Linux/UNIX IPC Programming

Alternative I/O Models: *epoll*

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Rev: # 8d7fc39ab521

Outline

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
    - ⇒ 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
- *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

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 (2003)

[TLPI §63.4]

epoll instances

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

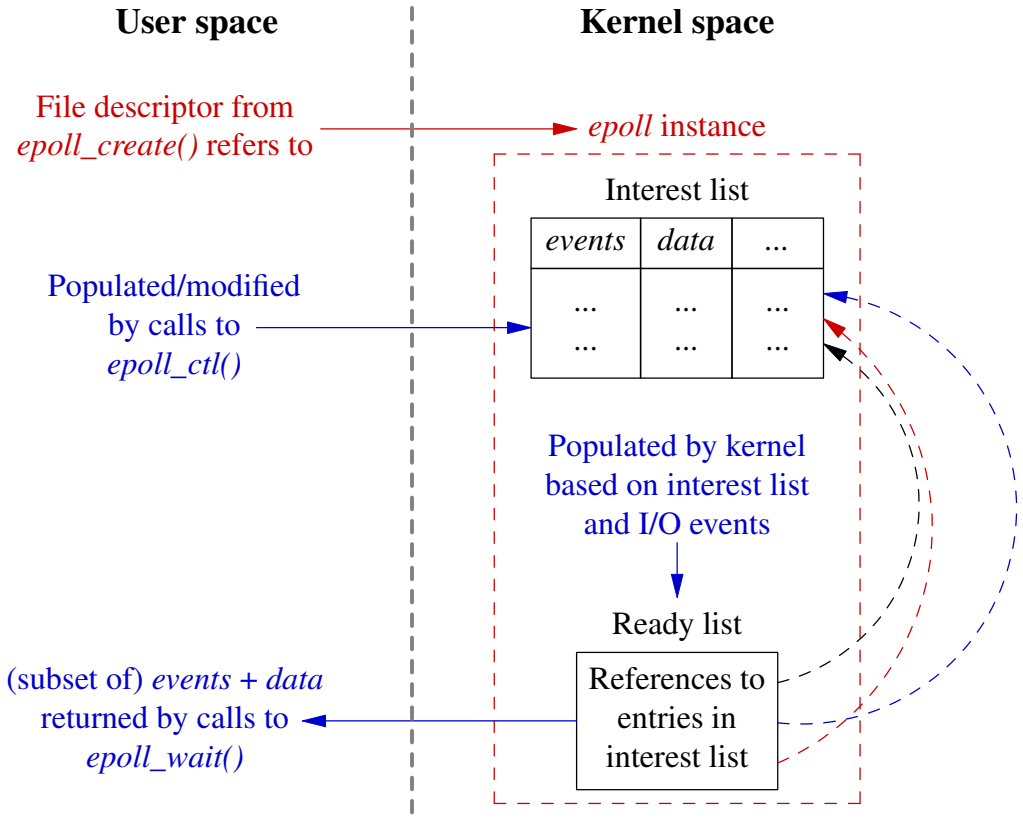
- **Persistent** data structure **maintained in kernel space**
  - Referred to in user space via file descriptor
- Can (abstractly) be considered as container for two lists:
  - **Interest list**: list of FDs to be monitored
  - **Ready list**: list of FDs that are ready for I/O
    - Ready list is (dynamic)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



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 manual page

[TLPI §63.4.1]

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

[TLPI §63.4.2]

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:

```
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 = epoll_create(5);

struct epoll_event ev;
ev.data.fd = fd;
ev.events = EPOLLIN;            /* Monitor for readability */

epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```

Outline

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
    - *epoll_pwait2()* (Linux 5.11) allows timeout with nanosecond precision
- 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 > *maxevents* FDs are ready, successive *epoll_wait()* calls round-robin through FDs
  - Helps prevent file descriptors being starved of attention
- 👍 In multithreaded programs:
  - While one thread is blocked in *epoll_wait()*, another thread can modify interest list (*epoll_ctl()*)
  - *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()*

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

- Other than EPOLLONESHOT and EPOLLET, 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]

Example: altio/epoll_read.c

```
./epoll_read 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 foreground on those terminals, to prevent shell stealing input)

Example: altio/epoll_read.c (1)

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

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

    struct epoll_event ev;
    ev.events = EPOLLIN;
    ev.data.fd = fd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
}
int numOpenFds = argc - 1;
```

- Create an *epoll* instance, obtaining *epoll* FD
- Open each of the files named on command line
- Monitor each file for input (EPOLLIN)
- Put *fd* into *ev.data*, so it is returned by *epoll_wait()*
- Add the FD to *epoll* interest list (*epoll_ctl()*)
- Track number of open FDs (in *numOpenFds*)

Example: altio/epoll_read.c (2)

```
while (numOpenFds > 0) {
    const int MAX_EVENTS = 5;
    struct epoll_event evlist[MAX_EVENTS];

    printf("About to epoll_wait()\n");
    int 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 no more FDs are open
- *epoll_wait()* call places up to MAX_EVENTS events in *evlist*
  - *timeout* == *–1* ⇒ infinite timeout
- Return value from *epoll_wait()* is number of ready FDs

Example: altio/epoll_read.c (3)

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

        const int BUF_SIZE = 10;
        char buf[BUF_SIZE];
        ssize_t nr = read(evlist[j].data.fd, buf, BUF_SIZE);
        if (nr == -1)
            errExit("read");
        ...
    }
```

- Iterate through *ready* items in *evlist*
- Display *events* bits for each ready FD
- Read from ready FD
  - Note that we don't even need to check *events*
    - EPOLLIN ⇒ *read()* won't block
    - EPOLLHUP ⇒ *read()* will return 0 (without blocking)

Example: altio/epoll_read.c (4)

```
    for (int j = 0; j < ready; j++) {
        ...
        if (nr == 0) {                  /* read() indicated end-of-file */
            printf("    closing fd %d\n", evlist[j].data.fd);

            epoll_ctl(epfd, EPOLL_CTL_DEL, evlist[j].data.fd, NULL);
            close(evlist[j].data.fd);
            numOpenFds--;
        } else {
            printf("    read %zd bytes: %.*s\n", nr, (int) nr, buf);
        }
    }
}
```

- If *read()* returned 0 (EOF):
  - Remove FD from *epoll* interest list
  - Close FD
- Otherwise, display data that was read
  - `%.*s` ⇒ field width taken from argument list (*s*)

Outline

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.)

   [Exercise continues on next slide]

Exercises

- Both the terminal and the socket will indicate as readable (EPOLLIN) when input becomes available or when an end-of-file condition occurs.
- The program should terminate if it detects end-of-file on either file descriptor.
- Calling *epoll_wait()* with *maxevents==1* will simplify the code!

```
struct epoll_event rev;
epoll_wait(epfd, &rev, 1, -1);
```

  (This is simpler, because then you don't have to iterate through an array that would in any case contain at most two entries.)
- 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

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

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.