دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

عنوان:

پیاده سازی وب سرور چندریسه

شماره گروه: ۲۴

اعضای گروه

روزبه پیراعیادی
آرین احدی نیا
حمیدرضا دهباشی

نام درس

سیستم‌های عامل

نیم‌سال اول ۱۴۰۱-۱۴۰۲

نام استاد درس

حسین اسدی

۱ مقدمه

یکی از مسائل مهم در جهان امروز وب‌سایت‌های اینترنتی هستند که بخش جدایی‌ناپذیری از زندگی انسان امروز را تشکیل داده‌اند. این وب‌سایت‌ها گاها تا چندین میلیارد کاربر همزمان دارند. بنابرین لازم است تا به صورتی بهینه برای سرویس‌دهی هر چه بیشتر پیاده‌سازی شوند. یکی از تکنیک‌های مورد استفاده برای افزایش کارایی چنین سیستم‌هایی استفاده از هم‌روندی در برنامه است. برنامه می‌تواند از چند ریسمان به صورت همزمان برای مدیریت درخواست‌ها استفاده کند تا کاربران مختلف بدون مسدود کردن راه یک‌دیگر به پردازش دستورهای خود بپردازند.

برای این پروژه قصد داریم تا یک نسخه ساده‌سازی شده از HTTP را پیاده‌سازی کنیم. به این صورت که هر کاربر ابتدا یک اتصال با سرور بر روی یک پورت پیش‌فرض برقرار کند و پس از آن یک رشته برای کاربر ساخته شود که کارهای مربوط به آن درخواست کاربر در آن سرور انجام شود و در نهایت رشته در یک حوضچه برای انجام درخواست‌های آتی قرار بگیرد. در این پروژه کاربر سطح بالا قابلیت اضافه کردن handler خواهد داشت و هندلر مورد نظر برای پردازش درخواست کاربر به صورت بلندترین تطابق انتخاب خواهد شد.

۲ ساختار کلی

ساختار این وب سرور ۴ بخش کلی دارد:

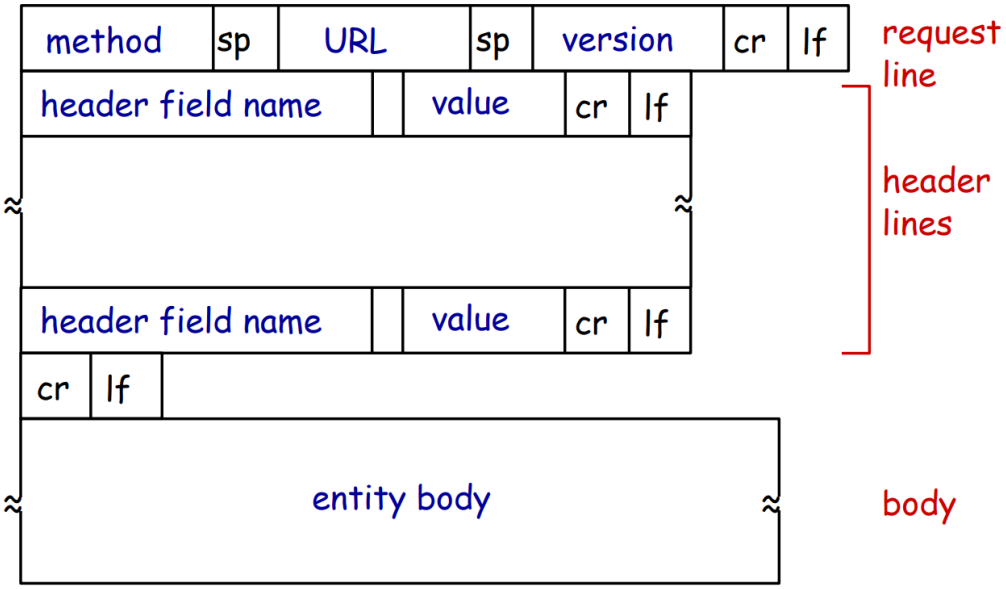
- دریافت و ارسال داده
- پردازش و parse کردن درخواست ها و پاسخ
- پردازش درخواست در سمت سرور و آماده سازی پاسخ
- پیاده سازی چندریسه ای و هندل کردن thread pool

در قسمت نخست با استفاده از سوکت و tcp ، بین کلاینت و سرور ارتباط برقرار میکنیم. توجه کنید که به دلیل اینکه از پروتکل HTTP استفاده میکنیم، همان browser درحقیقت نقش کلاینت را میتواند ایفا کند. توضیحات تکمیلی در ادامه آمده است.
در قسمت دوم، همانطور که اشاره شد باید درخواست ها را در قالب پروتکل HTTP دریافت و پردازش کنیم و پاسخ را نیز در همین قالب تحویل کلاینت دهیم. توضیحات تکمیلی در ادامه آمده است.
در قسمت سوم، به اینصورت است که کاربر میتواند با توجه به API مشخص شده، نوع درخواست خود را مشخص کند (در حال حاضر add, sub, mult, divide است) و سمت سرور نیز با استفاده از توابعی به نام Handler باید پاسخ متناظر با هر درخواست تولید شود.
در قسمت چهارم نیز با استفاده از یک thread pool و تعدادی worker و هماهنگی میان آن ها با استفاده از lock و semaphore ، توانستیم وب سرور را از تک ریسه به چندریسه تغییر دهیم.

۳ پروتکل HTTP

پروتکل HTTP یا Hypertext Transfer Protocol یک پروتکل لایه‌ی کاربرد است. امروزه این پروتکل بسیار فراگیر است و بخش بزرگی از اینترنت مبتنی بر همین پروتکل است. پروتکل http از پروتکل لایه‌ی انتقال cp استفاده می‌کند تا امکان انتقال امن اطلاعات را فراهم کند.

در ادامه نمایی کلی از یک درخواست http را مشاهده می‌کنید. ([۱])



خط اول که line request نامیده می‌شود، شاملmethod، URL و نسخه‌ی http است. بخش دوم مربوط ابه سرآیندهاست. این بخش به کمک دو کارکتر "\r" و "\n" از بخش قبل جدا می‌شود. در این بخش به صورت key و value مقادیر سرآیندها مشخص می‌شود.

بخش آخر نیز مربوط به بدنه‌ی پیام است. این بخش با دو کاراکتر پشت سر هم "\r" و "\n" از بخش قبل جدا می‌شود. دقت کنید که تمام این موارد به صورت یک رشته انتقال پیدا می‌کنند و صرفا برای نمایش بهتر به صورت خط به خط نشان داده شده است.

۴ پردازش درخواست‌های HTTP

- تابع parse request

```c
void parse_request(char * request, HttpRequest * http_request) {
    char * first_line_end = strstr(request, "\r\n");
    int first_line_length = first_line_end - request;
    char * first_line = malloc(first_line_length + 1);
    strncpy(first_line, request, first_line_length);
    first_line[first_line_length] = '\0';

    parse_first_line(first_line, first_line_length, http_request);
    free(first_line);

    char * header_end = strstr(first_line_end + 2, "\r\n\r\n");
    int header_length = header_end - first_line_end - 2;

    char * header = malloc(header_length + 1);
    strncpy(header, first_line_end + 2, header_length);
    header[header_length] = '\0';
    http_request -> headers = malloc(sizeof(Header));

    parse_header(header, header_length, http_request -> headers);
    free(header);

    int body_length = strlen(request) - (header_end - request) - 4;
    char * body = malloc(body_length + 1);
    strncpy(body, header_end + 4, body_length);
    body[body_length] = '\0';
    http_request -> body = body;
}
```

تابع parse request تابعی است که درخواست http را به صورت یک رشته گرفته و آن را به صورت یک struct از نوع HTTPRequest در می‌آورد. این تابع خروجی ندارد و نتیجه‌ی پردازش درخواست در همان struct ورودی گرفته شده ذخیره می‌شود.

این تابع ابتدا با پیدا کردن اولین "\r \n" خط اول را جدا می‌کند و برای پردازش آن از تابع parse first line استفاده می‌کند. سپس با پیدا کردن جایی که یک خط خالی گذاشته شده است یا به عبارتی از "\r \n \r \n" استفاده شده است، بخش مربوط به سرآیندها را از بخش بدنه جدا می‌کند. در ادامه برای پردازش سرآیندها از تابع parse header استفاده می‌کند.

