



اهداف تمرین

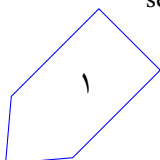
- پیاده سازی یک http server
- آشنایی با مفهوم ریسه و کاربرد آن

۱. مقدمه

امروزه پروتکل انتقال ابرمتن (HTTP) رایج‌ترین پروتکل سطح برنامه‌مورد استفاده است. مانند بسیاری از پروتکل‌های شبکه، HTTP از مدل کارخواه^۱ - کارگزار^۲ استفاده می‌کند. کارخواه HTTP یک اتصال شبکه به یک کارگزار HTTP باز می‌کند و یک پیغام درخواست HTTP می‌فرستد. سپس، کارگزار با یک پیغام پاسخ که شامل بعضی منابع (فایل، متن، داده دودویی) خواسته شده توسط کارخواه می‌شود پاسخ می‌دهد. در این تمرین، شما یک کارگزار HTTP پیاده‌سازی خواهید کرد که به درخواست‌های HTTP GET رسیدگی می‌کند. شما این عملکرد را با استفاده از سرآیندهای درخواست HTTP، پشتیبانی از کدهای خطای HTTP، ساخت فهرست‌های دایرکتوری با HTML و ساخت یک پروکسی HTTP فراهم خواهید کرد.

* باسپاس از گروه دستیاران آموزشی

¹client
²server



۲. راه‌اندازی مقدمات

به ماشین مجازی خود در Vagrant وارد شده و همانند تمارین قبل، یک پوشه متناسب با این تمرین ایجاد کنید و در آن پوشه موارد مرتبط با این تمرین را قرار دهید.

```
1 cd ~/code/hw2
```

تیم دستیاران تمرین کد شروع کار بر روی httpserver و یک Makefile ساده را برای شما در قسمت skeleton قرار داده‌اند. (شما می‌توانید skeleton را از اینجا دریافت کنید.)
به منظور اجرای httpserver می‌بایست دستورهای زیر را اجرا کنید:

```
1 make
```

```
2 ./ httpserver
```

هم‌چنین به منظور خاتمه دادن به اجرای httpserver پس از شروع آن، می‌توانید CTRL-C را فشار دهید.
آدرس IP ماشین مجازی شما 192.168.162.162 است. شما باید قادر باشید تا دستور `ping 192.168.162.162` را بر روی رایانه‌ی میزبان اجرا کنید و پاسخ ping را از ماشین مجازی دریافت کنید. اگر قادر نیستید تا ماشین مجازی را ping کنید، می‌توانید در عوض port forwarding را در Vagrant راه‌اندازی کنید.^۳

۳. پیش زمینه

۱.۳ ساختار درخواست HTTP

قالب پیغام درخواست HTTP به صورت زیر است:

- یک خط درخواست HTTP (شامل یک متود، method یک رشته پرسش و نسخه پروتکل HTTP)
- صفر، یک یا چند خط برای سرآیند HTTP
- یک خط خالی (فقط شامل یک CRLF)

خط‌های درخواست‌های HTTP به CRLF ختم می‌شوند که در زبان C با `\r\n` نمایش داده می‌شود. در زیر یک مثال از درخواست HTTP که توسط مرورگر کروم به یک کارگزار وب HTTP که بر روی localhost (127.0.0.1) و پورت ۸۰۰۰ بالا آمده است مشاهده می‌کنید. (CRLF به صورت اسکپ شده نشان داده شده است):

^۳ برای اطلاعات بیشتر: https://www.vagrantup.com/docs/networking/forwarded_ports.html

```

1 GET /hello.html HTTP/1.0\r\n
2 Host: 127.0.0.1:8000\r\n
3 Connection: keep-alive\r\n
4 Accept: text/html, application/xhtml+xml, application/xml;q=0.9,
5 image/webp, */*;q=0.8\r\n
6 User-Agent: Chrome/45.0.2454.93\r\n
7 Accept-Encoding: gzip, deflate, sdch\r\n
8 Accept-Language: en-US,en;q=0.8\r\n
9 \r\n

