Alternative I/O Models: *epoll*

Michael Kerrisk, man7.org © 2023
March 2023
mtk@man7.org

**Outline**

- 21 Alternative I/O Models: *epoll* 21-1
  - 21.1 Problems with *poll()* and *select()* 21-3
  - 21.2 The *epoll* API 21-6
  - 21.3 *epoll* events 21-17
  - 21.4 *epoll*: edge-triggered notification 21-32
  - 21.5 *epoll*: API quirks 21-45
# 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.
- ⇒ 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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21 Alternative I/O Models: <em>epoll</em></td>
<td>21-1</td>
</tr>
<tr>
<td>21.1 Problems with <em>poll()</em> and <em>select()</em></td>
<td>21-3</td>
</tr>
<tr>
<td>21.2 The <em>epoll</em> API</td>
<td>21-6</td>
</tr>
<tr>
<td>21.3 <em>epoll</em> events</td>
<td>21-17</td>
</tr>
<tr>
<td>21.4 <em>epoll</em>: edge-triggered notification</td>
<td>21-32</td>
</tr>
<tr>
<td>21.5 <em>epoll</em>: API quirks</td>
<td>21-45</td>
</tr>
</tbody>
</table>

### 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
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
### Creating an `epoll` instance: `epoll_create()`

```c
#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()`

```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
  - Can be FD for pipe, FIFO, terminal, socket, POSIX MQ
  - Can also be an `epoll` FD
    - An `epoll` FD indicates as readable if interest list is nonempty
  - Can't be FD for a regular file or directory

### `epoll_ctl()` op argument

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

- **EPOLL_CTL_ADD**: add `fd` to interest list
  - `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
  - `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
  - Also removes corresponding entry in ready list, if present
  - `ev` is ignored
  - If `fd` is not in interest list ⇒ **ENOENT**
  - Closing FD automatically removes it from `epoll` interest lists
    - But this is not reliable: close does **not** occur in some cases! See later...
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);

ep.data.fd = fd;
ev.events = EPOLLIN; /* Monitor for readability */
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```
21 Alternative I/O Models: epoll

21.1 Problems with poll() and select() 21-3
21.2 The epoll API 21-6
21.3 epoll events 21-17
21.4 epoll: edge-triggered notification 21-32
21.5 epoll: API quirks 21-45

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
Waiting for events: \textit{epoll\_wait()}

\begin{verbatim}
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
\end{verbatim}

- \textit{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 \textit{timeout} milliseconds or until an FD in
    interest list becomes ready
    - \textit{epoll\_pwait2()} (Linux 5.11) allows nanosecond timeout

- Return value:
  - > 0: number of items placed in \textit{evlist}
  - 0: no FDs became ready within interval specified by \textit{timeout}
  - –1: an error occurred

Info about multiple FDs can be returned in the array \textit{evlist}

Each element of \textit{evlist} returns info about one file descriptor:

- \textit{events} is a bit mask of events that have occurred for FD
- \textit{data} is \textit{ev.data} value \textit{currently} associated with FD in the
  interest list

\textbf{NB}: the FD itself is \textbf{not} returned!

- Instead, we put FD into \textit{ev.data.fd} when calling \textit{epoll\_ctl()},
  so that it is returned via \textit{epoll\_wait()}
  - (Or, put FD into a structure pointed to by \textit{ev.data.ptr})
Waiting for events: `epoll_wait()`

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

- If `maxevents` FDs are ready, successive `epoll_wait()` calls round-robin through FDs
  - Helps prevent file descriptors being starved of attention
- 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[].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>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 `EPOLLONESHOT` and `EPOLLERR`, bits have same meaning as similarly named `poll()` bit flags
- `EPOLLIN, EPOLLPRI, EPOLLRDHUP, and EPOLLOUT` are returned by `epoll_wait()` only if specified when adding FD using `epoll_ctl()`

---

[TLPI §63.4.3]
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)

### Example: altio/epoll_input.c (1)

```c
#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)
- \textit{fd} placed in \textit{ev.data}, so it is returned by \textit{epoll_wait()}
- Add the \textit{FD} to \textit{epoll} interest list (\textit{epoll_ctl()})
- Track the number of open FDs

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 \textit{epoll} events and analyzing results
- Loop terminates when all FDs has been closed
- \textit{epoll_wait()} call places up to \textit{MAX_EVENTS} events in \textit{evlist}
  - \textit{timeout} == \texttt{-1} \Rightarrow infinite timeout
- Return value of \textit{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:

```bash
./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.

[Exercise continues on next slide]
Exercises

- Calling `epoll_wait()` with `maxevents==1` will simplify the code!
- As a simplification, you can assume that the socket is always writable (i.e., you don’t need to monitor for the socket for `EPOLLOUT`).
- Bonus points if you find a way to crash the server (reproducibly)!

Exercises

- 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 socket, it should remove that socket from the epoll interest list and close the socket.
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.