- تابع parse first line

```
void parse_first_line(char * first_line, int first_line_length,
        HttpRequest * http_request) {
    char * method = strtok(first_line, " ");
    char * path = strtok(NULL, " ");
    char * version = strtok(NULL, " ");
    http_request -> method = malloc(strlen(method) + 1);
    strcpy(http_request -> method, method);
    http_request -> path = malloc(strlen(path) + 1);
    strcpy(http_request -> path, path);
    http_request -> version = malloc(strlen(version) + 1); !
    strcpy(http_request -> version, version);
    http_request -> query_parameters = malloc(sizeof(QueryParameters));
    http_request -> query_parameters -> num_params = 0;
    http_request -> path_without_query = parse_path(http_request ->
        query_parameters, path);
}
```

تابع parse first line این تابع method ، path و version را تشخیص می‌دهد. سپس باید در path به دنبال پارامترهایی که به عنوان ورودی قرار داده شده‌اند بگردیم که این کار توسط تابع parse path انجام می‌گیرد.

- تابع parse path

```
char * parse_path(QueryParameters * query_parameters, char * path) {
    char * query_start = strstr(path, "?");
    char * path_without_query;
    if (query_start != NULL) {
        int path_without_query_length = query_start - path;
        path_without_query = malloc(path_without_query_length + 1);
        strncpy(path_without_query, path, path_without_query_length);
        path_without_query[path_without_query_length] = '\0';

        char * query = query_start + 1;
        char * key = strtok(query, "=");
        char * value = strtok(NULL, "&");

        query_parameters -> keys = malloc(sizeof(char * ));
        query_parameters -> values = malloc(sizeof(char * ));

        query_parameters -> keys[0] =
               malloc(sizeof(char) * strlen(key) + 1);
        query_parameters -> values[0] =
               malloc(sizeof(char) * strlen(value) + 1);
        strcpy(query_parameters -> keys[0] , key);
        strcpy(query_parameters -> values[0] , value);

        query_parameters -> num_params = 1;
        while (value != NULL) {
            key = strtok(NULL, "=");
            value = strtok(NULL, "&");

            if (key != NULL && value != NULL) {
                query_parameters -> keys =
```

```
                realloc(query_parameters -> keys, sizeof(char * ) *
                    (query_parameters -> num_params + 1));
            query_parameters -> values =
                realloc(query_parameters -> values, sizeof(char * ) *
                    (query_parameters -> num_params + 1));

            int idx = query_parameters -> num_params;
            query_parameters -> keys[idx] =
                malloc(sizeof(char) * strlen(key) + 1);
            query_parameters -> values[idx] =
                malloc(sizeof(char) * strlen(value) + 1);
            strcpy(query_parameters -> keys[idx] , key);
            strcpy(query_parameters -> values[idx], value);

            query_parameters -> num_params++;
        }
    }
}
else{
    path_without_query = malloc(strlen(path) + 1);
    strcpy(path_without_query, path);
}

return path_without_query;
}
```

تابع parse path نیز بخش مربوط به query را از path جدا می‌کند. سپس بخش بدون query را بر می‌گرداند. بخش مربوط به query نیز به فرمت QueryParameters تبدیل می‌شود.

- تابع parse header

```
void parse_header(char * header, int header_length,
    Header * http_header) {
    char * key = strtok(header, ":");
    char * value = strtok(NULL, "\r");
    http_header -> keys = malloc(sizeof(char * ));
    http_header -> values = malloc(sizeof(char * ));

    http_header -> keys[0] = malloc(sizeof(char) * strlen(key) + 1);
    http_header -> values[0] = malloc(sizeof(char) * strlen(value) + 1);
    strcpy(http_header -> keys[0] , key);
    strcpy(http_header -> values[0] , value + 1);

    http_header -> num_headers = 1;
    while (value != NULL) {
        key = strtok(NULL, ":");
        value = strtok(NULL, "\r");
        if (key != NULL && value != NULL) {
            key = key + 1;
            value = value + 1;
            http_header -> keys = realloc(http_header -> keys,
                sizeof(char * ) * (http_header -> num_headers + 1));
            http_header -> values = realloc(http_header -> values,
                sizeof(char * ) * (http_header -> num_headers + 1));

            int idx = http_header -> num_headers;
            http_header -> keys[idx] =
                malloc(sizeof(char) * strlen(key) + 1);
            http_header -> values[idx] =
                malloc(sizeof(char) * strlen(value) + 1);
            strcpy(http_header -> keys[idx] , key);
```

```
            strcpy(http_header -> values[idx], value);

            http_header -> num_headers++;
        }
    }
}
```

تابع parse header سرآیند را به صورت یک رشته ورودی می‌گیرد و آن را به کمک ":" به key ها و value ها تقسیم می‌کند و به صورت یک struct از نوع Header ذخیره می‌کند.

- تابع response

```
HttpResponse * response(int status_code, char * status_message,
            char * body) {
    HttpResponse * http_response = malloc(sizeof(HttpResponse));
    http_response -> status_code = status_code;

    http_response -> status_message = malloc(10);
    strcpy(http_response -> status_message, status_message);
    http_response -> body = body;
    http_response -> headers = malloc(sizeof(Header));
    http_response -> headers -> num_headers = 0;
    return http_response;
}
```

این تابع یک پاسخ http تولید می‌کند.

- تابع add header

```
void add_header(HttpResponse * http_response, char * key, char * value){
    if (http_response -> headers -> num_headers == 0){
        http_response -> headers -> keys = malloc(sizeof(char * ));
        http_response -> headers -> values = malloc(sizeof(char * ));
    }
    else{
        http_response -> headers -> keys =
            realloc(http_response -> headers -> keys, sizeof(char * ) *
                (http_response -> headers -> num_headers + 1));
        http_response -> headers -> values =
            realloc(http_response -> headers -> values, sizeof(char * ) *
                (http_response -> headers -> num_headers + 1));
    }

    int idx = http_response -> headers -> num_headers;
    http_response -> headers -> keys[idx] =
        malloc(sizeof(char) * strlen(key) + 1);
    http_response -> headers -> values[idx] =
        malloc(sizeof(char) * strlen(value) + 1);
    strcpy(http_response -> headers -> keys[idx] , key);
    strcpy(http_response -> headers -> values[idx], value);

    http_response -> headers -> num_headers++;
}
```

از تابع add header برای اضافه کردن سرآیند به پاسخ http از این تابع استفاده می‌شود.

- تابع serialize header

```
char * serialize_response(HttpResponse * http_response) {
    char * status_line = malloc(100);
    sprintf(status_line, "HTTP/1.1 %d %s\r\n",
    http_response -> status_code, http_response -> status_message);
    char * headers = malloc(1000);
    strcpy(headers, "");
    char * header = malloc(100);
    for (int i = 0; i < http_response -> headers -> num_headers; i++) {
        sprintf(header, "%s: %s\r\n", http_response -> headers ->
        keys[i], http_response -> headers -> values[i]);
        strcat(headers, header);
    }
    char * body = malloc(1000);
    strcpy(body, "");
    if (http_response -> body != NULL) {
        sprintf(body, "\r\n%s", http_response -> body);
    }
    char * response = malloc(strlen(status_line) + strlen(headers) +
    strlen(body) + 1);
    strcpy(response, status_line);
    strcat(response, headers);
    strcat(response, body);
    free(status_line);
    free(header);
    free(headers);
    free(body);
    return response;
}
```

این تابع پاسخ http را به صورت رشته در می‌آورد تا بتواند برای کلاینت ارسال کند.

- تابع get param

```
char * get_param(HttpRequest * http_request, char * key) {
    for (int i = 0; i < http_request -> query_parameters
                -> num_params; i++) {
        if (strcmp(http_request -> query_parameters -> keys[i], key)
                    == 0) {
            return http_request -> query_parameters -> values[i];
        }
    }
    return NULL;
}
```

این تابع با گرفتن key و یک درخواست http ، value آن را پیدا می‌کند.