```

خطوط سرآیند اطلاعاتی در مورد درخواست به ما می دهد. در زیر بعضی از انواع سرآیندهای درخواست HTTP آمده است:

- HOST: شامل بخش نام میزبان در URL درخواست HTTP می شود. (برای مثال `ce.sharif.edu` یا `127.0.0.1:8000`)

- User-Agent: برنامه کارخواه HTTP را مشخص می کند. به صورت "`Program name/x.xx`" می آید که `x.xx` نسخه برنامه می باشد. در مثال بالا مرورگر گوگل کروم این سرآیند را برابر `Chrome/45.0.2454.93` قرار می دهد.

۲.۳ ساختار پاسخ HTTP

قالب پیغام پاسخ HTTP به شکل زیر است:

- یک خط شامل نسخه پروتکل HTTP کد وضعیت^۴ و یک توصیف متنی از کد وضعیت

- صفر، یک یا چند خط سرآیند HTTP

- یک خط خالی (فقط شامل یک CSRF تنها)

- محتوای مورد درخواست توسط درخواست HTTP

هر خط در درخواست های HTTP به CRLF ختم می شود که در زبان C به صورت `\r\n` نمایش داده می شود. در زیر یک نمونه پاسخ HTTP با کد وضعیت ۲۰۰ و یک فایل HTML پیوست شده به پاسخ می باشد^۵:

^۴Status Code

^۵CRLF به صورت اسکیپ شده نشان داده شده است

```
1 HTTP/1.0 200OK\r\n
2 Content-Type: text/html\r\n
3 Content-Length: 128\r\n
4 \r\n
5 <html >\n
6 <body >\n
7 <h1>Hello World</h1 >\n
8 <p >\n
9 Let 's see if thi sworks \n
10 </p >\n
11 </body >\n
12 </html >\n
```

خطوط وضعیت ممکن است HTTP/1.0 200 OK (همان طور که در مثال بالا آمده است)، HTTP/1.0 404 Not Found یا موارد دیگری باشد. کد وضعیت یک عدد ۳ رقمی است که رقم اول آن دسته‌بندی عمومی پاسخ را مشخص می‌کند:

- 1xx: پیامی فقط شامل یک سری اطلاعات را نشان می‌دهد.
- 2xx: نشان از موفقیت درخواست است.
- 3xx: کارخواه را به URL دیگری تغییر مسیر می‌دهد.
- 4xx: خطایی از طرف کارخواه را نشان می‌دهد.
- 5xx: خطایی از طرف کارگزار را نشان می‌دهد.

خطوط سرآیند اطلاعاتی درباره‌ی پاسخ فراهم می‌کنند. در زیر تعدادی از انواع سرآیند پاسخ HTTP آمده است:

- Content-Type: نوع MIME داده‌ی پیوست‌شده به پاسخ را نشان می‌دهد. برای مثال می‌تواند `text/html` یا `text/plain` باشد.
- Content-Length: تعداد بایت بدنه‌ی پاسخ را نشان می‌دهد.

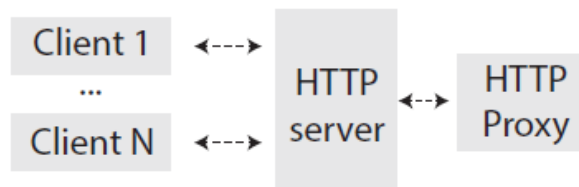
۴. وظیفه‌ی شما

۱.۴. طرح کلی از کارگزار وب HTTP

از نظر شبکه، کارگزار وب HTTP پایه‌ی شما باید موارد زیر را پیاده کند:

۱. یک سوکت بسازد که بر روی یک پورت گوش فرا دهد.
۲. صبر کند تا یک کارخواه به آن پورت متصل شود.
۳. کارخواه را بپذیرد و یک اتصال سوکت جدید فراهم کند.
۴. از سوکت بخواند و درخواست HTTP را پارس کند.
۵. با توجه به آرگومان‌های دستور، یکی از دو کار زیر را انجام دهد:

