

Linux/UNIX IPC Programming

Alternative I/O Models

Michael Kerrisk, man7.org © 2020

mtk@man7.org

February 2020

Outline

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: poll()	7-14
7.5	Problems with poll() and select()	7-31
7.6	The epoll API	7-34
7.7	epoll events	7-45
7.8	epoll: edge-triggered notification	7-59
7.9	epoll: API quirks	7-70
7.10	Event-loop programming	7-76

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: <code>poll()</code>	7-14
7.5	Problems with <code>poll()</code> and <code>select()</code>	7-31
7.6	The <code>epoll</code> API	7-34
7.7	<code>epoll</code> events	7-45
7.8	<code>epoll</code> : edge-triggered notification	7-59
7.9	<code>epoll</code> : API quirks	7-70
7.10	Event-loop programming	7-76

The traditional file I/O model

- I/O on **one file at a time**
 - `read()`, `write()`, etc. operate on single descriptor
- **Blocking I/O**
 - I/O not possible \Rightarrow call blocks until I/O becomes possible
 - Examples:
 - `write()` to pipe blocks if insufficient space
 - `read()` from socket that has no data available
- But sometimes, we want to:
 - **Check if I/O is possible without blocking** if it is not
 - **Monitor multiple file descriptors** to see if I/O is possible on any of them

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: <code>poll()</code>	7-14
7.5	Problems with <code>poll()</code> and <code>select()</code>	7-31
7.6	The <code>epoll</code> API	7-34
7.7	<code>epoll</code> events	7-45
7.8	<code>epoll</code> : edge-triggered notification	7-59
7.9	<code>epoll</code> : API quirks	7-70
7.10	Event-loop programming	7-76