۵ حوضچه‌ی ریسمان

همان‌گونه که مستحضر هستید در برنامه‌های تحت وب تعداد کاربران بسیار زیاد است و گاهی به میلیاردها نفر می‌رسند. این باعث می‌شود تا ما نیاز به مقیاس کردن برنامه در ابعاد بالا داشته باشیم. از دو راهکار مقیاس عمودی و افقی که می‌توان برای مقیاس کردن برنامه پیاده کرد، راهکار مقیاس کردن افقی هزینه کمتری دارد. از آنجایی که پروتکل HTTP بدون حالت یا به عبارتی Stateless است، این مقیاس‌پذیری در سطح پروتکل چالش جدی ندارد. برای مقیاس کردن می‌توانیم چند Instance همزمان از برنامه داشته باشیم اما داشتن یک Instance که چند ریسمان را اجرا می‌کند، هم نیاز به منابع کمتری دارد هم مدیریت آنها ساده‌تر خواهد بود و همه برنامه‌ها به یک درگاه[۱] مسیریابی خواهند شد.

برای جلوگیری از ایجاد سربار ایجاد رشته به ازای هر درخواست،می‌توانیم از حوضچه ریسمان استفاده کنیم. به این صورت که صفی از درخواست‌ها داشته باشیم و چند ریسمان هر یک از این صف‌ها درخواست برداشته و به آن پاسخ دهند. البته توجه کنید که پیاده‌سازی حوضچه ریسمان و صف مستقل از بحث وب‌سرور می‌تواند صورت بگیرد. در ادامه این بخش به بررسی پیاده‌سازی صف و حوضچه ریسمان می‌پردازیم.

ابتدا نیاز داریم که یک Struct داشته باشیم که در آن Task تعریف شود. هر تسک یک تابع است که در زمان مقتضی باید با پارامترهای مربوطه فراخوانی شود. این Task توسط یک رشته کارگر در اولین فرصت انجام خواهد شد.

```
typedef struct {
    void * ( * worker)(void * );
    void * args;
}
Task;
```

حال نیاز داریم تا صفی داشته باشیم که درخواست‌ها را در خود نگه دارد. برای این منظور آرایه‌ای از اشاره‌گرها به Task تعریف شده است. یک Lock جهت اینکه دستکاری مشترک روی صف نداشته باشیم تعریف شده است و یک Semaphore جهت اینکه تعداد عناصر در صف را نگه دارد و اجازه ندهد قبل از وارد شدن تسک به صف، تسکی از آن خارج شود تعریف شده است. به عبارت دیگر درخواست خروج از صف تا زمانی که صف خالی باشد Block خواهد شد.

---

[۱] Port

```
typedef struct {
    Task ** tasks;
    int size;
    int start;
    int end;
    pthread_mutex_t lock;
    sem_t semaphore;
}
TaskQueue;
```

حال ما یک ساختار برای خود حوضچه ریسمان داریم که لیستی از رشته‌ها به همراه State آنها را نگه داری می‌کند.

```
typedef struct {
    TaskQueue * taskQueue;
    int numThreads;
    int open;
    pthread_t * workerThreads;
}
ThreadPool;
```

در تابع زیر صف را ایجاد می‌کنیم. دقت کنید که مقدار اولیه سمافور از آنجایی که صف خالی است برابر صفر است.

```
void createTaskQueue(TaskQueue * taskQueue, int size) {
    taskQueue -> tasks = (Task ** ) malloc(sizeof(Task * ) * size);
    for (size_t i = 0; i < size; i++) taskQueue -> tasks[i] = NULL;
    taskQueue -> start = 0;
    taskQueue -> end = 0;
    taskQueue -> size = size;
    sem_init( & taskQueue -> semaphore, 1, 0);
    pthread_mutex_init( & taskQueue -> lock, NULL);
}
```

تابع زیر وظیفه اضافه کردن Task به صف را دارد. این تابع ابتدای امر قفل را دریافت می‌کند. اگر صف پر باشد درخواست Drop می‌شود. در صورتی که با موفقیت وظیفه به صف اضافه شود، سمافور یک واحد افزایش پیدا می‌کند تا بتوان از صف عنصری را خارج گرد. دقت کنید که همین ترتیب که برای اضافه کردن سمافور و آزاد کردن قفل در اینجا استفاده کرده‌ایم، در تابع dequeue نیز پیاده‌سازی شده است. دقت کنید که این صف به صورت Circular است و در صورتی که پر شود، دوباره از سر آن شروع به Insert می‌کنیم.

```
int enqueueTaskQueue(TaskQueue * taskQueue, Task * task) {
    pthread_mutex_lock( & taskQueue -> lock);
    if (taskQueue -> start == (taskQueue -> end + 1) % taskQueue -> size) {
        pthread_mutex_unlock( & taskQueue -> lock);
        return 0;
    }
    taskQueue -> tasks[taskQueue -> end] = task;
    taskQueue -> end = (taskQueue -> end + 1) % taskQueue -> size;
    sem_post( & taskQueue -> semaphore);
    pthread_mutex_unlock( & taskQueue -> lock);
    return 1;
}
```

در تابع dequeue به محض اینکه سمافور آزاد شود، منتظر دریافت قفل اختصاصی می‌شویم و پس از دریافت قفل، عنصر مورد نظر را از صف خارج می‌کنیم. در نهایت نیز قفل را آزاد می‌کنیم. دقت بفرمایید که این پیاده‌سازی نیز متناسب با Circular بودن صف است.

```
Task * dequeueTaskQueue(TaskQueue * taskQueue) {
    sem_wait( & taskQueue -> semaphore);
    pthread_mutex_lock( & taskQueue -> lock);
    Task * task = taskQueue -> tasks[taskQueue -> start];
    taskQueue -> tasks[taskQueue -> start] = NULL;
    taskQueue -> start = (taskQueue -> start + 1) % taskQueue -> size;
    pthread_mutex_unlock( & taskQueue -> lock);
    return task;
}
```

برای ایجاد حوضچه ریسمان، آرایه‌ای از ریسمان‌ها ایجاد می‌کنیم که هر یک تابع worker را اجرا می‌کنند. ورودی این تابع خود Thread Pool است تا همه به مقدار متغیر Open دسترسی داشته باشند.

```
void createThreadPool(ThreadPool * threadPool, TaskQueue * taskQueue,
                      int( * priority)(void * ), int numThreads,
                      int queueSize) {
    threadPool -> taskQueue = taskQueue;
    threadPool -> numThreads = numThreads;
    threadPool -> open = 1;
    threadPool -> workerThreads = (pthread_t * ) malloc(
        sizeof(pthread_t) * numThreads
    );
    for (size_t i = 0; i < numThreads; i++)
        pthread_create( & threadPool -> workerThreads[i], NULL,
        worker, threadPool);
}
```

در نهایت تابع worker تا زمانی که حوضچه ریسمان باز باشد، درخواست‌ها را از صف بر می‌دارد و آنها را اجرا می‌کند.

```
void * worker(void * args) {
    ThreadPool * threadPool = (ThreadPool * ) args;
    while (threadPool -> open) {
        Task * task = dequeueTaskQueue(threadPool -> taskQueue);
        task -> worker(task -> args);
    }
    return NULL;
}
```

۶ وب‌سرور

برنامه‌نویس سطح بالا وقتی میخواهد یک وب‌سرور پیاده‌سازی کند، نباید با مفاهیم سطح پایین مانند Socket، مسیریابی و مواردی از این دست مواجه شود. برای اینکه برنامه‌نویس با این مسایل مواجه نباشد، ما چارچوبی برای پیاده‌سازی وب‌سرور ارائه می‌دهیم تا برنامه نویس تنها بر موارد سطح بالا و منطق برنامه کاربردی خود متمرکز باشد. در این پیاده‌سازی برنامه‌نویس هیچ درگیری با مسیریابی و سوکت ندارد و مستقیما می‌تواند کار خود را انجام دهد. در ادامه این بخش به توضیح پیاده‌سازی انجام شده می‌پردازیم.