- یک فایل را از فایل سیستم محلی ارائه کند یا پیام 404 Not Found برگرداند
- درخواست را به یک کارگزار HTTP دیگر پروکسی کند



شکل ۱: وقتی به عنوان پروکسی استفاده می‌شود، کارگزار درخواست‌ها را با فرستادن آن‌ها به یک کارگزار HTTP از راه دور (پروکسی) ارائه می‌کند. پاسخ‌های ارسالی از پروکسی به کارخواه‌ها پس فرستاده می‌شود

کارگزار HTTP در حالت فایل یا پروکسی خواهد بود و هیچگاه در هر دو حالت عمل نخواهد کرد.

۶. سرآیند پاسخ HTTP متناسب و فایل پیوست آن را به کارخواه پس می‌فرستد (یا در صورت درخواست یک فایل ناموجود، یک پیغام خطا با ساختار صحیح می‌فرستد)

اسکلت کد در حال حاضر شامل گام‌های ۱ تا ۴ می‌شود. شما باید گام‌های ۵ و ۶ و یک استخر ریسه^۶ را برای پشتیبانی از چند درخواست HTTP هم‌زمان پیاده‌سازی کنید. httplib.c/h به شما برای گام‌های ۵ و ۶ و wq.c/h برای استخر ریسه کمک خواهد کرد.

۲.۴ استفاده از `./httpserver`

در زیر چگونگی استفاده از `httpserver` آمده است. برای راحتی شما، کد پارس کردن آرگومان‌ها برای شما از قبل پیاده‌سازی شده است:

```

1 ./ httpserver --help
2 Usage: ./ httpserver --files files/ --port 8000 [--num-threads 5]
3 ./ httpserver --proxy ce.sharif.edu:80 --port 8000 [--num-threads 5]

```

^۶Thread Pool

گزینه‌های موجود به شرح زیر می‌باشند:

- `--files` یک مسیر که فایل‌ها از آن ارائه می‌شوند را انتخاب می‌کند. شما باید فایل‌ها را از پوشه‌ی `files/` ارائه کنید. (مسیر نسبت به `CWD` حساب می‌شود، برای مثال اگر شما در حال حاضر `cd hw2/` کرده‌اید باید از `"/files files/` استفاده کنید)
- `--proxy` یک کارگزار بالادست برای پروکسی انتخاب می‌کند. آدرس پاس داده‌شده به عنوان آرگومان می‌تواند یک شماره پورت نیز بعد از کاراکتر `:` داشته‌باشد. (برای مثال `ce.sharif.edu:80`) اگر شماره پورت مشخص نشده بود پیش‌فرض پورت ۸۰ است.
- `--port` پورته‌ی که کارگزار HTTP روی آن برای اتصالات پیش‌رو گوش فرا می‌دهد را مشخص می‌کند. در هر دو حالت ارسال فایل و پروکسی از آن استفاده می‌شود. (این با پورت پروکسی متفاوت است)
- `--num-threads` تعداد ریسه‌های استخر ریسه که ارائه کردن درخواست‌های کارخواه‌ها را به صورت همزمان میسر می‌سازد را مشخص می‌کند. این آرگومان در ابتدا بدون استفاده است و به شما بستگی دارد که از آن استفاده کنید یا خیر.

شما نباید هم‌زمان از هر دو گزینه `--files` و `--proxy` استفاده کنید، یا در غیر این صورت آخرین گزینه از بین این دو، اولی را بازنویسی می‌کند. گزینه‌ی `--proxy` می‌تواند آدرس IP نیز بگیرد. در ابتدا `--num-threads` بلااستفاده و دل‌بخواهی است. وظیفه‌ی شماست که از آن برای پیاده‌سازی استخر ریسه‌تان استفاده کنید. مشکلی ندارد اگر در نهایت کارکرد تک ریسه‌ای را حذف و آرگومان `--num-threads` را اجباری کنید. اگر می‌خواهید از شماره پورته‌ی از ۰ تا ۱۰۲۳ استفاده کنید نیاز دارید تا `httpserver` را در حالت `root` اجرا کنید. این پورت‌ها، پورت‌های رزرو شده می‌باشند و تنها می‌توانند به یوزر `root` اختصاص داده شوند. این کار را می‌توانید با اجرای دستور `"sudo ./httpserver --files files/"` انجام دهید.

