there was already unread input available).
- The peer closes its writing half of the connection using *shutdown()*, or closes its socket altogether using *close()*.
- Output is possible on the socket (e.g., space has become available in the socket send buffer).
- An asynchronous error occurs on the socket.

The epoll API

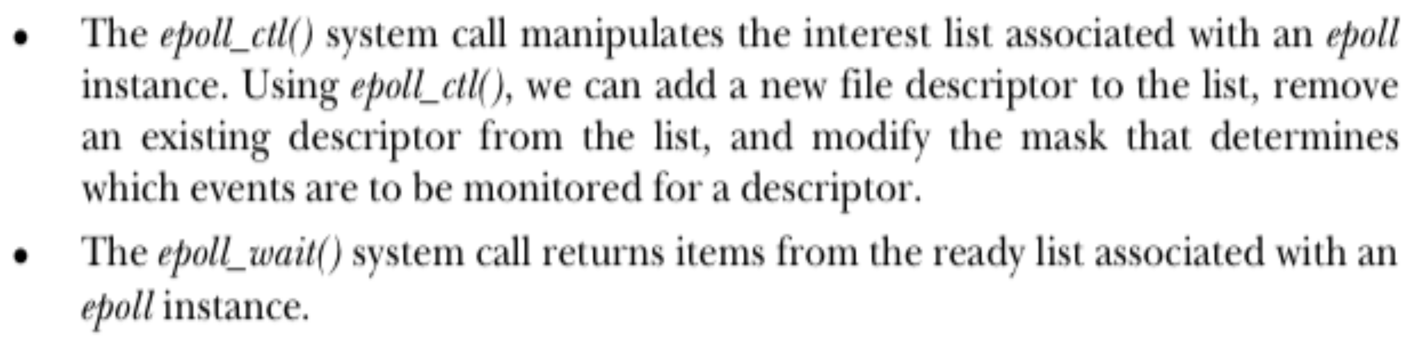
Performance of ePoll scales much better when watching many file descriptors. Unlike, select(), ePoll registers the interesting set of descriptors with the kernel.

For each file descriptor it is possible to register an interested events list.

ePoll consists of three system calls:

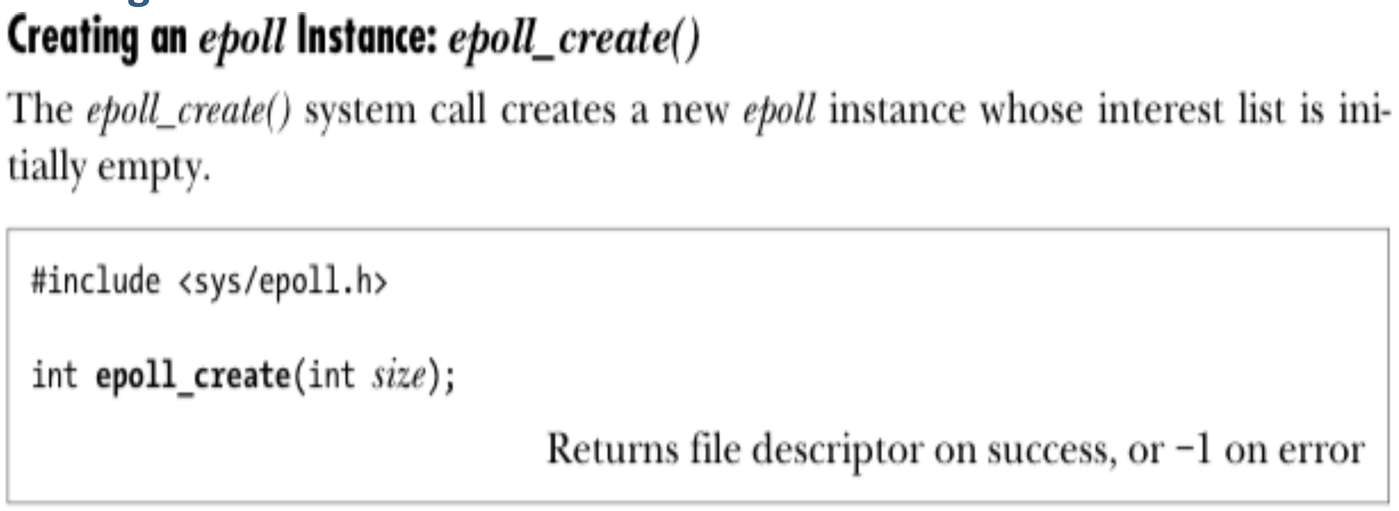


- The *epoll_create()* system call creates an *epoll* instance and returns a file descriptor referring to the instance.