ابتدا یک ساختار برای Handler پیاده‌سازی می‌کنیم. هر درخواست Http توسط یک Handler پردازش می‌شود. هر هندلر در یک مسیر است. هر درخواست با هندلری پردازش می‌شود که بلندترین تطابق را داشته باشد. همچنین هر هندلر یک تابع دارد که کار پردازش درخواست را عهده‌دار است. ورودی این تابع درخواست HTTP و خروجی آن پاسخ HTTP است.

```
typedef struct {
    char * path;
    HttpResponse * ( * handler_function)(HttpRequest * );
}
Handler;
```

وب سرور تعدادی هندلر دارد که درخواست ورودی را با آنها تطبیق می‌دهد. همچنین وب‌سرور یک حوضچه ریسمان دارد که وظیفه پردازش هر درخواست توسط یکی از آنها انجا می‌شود.

```
typedef struct {
    int port;
    ThreadPool * threadPool;
    Handler * handlers;
    int num_handlers;
    int open;
}
WebServer;
```

هر وب‌سرور زمانی که ساخته می‌شود باید یک حوضچه ریسمان و یک صف برای کارها بسازد و کارها را در آن قرار دهد.

```
void create_web_server(WebServer * webServer, int port, int num_threads,
                  int queue_size, int max_num_handlers)
{
    webServer -> port = port;
    TaskQueue * taskQueue = (TaskQueue * ) malloc(sizeof(TaskQueue));
    createTaskQueue(taskQueue, queue_size);
    webServer -> threadPool = (ThreadPool * ) malloc(sizeof(ThreadPool));
    createThreadPool(webServer -> threadPool, taskQueue, NULL,
                  num_threads, queue_size);
    webServer -> handlers = (Handler * ) malloc(
        sizeof(Handler) * max_num_handlers
    );
    webServer -> num_handlers = 0;
    webServer -> open = 1;
}
```

برنامه‌نویس پس از ساختن وب‌سرور می‌تواند به آن Handler اضافه کند تا درخوات وقتی وارد می‌شود توسط هندلری که بلندترین تطابق را دارد اجرا شود.

```
void add_new_handler(WebServer * webServer, char * path,
                  HttpResponse * ( * handler_function)(HttpRequest * ))
{
    Handler * handler = & webServer -> handlers[webServer -> num_handlers++];
    handler -> path = path;
    handler -> handler_function = handler_function;
}
```

پس از ایجاد وب‌سرور می‌توانیم سوکت TCP را باز کنیم که میزبان درخواست ها باشد. این کار را در تابع start_web_server انجام می‌دهیم. خروجی این تابع از ساختار Connection-Descriptor است که اطلاعات سوکت ساخته شده برمیگرداند. در این جا از این جهت که تابع چند خروجی دارد، چنین ساختاری را تعریف کردیم تا بتوانیم تمام خروجی‌ها را در این قالب به صورت لفاف‌بندی شده باز گردانیم.

```c
typedef struct {
    int sockfd;
    struct sockaddr_in host_addr;
    int host_addrlen;
}
ConnectionDescriptor;
```

تابع start_web_server ابتدا یک سوکت را می‌سازد. سپس پورت مورد نظر را به آن آدرس متصل می‌کنیم. سپس پورت را باز می‌کنیم تا به درخواست‌های روی آن پورت پاسخ دهد.

```c
ConnectionDescriptor * start_web_server(WebServer * webServer) {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("Cannot initialize socket.\n");
        exit(1);
    }
    printf("Socket successfully initialized\n");

    struct sockaddr_in host_addr;
    int host_addrlen = sizeof(host_addr);

    host_addr.sin_family = AF_INET;
    host_addr.sin_port = htons(webServer -> port);
    host_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sockfd, (struct sockaddr * ) & host_addr, host_addrlen) != 0) {
        perror("Cannot bind socket to address.\n");
        exit(1);
    }
    printf("Socket successfully binded\n");

    if (listen(sockfd, SOMAXCONN) != 0) {
```

```
        perror("Cannot listen for connections.\n");
        exit(1);
    }
    printf("Listening for connections\n");

    ConnectionDescriptor * connectionDescriptor = (ConnectionDescriptor * )
        malloc(sizeof(ConnectionDescriptor));
    connectionDescriptor -> sockfd = sockfd;
    connectionDescriptor -> host_addr = host_addr;
    connectionDescriptor -> host_addrlen = host_addrlen;
    return connectionDescriptor;
}
```

زمانی که درخواست TCP قبول می‌شود، یک سوکت جدید ساخته می شود و ارتباطات روی سوکت جدید انجام می‌شود. برای اینکه این ارتباطات توسط رشته کارگر انجام شود و سربار روی روی رشته مادر نداشته باشد، زمانی که کانکشن برقرار می‌شود. اطلاعات پورت جدید در قالب داده ساختار زیر به رشته کارگز ارسال می‌شود تا مورد پردازش قرار بگیرد.

```
typedef struct {
    int id;
    int newsockfd;
    WebServer * webServer;
}
HandleRequestArgs;
```

هر درخواست بلافاصله پس از پذیرش به یک ترد کارگر سپرده می‌شود تا امور مربوط به آن انجام شود. در ابتدا از اتصال برقرار شده متن درخواست خوانده می‌شود. سپس متن درخواست Parse میشود.

پس از آن با توجه به آدرسی که کاربر وارد کرده است، هندلر مربوطه به صورت longest prefix matching پیدا می‌شود. اگر هندلر مربوطه پیدا نشد، ارور ۴۰۴ به کاربر برگردانده می‌شود. اگر که هندلر پیدا شد، تابع مربوط به پردازش آن هندلر اجرا می‌شود. در نهایت پاسخ تولید شده سریالایز می‌شود و به کاربر برگردانده می‌شود.

```
void * handle_request(void * args) {
    HandleRequestArgs * handleRequestArgs = (HandleRequestArgs * ) args;
    int id = handleRequestArgs -> id;
    int newsockfd = handleRequestArgs -> newsockfd;
    WebServer * webServer = handleRequestArgs -> webServer;

    struct timeval time_thread_dedicated;
    gettimeofday(&time_thread_dedicated, NULL);
    printf("Id: %d, Time thread dedicated: %ld\n",
          id, time_thread_dedicated.tv_sec * 1000000
          + time_thread_dedicated.tv_usec);
    fprintf(log_file, "Id: %d, Time thread dedicated: %ld\n",
           id, time_thread_dedicated.tv_sec * 1000000
           + time_thread_dedicated.tv_usec);
    fflush(log_file);

    char * buffer = malloc(BUFFER_SIZE);

    int valread = read(newsockfd, buffer, BUFFER_SIZE);
    if (valread < 0) {
        perror("Cannot read from socket.\n");
        return NULL;
    }

    HttpRequest * http_request = malloc(sizeof(HttpRequest));
    parse_request(buffer, http_request);
    http_request -> id = id;

    Handler * handler = NULL;
    for (int i = 0; i < webServer -> num_handlers; i++) {
        if (strncmp(webServer -> handlers[i].path,
```

```
                http_request -> path_without_query,
                strlen(webServer -> handlers[i].path)) == 0) {
        if (handler == NULL ||
            strlen(webServer -> handlers[i].path)
            > strlen(handler -> path)) {
            handler = & webServer -> handlers[i];
        }
    }
}


HttpResponse * http_response = NULL;

if (handler != NULL) {
    http_response = handler -> handler_function(http_request);
} else {
    http_response = response(404, "Not Found", "");
}

free_request(http_request);

char * response_string = serialize_response(http_response);

struct timeval time_process_finished;
gettimeofday(&time_process_finished, NULL);
printf("Id: %d, Time thread dedicated: %ld\n",
       id, time_process_finished.tv_sec * 1000000
       + time_process_finished.tv_usec);
fprintf(log_file, "Id: %d, Time thread dedicated: %ld\n",
        id, time_process_finished.tv_sec * 1000000
        + time_process_finished.tv_usec);
fflush(log_file);
```

```
64
65      int valwrite = write(
66          newsockfd,
67          response_string,
68          strlen(response_string)
69      );
70      if (valwrite < 0) {
71          perror("Cannot write to socket.\n");
72          return NULL;
73      }
74
75      free_response(http_response);
76      close(newsockfd);
77      free(buffer);
78      free(response_string);
79      return NULL;
80  }
```