۳.۴ دسترسی به کارگزار HTTP

با رفتن به آدرس `HTTP:192.168.162.162:8000` چک کنید که کارگزار HTTP کار می‌کند. همچنین می‌توانید درخواست‌های HTTP خود را با برنامه `curl` که بر روی ماشین مجازیتان نصب شده است بفرستید. مثالی برای استفاده از `curl` در زیر آمده است:

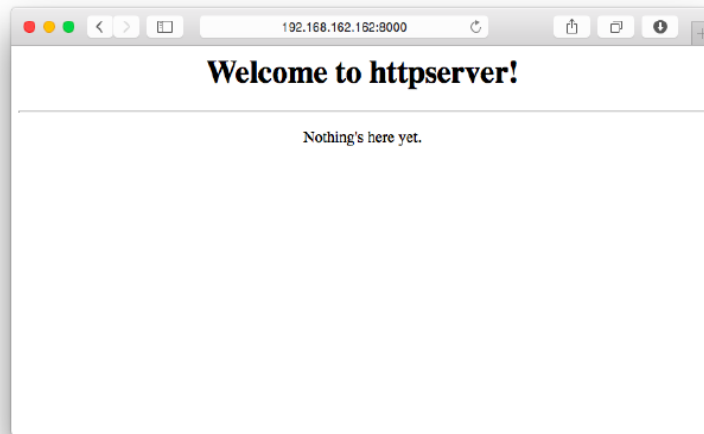
```
1 curl -v http://192.168.162.162:8000/
```

می‌توانید یک اتصال به کارگزار HTTPتان به طور مستقیم بر روی سوکت شبکه با استفاده از `netcat(nc)` ایجاد کنید و درخواست HTTPتان را تایپ کنید. (یا از یک فایل پایپ کنید)

```
1 nc -v 192.168.162.162 8000
```

```
2 Connection to 192.168.162.162 8000 port [tcp/*] succeeded!
```

(سپس می‌توانید درخواستتان را بنویسید تا به کارگزار ارسال شود)



شکل ۲

۴.۴ پیغام‌های خطای رایج

۱.۴.۴ Failed to bind on socket: Address already in use

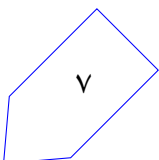
این پیام بدین معنی است که شما یک `httpserver` در حال اجرا در پس‌زمینه دارید. به عبارت دیگر، برنامه دیگری در حال استفاده از پورت موردنظر شماست. این می‌تواند زمانی اتفاق بیفتد که برنامه شما پرده‌هایی که سوکت را در اختیار دارند را `leak` کند، یا برای مثال بدون خاموش کردن `httpserver` ماشین مجازی را خاموش کنید. دستور `netstat -tulpn` در لینوکس به شما کمک می‌کند تا متوجه شوید در حال حاضر کدام برنامه‌ها کدام سوکت‌ها را در اختیار دارند. برای بستن پرده‌ها می‌توانید از دستور `kill -9 <PID>` استفاده کنید و یا اگر برنامه خود شما همچنان در حال اجرا در پس‌زمینه است می‌توانید این مشکل را با اجرای دستور `pskill -9 httpserver` برطرف کنید. اگر این دستور کار نکرد می‌توانید یک پورت دیگری را با اجرای `httpserver --files files/ --port 8001` مشخص کنید یا ماشین مجازی را با `vagrant reload` مجدداً بارگذاری کنید.