- The *epoll_ctl()* system call manipulates the interest list associated with an *epoll* instance. Using *epoll_ctl()*, we can add a new file descriptor to the list, remove an existing descriptor from the list, and modify the mask that determines which events are to be monitored for a descriptor.
- The *epoll_wait()* system call returns items from the ready list associated with an *epoll* instance.

Creating an ePoll instance



Creating an *epoll* Instance: *epoll_create()*

The *epoll_create()* system call creates a new *epoll* instance whose interest list is initially empty.

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

int epoll_create(int size);
```

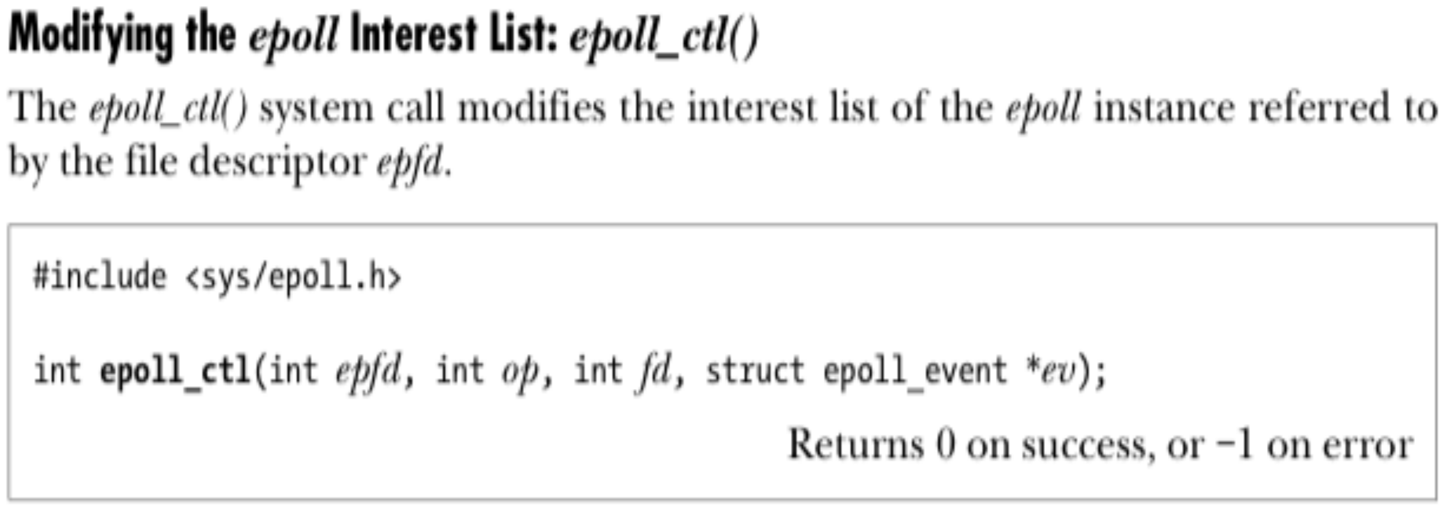
Returns file descriptor on success, or –1 on error

Size refers to an initial set size for the file descriptors.

Epoll returns the file descriptor referring to the new epoll instance.

When all file descriptors referring to the epoll instance is closed, the epoll instance is destroyed.

Modifying an ePoll instance



Modifying the *epoll* Interest List: *epoll_ctl()*

The *epoll_ctl()* system call modifies the interest list of the *epoll* instance referred to by the file descriptor *epfd*.

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

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *ev);
```

Returns 0 on success, or –1 on error

The *op* argument specifies the operation to be performed, and has one of the following values:

`EPOLL_CTL_ADD`

Add the file descriptor *fd* to the interest list for *epfd*. The set of events that we are interested in monitoring for *fd* is specified in the buffer pointed to by *ev*, as described below. If we attempt to add a file descriptor that is already in the interest list, *epoll_ctl()* fails with the error `EEXIST`.