حال باید تابعی داشته باشیم که بر روی پورت مورد نظر منتظر باشد و بلافاصله که درخواست آمد، اتصال جدید را بپذیرد و آن را به یک رشته کارگر بسپارد. تابع accept_connection پس از ساختن پورت سرور بر روی آن منتظر میماند و هر زمان که درخواستی برای براقراری ارتباط آمد، آن را می‌پذیرد و به یک رشته کارگر می‌سپارد.

```
1  void accept_connections(WebServer * webServer,
2                          ConnectionDescriptor * connectionDescriptor) {
3      int id_counter = 0;
4
5      int sockfd = connectionDescriptor -> sockfd;
6      struct sockaddr_in host_addr = connectionDescriptor -> host_addr;
7      int host_addrlen = connectionDescriptor -> host_addrlen;
8
9      while (1) {
```

```
10          int newsockfd = accept(sockfd, (struct sockaddr * )
11              & host_addr, (socklen_t * ) & host_addrlen);
12          if (newsockfd < 0) {
13              perror("Cannot accept connection.\n");
14              continue;
15          }
16          printf("Connection accepted\n");
17
18          int id = id_counter++;
19
20          struct timeval time_accepted;
21          gettimeofday(&time_accepted, NULL);
22          printf("Id: %d, Time accepted: %ld\n", id,
23              time_accepted.tv_sec * 1000000 + time_accepted.tv_usec);
24          fprintf(log_file, "Id: %d, Time accepted: %ld\n", id,
25              time_accepted.tv_sec * 1000000 + time_accepted.tv_usec);
26          fflush(log_file);
27
28          HandleRequestArgs * handleRequestArgs = (HandleRequestArgs * )
29          malloc(sizeof(HandleRequestArgs));
30          handleRequestArgs -> id = id;
31          handleRequestArgs -> newsockfd = newsockfd;
32          handleRequestArgs -> webServer = webServer;
33          addTaskThreadPool(webServer -> threadPool,
34                  handle_request, handleRequestArgs);
35
36          if (!webServer -> open)
37              break;
38      }
39  }
```

API ۷

در این قسمت میخواهیم توابعی را که کاربر میتواند در سمت سرور اجرا کند و پاسخ آن را دریافت کند توضیح دهیم.
درخواستی که کاربر ارسال میکند، باید به صورت زیر باشد:

URL:Port/HandlerPath/a=value1?b=value2

for example: 127.0.0.1:8080/Add/a=7?b=34

پس از parse کردن درخواست کاربر به یک http request میرسیم. سپس با استفاده از این http request ، تلاش میشود تا handler مناسب برای درخواست انتخاب شود. ساختار Handler به صورت زیر است:

```
typedef struct {
    char * path;
    HttpResponse * ( * handler_function)(HttpRequest * );
}
Handler;
```

در ساختار بالا، path همان HandlerPath موجود در درخواست کاربر است. پارامتر دوم آن نیز تابعی است که باید در صورت وجود HandlerPath مورد نظر در درخواست، روی دو ورودی داده شده اجرا شود.
همچنین با استفاده از تابع add-new-handler در سمت سرور، امکان تعریف handler های بیشتر توسط سرور و ارائه خدمات بیشتر وجود دارد. کافی است تابعی را که میخواهیم اجرا شود به همراه Path مطابق زیر تعریف کنیم و این تابع را در ابتدای بالا آمدن سرور، صدا بزنیم.

```
void add_new_handler(WebServer * webServer, char * path,
  HttpResponse * ( * handler_function)(HttpRequest * )) {
   Handler * handler = & webServer -> handlers[webServer -> num_handlers++];
   handler -> path = path;
   handler -> handler_function = handler_function;
   }
```

نحوه اضافه کردن هندلر به وب سرور هم در ابتدای برنامه هم به صورت زیر است:

```
add_new_handler(webServer, "/ping", ping_handler);
add_new_handler(webServer, "/add", add_handler);
add_new_handler(webServer, "/sub", sub_handler);
add_new_handler(webServer, "/mult", mult_handler);
add_new_handler(webServer, "/div", div_handler);
```

ساختار توابع هندلر نیز به اینصورت است که http-request را به عنوان ورودی میگیرند و http-response را خروجی میدهند.

در حال حاضر همانطور که در کد بالا مشخص است، وب سرور ارائه شده ۵ تابع هندلر را پیاده‌سازی کرده است. توابع جمع، تفریق، ضرب، تقسیم و پینگ. کد پیاده سازی شده تقسیم در زیر آورده شده است:

```

HttpResponse * div_handler(HttpRequest * http_request) {
    char * a = get_param(http_request, "a");
    char * b = get_param(http_request, "b");
    int a_int = atoi(a);
    int b_int = atoi(b);
    if (b_int == 0) {
        char * cannot_string = malloc(50* sizeof(char));
        strcpy(cannot_string, "Cannot divide by 0");
        HttpResponse * http_response = response(400, "Bad Request",
        cannot_string);
        add_header(http_response, "Content-Type", "text/plain");
        return http_response;
    }
    int div = a_int / b_int;
    char * div_string = malloc(10);
    sprintf(div_string, "%d", div);
    HttpResponse * http_response = response(200, "OK", div_string);
    add_header(http_response, "Content-Type", "text/plain");
```

```
20      return http_response;
21  }
```

همانطور که مشاهده میکنید در این قطعه کد، a و b را با استفاده از http-request دریافت میکنیم و آنها را به int تبدیل میکنیم. سپس در خط ۷ بررسی میشود که مقسوم علیه ۰ نباشد. در صورت ۰ بودن مقسوم علیه (آرگومان دوم)، یک response با خطای Bad Request ساخته میشود و برگردانده میشود. در غیر اینصورت تقسیم انجام میشود و در نهایت پاسخ نهایی تبدیل به رشته شده و div-string ریخته میشود. سپس یک response با هدر OK و بدنه div-string ساخته میشود و در نهایت این response را برمیگرداند.

استفاده از این توابع نیز به اینصورت است که پس از parse کردن درخواست کاربر، طبق کد زیر handler مناسب درخواست را پیدا میکنیم (با توجه به پارامتر path موجود در handler)، سپس آن را نگهداری میکنیم.

```
1  Handler * handler = NULL;
2      for (int i = 0; i < webServer -> num_handlers; i++) {
3          if (strncmp(webServer -> handlers[i].path,
4              http_request -> path_without_query,
5              strlen(webServer -> handlers[i].path)) == 0) {
6               if (handler == NULL ||
7                strlen(webServer -> handlers[i].path) > strlen(handler -> path))
8               {
9                   handler = & webServer -> handlers[i];
10              }
11          }
12      }
```

در گام بعد بررسی میکنیم که اگر handler مشخصی پیدا نشد، یک response با خطای Not Found ایجاد میشود.

```
HttpResponse * http_response = NULL;


    if (handler != NULL) {
        http_response = handler -> handler_function(http_request);
    } else {
        http_response = response(404, "Not Found", "");
    }
```

۸ نتایج

در این قسمت میخواهیم یک سری از تست ها و بررسی هایی را که انجام داده ایم را توضیح دهیم.

در ابتدا همانطور که در داک گفته شده بود، یک سری آماره را باید لاگ میکردیم و نمونه لاگ در پوشه پروژه با نام log.txt موجود است.

همچنین یک کد پایتون را پیاده سازی کردیم (با نام statistic.py در پوشه پروژه) که لاگ تولید شده توسط وب سرور را دریافت میکند و میانگین زمان انتظار و اجرای تسک ها توسط هر ریسمان را نشان میدهد و در فایل result.txt چاپ میکند.

به ازای ۴ کانفیگ متفاوت تست انجام شد. در هر کدام از تست ها ۲۱۵ عدد درخواست را از سمت کلاینت به سرور میزدیم و خروجی آن ها را لاگ گرفتیم و با استفاده از کد پایتونی که زده بودیم، آماره های میانگین زمان انتظار و میانگین انجام هر تسک را دریافت کردیم. نتایج این آزمایش به صورت زیر است(لاگ مربوط به این آزمایش در پوشه پروژه در پوشه Results آورده شده است):

1. Number of Thread = 1, Queue size = 2:
   waitng time=174.7, each-task time=194976
2. Number of Thread = 20, Queue size = 2:
   waitng time=154.8, each-task time=223654
3. Number of Thread = 1, Queue size = 20:
   waitng time=1032.3, each-task time=176223
4. Number of Thread = 20, Queue size = 20:
   waitng time=296.1, each-task time=170350

- همانطور که مشاهده میکنید با افزایش تعداد ریسمان ها، زمان انتظار کاهش می‌یابد. برای مثال در حالت ۱ به ۲ که تعداد ریسمان ها را زمانی که طول صف برابر ۲ است، ۲۰ برابر میکنیم، زمان انتظار از 174.7 به 154.8 میرسد و به نوعی کاهشی ۱۲ درصدی دارد. در صورتیکه اگر همین امر را در ارتباط با حالت ۳ و ۴ بررسی کنیم که طول صف برابر ۲۰ است، آنگاه این کاهش به ۷۲ درصد میرسد. این نشان میدهد هر چقدر طول صف بیشتر باشد، تاثیر تعداد ریسمان در زمان انتظار بیشتر است.