۲.۴.۴ Failed to bind on socket: Permission denied

اگر از پورتی با شماره کمتر از ۱۰۲۴ استفاده کنید ممکن است با این خطا مواجه شوید. فقط کاربر `root` می‌تواند این شماره پورت‌ها را استفاده کند، در نتیجه شما باید شماره پورت بزرگتری را انتخاب کنید. (از ۱۰۲۴ تا ۶۵۵۳۶)

۵.۴ وظیفه‌ی شما

۱. برای رسیدگی به درخواست‌های `GET HTTP` برای فایل‌ها `handle_files_request(int fd)` را پیاده‌سازی کنید. این تابع سوکت `fd` را که از گام ۳ توضیحات قبل به دست آمده، به عنوان ورودی می‌گیرد.



موارد زیر باید رسیدگی شوند:

- از مقدار آرگومان ورودی `--files` که شامل مسیری که فایل‌ها در آن قرار دارند می‌شود، استفاده کنید. (این مسیر در متغیر سراسری `*server_files_directory` ذخیره شده است)
- اگر مسیر درخواست HTTP، مربوط به یک فایل باشد با پیام `200 OK` و محتوای فایل پاسخ دهید. (برای مثال اگر `/index.html` مورد درخواست باشد و فایلی با نام `index.html` در مسیر فایل‌ها موجود باشد) همچنین باید قادر باشید به درخواست فایل‌هایی که در زیرپوشه‌ی مسیر فایل‌ها قرار دارد نیز رسیدگی کنید.

راهنمایی:

— تعدادی تابع سودمند در `libhttp.h` موجود است! مثال‌هایی از طریق استفاده و مستندات آن را در پیوست می‌توانید بیابید.

— مطمئن شوید که سرآیند `Content-Type HTTP` را به درستی مقداردهی کرده اید. تابعی سودمند برای این کار در `libhttp.h` موجود است که نوع `MIME` فایل را برمی‌گرداند. (این تنها فایل سرآیندی است که نیاز دارید برای نشان دادن فایل‌ها/سندها استفاده کنید)

— همچنین اطمینان یابید که سرآیند `Content-Length HTTP` را به درستی مقداردهی کرده اید. مقدار این سرآیند برابر اندازه‌ی بدنه‌ی پاسخ `HTTP` به بایت خواهد بود. برای مثال

`Content-Length: 7810`^۷

— مسیرهای درخواست `HTTP` همیشه با `/` آغاز می‌شوند؛ حتی اگر صفحه‌ی اصلی مورد درخواست باشد. (برای مثال برای `http://ce.sharif.edu/` مسیر درخواست `/` خواهد بود)

- اگر درخواست مربوط به یک مسیر بود و این مسیر شامل فایل `index.html` باشد، با یک پیام `200 OK` و محتوای فایل `index.html` پاسخ دهید. (حواستان باشد که فرض نکنید مسیر درخواست‌ها همیشه با `/` خاتمه می‌یابند)

— برای فرق گذاشتن بین فایل‌ها و مسیرها احتمالاً از تابع `stash()` و ماکروهای `S_ISDIR` یا `S_ISREG` استفاده خواهید کرد.

— نیازی نیست به اشیاء دیگر فایل سیستم غیر از فایل‌ها و مسیرها رسیدگی کنید. (برای مثال نیازی نیست به لینک‌های سمبلیک، پایپ‌ها و فایل‌های خاص رسیدگی کنید)

— سعی کنید بخش‌هایی از کدتان را که زیاد تکرار می‌شوند را حتماً به صورت تابع دربیابید تا راحت‌تر خطایابی شود.

- اگر درخواست مربوط به مسیری بود که شامل فایل `index.html` نیست، با یک صفحه `HTML` شامل لینک به فرزندان مستقیم این مسیر (مانند `ls -1`) و پدر آن پاسخ دهید. (برای مثال لینک به

^۷ برای تست کردن کارگزار خود می‌توانید پرونده مورد درخواست را با دستور `wc -c` لینوکس بررسی کنید و اطمینان حاصل کنید همین عدد برای سرآیند `Content-Length` فرستاده می‌شود.

