

1 Asynchronous I/O

1.1 Asynchronous I/O

Asynchronous I/O is a major part of the difficulty in creating a server implementation in C, particularly when using threads is not a viable option. We don't want the server to be stuck while we wait for it to perform some I/O operation like reading or writing to an existing connection when the server has other tasks it could be doing, like establishing new client connections or parsing an already-received message. In C, one of the primary ways of performing asynchronous I/O is using the `epoll` API. Using this API, we are able to keep a list of file descriptors we want the current process to monitor, as well as a list of file descriptors that are ready for I/O. However, this process is extraordinarily tedious, requiring many expensive system calls to set up and maintain the `epoll` instance. When compared with other, more modern languages, the `epoll` API is both more verbose and more difficult to use. Consider a more modern language like Go or Javascript. Both of these languages have their own facilities for asynchronous operation. In Go, we use several constructs, including the `go` (to start *goroutines*, which are run by the runtime engine either on the current thread or automatically on another thread) and `select` (to wait for any operations from some set of operations to complete) keywords, to implement various aspects of I/O. In Javascript, we utilize both the `async` and `await` keywords, as well as the idea of *promises* in order to achieve some measure of asynchronous operation. The goal of our extension is to include these same facilities in ABLEC, allowing for a programmer to more easily write and understand the code performing asynchronous I/O operations.

1.2 Asynchronous I/O Concepts

In this section, we discuss the mechanics of the Asynchronous I/O extension, including the specifics of the translation from ABLEC code to plain C code.

This extension is less focused on redesigning the fundamentals of asynchronous I/O and

is more focused on providing syntax that is both easy to understand and easy to write for a programmer utilizing the extension. To this end, we retain some of the same principles utilized by the `epoll` API, but introduce similar syntax to modern languages with the `spawn` and `await` keywords.

The `spawn` keyword has similar syntax to the `spawn` keyword in Cilk [2], the `spawn()` method in Ruby, or the `spawn()` method in Rust. All of these keywords have a similar idea that underlies them. Each of these languages uses `spawn` to indicate starting a new process, thread or function call. The syntax for all three is also similar in concept, as well. Each has the form `spawn <foo>`, where what is in `foo` is either a function or a closure (in Rust). This starts program execution on whatever task is passed through `foo`.

The `await` keyword is at least partially inspired by the syntax of Javascript. After we spawn several tasks using `spawn`, we can then specify that we would like to wait for some of those tasks using `await`. The syntax is similar to Javascript: we simply write `await <foo>`, where `foo`, after the `await` call, is some timevalue object indicating how long to wait. In other words, `await` simply waits until at least one of the tasks we have spawned finishes, then returns, the same as a call to `io_dispatch` would.

1.3 Asynchronous I/O Extension

Here we consider what changes are actually made by our extension. Consider the code snippets from the `io.c` file, shown below in Figure 5 and the code from the extension in Figure 6. The major changes between the figures are on lines 23, 45, and 50.

Note in the first example that we utilize the default `io_event_add()` and `io_dispatch()` functions. A particular issue that arises when dealing with these functions is the lack of high-level transparency in their calls. Perhaps `io_event_add()` is somewhat clear, but the `io_dispatch()` function is particularly opaque. We make a call to the function with some `timeval`, but we cannot determine the use of this function without substantial effort on the part of the programmer.

On the other hand, consider the implementation using the asynchronous extension. We are using the `spawn` keyword to create new tasks, in this case, calling out to a `read_helper()` or `write_helper()` function. These functions are helper functions that simply perform a write or a read to some socket file descriptor, and we are passing them to `spawn` as the function to be run in a task. Note that because of these helper functions, we do not require any knowledge of constants like `IO_WANTREAD` or `IO_WANTWRITE`, instead simply requiring the programmer to pass in whatever function the programmer desires to run asynchronously.