`EPOLL_CTL_MOD`

Modify the events setting for the file descriptor *fd*, using the information specified in the buffer pointed to by *ev*. If we attempt to modify the settings of a file descriptor that is not in the interest list for *epfd*, *epoll_ctl()* fails with the error `ENOENT`.

`EPOLL_CTL_DEL`

Remove the file descriptor *fd* from the interest list for *epfd*. The *ev* argument is ignored for this operation. If we attempt to remove a file descriptor that is not in the interest list for *epfd*, *epoll_ctl()* fails with the error `ENOENT`. Closing a file descriptor automatically removes it from all of the *epoll* interest lists of which it is a member.

The *ev* argument is a pointer to a structure of type *epoll_event*, defined as follows:

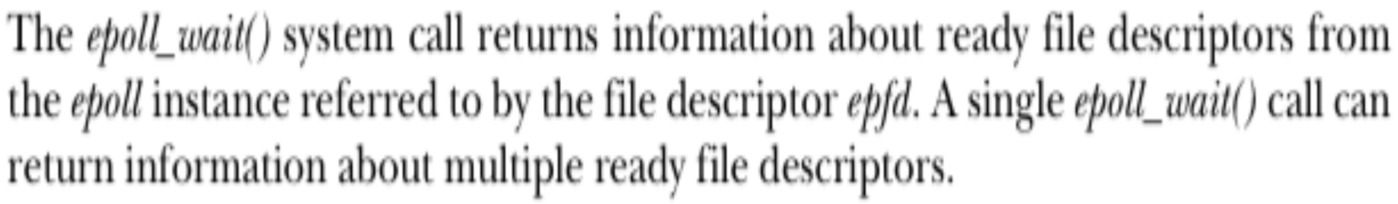
```
struct epoll_event {
    uint32_t     events;        /* epoll events (bit mask) */
    epoll_data_t data;          /* User data */
};
```

The *data* field of the *epoll_event* structure is typed as follows:

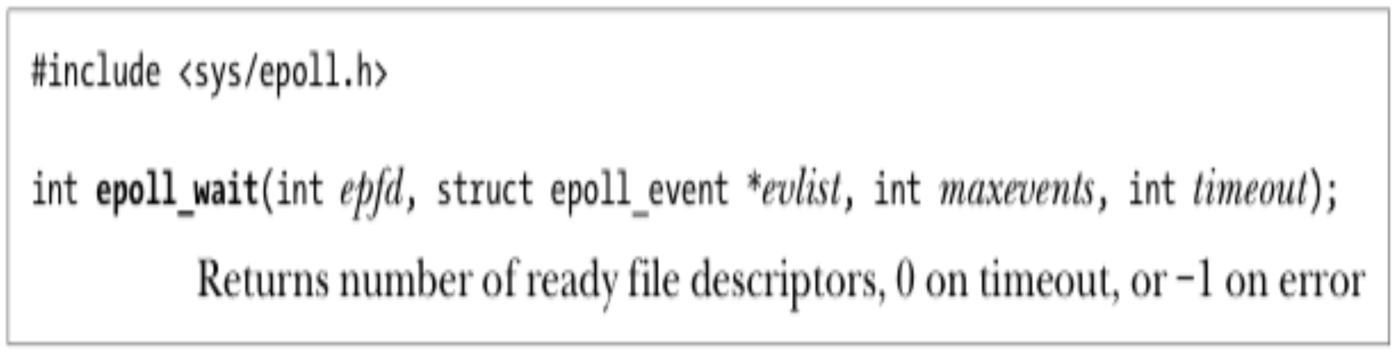
```
typedef union epoll_data {
    void        *ptr;           /* Pointer to user-defined data */
    int          fd;            /* File descriptor */
    uint32_t     u32;           /* 32-bit integer */
    uint64_t     u64;           /* 64-bit integer */
} epoll_data_t;
```

```
int epfd;
struct epoll_event ev;

epfd = epoll_create(5);
if (epfd == -1)
    errExit("epoll_create");

ev.data.fd = fd;
ev.events = EPOLLIN;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, ev) == -1)
    errExit("epoll_ctl");
```