پدرش به شکل `Parent directory` در می‌آیند.)
راهنمایی:

- برای لیست کردن محتوای یک مسیر توابع `opendir()` و `readdir()` مفیدند.
 - مسیرهای لینک‌ها می‌توانند مطلق یا نسبی باشند. مثل اینکه دستورات `cd usr/` و `cd /usr/` دو کار متفاوت انجام می‌دهند
 - نیازی نیست نگران / های اضافی در لینک‌هایتان باشید. (برای مثال `//files///a.jpg` مشکلی ندارد) هم فایل سیستم و هم مرورگر این را تحمل می‌کنند.
 - فراموش نکنید که سرآیند Content-Type را مقداردهی کنید.
 - در غیراین صورت با پیغام `404 Not Found` پاسخ دهید. (بدنه‌ی HTTP اختیاری است) بسیاری از چیزها ممکن است در درخواست HTTP به خطا بیانجامند ولی ما فقط انتظار داریم که از دستور خطای `404 Not Found` برای فایل ناموجود پشتیبانی کنید.
 - در حالت ارائه‌کردن فایل، لازم است تنها یک درخواست یا پاسخ را به ازای هر اتصال، پشتیبانی کنید. نیازی نیست اتصال `keep-alive` یا `pipelining` را برای این قسمت پیاده‌سازی کنید.
۲. یک استخر ریسه به اندازه‌ی ثابت پیاده‌سازی کنید که به درخواست‌های کارخواه‌ها به طور همزمان رسیدگی کند.

- برای این کار از کتابخانه‌ی `pthread` استفاده کنید.
- استخر ریسه شما باید قادر باشد به دقت و نه بیشتر `--num-threads` کارخواه به طور همزمان رسیدگی کند. توجه کنید که ما معمولا از `1 + --num-threads` ریسه در برنامه‌مان استفاده می‌کنیم: ریسه اصلی مسئول پذیرفتن `accept()` اتصالات کارخواه‌ها در یک حلقه بی‌نهایت و توزیع درخواست‌ها به استخر ریسه‌هاست تا ریسه‌های دیگر به آن‌ها رسیدگی کنند.
- با مشاهده‌ی توابع داخل `wq.c/h` کار خود را آغاز کنید.
- ریسه اصلی (ریسه‌ای که شما برنامه `httpserver` را با آن شروع می‌کنید) باید هنگام پذیرفتن یک اتصال جدید در سوکت، توصیف‌گر فایل^۸ آن سوکت را در صف `work_queue` (که در ابتدای `httpserver.c` و نیز در `wq.c/h` تعریف شده) به کمک دستور `wq_push()` وارد کند.
- سپس، ریسه‌های موجود در استخر ریسه‌ها با استفاده از دستور `wq_pop()` به توصیف‌گر فایل سوکت کارخواه رسیدگی می‌کنند.
- بیشتر عملکرد صف‌کارها^۹ در `wq.c` پیاده‌سازی شده است. اما اسکلت پیاده‌سازی `wq_pop` بدون انسداد است (در حالی که باید این ویژگی را داشته‌باشد) و `wq_pop()` و `wq_push()`

^۸File Descriptor

^۹Work Queue

هیچ‌کدام Thread Safe نیستند. شما باید این مشکل را برطرف کنید.

- علاوه بر پیاده سازی صف کارهای مسدودشونده شما نیاز دارید به تعداد `--num-threads` ریشه‌ی جدید بسازید که در یک حلقه، کارهای زیر را انجام می‌دهند:

– برای توصیفگر فایل بعدی کارخواه، فراخوانی های مسدود شونده ای به `wq_pop` انجام دهد.

– بعد از `pop` کردن موفقیت آمیز یک توصیفگر فایل کارخواهی که باید خدمت رسانی شود، `handler request` مناسب را برای رسیدگی به درخواست کارخواه فراخوانی کنید.

راهنمایی‌ها:

– صفحه‌ی مستندات man برای همگام‌سازی را با دستور زیر بگیرید:

```
$ sudo apt-get install glibc-doc
```

– برای `pthread_cond_init` و `pthread_mutex_init` صفحات man را بخوانید. (یا از Google-fu استفاده کنید) شما به هر دوی اینها نیاز خواهید داشت.

