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
epoll instances

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

epoll APIs

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

User space

- File descriptor from `epoll_create()` refers to
- Populated/modified by calls to `epoll_ctl()`

Kernel space

- Interest list
  - Events | Data | ...
  - ... | ... | ...
  - ... | ... | ...

- Ready list
- Populated by kernel based on interest list and I/O events
- References to entries in interest list

Creating an epoll instance: `epoll_create()`

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

- Creates an epoll instance; returns FD referring to instance
- `size`:
  - Since Linux 2.6.8: serves no purpose, but must be \( \geq 0 \)
  - 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

[TLPI §63.4.1]
Modifying the `epoll` interest list: `epoll_ctl()`

```c
#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)

**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:

```c
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()`

```c
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

22 Alternative I/O Models 22-1
22.1 Overview 22-3
22.2 Nonblocking I/O 22-5
22.3 Signal-driven I/O 22-11
22.4 I/O multiplexing: poll() 22-14
22.5 Problems with poll() and select() 22-31
22.6 The epoll API 22-34
22.7 epoll events 22-45
22.8 epoll: edge-triggered notification 22-58
22.9 epoll: API quirks 22-67
22.10 Event-loop programming 22-71

Waiting for events: `epoll_wait()`

```c
#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
  - If > `maxevents` FDs are ready, successive `epoll_wait()` calls
    round-robin through FDs

[TLPI §63.4.3]
Waiting for events: **epoll_wait()**

```c
#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

[TLPI §63.4.3]

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**)

Linux/UNIX System Programming ©2017, Michael Kerrisk Alternative I/O Models 22-47 §22.7
**epoll events**

Following table shows:

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

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

- 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`
# 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

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)

```c
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)

```c
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)!

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.)
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.
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:

```c
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

-----

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 ⇒ 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: reenable 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.
Entries in `epoll` interest list are associated with open file descriptions, not file descriptors.

Can provide some surprises when FDs are duplicated...
Suppose that `fd` in code below refers to a socket...

```c
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`
  - Open file description is still alive (because of `newfd`)
-⚠️ Notifications return data given in registration of `fd`!!
-⚠️ Can't `EPOLL_CTL_DEL fd` after `close()` (`EBADF`)
  - Must either: close duplicate FDs
  - Or: manually `EPOLL_CTL_DEL fd` before closing it

Analogous scenarios possible with `fork()`:

```c
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, ...);
}
```