Nonblocking I/O

- Nonblocking I/O \Rightarrow **return error instead of blocking**
 - `EAGAIN` error for `read()`, `write()`, and similar
- Enabled via `O_NONBLOCK` file status flag
 - Set during `open()`; can also be enabled via `fcntl()`:

```
flags = fcntl(fd, F_GETFL);
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

- Recall: file status flags reside in open file description
- Many APIs that create FDs also have a flag that allows nonblocking mode to be set at time FD is created
 - E.g., `eventfd()`, `inotify_init1()`, `open()`, `pipe2()`, `signalfd()`, `socket()`, `timerfd_create()`

EAGAIN vs EWOULDBLOCK

- On BSD, `EWOULDBLOCK` was/is returned instead of `EAGAIN`
- Many modern systems address this portability issue by making `EAGAIN` and `EWOULDBLOCK` synonyms
 - POSIX explicitly permits this
 - Linux does this

Use cases for nonblocking I/O

- Check if I/O is possible without blocking if not (“**polling**”)
 - Mark file descriptor nonblocking
 - Perform I/O system call
 - If I/O call fails, try again later
- Perform as much I/O as possible, without blocking on completion
 - Mark file descriptor nonblocking
 - Perform I/O in a loop until `EAGAIN` encountered
- Nonblocking `accept()`
 - Make listening socket nonblocking
 - \Rightarrow `accept()` returns with `EAGAIN`/`EWOULDBLOCK` if no pending connection
- We’ll see some other valid use cases for nonblocking I/O
 - E.g., I/O while employing edge-triggered `epoll` notification

Problems with nonblocking I/O

- Using nonblocking I/O for **repeatedly polling multiple file descriptors is problematic**
 - Frequent polling \Rightarrow CPU cycles wasted
 - Infrequent polling \Rightarrow high I/O latency
- We need better techniques...

Better techniques for managing multiple file descriptors

- *poll()*, *select()* (“I/O multiplexing”):
 - **Simultaneously monitor multiple FDs** to see if I/O is possible on any of them
- **Signal-driven I/O**:
 - Kernel sends process a **signal when I/O is possible** on FD
 - Better performance than *select()* / *poll()*
- *epoll*:
 - Monitor multiple FDs (like *select()* / *poll()*)
 - Better performance and more features than *select()* / *poll()*
 - Simpler to program than signal-driven I/O
 - Linux-specific (since kernel 2.6.0)
- Above techniques only **monitor** FDs to see if I/O is possible
 - Actual I/O is performed using traditional system calls

Outline

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: <code>poll()</code>	7-14
7.5	Problems with <code>poll()</code> and <code>select()</code>	7-31
7.6	The <code>epoll</code> API	7-34
7.7	<code>epoll</code> events	7-45
7.8	<code>epoll</code> : edge-triggered notification	7-59
7.9	<code>epoll</code> : API quirks	7-70
7.10	Event-loop programming	7-76

Signal-driven I/O

- *Somewhat* portable technique for monitoring multiple FDs
- Process performs following steps:
 - Establish signal handler (default notification signal is `SIGIO`)
 - Mark itself as “owner” of FD (process that is to receive signals)
 - `fcntl(fd, F_SETOWN, pid)` operation
 - Enable signaling when I/O is possible on FD
 - Set `O_ASYNC` flag using `fcntl(fd, F_SETFL, flags)`
 - Carry on to do other tasks
 - When I/O becomes possible, signal handler is invoked
- Can enable I/O signaling on multiple FDs
- Better performance than `poll()/select()`
 - (For same reasons as `epoll`, as explained later)

[TLPI §63.3]

- Fully exploiting signal-driven I/O **requires use of Linux-specific features**
 - Choosing (realtime) signal via *fcntl(fd, F_SETSIG, sig)*
 - Default signal (**SIGIO**) is a **nonqueuing** traditional signal
 - Use **SA_SIGINFO** handler
 - \Rightarrow obtain file descriptor via *si_fd* field of *siginfo_t* structure
- **epoll API is more feature-rich** for task of monitoring multiple FDs
- \Rightarrow We'll ignore signal-driven I/O
 - (See TLPI §63.3 for more info + example program)

Outline

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: <code>poll()</code>	7-14
7.5	Problems with <code>poll()</code> and <code>select()</code>	7-31
7.6	The <code>epoll</code> API	7-34
7.7	<code>epoll</code> events	7-45
7.8	<code>epoll</code> : edge-triggered notification	7-59
7.9	<code>epoll</code> : API quirks	7-70
7.10	Event-loop programming	7-76

I/O multiplexing

- Monitor multiple file descriptors to see if I/O is possible on any of them
- Terminology: the FD is “**ready**” for I/O
 - Often, we’ll talk of monitoring I/O events, but...
 - Strictly speaking, **these APIs tell us whether an I/O system call would block**
- Two traditional techniques:
 - `select()` (4.2BSD, 1983)
 - `poll()` (System V Release 3, 1986)
 - Both specified in POSIX and widely available
- Can be applied to any file type
 - Pipes, FIFOs, terminals, devices, sockets...
 - Applicable to regular files, but not very useful

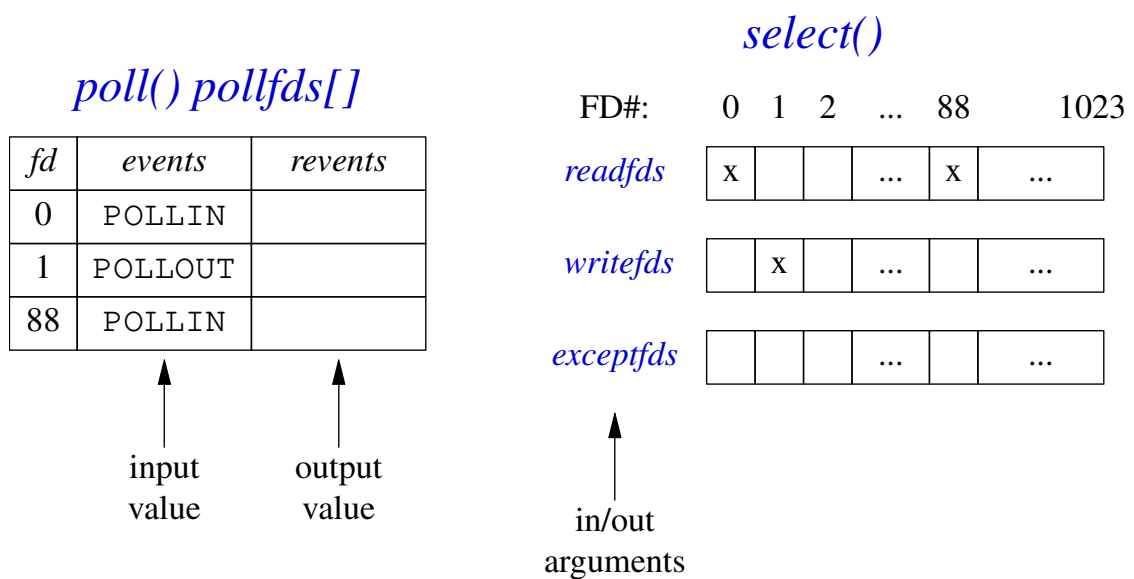
[TLPI §63.2]

poll() and *select()*

- *select()* and *poll()* perform same task
- Differ primarily in how FDs are specified:
 - *select()*:
 - Arguments: 3 FD sets for 3 classes of readiness
 - Each FD set contains a set of FDs
 - *poll()*:
 - Argument: list (array) of file descriptors
 - Each array element specifies type of readiness to test

[TLPI §63.2.2]

Arguments of *poll()* and *select()*



poll() vs *select()*

- *poll()* fixes some of the API problems of *select()*
 - *select()* uses fixed-size FD sets
 - Only FDs < 1024 can be monitored
 - Limitation of glibc, not kernel
 - *select()* uses same arguments for input and output
 - (Must reinitialize on each call inside a loop)
- ⇒ We'll focus on *poll()*

[TLPI §63.2.2]

poll()

```
#include <poll.h>
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

- *fds*: list of file descriptors to be monitored
- *nfds*: number of elements in *fds*
- *timeout*: timeout if call blocks because no FD is yet ready for I/O

[TLPI §63.2.2]

The *pollfd* array

```
struct pollfd {
    int    fd;        /* File descriptor */
    short  events;     /* Requested events bit mask */
    short  revents;    /* Returned events bit mask */
};
```

- *fds* argument to *poll()* is list of file descriptors to monitor
- For each list element:
 - *events*: bit mask of **events to monitor** for *fd*
 - Input value, initialized by caller
 - *revents*: returned bit mask of **events that occurred** for *fd*
 - Output value, set by kernel

poll() events bits

Bit	Input in <i>events</i> ?	Output in <i>revents</i> ?	Description
POLLIN	•	•	Normal-priority data can be read
POLLPRI	•	•	High-priority data/exceptional condition
POLLRDHUP	•	•	Shutdown on peer socket
POLLOUT	•	•	Data can be written
POLLERR		•	An error has occurred
POLLHUP		•	A hangup occurred
POLLNVAL		•	File descriptor is not open

