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 interface and scales well to large numbers of watched file descriptors. The following system calls are provided to create and manage an **epoll** instance:

* **epoll_create**(2) creates an **epoll** instance and returns a file descriptor referring to that instance. (The more recent **epoll_create1**(2) extends the functionality of **epoll_create**(2).)
* Interest in particular file descriptors is then registered via **epoll_ctl**(2). The set of file descriptors currently registered on an **epoll** instance is sometimes called an *epoll* set.
* **epoll_wait**(2) waits for I/O events, blocking the calling thread if no events are currently available.

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 only delivers events when changes occur on the monitored file descriptor. So, in step **5** the caller might end up waiting 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 operation 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:

- **i** with nonblocking file descriptors; and
- **ii** by waiting for an event only after **read**(2) or **write**(2) return **EAGAIN**.

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 semantics.

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**.

/proc interfaces

The following interfaces can be used to limit the amount of kernel memory 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 system. The limit is per real user ID. Each registered file descriptor 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, divided 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 requires more clarification to avoid stalls in the application event loop. In this example, listener is a nonblocking socket on which **listen**(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;

/* Set up listening socket, 'listen_sock' (socket(),
   bind(), listen()) */

epollfd = epoll_create(10);
if (epollfd == −1) {
    perror("epoll_create");
    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_pwait");
        exit(EXIT_FAILURE);
    }

    for (n = 0; n < nfds; ++n) {
        if (events[n].data.fd == listen_sock) {
            conn_sock = accept(listen_sock,
                            (struct sockaddr *) &local, &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

Q0 What is the key used to distinguish the file descriptors registered in an **epoll** set?

A0 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).

Q1 What happens if you register the same file descriptor on an **epoll** instance twice?

A1 You will probably get **EEXIST**. However, it is possible to add a duplicate (**dup**(2), **dup2**(2), **fcntl**(2) **F_DUPFD**) descriptor to the same **epoll** instance. This can be a useful technique for filtering events, if the duplicate file descriptors are registered with different *events* masks.

Q2 Can two **epoll** instances wait for the same file descriptor? If so, are events reported to both **epoll** file descriptors?

A2 Yes, and events would be reported to both. However, careful programming may be needed to do this correctly.

Q3 Is the **epoll** file descriptor itself poll/epoll/selectable?

A3 Yes. If an **epoll** file descriptor has events waiting then it will indicate as being readable.

Q4 What happens if one attempts to put an **epoll** file descriptor into its own file descriptor set?

A4 The **epoll_ctl**(2) call will fail (**EINVAL**). However, you can add an **epoll** file descriptor inside another **epoll** file descriptor set.

Q5 Can I send an **epoll** file descriptor over a UNIX domain socket to another process?

A5 Yes, but it does not make sense to do this, since the receiving process would not have copies of the file descriptors in the **epoll** set.

Q6 Will closing a file descriptor cause it to be removed from all **epoll** sets automatically?

A6 Yes, but be aware of the following point. A file descriptor is a reference to an open file description (see **open**(2)). Whenever a 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 continues to exist until all file descriptors referring to it have been closed. A file descriptor is removed from an **epoll** set only after all the file descriptors referring to the underlying open file description have been closed (or before if the descriptor is explicitly removed using **epoll_ctl**(2) **EPOLL_CTL_DEL**). This means that even after a file descriptor that is part of an **epoll** set has been closed, events may be reported for that file descriptor if other file descriptors referring to the same underlying file description remain open.

Q7 If more than one event occurs between **epoll_wait**(2) calls, are they combined or reported separately?

A7 They will be combined.

Q8 Does an operation on a file descriptor affect the already collected but not yet reported events?

A8 You can do two operations on an existing file descriptor. Remove would be meaningless for this case. Modify will reread available I/O.

Q9 Do I need to continuously read/write a file descriptor until **EAGAIN** when using the **EPOLLET** flag (edge-triggered behavior) ?

A9 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-oriented 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 application to remember which files need to be processed but still round robin amongst all the ready files. This also supports ignoring subsequent 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 previously 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*.

SEE ALSO

epoll_create(2), **epoll_create1**(2), **epoll_ctl**(2), **epoll_wait**(2)

COLOPHON

This page is part of release 3.44 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.