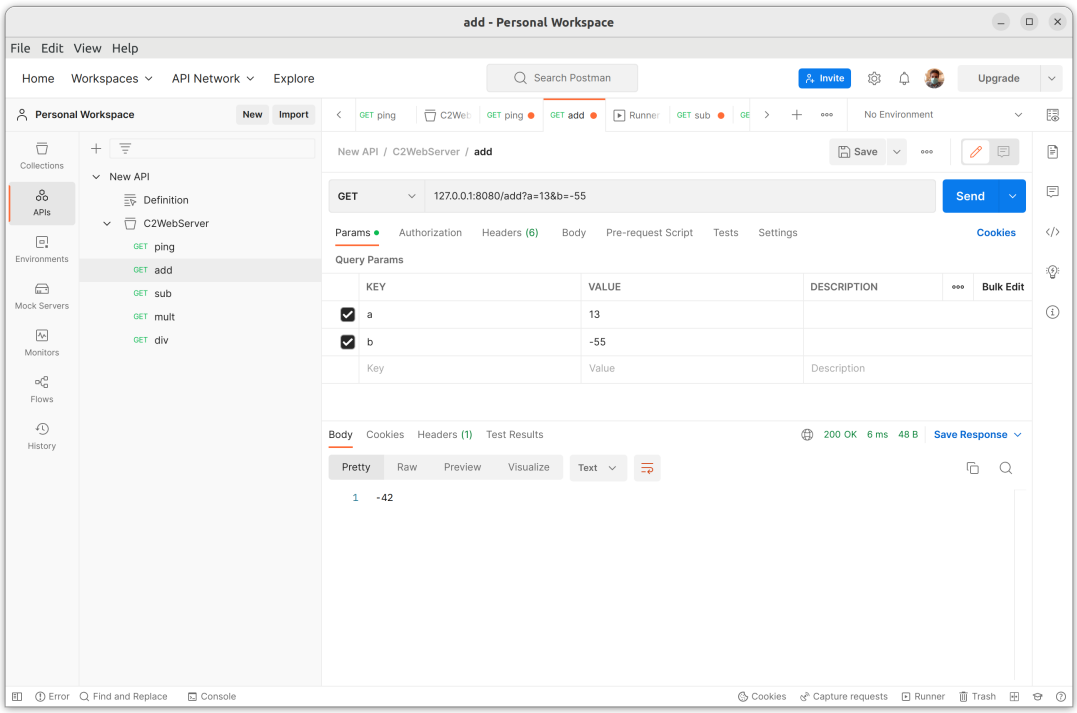
- همچنین نکته دیگر اینکه در حالتی که تعداد ریسمان برابر ۱ است و طول صف زیاد میشود، زمان انتظار رشد نسبتا زیادی خواهد داشت و این امر ضرورت استفاده از وب سرور چندریسه‌ای را نشان میدهد. از طرفی میدانیم طول صف با تعداد درخواست هایی که نادیده گرفته میشوند (به دلیل حجم بالای درخواست ها و پر شدن صف)، رابطه معکوس دارد و به همین دلیل در کاربردهای واقعی برای جلوگیری از miss شدن درخواست ها، صف ها معمولا طولانی هستند و همین امر نیاز به چندریسه بودن سرور را مضاعف میکند.
- نکته دیگری که میتوان به آن اشاره کرد سربار افزایش تعداد ریسمان است که در آماره -each task-time مشخص شده است. همانطور که مشاهده میشود، با افزایش ریسمان، زمان لازم برای انجام هر تسک افزایش می یابد و این اتفاق به دلیل سربار افزایش ریسمان است.

مسئله دیگری که بررسی کردیم، Memory Leakage بود. به اینصورت که در ابتدا متغیرهایی که حافظه به آن ها تخصیص میدادیم را به صورت دقیق و سازمان دهی شده آزاد نکرده بودیم. همین امر باعث شده بود تا به سرعت به مشکل حافظه بربخوریم و با اجرای دستور مربوط به valgrind، متوجه شدیم حدود ۸ مگابایت در طی یک اجرای کوتاه، حافظه را آزاد نکرده بودیم. سپس با ارائه بهبودهایی و اضافه کردن free به صورت مدون و سازمان دهی شده به کد، این مقدار را تا ۳۴۰ کیلوبایت کاهش دادیم:

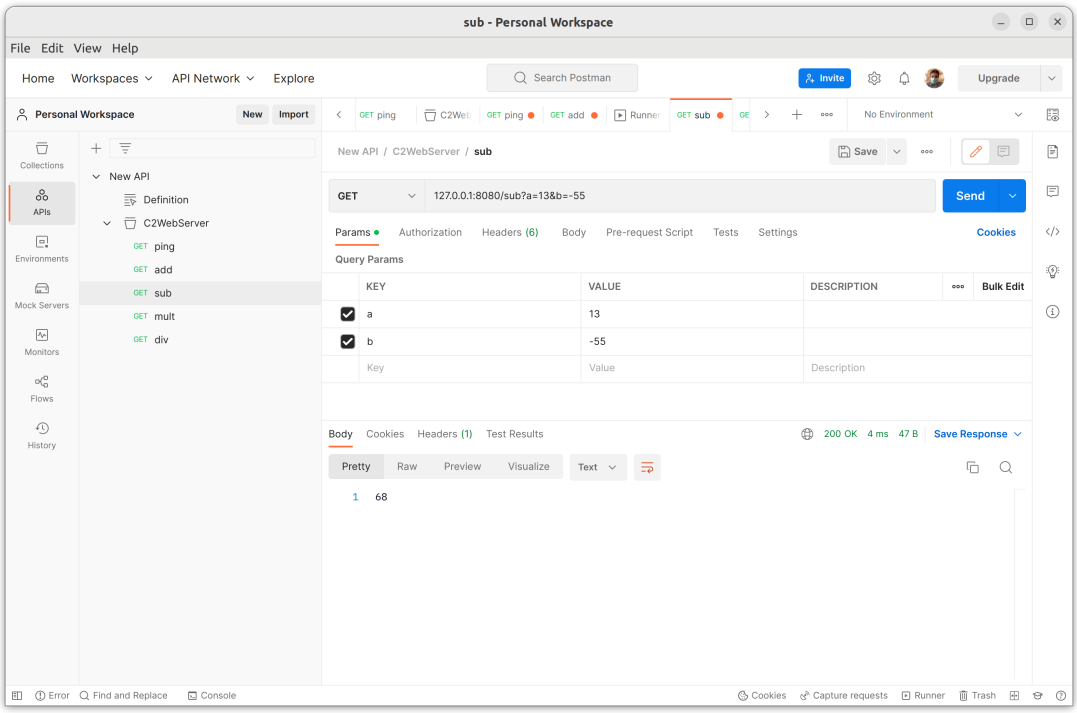
```
==42626== LEAK SUMMARY:
==42626==  definitely lost: 346,752 bytes in 21,672 blocks
==42626==  indirectly lost: 346,752 bytes in 21,672 blocks
==42626==    possibly lost: 544 bytes in 2 blocks
==42626==  still reachable: 14,200 bytes in 14 blocks
==42626==       suppressed: 0 bytes in 0 blocks
==42626== Reachable blocks (those to which a pointer was found) are not shown
```

در ادامه پاسخ مشاهده شده توسط وب سرور را به ازای درخواست‌های متفاوت نشان داده‌ایم:

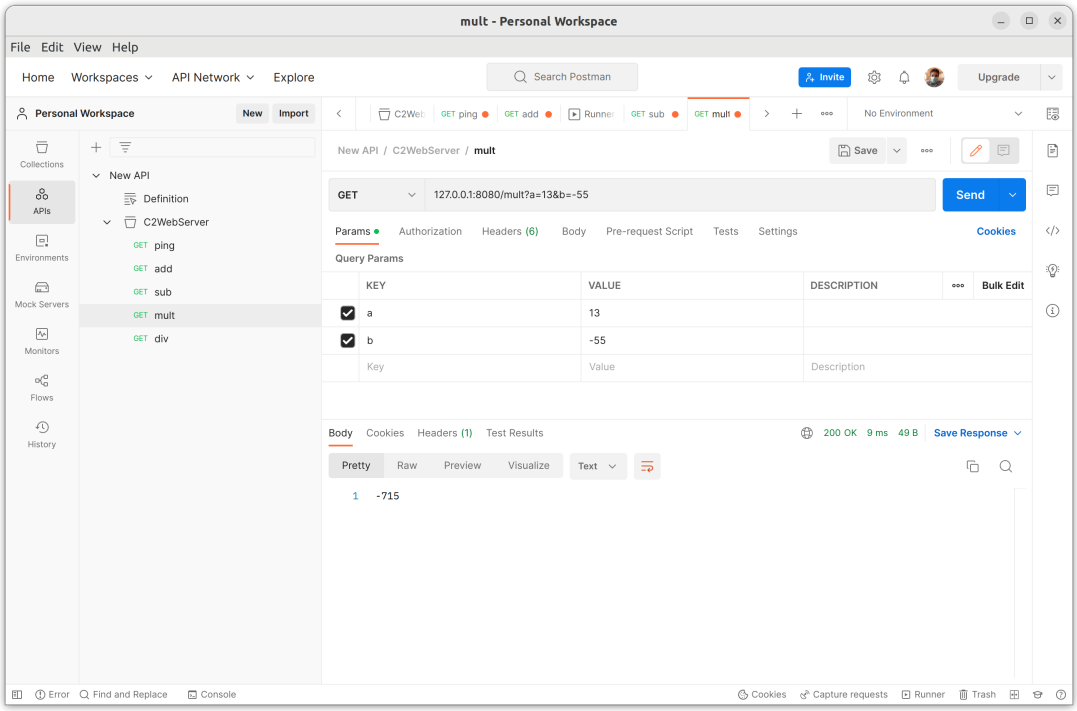
- add



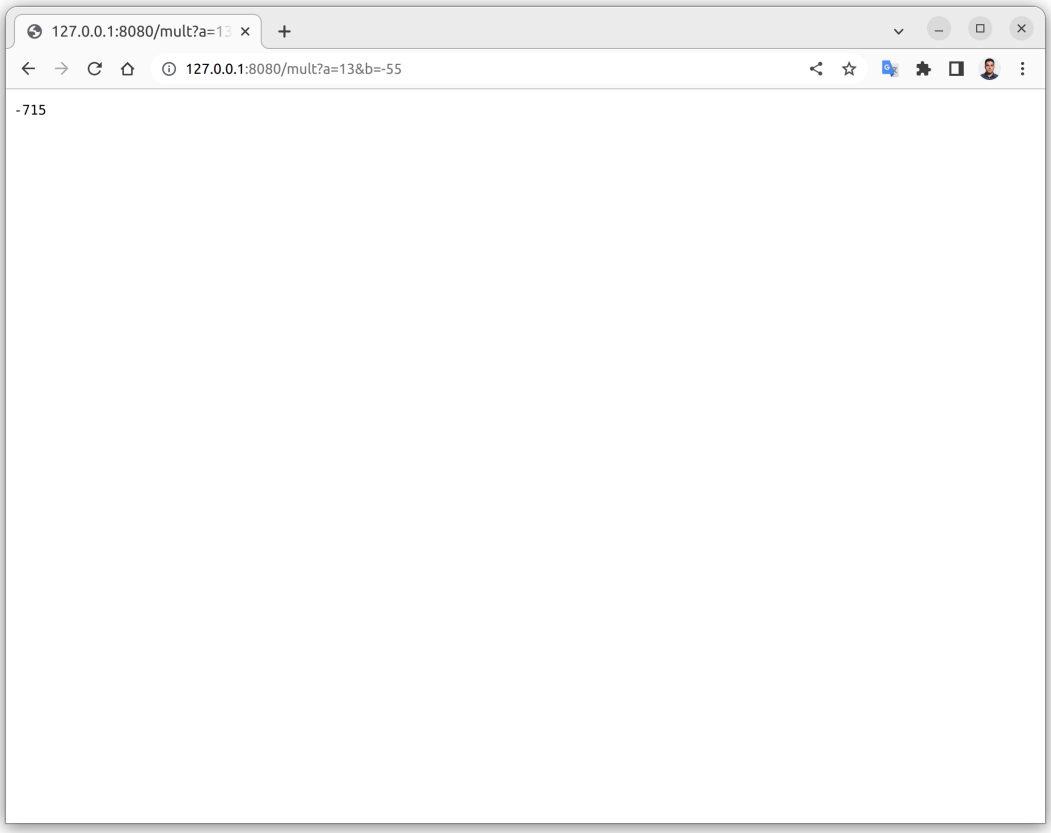
- sub



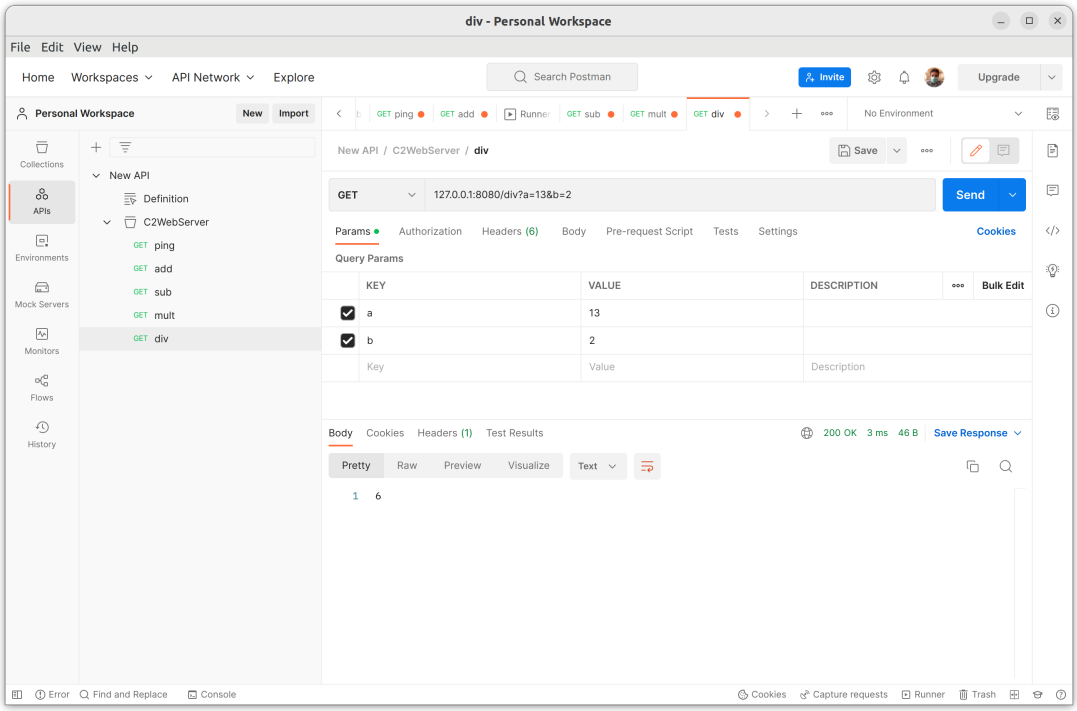
mult •


mult - Personal Workspace
File Edit View Help
Home Workspaces API Network Explore
Search Postman
Invite
Upgrade
Personal Workspace
New
Import
No Environment
Collections
APIs
Environments
Mock Servers
Monitors
Flows
History
New API
Definition
C2WebServer
GET ping
GET add
GET sub
GET mult
GET div
New API / C2WebServer / mult
Save
GET
127.0.0.1:8080/mult?a=13&b=-55
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
KEY
VALUE
DESCRIPTION
Bulk Edit
a
13
b
-55
Key
Value
Description
Body
Cookies
Headers (1)
Test Results
200 OK
9 ms
49 B
Save Response
Pretty
Raw
Preview
Visualize
Text
1 -715
Error
Find and Replace
Console
Cookies
Capture requests
Runner
Trash