- POLLIN, POLLPRI, and POLLRDHUP indicate **input** events
- POLLOUT indicates an **output** event
- POLLERR, POLLHUP, and POLLNVAL are returned in *revents* to provide **additional info** about FD
 - Ignored if specified in *events*

poll() events bits

A few *poll()* events bits need some explanation:

- **POLLPRI:**
 - State change on pseudoterminal master in packet mode
 - Out-of-band data on stream socket
 - (Rarely used)
- **POLLHUP:**
 - Returned on read end of pipe/FIFO if write end is closed
- **POLLERR:**
 - Returned on write end of pipe/FIFO if read end is closed
- **POLLRDHUP:**
 - Stream socket peer has closed (writing half of) connection
 - Linux-specific, since kernel 2.6.17
 - Useful with *epoll* edge-triggered mode (see *epoll_ctl(2)*)
- POSIX is vague on specifics; details vary across systems

[TLPI §63.2.3]

poll() timeout

- *timeout* determines blocking behavior of *poll()*:
 - -1: block indefinitely
 - 0: don't block ("poll" current state of descriptors)
 - > 0: block for up to *timeout* milliseconds
- When blocking, *poll()* waits until either:
 - A file **descriptor becomes ready**
 - A **signal handler interrupts** the call
 - The **timeout** is reached

poll() return value

Return value from *poll()* is one of:

- > 0 : number of ready FDs
 - I.e., number of elements in *pollfd* array that have *revents* $\neq 0$
- 0 : *poll()* timed out without any FD becoming ready
- -1 : error

Example: altio/poll_pipes.c

```
./poll_pipes num-pipes [num-writes]
```

- Create *num-pipes* pipes
- Loop *num-writes* times, each time writing a single byte to the write end of a randomly selected pipe
- Employ *poll()* to monitor all of the pipe read ends to see which pipes are readable
- Scan the *pollfd* array returned by *poll()* and print list of readable pipes

Example: altio/poll_pipes.c

```
1 int numPipes, ready, randPipe, numWrites, j;
2 struct pollfd *pollFd;
3 int (*pfd)[2]; /* File descriptors for all pipes */
4
5 numPipes = getInt(argv[1], GN_GT_0, "num-pipes");
6 numWrites = (argc > 2) ?
7             getInt(argv[2], GN_GT_0, "num-writes") : 1;
8
9 pfd = calloc(numPipes, sizeof(int [2]));
10 pollFd = calloc(numPipes, sizeof(struct pollfd));
```

- Because number of pipes is selected at run-time, we must allocate structures at run time
- `getInt()` converts string to integer
- Allocate array for pipe pairs
 - `calloc()` == `malloc(nmemb * size)`, and also zeroes memory
- Allocate `pollfd` array

Example: altio/poll_pipes.c

```
1 for (j = 0; j < numPipes; j++)
2     pipe(pfd[j]);
3
4 srand((int) time(NULL)); /* Seed RNG */
5 for (j = 0; j < numWrites; j++) {
6     randPipe = random() % numPipes;
7     printf("Writing to fd: %3d (read fd: %3d)\n",
8           pfd[randPipe][1], pfd[randPipe][0]);
9     write(pfd[randPipe][1], "a", 1);
10 }
```

- Create pipe pairs
- Loop `num-writes` times, writing a byte to a randomly selected pipe
 - Display FD for write and read end of pipe

Example: altio/poll_pipes.c

```
1 for (j = 0; j < numPipes; j++) {
2     pollFd[j].fd = pfd[j][0];
3     pollFd[j].events = POLLIN;
4 }
5 ready = poll(pollFd, numPipes, 0);
6
7 printf("poll() returned: %d\n", ready);
8
9 for (j = 0; j < numPipes; j++)
10     if (pollFd[j].revents & POLLIN)
11         printf("Readable: %3d\n", pollFd[j].fd);
```

- Build *pollfd* array containing all pipe read ends
 - Monitor to see if input is possible (*POLLIN*)
- Call *poll()* with zero *timeout*
- Return value from *poll()* is number of ready FDs
- Walk through *revents* fields in *pollfd* array, to see which FDs are ready for reading

Exercise

- 1 Write a program (*[template: altio/ex.poll_pipes_write.c]*) that has the following command-line syntax:

```
./poll_pipes_write num-pipes [num-writes [block-size]]
```

The program should create *num-pipes* pipes, and make the write ends of each pipe nonblocking (set the *O_NONBLOCK* flag with *fcntl(F_SETFL)*; see slide 7-6).

The program should then loop *num-writes* (default: 1) times, each time writing *block-size* (arbitrary) bytes (default: 100) to a randomly selected pipe. During the loop, the program should count the number of writes that failed because the pipe was full (*write()* failed with *EAGAIN* in *errno*) and the number of partial writes (*write()* wrote fewer bytes than requested).

After the above loop completes, the program should employ a (nonblocking) *poll()* call to monitor all of the pipe **write** ends to see which pipes are still writable, and then report the following:

- A list of the pipes that are writable
- The total number of partial writes
- The total number of times that *write()* failed with *EAGAIN*

Vary the command-line arguments until you see instances of *EAGAIN* errors and partial writes. Can you discover any rule about the minimum *block-size* needed in order to see partial writes?

