NAME

epoll – I/O event notification facility

SYNOPSIS

#include <sys/epoll.h>

DESCRIPTION

epoll is a variant of **poll**(2) that can be used either as Edge or Level Triggered interface and scales well to large numbers of watched fds. Three system calls are provided to set up and control an **epoll** set: **epoll_create**(2)**, epoll_ctl**(2)**, epoll_wait**(2)**.**

An **epoll** set is connected to a file descriptor created by **epoll_create**(2)**.** Interest for certain file descriptors is then registered via **epoll_ctl(2)**. Finally, the actual wait is started by **epoll_wait**(2)**.**

NOTES

The **epoll** event distribution interface is able to behave both as Edge Triggered (ET) and Level Triggered (LT). The difference between ET and LT event distribution mechanism can be described as follows. Suppose that this scenario happens :

1 The file descriptor that represent the read side of a pipe (**RFD**) is added inside the **epoll** device.

2 Pipe writer writes 2Kb of data on the write side of the pipe.

3 A call to **epoll_wait**(2) is done that will return **RFD** as ready file descriptor.

4 The pipe reader reads 1Kb 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** flag, the call to **epoll_wait**(2) done in step **5** will probably hang because of the available data still present in the file input buffers and the remote peer might be expecting a response based on the data it already sent. The reason for this is that Edge Triggered event distribution delivers events only when events happens on the monitored file. 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 lock indefinitely. The **epoll** interface, when used with the **EPOLLET** flag (Edge Triggered) should use non-blocking file descriptors to avoid having a blocking read or write starve the task that is handling multiple file descriptors. The suggested way to use **epoll** as an Edge Triggered (**EPOLLET**) interface is below, and possible pitfalls to avoid follow.

i with non-blocking file descriptors

ii by going to wait for an event only after **read**(2) or **write**(2) return EAGAIN

On the contrary, when used as a Level Triggered interface, **epoll** is by all means a faster **poll**(2), and can be used wherever the latter is used since it shares the same semantics. Since even with the Edge Triggered **epoll** multiple events can be generated up on receival 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 receival of an event with **epoll_wait**(2). When the **EPOLLONESHOT** flag is specified, it is caller responsibility to rearm the file descriptor using **epoll_ctl**(2) with **EPOLL_CTL_MOD.**

EXAMPLE FOR SUGGESTED USAGE

While the usage of **epoll** when employed like a Level Triggered interface does have the same semantics of **poll**(2), an Edge Triggered usage requires more clarifiction to avoid stalls in the application event loop. In this example, listener is a non-blocking 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.

```
struct epoll_event ev, *events;

for(;;) {
   nfds = epoll_wait(kdpfd, events, maxevents, -1);

   for(n = 0; n < nfds; ++n) {
      if(events[n].data.fd == listener) {
         client = accept(listener, (struct sockaddr *) &local,
                     &addrlen);
         if(client < 0){
            perror("accept");
            continue;
         }
         setnonblocking(client);
         ev.events = EPOLLIN | EPOLLET;
         ev.data.fd = client;
         if (epoll_ctl(kdpfd, EPOLL_CTL_ADD, client, &ev) < 0) {
            fprintf(stderr, "epoll set insertion error: fd=%d0,
                  client);
            return -1;
         }
      }
      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 (from linux-kernel)

Q1 What happens if you add the same fd to an epoll_set twice?

A1 You will probably get EEXIST. However, it is possible that two threads may add the same fd twice. This is a harmless condition.

Q2 Can two **epoll** sets wait for the same fd? If so, are events reported to both **epoll** sets fds?

A2 Yes. However, it is not recommended. Yes it would be reported to both.

Q3 Is the **epoll** fd itself poll/epoll/selectable?

A3 Yes.

Q4 What happens if the **epoll** fd is put into its own fd set?

A4 It will fail. However, you can add an **epoll** fd inside another epoll fd set.

Q5 Can I send the **epoll** fd over a unix-socket to another process?

A5 No.

Q6 Will the close of an fd cause it to be removed from all **epoll** sets automatically?

A6 Yes.

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

A7 They will be combined.

Q8 Does an operation on an fd affect the already collected but not yet reported events?

A8 You can do two operations on an existing fd. Remove would be meaningless for this case. Modify will re-read available I/O.

Q9 Do I need to continuously read/write an fd until EAGAIN when using the **EPOLLET** flag (Edge Triggered behaviour) ?

A9 No you don't. Receiving an event from **epoll_wait**(2) should suggest to you that such file descriptor is ready for the requested I/O operation. You have simply to consider it ready until you will receive the next EAGAIN. When and how you will use such file descriptor is entirely up to you. Also, the condition that the read/write I/O space is exhausted can be detected by checking the amount of data read/write from/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 to have exhausted the read I/O space for such file descriptor. Same is valid when writing using the **write**(2) function.

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 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 fd's that are already ready.

o If using an event cache...

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

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

CONFORMING TO

epoll(4) is a new API introduced in Linux kernel **2.5.44**. Its interface should be finalized in Linux kernel **2.5.66**.

SEE ALSO

epoll_create(2) **epoll_ctl**(2) **epoll_wait**(2)