Design Document: Multithreaded Server

1. Goals

The goal of this program is to modify the http server to handle multithreading and logging using pthreads. Argument flags are handled by getopt().

2. Design

The design is separated into parts. The program first initialize the server using arguments. A “dispatch” thread listens to the connection. When a connection is made, a “worker” thread will handle the connection while dispatch goes back to listening.

2.1 Handling Arguments

In order to enable the use of flags, getopt() is used to parse the argument array. Arguments handling is shown in Algorithm 1. After checking for flags, program checks that at least the address is given as parameter. Then the address and port string are passed to socket_setup().

```
procedure argument handling
        string log_name
        unsigned opt
        unsigned arg_count = argc
        unsigned lflag = 0
        unsigned thread_count = 0
        while opt = getopt(argc, argv, “N:l”) != -1 then
                switch(opt)
                        case ‘N’
                                --arg_count
                                if optarg then
                                        thread_count = optarg
                                        --arg_count
                                break;
                        case ‘l’
                                --arg_count
                                ++lflag;
                                if optarg then
                                        log_name = optarg
                                        --arg_count
                                break;
                        default:
                                break;
        end
        if arg_count < 1 or thread_count < 1 then
                err(1)
        end
        socket = socket_setup(arg_count, argv[optind], argv[optind + 1])
```

Algorithm 1. Handling Arguments

2.2 Socket Setup

The optional port number is port 80 by default. getaddrinfo() is used to get the information needed into struct addrs. Members of addrs is then passed to socket and bind. Each networking function has their own error handling if statement. Address and port are put in struct sockaddr_in, the struct instance is passed to bind(). Then listen() waits for a connection from a client. At the end the socket is returned so main() could start the dispatch loop.

```
procedure socket_setup (arg_count, address, port)
        struct addrinfo *addrs, hints = {};
        hints.ai_family = AF_INET;
        hints.ai_socktype = SOCK_STREAM;
        if arg_count == 3 then
                getaddrinfo(address , port , &hints, &addrs);
        else
                getaddrinfo(address , "80", &hints, &addrs);
        end
        s_fd = socket(AF_INET, SOCK_STREAM, 0);
        setsockopt(main_socket)
        bind(main_socket)
        listen(main_socket)
        return main_socket
```

Algorithm 2. Socket Setup

2.3 Dispatcher and worker

Mutex and conditional variables are used to synchronize the dispatch and worker threads. -1 in buffer indicates buffer is not being used.

```
threads thread[N]
struct parameters
        unsigned id
        unsigned log
conditional variable empty
conditional variable full
mutex mutex
unsigned active_threads = 0
unsigned waiting_threads = 0
signed buffer[maxbuffer] = {-1}
unsigned state[N]
```

Algorithm 3. shared resources

Dispatch creates child threads and a while loop that handle incoming connections using up to N threads. Dispatch sleeps if all threads are currently being used. If one or more threads are waiting, dispatch will loop through 1 to N until a waiting thread is found. Then using mutex and conditional variable, dispatch signals the worker thread and set its state to working.

```
procedure dispatch
        thread thread_pool = malloc()
        for n = 1 to thread_count
                struct parameters p
                p.id = n
                p.log = lflag;
                pthread_create(thread_pool, worker, p)
        end
        unsigned i
        while(1)
                acc_soc = accept()
                mutex.lock()
                if active == N then
                        empty.wait()
                end
                for i = 0, i < N, ++i
                        if thread_state[i] == waiting then
                                break
                        end
                end
                buffer[i] = acc_soc
                ++active_threads
                thread_state[i] = working
                for i = 1 to waiting_threads
                        full.signal(i)
```

```
        mutex.unlock()
    end
```

Algorithm 4. dispatcher

The worker thread waits on the conditional variable. Once received, the thread stores socket_fd so mutex can be released. Then handle_socket is called to handle the connection. The input id tells the worker thread which element of the buffer array to use.

```
procedure worker ( id )
    uint8_t soc
    while(1) {
        mutex.lock()
        while state[i] == waiting
            ++waiting_threads
            full.wait()
            --waiting_threads
        soc = buffer[i]
        handle_socket( soc )
        buffer[i] = -1
        state[i] = state[i] -1
        --active_thread
        empty.signal()
        mutex.unlock()
    end
```

Algorithm 5. worker

2.4 handle_client()

Inside the while loop with accept, handle_client reads the message are identify the request and filename. A response is made using concat(). sscanf() detects the request in buffer.