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: <code>poll()</code>	7-14
7.5	Problems with <code>poll()</code> and <code>select()</code>	7-31
7.6	The <code>epoll</code> API	7-34
7.7	<code>epoll</code> events	7-45
7.8	<code>epoll</code> : edge-triggered notification	7-59
7.9	<code>epoll</code> : API quirks	7-70
7.10	Event-loop programming	7-76

Problems with `poll()` and `select()`

- `poll()` + `select()` are portable, long-standing, and widely used
- But, there are scalability problems when monitoring many FDs, because, on each call:
 - 1 Program passes a data structure to kernel describing **all** FDs to be monitored
 - 2 The kernel must recheck **all** specified FDs for readiness
 - This includes hooking (and subsequently unhooking) all FDs to handle case where it is necessary to block
 - 3 The kernel passes a modified data structure describing readiness of **all** FDs back to program in user space
 - 4 After the call, the program must inspect readiness state of **all** FDs in modified data
- \Rightarrow Cost of `select()` and `poll()` scales with number of FDs being monitored

[TLPI §63.2.5]

Problems with *poll()* and *select()*

- *poll()* and *select()* have a design problem:
 - Typically, set of FDs monitored by application is static
 - (Or set changes only slowly)
 - But, kernel doesn't remember monitored FDs between calls
 - \Rightarrow Info on all FDs must be copied back & forth on each call
- *epoll* improves performance by fixing this design problem
 - Kernel maintains a persistent set of FDs that application is interested in
 - Application can **incrementally** change "interest list"
- *epoll* cost **scales according to number of I/O events**
 - **Much better performance when monitoring many FDs!**
 - Signal-driven I/O scales similarly, for same reasons

[TLPI §63.4.5]

Outline

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: <code>poll()</code>	7-14
7.5	Problems with <code>poll()</code> and <code>select()</code>	7-31
7.6	The <code>epoll</code> API	7-34
7.7	<code>epoll</code> events	7-45
7.8	<code>epoll</code> : edge-triggered notification	7-59
7.9	<code>epoll</code> : API quirks	7-70
7.10	Event-loop programming	7-76

Overview

- Like `select()` and `poll()`, `epoll` can monitor multiple FDs
- `epoll` returns readiness information in similar manner to `poll()`
- Two main **advantages**:
 - `epoll` provides **much better performance** when monitoring large numbers of FDs (see TLPI §63.4.5)
 - `epoll` provides two **notification modes**: **level-triggered** and **edge-triggered**
 - Default is level-triggered notification
 - `select()` and `poll()` provide only level-triggered notification
 - (Signal-driven I/O provides only edge-triggered notification)
- Linux-specific, since kernel 2.6.0

[TLPI §63.4]

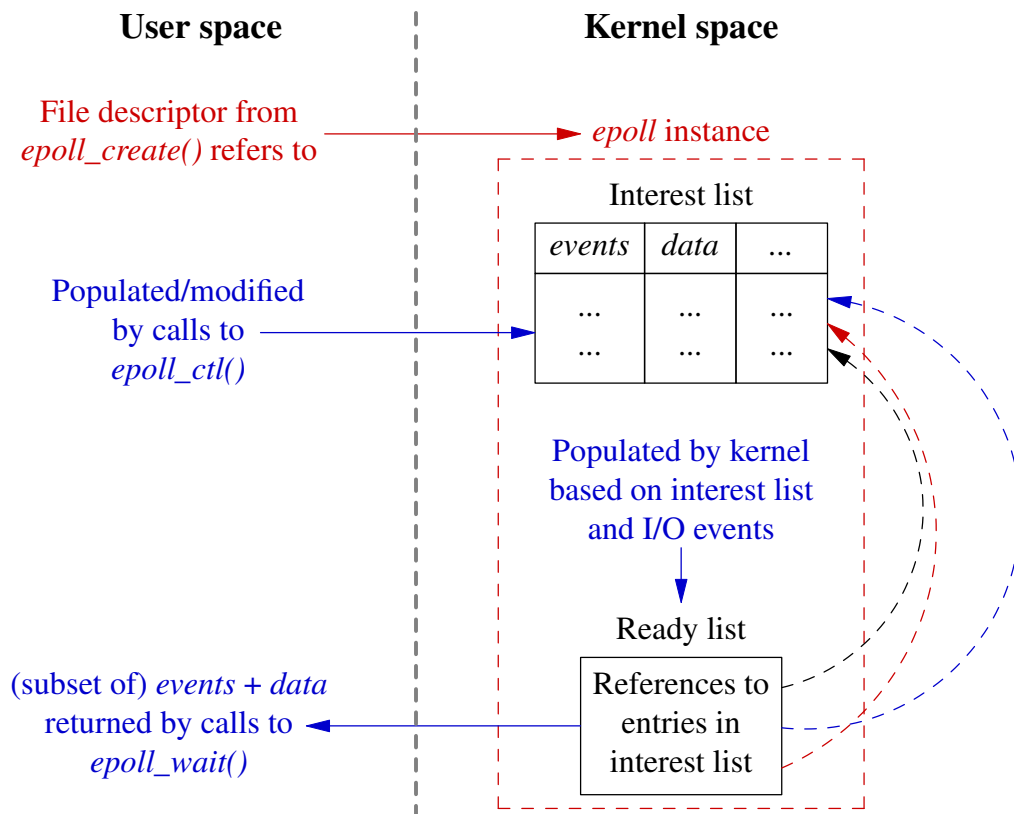
Central data structure of *epoll* API is an *epoll* instance

