Asynchronous Network I/O – epoll

- The time-tested, traditional method of implementing TCP servers is to use the "one thread/process per connection" model. But this approach does not perform very well at all with today's high-end hardware and high-speed networks with high loads.
- A much more improved model to the above is to use set non-blocking mode on all network sockets, and then use select (or poll) to determine which socket has received data.
- With this scheme, the kernel will inform the application as to whether or not a file descriptor has had an event, such as data arriving on a connected socket. This is referred to as "level triggering".
- A serious bottleneck using this method is that read or send will almost certainly block (even in non-blocking mode) if the particular page is not in core at the moment.
- As soon as the server requires disk I/O, its process blocks, all clients must wait, and all raw non-threaded performance goes to waste.
- The latest high-performance kernel-based improvement to the above methods is to use edge-triggered readiness notification. This is also referred to as "readiness change notification".
- An *edge-triggered* call will only return a file descriptor as being available for I/O after a change has happened on that file descriptor.
- The epoll system call in the current implementation of an edge-triggered call and it is integrated into the Linux 2.6 kernel and above.
- It is designed to replace the deprecated select (and also poll). Unlike these earlier system calls, which are O(n), epoll is an O(1) algorithm - this means that it scales well as the number of watched file descriptors increase.
- The select call uses a linear search through the list of watched file descriptors, which causes its O(n) behavior, whereas epoll uses callbacks in the kernel file structure.
- The edge-triggered nature of epoll means that the application issuing the call will receive "hints" when the kernel believes the file descriptor has become ready for I/O (in other words it is a certainty), as opposed to being told "I/O can be carried out on this file descriptor".

- In this way the kernel space does not need to keep track of the state of the file descriptor, although it might just push that problem into user space. However, user space programs can be more flexible (e.g. the readiness change notification can just be ignored).

- Note that the epoll event interface can be configured 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. Consider the following classic reader-write scenario:
    1. The file descriptor that represents the read side of a pipe (read_fd) is added inside the epoll device.
    2. The writer process writes 2KBytes of data on the write side of the pipe.
    3. A call to epoll_wait() is executed, which will return read_fd as ready file descriptor.
    4. The reader process reads 1KByte of data from read_fd.
    5. A call to epoll_wait() is executed.

- If the read_fd file descriptor has been added to the epoll interface using the EPOLLET flag, the call to epoll_wait() in step 5 will most likely block since there is available data 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 event notification only when events happen on the monitored file.
- Thus, 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 read_fd 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() done in step 5 might block 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 following steps illustrate the suggested way to use epoll as an Edge Triggered (EPOLLET) interface:
    - Use non-blocking file descriptors
    - Only issue a wait for an event after the read(2) or write(2) calls return EAGAIN

- On the contrary, when used as a Level Triggered interface, epoll is by all means a faster poll(), 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 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().
- When the EPOLLONESHOT flag is specified, it is caller responsibility to rearm the file descriptor using epoll_ctl() with EPOLL_CTL_MOD.

Summary of epoll system calls

- Create a specific file descriptor for all subsequent epoll calls:

```
epfd = epoll_create(EPOLL_QUEUE_LEN);
```

- Where `EPOLL_QUEUE_LEN` is the maximum number of connection descriptors that are expected to be managed at one time.
- The return value is a file descriptor that will be used in epoll calls later. This descriptor can be closed with `close()` when you do not longer need it.
- Add the descriptors to epoll:

```
static struct epoll_event ev;

int client_sock;

...

ev.events = EPOLLIN | EPOLLPRI | EPOLLERR | EPOLLHUP;

ev.data.fd = client_sock;

int res = epoll_ctl(epfd, EPOLL_CTL_ADD, client_sock, &ev);
```

- Where `ev` is epoll event configuration structure, `EPOLL_CTL_ADD` - predefined command constant to add sockets to epoll.
- Detailed description of `epoll_ctl` flags can be found in `epoll_ctl()` man page. When `client_sock` descriptor will be closed, it will be automatically deleted from epoll descriptor.

- After all the descriptors have been added to epoll, the server process can block and wait for event notification on socket descriptors being monitored by epoll:

```
while (TRUE)

{

      // wait for something to do...

      int nfds = epoll_wait(epfd, events,
      AX_EPOLL_EVENTS_PER_RUN,RUN_TIMEOUT);

      if (nfds < 0)

            perror("Error in epoll_wait!");


      // for each ready socket

      for(int i = 0; i < nfds; i++)

      {

            int fd = events[i].data.fd;

            handle_io_on_socket(fd);

      }

}
```

- The following describes a typical architecture of the networking portion of the application. This architecture allow almost unlimited scalability of your application on single and multi-processor systems:
    - **Listener** - thread that performs `bind()` and `listen()` calls and waits for incoming conncetions. Then new connection arrives, this thread can do `accept()` on listening socket an send accepted connection socket to one of the **I/O-workers**.
    - **I/O-Worker(s)** - one or more threads to receive connections from **listener** and to add them to epoll. Main loop of the generic **I/O-worker** looks like last step of epoll using pattern described above.
    - **Data Processing Worker(s)** - one or more threads to receive data from and send data to **I/O-workers** and to perform data processing.

- The `epoll()` API is very simple (albeit poorly documented) but provides a linear scalability that will allow high-performance servers to manage large amounts of concurrent connections.
- The two examples provided summarize the use of epoll calls using a typical echo client-server model.

Useful Resources

- epoll Scalability Web Page at Sourceforge.
- The C10K problem
- libevent: high-level event-handling library that can be used on top of epoll. This page contains some information about performance tests of epoll.