Full credit is given to the above companies including the OS that this PDF file was generated!

Rocky Enterprise Linux 9.2 Manual Pages on command 'epoll.7'

$ man epoll.7

EPOLL(7) Linux Programmer's Manual EPOLL(7)

NAME

epoll - I/O event notification facility

SYNOPSIS

#include <sys/epoll.h>

DESCRIPTION

The epoll API performs a similar task to poll(2): monitoring multiple

file descriptors to see if I/O is possible on any of them. The epoll

API can be used either as an edge-triggered or a level-triggered inter?

face and scales well to large numbers of watched file descriptors.

The central concept of the epoll API is the epoll instance, an in-ker?

nel data structure which, from a user-space perspective, can be consid?

ered as a container for two lists:

? The interest list (sometimes also called the epoll set): the set of

file descriptors that the process has registered an interest in moni?

toring.

? The ready list: the set of file descriptors that are "ready" for I/O.

The ready list is a subset of (or, more precisely, a set of refer?

ences to) the file descriptors in the interest list. The ready list is dynamically populated by the kernel as a result of I/O activity on those file descriptors.

The following system calls are provided to create and manage an epoll instance:

? epoll_create(2) creates a new epoll instance and returns a file de? scriptor referring to that instance. (The more recent epoll_cre? ate1(2) extends the functionality of epoll_create(2).)

? Interest in particular file descriptors is then registered via epoll_ctl(2), which adds items to the interest list of the epoll in? stance.

? epoll_wait(2) waits for I/O events, blocking the calling thread if no events are currently available. (This system call can be thought of as fetching items from the ready list of the epoll instance.)

Level-triggered and edge-triggered

The epoll event distribution interface is able to behave both as edge-triggered (ET) and as level-triggered (LT). The difference between the two mechanisms can be described as follows. Suppose that this scenario happens:

1. The file descriptor that represents the read side of a pipe (rfd) is registered on the epoll instance.
2. A pipe writer writes 2 kB of data on the write side of the pipe.
3. A call to epoll_wait(2) is done that will return rfd as a ready file descriptor.
4. The pipe reader reads 1 kB of data from rfd.
5. A call to epoll_wait(2) is done.

If the rfd file descriptor has been added to the epoll interface using the EPOLLET (edge-triggered) flag, the call to epoll_wait(2) done in step 5 will probably hang despite the available data still present in the file input buffer; meanwhile the remote peer might be expecting a response based on the data it already sent. The reason for this is that edge-triggered mode delivers events only when changes occur on the monitored file descriptor. So, in step 5 the caller might end up wait?

ing for some data that is already present inside the input buffer. In the above example, an event on rfd will be generated because of the write done in 2 and the event is consumed in 3. Since the read opera? tion done in 4 does not consume the whole buffer data, the call to epoll_wait(2) done in step 5 might block indefinitely.

An application that employs the EPOLLET flag should use nonblocking file descriptors to avoid having a blocking read or write starve a task that is handling multiple file descriptors. The suggested way to use epoll as an edge-triggered (EPOLLET) interface is as follows:

a) with nonblocking file descriptors; and

b) by waiting for an event only after read(2) or write(2) return EA? GAIN.

By contrast, when used as a level-triggered interface (the default, when EPOLLET is not specified), epoll is simply a faster poll(2), and can be used wherever the latter is used since it shares the same seman? tics.

Since even with edge-triggered epoll, multiple events can be generated upon receipt of multiple chunks of data, the caller has the option to specify the EPOLLONESHOT flag, to tell epoll to disable the associated file descriptor after the receipt of an event with epoll_wait(2). When the EPOLLONESHOT flag is specified, it is the caller's responsibility to rearm the file descriptor using epoll_ctl(2) with EPOLL_CTL_MOD.

If multiple threads (or processes, if child processes have inherited the epoll file descriptor across fork(2)) are blocked in epoll_wait(2) waiting on the same epoll file descriptor and a file descriptor in the interest list that is marked for edge-triggered (EPOLLET) notification becomes ready, just one of the threads (or processes) is awoken from epoll_wait(2). This provides a useful optimization for avoiding "thun? dering herd" wake-ups in some scenarios.

Interaction with autosleep

If the system is in autosleep mode via /sys/power/autosleep and an event happens which wakes the device from sleep, the device driver will keep the device awake only until that event is queued. To keep the de?

vice awake until the event has been processed, it is necessary to use

the epoll_ctl(2) EPOLLWAKEUP flag.