Waiting for Events on ePoll



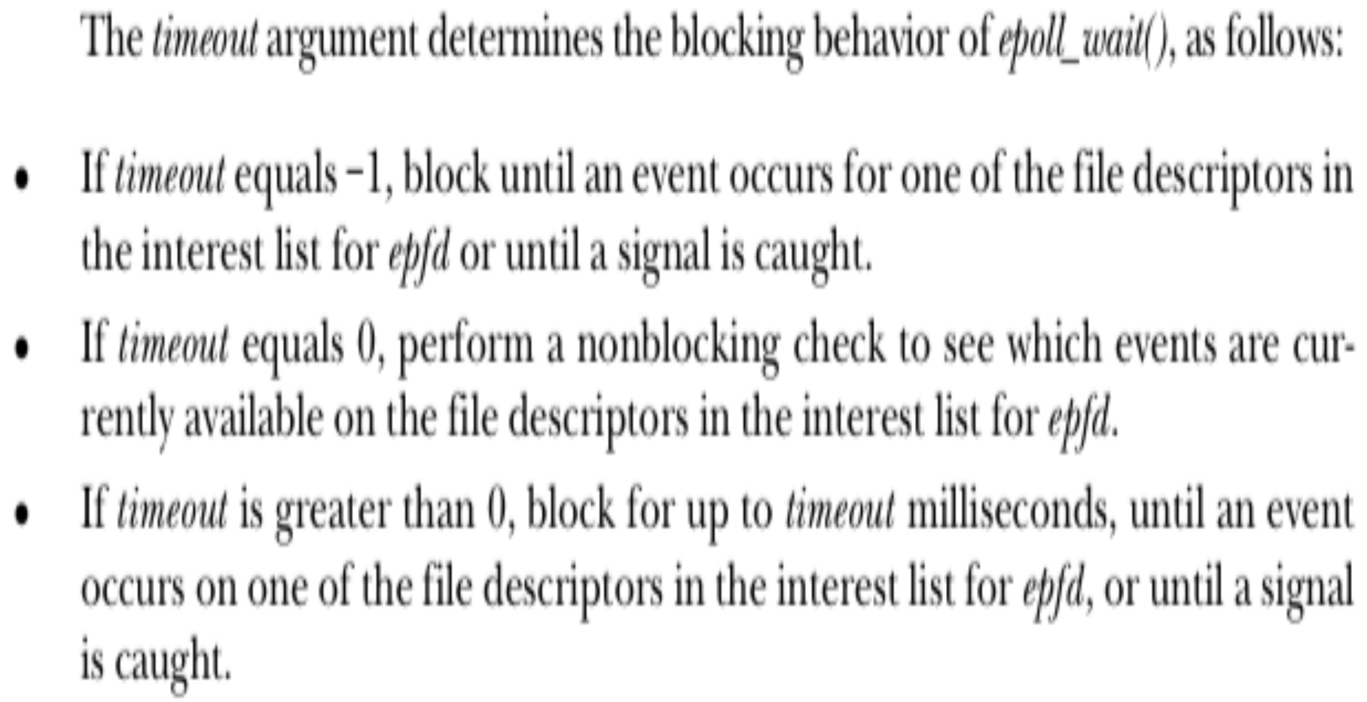
The *epoll_wait()* system call returns information about ready file descriptors from the *epoll* instance referred to by the file descriptor *epfd*. A single *epoll_wait()* call can return information about multiple ready file descriptors.



