12. The Event Loop and Timers

12.1 Timeouts and Timers

There is one major thing missing in our server: timeouts. Every networked application needs to handle timeouts since the other side of the network can just disappear. Not only do ongoing IO operations like read/write need timeouts, but it is also a good idea to kick out idle TCP connections. To implement timeouts, the event loop must be modified since the `poll` is the only thing that is blocking.

Looking at our existing event loop code:

```
int rv = poll(poll_args.data(), (nfds_t)poll_args.size(), 1000);
```

The `poll` syscall takes a timeout argument, which imposes an upper bound of time spent on the `poll` syscall. The timeout value is currently an arbitrary value of 1000 ms. If we set the timeout value according to the timer, `poll` should wake up at the time it expires, or before that; then we have a chance to fire the timer in due time.

The problem is that we might have more than one timer, the timeout value of `poll` should be the timeout value of the nearest timer. Some data structure is needed for finding the nearest timer. The heap data structure is a popular choice for finding the min/max value and is often used for such purpose. Also, any data structure for sorting can be used. For example, we can use the AVL tree to order timers and possibly augment the tree to keep track of the minimum value.

Let's start by adding timers to kick out idle TCP connections. For each connection there is a timer, set to a fixed timeout into the future, every time there are IO activities on the connection, the timer is renewed to a fixed timeout.

Notice that when we renew a timer, it becomes the most distant one; therefore, we can exploit this fact to simplify the data structure; a simple linked list is sufficient to keep the order of timers: the new or updated timer simply goes to the end of the list, and the list maintains sorted order. Also, operations on linked lists are `O(1)`, which is better than sorting data structures.

12.2 The Linked List

Defining the linked list is a trivial task:

```
struct DList {
    DList *prev = NULL;
    DList *next = NULL;
};

inline void dlist_init(DList *node) {
    node->prev = node->next = node;
}

inline bool dlist_empty(DList *node) {
    return node->next == node;
}

inline void dlist_detach(DList *node) {
    DList *prev = node->prev;
    DList *next = node->next;
    prev->next = next;
    next->prev = prev;
}

inline void dlist_insert_before(DList *target, DList *rookie) {
    DList *prev = target->prev;
    prev->next = rookie;
    rookie->prev = prev;
    rookie->next = target;
    target->prev = rookie;
}
```

12.3 Event Loop Overview

The next step is adding the list to the server and the connection struct.

```
// global variables
static struct {
    HMap db;
    // a map of all client connections, keyed by fd
    std::vector<Conn *> fd2conn;
    // timers for idle connections
    DList idle_list;
} g_data;


struct Conn {
    // code omitted...
    uint64_t idle_start = 0;
    // timer
    DList idle_list;
};
```

An overview of the modified event loop:

```
int main() {
    // some initializations
    dlist_init(&g_data.idle_list);

    int fd = socket(AF_INET, SOCK_STREAM, 0);
    // bind, listen & other miscs
    // code omitted...

    // the event loop
    std::vector<struct pollfd> poll_args;
    while (true) {
        // prepare the arguments of the poll()
        // code omitted...

        // poll for active fds
        int timeout_ms = (int)next_timer_ms();
        int rv = poll(poll_args.data(), (nfds_t)poll_args.size(),
        timeout_ms);
        if (rv < 0) {
            die("poll");
        }

        // process active connections
```

```
        for (size_t i = 1; i < poll_args.size(); ++i) {
            if (poll_args[i].revents) {
                Conn *conn = g_data.fd2conn[poll_args[i].fd];
                connection_io(conn);
                if (conn->state == STATE_END) {
                    // client closed normally, or something bad happened.
                    // destroy this connection
                    conn_done(conn);
                }
            }
        }

        // handle timers
        process_timers();

        // try to accept a new connection if the listening fd is active
        if (poll_args[0].revents) {
            (void)accept_new_conn(fd);
        }
    }

    return 0;
}
```

A couple of things were modified:

1. The timeout argument of `poll` is calculated by the `next_timer_ms` function.
2. The code for destroying a connection was moved to the `conn_done` function.
3. Added the `process_timers` function for firing timers.
4. Timers are updated in `connection_io` and initialized in `accept_new_conn`.