When the EPOLLWAKEUP flag is set in the events field for a struct

epoll_event, the system will be kept awake from the moment the event is

queued, through the epoll_wait(2) call which returns the event until

the subsequent epoll_wait(2) call. If the event should keep the system

awake beyond that time, then a separate wake_lock should be taken be?

fore the second epoll_wait(2) call.

/proc interfaces

The following interfaces can be used to limit the amount of kernel mem?

ory consumed by epoll:

/proc/sys/fs/epoll/max_user_watches (since Linux 2.6.28)

This specifies a limit on the total number of file descriptors

that a user can register across all epoll instances on the sys?

tem. The limit is per real user ID. Each registered file de?

scriptor costs roughly 90 bytes on a 32-bit kernel, and roughly

160 bytes on a 64-bit kernel. Currently, the default value for

max_user_watches is 1/25 (4%) of the available low memory, di?

vided by the registration cost in bytes.

Example for suggested usage

While the usage of epoll when employed as a level-triggered interface

does have the same semantics as poll(2), the edge-triggered usage re?

quires more clarification to avoid stalls in the application event

loop. In this example, listener is a nonblocking socket on which lis?

ten(2) has been called. The function do_use_fd() uses the new ready

file descriptor until EAGAIN is returned by either read(2) or write(2).

An event-driven state machine application should, after having received

EAGAIN, record its current state so that at the next call to

do_use_fd() it will continue to read(2) or write(2) from where it

stopped before.

```
#define MAX_EVENTS 10

struct epoll_event ev, events[MAX_EVENTS];

int listen_sock, conn_sock, nfds, epollfd;
```

```
/* Code to set up listening socket, 'listen_sock',
   (socket(), bind(), listen()) omitted */
epollfd = epoll_create1(0);
if (epollfd == -1) {
    perror("epoll_create1");
    exit(EXIT_FAILURE);
}
ev.events = EPOLLIN;
ev.data.fd = listen_sock;
if (epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev) == -1) {
    perror("epoll_ctl: listen_sock");
    exit(EXIT_FAILURE);
}
for (;;) {
    nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
    if (nfds == -1) {
        perror("epoll_wait");
        exit(EXIT_FAILURE);
    }
    for (n = 0; n < nfds; ++n) {
        if (events[n].data.fd == listen_sock) {
            conn_sock = accept(listen_sock,
                               (struct sockaddr *) &addr, &addrlen);
            if (conn_sock == -1) {
                perror("accept");
                exit(EXIT_FAILURE);
            }
            setnonblocking(conn_sock);
            ev.events = EPOLLIN | EPOLLET;
            ev.data.fd = conn_sock;
            if (epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock,
                        &ev) == -1) {
                perror("epoll_ctl: conn_sock");
```

```
                exit(EXIT_FAILURE);
            }
        } else {
            do_use_fd(events[n].data.fd);
        }
    }
}
```

When used as an edge-triggered interface, for performance reasons, it is possible to add the file descriptor inside the epoll interface (EPOLL_CTL_ADD) once by specifying (EPOLLIN|EPOLLOUT). This allows you to avoid continuously switching between EPOLLIN and EPOLLOUT calling epoll_ctl(2) with EPOLL_CTL_MOD.

Questions and answers

0. What is the key used to distinguish the file descriptors registered in an interest list?

   The key is the combination of the file descriptor number and the open file description (also known as an "open file handle", the kernel's internal representation of an open file).

1. What happens if you register the same file descriptor on an epoll instance twice?

   You will probably get EEXIST. However, it is possible to add a du? plicate (dup(2), dup2(2), fcntl(2) F_DUPFD) file descriptor to the same epoll instance. This can be a useful technique for filtering events, if the duplicate file descriptors are registered with dif? ferent events masks.

2. Can two epoll instances wait for the same file descriptor? If so, are events reported to both epoll file descriptors?

   Yes, and events would be reported to both. However, careful pro? gramming may be needed to do this correctly.

3. Is the epoll file descriptor itself poll/epoll/selectable?

   Yes. If an epoll file descriptor has events waiting, then it will indicate as being readable.

4. What happens if one attempts to put an epoll file descriptor into

its own file descriptor set?

The epoll_ctl(2) call fails (EINVAL). However, you can add an epoll file descriptor inside another epoll file descriptor set.