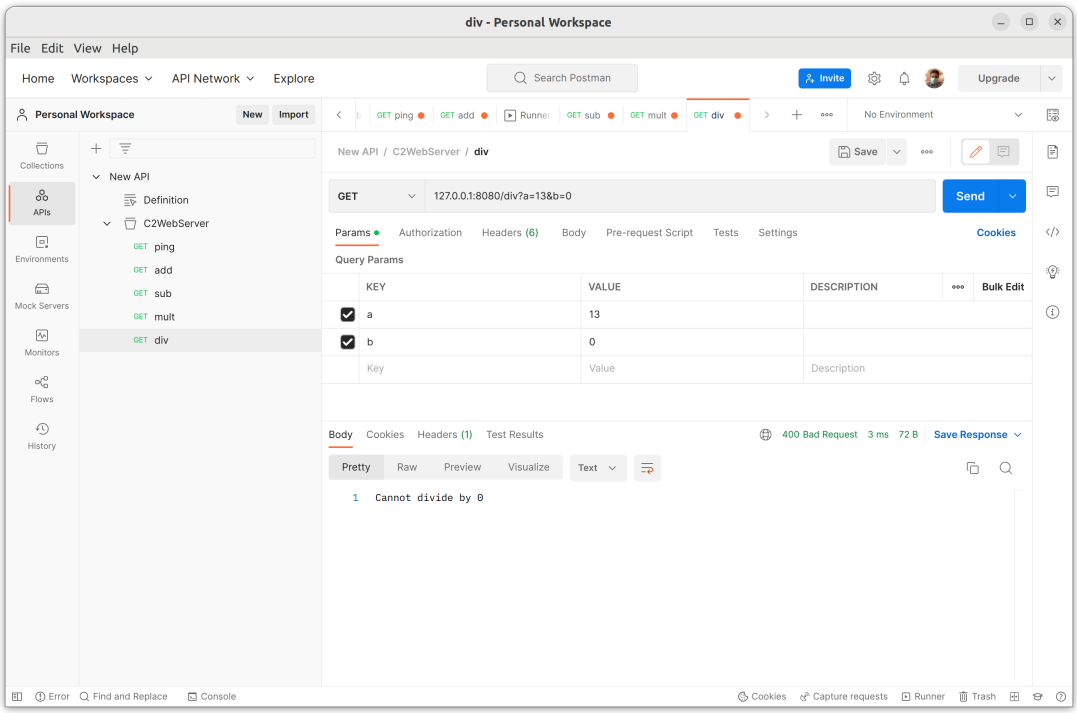
mult •


127.0.0.1:8080/mult?a=13
127.0.0.1:8080/mult?a=13&b=-55
-715

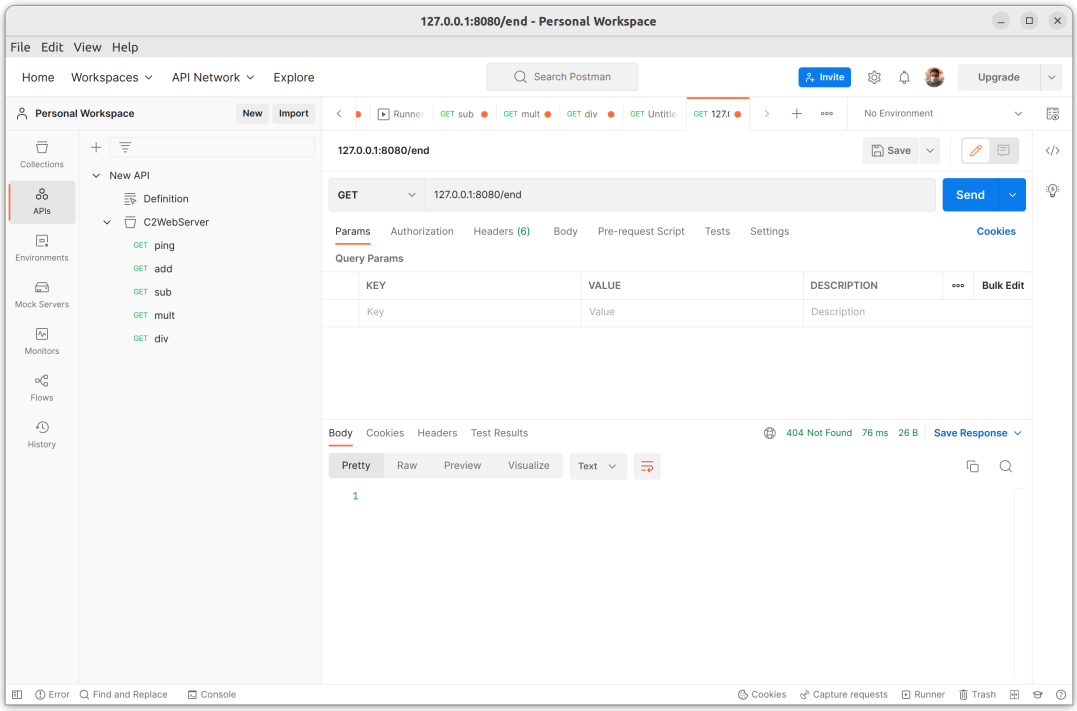
div •



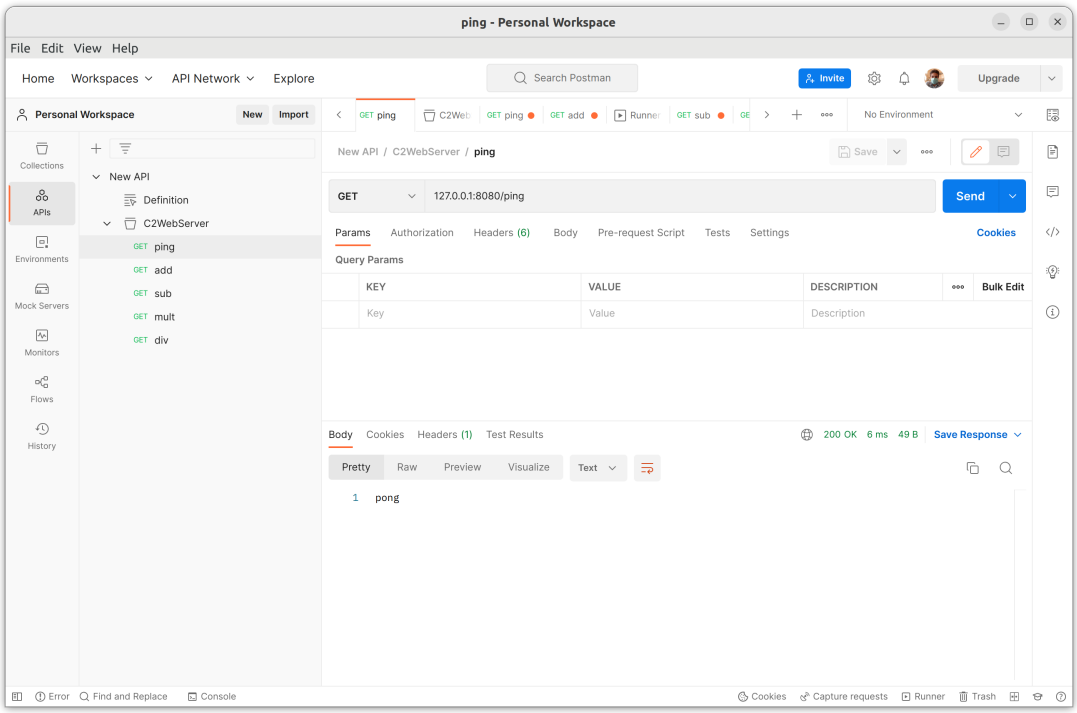
error with div •



404 not found •



ping •



References

[1] J. Kurose and K. Ross. *computer networking a top-down approach 8th edition slides*. 2022.