This extension is also a much more flexible implementation than what is available with the `epoll` API. As an example, consider if a programmer wanted to modify the server to do something other than just read or write data using the `epoll` API. In the old C code, this would require defining a set of new constants and substantial code modification. In our new extension, all the programmer must be concerned with is writing whatever function they desire to be executed asynchronously. Then, utilize `spawn` with that function to register that function for asynchronous execution. There is no need to define any new constants or for the end programmer to do any substantial coding (outside of the function they desire to run asynchronously, which would have been written anyways without the extension).

```

1 GLOBAL void Conn_Handler(void) {
2     int i;
3     size_t wdatalen;
4     struct timeval tv;
5     time_t t;
6     bool command_available;
7     while (!NGIRCD_SignalQuit && !NGIRCD_SignalRestart) {
8         t = time(NULL);
9         command_available = false;
10        //Utility checks omitted for brevity
11        for (i = 0; i < Pool_Size; i++) { // Look for non-empty read buffers
12            if ((My_Connections[i].sock > NONE)
13                && (array_bytes(&My_Connections[i].rbuf) > 0)) {
14                Handle_Buffer(i); // handle the received data
15            }
16        }
17        for (i = 0; i < Pool_Size; i++) { // Look for non-empty write buffers
18            if (My_Connections[i].sock <= NONE)
19                continue;
20            wdatalen = array_bytes(&My_Connections[i].wbuf);
21            if (wdatalen > 0) {
22                //SSL Code omitted for brevity
23                io_event_add(My_Connections[i].sock, IO_WANTWRITE);
24            }
25        }
26        for (i = 0; i < Pool_Size; i++) { //Check sockets for readability
27            if (My_Connections[i].sock <= NONE)
28                continue;
29            //SSL code omitted for brevity
30            if (Proc_InProgress(&My_Connections[i].proc_stat)) {
31                io_event_del(My_Connections[i].sock, IO_WANTREAD); //Wait on subprocesses
32                continue;
33            }
34            if (Conn_OPTION_ISSET(&My_Connections[i], CONN_ISCONNECTING))
35                continue; //Wait for connect() to complete
36            if (My_Connections[i].delaytime > t) { //penalty set, ignore socket
37                io_event_del(My_Connections[i].sock, IO_WANTREAD);
38                continue;
39            }
40            if (array_bytes(&My_Connections[i].rbuf) >= COMMANDLEN) {
41                io_event_del(My_Connections[i].sock, IO_WANTREAD);
42                command_available = true;
43                continue;
44            }
45            io_event_add(My_Connections[i].sock, IO_WANTREAD);
46        }
47        tv.tv_usec = 0;
48        tv.tv_sec = command_available ? 0 : 1;
49        i = io_dispatch(&tv);
50        if (i == -1 && errno != EINTR) { exit(1); } //fatal errors
51        if (Conf_IdleTimeout > 0 && NumConnectionsAccepted > 0
52            && idle_t > 0 && time(NULL) - idle_t >= Conf_IdleTimeout) {
53            NGIRCD_SignalQuit = true;
54        }
55    }
56    //Server shutdown messages omitted
57 } /* Conn_Handler */

```

Figure 1: A file utilizing the asynchronous I/O Interface before implementing the extension

```

1 GLOBAL void Conn_Handler(void) {
2     int i;
3     size_t wdatalen;
4     struct timeval tv;
5     time_t t;
6     bool command_available;
7     while (!NGIRCD_SignalQuit && !NGIRCD_SignalRestart) {
8         t = time(NULL);
9         command_available = false;
10        //Utility checks omitted for brevity
11        for (i = 0; i < Pool_Size; i++) { // Look for non-empty read buffers
12            if ((My_Connections[i].sock > NONE)
13                && (array_bytes(&My_Connections[i].rbuf) > 0)) {
14                Handle_Buffer(i); // handle the received data
15            }
16        }
17        for (i = 0; i < Pool_Size; i++) { // Look for non-empty write buffers
18            if (My_Connections[i].sock <= NONE)
19                continue;
20            wdatalen = array_bytes(&My_Connections[i].wbuf);
21            if (wdatalen > 0) {
22                //SSL Code omitted for brevity
23                /* CHANGE */
24                spawn write_helper(My_Connections[i].sock)
25            }
26        }
27        for (i = 0; i < Pool_Size; i++) { //Check sockets for readability
28            if (My_Connections[i].sock <= NONE)
29                continue;
30            //SSL Code omitted for brevity
31            if (Proc_InProgress(&My_Connections[i].proc_stat)) {
32                io_event_del(My_Connections[i].sock, IO_WANTREAD); //Wait on subprocesses
33                continue;
34            }
35            if (Conn_OPTION_ISSET(&My_Connections[i], CONN_ISCONNECTING))
36                continue; //Wait for connect() to complete
37            if (My_Connections[i].delaytime > t) { //penalty set, ignore socket
38                io_event_del(My_Connections[i].sock, IO_WANTREAD);
39                continue;
40            }
41            if (array_bytes(&My_Connections[i].rbuf) >= COMMANDLEN) {
42                io_event_del(My_Connections[i].sock, IO_WANTREAD);
43                command_available = true;
44                continue;
45            }
46            /* CHANGE */
47            spawn read_helper(My_Connections[i].sock);
48        }
49        tv.tv_usec = 0;
50        tv.tv_sec = command_available ? 0 : 1;
51        /*CHANGE*/
52        i = await tv; //Waits for some specified timevalue
53        if (i == -1 && errno != EINTR) { exit(1); } //fatal errors
54        if (Conf_IdleTimeout > 0 && NumConnectionsAccepted > 0
55            && idle_t > 0 && time(NULL) - idle_t >= Conf_IdleTimeout) {
56            NGIRCD_SignalQuit = true;
57        }
58    }
59    //Server shutdown messages omitted
60 } /* Conn_Handler */

```

Figure 2: A file utilizing the asynchronous I/O interface after implementing the extension