5. Can I send an epoll file descriptor over a UNIX domain socket to another process?

   Yes, but it does not make sense to do this, since the receiving process would not have copies of the file descriptors in the inter? est list.

6. Will closing a file descriptor cause it to be removed from all epoll interest lists?

   Yes, but be aware of the following point. A file descriptor is a reference to an open file description (see open(2)). Whenever a file descriptor is duplicated via dup(2), dup2(2), fcntl(2) F_DUPFD, or fork(2), a new file descriptor referring to the same open file description is created. An open file description contin? ues to exist until all file descriptors referring to it have been closed.

   A file descriptor is removed from an interest list only after all the file descriptors referring to the underlying open file descrip? tion have been closed. This means that even after a file descrip? tor that is part of an interest list has been closed, events may be reported for that file descriptor if other file descriptors refer? ring to the same underlying file description remain open. To pre? vent this happening, the file descriptor must be explicitly removed from the interest list (using epoll_ctl(2) EPOLL_CTL_DEL) before it is duplicated. Alternatively, the application must ensure that all file descriptors are closed (which may be difficult if file de? scriptors were duplicated behind the scenes by library functions that used dup(2) or fork(2)).

7. If more than one event occurs between epoll_wait(2) calls, are they combined or reported separately?

   They will be combined.

8. Does an operation on a file descriptor affect the already collected

but not yet reported events?

You can do two operations on an existing file descriptor. Remove would be meaningless for this case. Modify will reread available I/O.

9. Do I need to continuously read/write a file descriptor until EAGAIN when using the EPOLLET flag (edge-triggered behavior)?

Receiving an event from epoll_wait(2) should suggest to you that such file descriptor is ready for the requested I/O operation. You must consider it ready until the next (nonblocking) read/write yields EAGAIN. When and how you will use the file descriptor is entirely up to you.

For packet/token-oriented files (e.g., datagram socket, terminal in canonical mode), the only way to detect the end of the read/write I/O space is to continue to read/write until EAGAIN.

For stream-oriented files (e.g., pipe, FIFO, stream socket), the condition that the read/write I/O space is exhausted can also be detected by checking the amount of data read from / written to the target file descriptor. For example, if you call read(2) by asking to read a certain amount of data and read(2) returns a lower number of bytes, you can be sure of having exhausted the read I/O space for the file descriptor. The same is true when writing using write(2). (Avoid this latter technique if you cannot guarantee that the monitored file descriptor always refers to a stream-ori? ented file.)

Possible pitfalls and ways to avoid them

o Starvation (edge-triggered)

If there is a large amount of I/O space, it is possible that by trying to drain it the other files will not get processed causing starvation. (This problem is not specific to epoll.)

The solution is to maintain a ready list and mark the file descriptor as ready in its associated data structure, thereby allowing the appli? cation to remember which files need to be processed but still round robin amongst all the ready files. This also supports ignoring subse?

quent events you receive for file descriptors that are already ready.

o If using an event cache...

If you use an event cache or store all the file descriptors returned from epoll_wait(2), then make sure to provide a way to mark its closure dynamically (i.e., caused by a previous event's processing). Suppose you receive 100 events from epoll_wait(2), and in event #47 a condition causes event #13 to be closed. If you remove the structure and close(2) the file descriptor for event #13, then your event cache might still say there are events waiting for that file descriptor causing confusion.

One solution for this is to call, during the processing of event 47, epoll_ctl(EPOLL_CTL_DEL) to delete file descriptor 13 and close(2), then mark its associated data structure as removed and link it to a cleanup list. If you find another event for file descriptor 13 in your batch processing, you will discover the file descriptor had been previi? ously removed and there will be no confusion.

VERSIONS

The epoll API was introduced in Linux kernel 2.5.44. Support was added to glibc in version 2.3.2.

CONFORMING TO

The epoll API is Linux-specific. Some other systems provide similar mechanisms, for example, FreeBSD has kqueue, and Solaris has /dev/poll.

NOTES

The set of file descriptors that is being monitored via an epoll file descriptor can be viewed via the entry for the epoll file descriptor in the process's /proc/[pid]/fdinfo directory. See proc(5) for further details.

The kcmp(2) KCMP_EPOLL_TFD operation can be used to test whether a file descriptor is present in an epoll instance.

SEE ALSO

epoll_create(2), epoll_create1(2), epoll_ctl(2), epoll_wait(2), poll(2), select(2)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.