12.4 Sorting with a Linked List

12.4.1 Find the Nearest Timer

The `next_timer_ms` function takes the first (nearest) timer from the list and uses it the calculate the timeout value of `poll`.

```
const uint64_t k_idle_timeout_ms = 5 * 1000;

static uint32_t next_timer_ms() {
```

```
    if (dlist_empty(&g_data.idle_list)) {
        return 10000;   // no timer, the value doesn't matter
    }

    uint64_t now_us = get_monotonic_usec();
    Conn *next = container_of(g_data.idle_list.next, Conn, idle_list);
    uint64_t next_us = next->idle_start + k_idle_timeout_ms * 1000;
    if (next_us <= now_us) {
        // missed?
        return 0;
    }

    return (uint32_t)((next_us - now_us) / 1000);
}
```

`get_monotonic_usec` is the function for getting the time. Note that the timestamp must be monotonic. Timestamp jumping backward can cause all sorts of troubles in computer systems.

```
static uint64_t get_monotonic_usec() {
    timespec tv = {0, 0};
    clock_gettime(CLOCK_MONOTONIC, &tv);
    return uint64_t(tv.tv_sec) * 1000000 + tv.tv_nsec / 1000;
}
```

12.4.2 Fire Timers

At each iteration of the event loop, the list is checked in order to fire timers in due time.

```
static void process_timers() {
    uint64_t now_us = get_monotonic_usec();
    while (!dlist_empty(&g_data.idle_list)) {
        Conn *next = container_of(g_data.idle_list.next, Conn, idle_list);
        uint64_t next_us = next->idle_start + k_idle_timeout_ms * 1000;
        if (next_us >= now_us + 1000) {
            // not ready, the extra 1000us is for the ms resolution of
poll()
            break;
        }

        printf("removing idle connection: %d\n", next->fd);
```

```
            conn_done(next);
        }
    }
```

12.4.3 Maintain Timers

Timers are updated in the `connection_io` function:

```
static void connection_io(Conn *conn) {
    // waked up by poll, update the idle timer
    // by moving conn to the end of the list.
    conn->idle_start = get_monotonic_usec();
    dlist_detach(&conn->idle_list);
    dlist_insert_before(&g_data.idle_list, &conn->idle_list);

    // do the work
    if (conn->state == STATE_REQ) {
        state_req(conn);
    } else if (conn->state == STATE_RES) {
        state_res(conn);
    } else {
        assert(0);  // not expected
    }
}
```

Timers are initialized in the `accept_new_conn` function:

```
static int32_t accept_new_conn(int fd) {
    // code omitted...

    // creating the struct Conn
    struct Conn *conn = (struct Conn *)malloc(sizeof(struct Conn));
    if (!conn) {
        close(connfd);
        return -1;
    }
    conn->fd = connfd;
    conn->state = STATE_REQ;
    conn->rbuf_size = 0;
    conn->wbuf_size = 0;
```

```
    conn->wbuf_sent = 0;
    conn->idle_start = get_monotonic_usec();
    dlist_insert_before(&g_data.idle_list, &conn->idle_list);
    conn_put(g_data.fd2conn, conn);
    return 0;
}
```

Don't forget to remove the connection from the list when done:

```
static void conn_done(Conn *conn) {
    g_data.fd2conn[conn->fd] = NULL;
    (void)close(conn->fd);
    dlist_detach(&conn->idle_list);
    free(conn);
}
```

12.5 Testing

We can test the idle timeouts using the `nc` or `socat` command:

```
$ ./server
removing idle connection: 4
```

```
$ socat tcp:127.0.0.1:1234 -
```

The server should close the connection by 5s.

Exercises:

1. Add timeouts to IO operations (read & write).
2. Try to implement more generic timers using sorting data structures.

Source code:

> - 12_server.cpp
> - avl.cpp
> - avl.h
> - common.h
> - hashtable.cpp

- hashtable.h
- list.h
- zset.cpp
- zset.h