- **Persistent** data structure **maintained in kernel space**
 - Referred to in user space via file descriptor
- Can (abstractly) be considered as container for two lists:
 - **Interest list**: list of FDs to be monitored
 - **Ready list**: list of FDs that are ready for I/O
 - Ready list is (dynamic) subset of interest list

The key *epoll* APIs are:

- *epoll_create()*: create a new *epoll* instance and return FD referring to instance
 - FD is used in the calls below
- *epoll_ctl()*: modify interest list of *epoll* instance
 - Add FDs to/remove FDs from interest list
 - Modify events mask for FDs currently in interest list
- *epoll_wait()*: return items from ready list of *epoll* instance

epoll kernel data structures and APIs



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

```
#include <sys/epoll.h>
int epoll_create(int size);
```

- Creates an *epoll* instance
- *size*:
 - Since Linux 2.6.8: serves no purpose, but must be > 0
 - Before Linux 2.6.8: an *estimate* of number of FDs to be monitored via this *epoll* instance
- Returns file descriptor on success, or -1 on error
 - When FD is no longer required, it should be closed via *close()*
- Since Linux 2.6.27, *epoll_create1()* provides improved API
 - See the man page

Modifying the *epoll* interest list: *epoll_ctl()*

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd,
              struct epoll_event *ev);
```

- Modifies the interest list associated with *epoll* FD, *epfd*
- *fd*: identifies which FD in interest list is to have its settings modified
 - E.g., FD for pipe, FIFO, terminal, socket, POSIX MQ, or even another *epoll* FD
 - (Can't be FD for a regular file or directory)
- *op*: operation to perform on interest list
- *ev*: (Later)

[TLPI §63.4.2]

epoll_ctl() *op* argument

The *epoll_ctl()* *op* argument is one of:

- **EPOLL_CTL_ADD**: add *fd* to interest list of *epfd*
 - *ev* specifies events to be monitored for *fd*
 - If *fd* is already in interest list ⇒ **EEXIST**
- **EPOLL_CTL_MOD**: modify settings of *fd* in interest list of *epfd*
 - *ev* specifies new settings to be associated with *fd*
 - If *fd* is not in interest list ⇒ **ENOENT**
- **EPOLL_CTL_DEL**: remove *fd* from interest list of *epfd*
 - Also removes corresponding entry in ready list, if present
 - *ev* is ignored
 - If *fd* is not in interest list ⇒ **ENOENT**
 - **Closing an FD automatically removes it from all *epoll* interest lists**
 - ⚠ But see later! Manual deletion is sometimes required

The `epoll_event` structure

`epoll_ctl()` *ev* argument is pointer to an `epoll_event` structure:

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

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;
```

- ***ev.events***: bit mask of events to monitor for *fd*
 - (Similar to *events* mask given to `poll()`)
- ***data***: info to be passed back to caller of `epoll_wait()` when *fd* later becomes ready
 - **Union field**: value is specified in *one* of the members

Example: using `epoll_create()` and `epoll_ctl()`

```
int epfd;
struct epoll_event ev;

epfd = epoll_create(5);

ev.data.fd = fd;
ev.events = EPOLLIN; /* Monitor for input available */
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```

Outline

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: poll()	7-14
7.5	Problems with poll() and select()	7-31
7.6	The epoll API	7-34
7.7	epoll events	7-45
7.8	epoll: edge-triggered notification	7-59
7.9	epoll: API quirks	7-70
7.10	Event-loop programming	7-76

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- Returns info about ready FDs in interest list of *epoll* instance of *epfd*
- Blocks until at least one FD is ready
- Info about ready FDs is returned in array *evlist*
 - I.e., can get information about multiple ready FDs with one *epoll_wait()* call
 - (Caller allocates the *evlist* array)
- *maxevents*: size of the *evlist* array

[TLPI §63.4.3]

Waiting for events: `epoll_wait()`

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- *timeout* specifies a timeout for call:
 - -1: block until an FD in interest list becomes ready
 - 0: perform a nonblocking “poll” to see if any FDs in interest list are ready
 - > 0: block for up to *timeout* milliseconds or until an FD in interest list becomes ready
- Return value:
 - > 0: number of items placed in *evlist*
 - 0: no FDs became ready within interval specified by *timeout*
 - -1: an error occurred

Waiting for events: `epoll_wait()`

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- Info about **multiple** FDs can be returned in the array *evlist*
- Each element of *evlist* returns info about one file descriptor:
 - *events* is a bit mask of events that have occurred for FD
 - *data* is *ev.data* value *currently* associated with FD in the interest list
- **NB:** the FD itself is **not** returned!
 - Instead, we put FD into *ev.data.fd* when calling `epoll_ctl()`, so that it is returned via `epoll_wait()`
 - (Or, put FD into a structure pointed to by *ev.data.ptr*)

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- 🍷 If $> \text{maxevents}$ FDs are ready, successive *epoll_wait()* calls round-robin through FDs
 - Helps prevent file descriptor starvation
- 🍷 In multithreaded programs:
 - One thread can modify interest list (*epoll_ctl()*) while another thread is blocked in *epoll_wait()*
 - *epoll_wait()* call will return if a newly added FD becomes ready

epoll events

Following table shows:

- Bits given in *ev.events* to *epoll_ctl()*
- Bits returned in *evlist[i].events* by *epoll_wait()*

Bit	<i>epoll_ctl()</i> ?	<i>epoll_wait()</i> ?	Description
EPOLLIN	•	•	Normal-priority data can be read
EPOLLPRI	•	•	High-priority data can be read
EPOLLRDHUP	•	•	Shutdown on peer socket
EPOLLOUT	•	•	Data can be written
EPOLLONESHOT	•		Disable monitoring after event notification
EPOLLET	•		Employ edge-triggered notification
EPOLLERR		•	An error has occurred
EPOLLHUP		•	A hangup occurred

- Other than **EPOLLOUT** and **EPOLLET**, bits have same meaning as similarly named *poll()* bit flags

Example: altio/epoll_input.c