The first line of the header is read to get the request. Using strstr(), the pointer to the beginning and end of line “content-length” is found. The size of the content is saved and converted with atoi().

If the request is PUT, a file is made using write() with the filesize of content-length and data from the received header. If the request is GET, read() tries to find the file with the same name. If the file exists, the content is copied into a buffer. strcat() concatenate the buffer into the response. Finally, the response is sent using send().

Finally, a while loop is used to read and send content requested if necessary. read() and send() would use the same buffer and size to be read. A counter decreases to keep track of the data remaining to be read and send. When complete, close() is used to close the file descriptor.

```
procedure handle_client(acc_soc)
        read( acc_soc, buffer, sizeof(buffer));
        sscanf(buffer, "%s %s", command, filename, size, data);
        read(acc_soc, buffer, sizeof(buffer));
        substring_start = strstr(buffer, "Content-Length: ");
        if substring_start != nullptr then
                substring_end = strstr(substring_start, "\r");
                sub_len = substring_end - substring_start - 16;
                strncpy(cont_len_substr, substring_start + 16, sub_len);
                size = atoi(cont_len_substr);
                if size > 0 then
                        read(soc_fd, (char *)payload, sizeof(payload));
                end
        end
           strcpy(header, “HTTP/1.1”);
           if filename = “/” or filename size not 27 then
                strcat(header, 403 forbidden\r\nContent-Length: 0\r\n”);
           else if strcmp(command, "PUT”) == 0 then
                if access(filename, W_OK) == 0 then
                        remove(filename)
                end
                fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR);
                if fd == ERR then
                        strcat(header, “400 bad request\r\n”);
                else
                        write(fd, data, sizeof(data))
                        strcat(header, “201 Created\r\n”);
                end
        else if strcmp(command, “GET”) == 0 then
```

```
        fd = open(filename, O_RDONLY);
        if (fd == -1) then
                strcat(header, “400 bad request\r\n”)
        else
                strcat(header, “200 ok\r\n”)
                fileSize = lseek(fd, 0, SEEK_END)
                lseek(fd, 0, 0);
                char fileData[fileSize]
                close(fd)
                sprintf(buffer, "Content-Length: %d\r\n%s\r\n", sizeof(data), data);
                strcat((char *)header, (char *)buffer);
        end
        strcat((char *)header, "500 Internal Server Error\r\n");
end
strcpy(response, (char*)header);
send(soc_fd, (char*)header, headersize, 0);
if (payloadSize > 0) then
        read(fd, payload, BUFMAX);
        send(soc_fd, payload, BUFMAX, 0)
        close(fd);
        payloadSize = payloadSize - BUFMAX;
end
```

Algorithm 6. handle_client()

2.5 Logging

Dispatch includes the logging flag and logging offset in the structure passed to the worker thread. Logging will occur inside handle_client() if the logging flag is true. A semaphore is used in handle_client() to reserve space in the log file.

```
procedure logging (file descriptor, operation, filename, length, offset)
        string buffer
        string output
        unsigned count, i , j, k
        format(buffer, "%s %s length %d", operation, filename, length)
        concat(output, buffer)
        j = 0
        for i = 0 ; i < length ; i += 20
                format(buffer, "%08d", count)
                count += 20
                concat(output, buffer)
                for k = 0, k< 20 or remaining characters, ++k
                        format (buffer, "%h ", character)
                        concat(output, buffer)
                        ++j
                end
                concat(output, newline)
        end
        concat(output "========\0")
        file_write_offset(fd, output, strlen(output), offset)
```

Algorithm 7. logging

```
semaphore sem
unsigned file_offset
procedure logging_synchronization
        unsigned my_offset
        sem.down
        my_offset = file_offset
        sem.up
        logging(my_offset)
```

Algorithm 8. synchronizing file offset