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

int epoll_wait(int epfd, struct epoll_event *evlist, int maxevents, int timeout);
```

Returns number of ready file descriptors, 0 on timeout, or –1 on error

Evlist is allocated by the caller. Each item returns information about a ready file descriptor.



The *timeout* argument determines the blocking behavior of *epoll_wait()*, as follows:

- If *timeout* equals –1, block until an event occurs for one of the file descriptors in the interest list for *epfd* or until a signal is caught.
- If *timeout* equals 0, perform a nonblocking check to see which events are currently available on the file descriptors in the interest list for *epfd*.
- If *timeout* is greater than 0, block for up to *timeout* milliseconds, until an event occurs on one of the file descriptors in the interest list for *epfd*, or until a signal is caught.

Bit	Input to *epoll_ctl()*?	Returned by *epoll_wait()*?	Description
`EPOLLIN`	•	•	Data other than high-priority data can be read
`EPOLLPRI`	•	•	High-priority data can be read
`EPOLLRDHUP`	•	•	Shutdown on peer socket (since Linux 2.6.17)
`EPOLLOUT`	•	•	Normal data can be written
`EPOLLET`	•		Employ edge-triggered event notification
`EPOLLONESHOT`	•		Disable monitoring after event notification
`EPOLLERR`		•	An error has occurred
`EPOLLHUP`		•	A hangup has occurred

Example ePoll program

```
#include <sys/epoll.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define MAX_BUF     1000        /* Maximum bytes fetched by a single read() */
#define MAX_EVENTS     5        /* Maximum number of events to be returned from
                                   a single epoll_wait() call */

int
main(int argc, char *argv[])
{
    int epfd, ready, fd, s, j, numOpenFds;
    struct epoll_event ev;
    struct epoll_event evlist[MAX_EVENTS];
    char buf[MAX_BUF];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file...\n", argv[0]);
```

Create an ePoll instance

```
①   epfd = epoll_create(argc - 1);
    if (epfd == -1)
        errExit("epoll_create");

    /* Open each file on command line, and add it to the "interest
       list" for the epoll instance */
```

```
②       for (j = 1; j < argc; j++) {
            fd = open(argv[j], O_RDONLY);
            if (fd == -1)
                errExit("open");
            printf("Opened \"%s\" on fd %d\n", argv[j], fd);

            ev.events = EPOLLIN;            /* Only interested in input events */
            ev.data.fd = fd;
③           if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev) == -1)
                errExit("epoll_ctl");
        }
```

```
            printf("Ready: %d\n", ready);

            /* Deal with returned list of events */

⑦           for (j = 0; j < ready; j++) {
                printf("  fd=%d; events: %s%s%s\n", evlist[j].data.fd,
                        (evlist[j].events & EPOLLIN)  ? "EPOLLIN "  : "",
                        (evlist[j].events & EPOLLHUP) ? "EPOLLHUP " : "",
                        (evlist[j].events & EPOLLERR) ? "EPOLLERR " : "");

⑧               if (evlist[j].events & EPOLLIN) {
                    s = read(evlist[j].data.fd, buf, MAX_BUF);
                    if (s == -1)
                        errExit("read");
                    printf("    read %d bytes: %.*s\n", s, s, buf);

⑨               } else if (evlist[j].events & (EPOLLHUP | EPOLLERR)) {

                    /* If EPOLLIN and EPOLLHUP were both set, then there might
                       be more than MAX_BUF bytes to read. Therefore, we close
                       the file descriptor only if EPOLLIN was not set.
                       We'll read further bytes after the next epoll_wait(). *

                    printf("    closing fd %d\n", evlist[j].data.fd);
⑩                   if (close(evlist[j].data.fd) == -1)
                        errExit("close");
                    numOpenFds--;
                }
            }
        }

        printf("All file descriptors closed; bye\n");
        exit(EXIT_SUCCESS);
    }
    numOpenFds = argc - 1;

④   while (numOpenFds > 0) {

        /* Fetch up to MAX_EVENTS items from the ready list */

        printf("About to epoll_wait()\n");
⑤       ready = epoll_wait(epfd, evlist, MAX_EVENTS, -1);
        if (ready == -1) {
⑥           if (errno == EINTR)
                continue;               /* Restart if interrupted by signal */
            else
                errExit("epoll_wait");
        }
```