```
./epoll_input file...
```

- Monitors one or more files using *epoll* API to see if input is possible
- Suitable files to give as arguments are:
 - FIFOs
 - Terminal device names
 - (May need to run *sleep* command in FG on the other terminal, to prevent shell stealing input)
 - Standard input
 - `/dev/stdin`

Example: altio/epoll_input.c (1)

```
#define MAX_BUF      1000    /* Max. bytes for read() */
#define MAX_EVENTS    5
    /* Max. number of events to be returned from
       a single epoll_wait() call */

int epfd, ready, fd, s, j, numOpenFds;
struct epoll_event ev;
struct epoll_event evlist[MAX_EVENTS];
char buf[MAX_BUF];

epfd = epoll_create(argc - 1);
```

- Declarations for various variables
- Create an *epoll* instance, obtaining *epoll* FD

Example: altio/epoll_input.c (2)

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

    ev.events = EPOLLIN;
    ev.data.fd = fd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
}

numOpenFds = argc - 1;
```

- Open each of the files named on command line
- Each file is monitored for input (**EPOLLIN**)
- *fd* placed in *ev.data*, so it is returned by *epoll_wait()*
- Add the FD to *epoll* interest list (*epoll_ctl()*)
- Track the number of open FDs

Example: altio/epoll_input.c (3)

```
while (numOpenFds > 0) {
    printf("About to epoll_wait()\n");
    ready = epoll_wait(epfd, evlist, MAX_EVENTS, -1);
    if (ready == -1) {
        if (errno == EINTR)
            continue;          /* Restart if interrupted
                                by signal */
        else
            errExit("epoll_wait");
    }
    printf("Ready: %d\n", ready);
}
```

- Loop, fetching *epoll* events and analyzing results
- Loop terminates when all FDs has been closed
- *epoll_wait()* call places up to **MAX_EVENTS** events in *evlist*
 - *timeout == -1* ⇒ infinite timeout
- Return value of *epoll_wait()* is number of ready FDs

Example: altio/epoll_input.c (4)

```
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 " : "");
    if (evlist[j].events & EPOLLIN) {
        s = read(evlist[j].data.fd, buf, MAX_BUF);
        printf("    read %d bytes: %.*s\n", s, s, buf);
    } else if (evlist[j].events & (EPOLLHUP | EPOLLERR)) {
        printf("    closing fd %d\n", evlist[j].data.fd);
        close(evlist[j].data.fd);
        numOpenFds--;
    }
}
```

- Scan up to *ready* items in *evlist*
- Display *events* bits
- If *EPOLLIN* event occurred, read some input and display it on *stdout*
 - *%.s* ⇒ print string with field width taken from argument list (*s*)
- Otherwise, if error or hangup, close FD and decrements FD count
- Code correctly handles case where both *EPOLLIN* and *EPOLLHUP* are set in *evlist[j].events*

Exercises

- 1 Write a client ([*template: altio/ex.is_chat_cl.c*]) that communicates with the TCP chat server program, *is_chat_sv.c*. The program should be run with the following command line:

```
./is_chat_cl <host> <port> [<nickname>]
```

The program should create a connection to the server, and then use the *epoll* API to monitor both the terminal and the TCP socket for input. All input that becomes available on the socket should be written to the terminal and vice versa.

- Each time the program sends input from the terminal to the socket, that input should be prepended by the nickname supplied on the command line. If no nickname is supplied, then use the string returned by *getlogin(3)*. (*snprintf(3)* provides an easy way to concatenate the strings.)
- The program should terminate if it detects end-of-file or an error condition on either file descriptor.
- Calling *epoll_wait()* with *maxevents==1* will simplify the code!
- Bonus points if you find a way to crash the server (reproducibly)!

Exercises

- 2 Write the chat server ([template: altio/ex.is_chat_sv.c]).

Note the following points:

- The program should take one command-line argument: the port number to which it should bind its listening socket.
- The program should accept and handle multiple simultaneous client connections. Input read from any client should be broadcast to all other clients.
- Use the *epoll* API to manage the file descriptors.
- You should use nonblocking file descriptors to ensure that the server never blocks when accepting connections or when reading or writing to clients.
- When the server detects end-of file or an error (other than *EAGAIN*) while reading or writing on a client connection, it should close that connection. (Remember that closing a file descriptor automatically removes it from any *epoll* interest lists of which it is a member.)

Exercises

- 3 Write a program ([template: altio/ex.epoll_pipes.c]) which performs the same task as the *altio/poll_pipes.c* program, but uses the *epoll* API instead of *poll()*.

Hints:

- After writing to the pipes, you will need to call *epoll_wait()* in a loop. The loop should be terminated when *epoll_wait()* indicates that there are no more ready file descriptors.
- After each call to *epoll_wait()*, you should display each ready pipe read file descriptor and then drain all input from that file descriptor so that it does not indicate as ready in future calls to *epoll_wait()*.
- In order to drain a pipe without blocking, you will need to make the file descriptor for the read end of the pipe nonblocking.

Outline

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: <code>poll()</code>	7-14
7.5	Problems with <code>poll()</code> and <code>select()</code>	7-31
7.6	The <code>epoll</code> API	7-34
7.7	<code>epoll</code> events	7-45
7.8	<code>epoll</code> : edge-triggered notification	7-59
7.9	<code>epoll</code> : API quirks	7-70
7.10	Event-loop programming	7-76

Edge-triggered notification

- By default, `epoll` provides **level-triggered** (LT) notification
 - Tells us whether an **I/O operation can be performed on FD without blocking**
 - Like `poll()` and `select()`
- **EPOLLET** provides **edge-triggered** (ET) notification
 - Has I/O **activity occurred since `epoll_wait()` last notified about this FD?**
 - Or, if no `epoll_wait()` since FD was added/modified by `epoll_ctl()`, then: is FD ready?
- Example:

```
struct epoll_event ev;  
ev.data.fd = fd  
ev.events = EPOLLIN | EPOLLET;  
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```

[TLPI §63.4.6]

Edge-triggered notification

- Illustration of difference between LT and ET notification:
 - ① Monitoring a socket for input possible (**EPOLLIN**)
 - ② Input arrives on socket
 - ③ We call `epoll_wait()`, which informs us that FD is ready
 - We *perhaps* consume some (**but not all**) available input
 - **No further input arrives on socket**
 - ④ We call `epoll_wait()` again
- LT notification: second `epoll_wait()` would (again) report FD as ready
 - Because outstanding data is still available for reading
- ET notification: second `epoll_wait()` would **not** report FD as ready
 - Because no I/O activity occurred since previous `epoll_wait()`

Uses for edge-triggered notification

- Can be more efficient: application is not repeatedly reminded that FD is ready
- Example: application that (periodically) generates data to be written to a socket
 - Application does not always have data to write
 - Application monitors socket for writability (**EPOLLOUT**)
 - Application is also monitoring other FDs for I/O possible
 - At some point, socket is full (output not possible)
 - Peer drains some data, socket becomes writable
 - LT notification: every `epoll_wait()` would (immediately) wake and say FD is writable
 - ET notification: only first `epoll_wait()` would say FD is writable
 - Application could cache that info for later action (e.g., when data is generated)

Edge-triggered notification provides an optimization

- Scenario: multiple threads/processes are `epoll_wait()`-ing on same `epoll` FD
 - E.g., `epoll` FD is monitoring listening socket
 - LT notification: **all** waiters are woken up when connection request arrives
 - ET notification: only **one** waiter is woken up
 - Avoids thundering herd problem
 - Code examples: `altio/multithread_epoll_wait.c`, `altio/epoll_flags_fork.c`
 - The `EPOLLEXCLUSIVE` flag provides a similar behavior in some scenarios when using level-triggered notification
 - Since Linux 4.5
 - See `epoll_ctl(2)` and `altio/epoll_flags_fork.c`

Edge-triggered notification and EPOLLONESHOT

- Scenario: monitoring socket for input available with `EPOLLET`
 - Assumption: we want to know when input is available, but don't want to read it **yet**
 - (So, we use `EPOLLET` to avoid repeated notifications)
- New input keeps appearing \Rightarrow ET notification keeps notifying
 - Really, we needed only **first** notification
- Solution: `EPOLLONESHOT`

One-shot monitoring: EPOLLONESHOT

- Specifying `EPOLLONESHOT` in *events* causes FD to be reported just once by `epoll_wait()`
- FD is then marked inactive in interest list
- FD remains in interest list, and can be reactivated using `epoll_ctl(EPOLL_CTL_MOD)`
 - Continuing previous example: reenables notification after draining input from socket

[TLPI §63.4.3]

Using edge-triggered notification

- Normally **employed with nonblocking I/O**
 - Can't monitor "I/O level", so must do nonblocking I/O calls until no more I/O is possible
 - Otherwise: risk blocking when doing I/O
- **Beware of FD starvation**
 - Scenarios where responding to a busy FD leaves other ready FDs starved of attention
 - (Starvation scenarios can also occur with level-triggered notification)
 - See TLPI §63.4.6

Exercises

The `altio/i_epoll.c` program can be used to perform epoll monitoring and file I/O operations on the objects named in its command-line arguments. The program is interactive, and supports the following commands:

```
p [<timeout>]
    Do epoll_wait() with millisecond timeout (default: 0)
e <fd> [<flags>]
    Modify epoll settings of <fd>; <flags> can include:
    'r' - EPOLLIN
    'w' - EPOLLOUT
    'e' - EPOLLET
    'o' - EPOLLONESHOT
    If no flags are given, disable <fd> in the interest list
r <fd> <size>
    Blocking read of <size> bytes from <fd>
R <fd> <size>
    Nonblocking read of <size> bytes from <fd>
w <fd> <size> [<char>]
    Blocking write of <size> bytes to <fd>; <char> is character
    to write (default: 'x')
W <fd> <size> [<char>]
    Nonblocking write of <size> bytes to <fd>
```

Each command-line argument has the form `<path>[:<flags>]` (to open a file) or `s%<host>%<port>[:<flags>]` (to connect a socket to a specified host/port). `<flags>` is as described above, and defaults to "r". (If testing with sockets, you will find the command `ncat -l <port>` useful, in order to create a server that you can connect to.)

Exercises

The following exercises are intended to demonstrate the effect of the `EPOLLET` and `EPOLLONESHOT` flags.

- 1 In separate windows, create two FIFOs and use `cat` to write to each FIFO:

```
mkfifo x
cat > x
```

```
mkfifo y
cat > y
```

- 2 Run the `i_epoll` program, using it to monitor both FIFOs for reading, specifying the `EPOLLET` flag for the FIFO `y`; note the file descriptor numbers used for each FIFO:

```
./i_epoll x:r y:re
```

- 3 Type some input into both `cat` commands, and then use the "p" command to perform an `epoll_wait()`:

```
i_epoll> p
```

You should find that both file descriptors report as ready for reading (`EPOLLIN`).


Exercises

- 4 Enter the “p” command again. You should find that only the FIFO `x` reports `EPOLLIN`. (`y` does not report as ready because no new input has appeared on the FIFO.)
- 5 Type some input into the `cat` command that is writing to the FIFO `y`, and once more use the “p” command to perform an `epoll_wait()`. You should find that both FIFOs report `EPOLLIN`. (`y` reports as ready again because new input has appeared on the FIFO.)
- 6 Switch the monitoring of the FIFO `y` to use `EPOLLET` and `EPOLLONESHOT` with the command “e <fd> reo”.
- 7 Type some input into the FIFO `y`, and then use the “p” command to perform an `epoll_wait()`. You should find that both `x` and `y` report `EPOLLIN`.
- 8 Type some more input into the FIFO `y`, and again use the “p” command to perform an `epoll_wait()`. You should find that `y` does not report as ready (because, after it reported as ready in the previous step, it was disabled in the interest list by `EPOLLONESHOT`).
- 9 Reenable the FIFO `y` in the interest list using the command “e <fd> re” and again use the “p” command to perform an `epoll_wait()`. You should find that `y` reports `EPOLLIN`.
- 10 Try any other experiments you might think of!

Outline

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: poll()	7-14
7.5	Problems with poll() and select()	7-31
7.6	The epoll API	7-34
7.7	epoll events	7-45
7.8	epoll: edge-triggered notification	7-59
7.9	epoll: API quirks	7-70
7.10	Event-loop programming	7-76

epoll and duplication of file descriptors

- Entries in *epoll* interest list are associated with **combination** of file descriptor and open file description
 - Not just FD alone
-  Lifetime of interest list entry == lifetime of OFD
 - Can provide some surprises when FDs are duplicated...

epoll and duplication of file descriptors

- Suppose that *fd* in code below refers to a socket...

```
ev.events = EPOLLIN;
ev.data.fd = fd
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
newfd = dup(fd);
close(fd);
epoll_wait(epfd, ...);
```

- What happens if some input now arrives on the socket?
- *epoll_wait()* might still return events for registration of *fd*
 - Because open file description is still alive and present in interest list
 - OFD is kept alive by *newfd*
 - ⚠ Notifications return data given in registration of *fd*!!

epoll and duplication of file descriptors

- Analogous scenarios possible with *fork()*:

```
ev.events = EPOLLIN;
ev.data.fd = fd
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
if (fork() == 0) {
    /* Child continues, does not close 'fd' */
} else {
    close(fd);
    epoll_wait(epfd, ...);
}
```

epoll and duplication of file descriptors

- ⚠ Can't `EPOLL_CTL_DEL` *fd* after `close()`
 - \Rightarrow `EBADF`
- Must either:
 - Close duplicate FDs
 - ⚠ But you may not know about duplicate if it was created by a library function that used `dup()` or `fork()`
 - Or manually `EPOLL_CTL_DEL` *fd* **before** closing it

7	Alternative I/O Models	7-1
7.1	Overview	7-3
7.2	Nonblocking I/O	7-5
7.3	Signal-driven I/O	7-11
7.4	I/O multiplexing: <code>poll()</code>	7-14
7.5	Problems with <code>poll()</code> and <code>select()</code>	7-31
7.6	The <code>epoll</code> API	7-34
7.7	<code>epoll</code> events	7-45
7.8	<code>epoll</code> : edge-triggered notification	7-59
7.9	<code>epoll</code> : API quirks	7-70
7.10	Event-loop programming	7-76

Event-loop programming

- *`select()`*/*`poll()`*/*`epoll`* lend themselves to **event-loop programming**
 - I.e., program just sits in a loop, waiting on events from file descriptors
 - Monitored FDs can include pipes, sockets, terminals, devices, inotify, and even other `epoll` instances
 - Events are processed synchronously
- Problem: some other events of interest are not (traditionally) synchronous/aren't monitorable via FDs:
 - Signals
 - Timer expirations
 - IPC synchronization events
 - E.g., semaphore is incremented (*`sem_post()`*)
 - Process state transitions
 - E.g., child process termination

Event-loop programming

- Linux solution: turn those other events into file descriptors:
 - Signals \Rightarrow `signalfd()`
 - Timers \Rightarrow `timerfd` (`timerfd_create()`, `timerfd_settime()`, ...)
 - Synchronization \Rightarrow `eventfd()`
 - Process state transitions \Rightarrow “PID” file descriptors
 - PID FDs are returned by `pidfd_open()`, `clone()`/`clone3()` `CLONE_PIDFD`
 - Currently (Linux 5.4), only process-termination transitions are notified
- Monitor FDs produced by those mechanisms along with other FDs, using `select()`/`poll()`/`epoll`

Notes