۳. تابع `handle_proxy_request(int fd)` را برای پروکسی کردن درخواست‌های HTTP به کارگزار

HTTP دیگر پیاده‌سازی کنید. در حال حاضر برایتان کد تنظیم اتصالات را آماده کرده‌ایم. شما باید آن را بخوانید و بفهمید اما نیازی نیست آن را تغییر دهید. به صورت مختصر این‌ها کارهایی‌ست که انجام داده ایم:

- از مقدار آرگومان `--proxy` که شامل آدرس و شماره پورت کارگزار HTTP بالادست است استفاده کردیم. (این دو مقدار در متغیرهای سراسری `server_proxy_hostname` و `int server_proxy_port` ذخیره شده‌اند)

- یک جستجوی DNS برای `server_proxy_hostname` انجام دادیم تا آدرس IP آن را بیابیم. (به تابع `gethostbyname2()` نگاه بیندازید)

- یک سوکت با آدرس به دست آمده در قسمت قبل ساختیم. توابع `socket()` و `connect()` را چک کنید.

- از تابع `htons()` برای تنظیم پورت سوکت استفاده شده‌است. (اعداد در حافظه به صورت little-endian ذخیره می‌شوند ولی در شبکه به صورت big-endian مورد انتظار می‌باشد) همچنین توجه کنید که HTTP یک پروتکل SOCK_STREAM است.

حال به قسمت شما می‌رسیم! در زیر چیزهایی که شما نیاز دارید به آن‌ها توجه کنید آمده است:

- برای داده‌ی جدید روی هر دو سوکت‌ها منتظر بمانید. (هم `fd` کارخواه HTTP و هم `fd` کارگزار HTTP بالادست) وقتی داده رسید شما باید سریعاً آن را از داخل یک بافر بخوانید و سپس آن را روی سوکت دیگر بنویسید. باید یک ارتباط دو طرفه بین کارخواه HTTP و کارگزار HTTP بالادست برقرار

سازید. پروکسی شما باید از چندین درخواست/پاسخ پشتیبانی کند.
راهنمایی:

– این کار از نوشتن داخل یک فایل یا خواندن از stdin سخت تر است، زیرا شما نمی دانید کدام طرف جریان دو طرف ابتدا داده را می نویسد یا داده ی بیشتری می خواهند بعد از دریافت پاسخ بفرستند. در حالت پروکسی، شما با این مواجه می شوید که برخلاف کارگزار HTTP تان که فقط نیاز دارد از یک درخواست/پاسخ به ازای هر ارتباط پشتیبانی کند، چندین درخواست/پاسخ روی ارتباط یکسان فرستاده می شوند.

– برای این قسمت نیز نیاز دارید از pthread استفاده کنید. در نظر بگیرید که از دو ریسره برای تسهیل کردن ارتباط دو طرفه استفاده کنید؛ یکی از A به B و دیگری از B به A. تا زمانی که پیاده سازی شما دقیقاً به --num-thread کارخواه خدمت رسانی کند، مشکلی ندارد اگر از چندین ریسره برای رسیدگی به یک درخواست پروکسی کارخواه استفاده کنید.

– نیاز دارید که از client_socket_fd استفاده کنید.

– از توابع select()، fcntl() یا شبیه آن استفاده نکنید. این روش می تواند گیج کننده باشد.

• اگر یکی از سوکت ها بسته شد، اتصال، دیگر برقرار نمی ماند. در نتیجه شما باید سوکت دیگر را ببندید و از پروسه ی فرزند خارج شوید.

۶.۴ . طریقه ی ارسال

برای ارسال و فرستادن برای نمره دهی ابتدا تغییرات خود را مطابق زیر کامیت کنید:

```
1 git push personal master
```

سپس می توانید نمره ی خود را در فایل grade.txt دریافت کنید.

۷.۴ . تحویل دادنی ها

تنها تحویل دادنی این تمرین یک httpserver است که می بایست قابلیت های مطرح شده در قسمت های قبل را فراهم کرده باشد. ۳ قابلیت مطرح شده را می بایست به صورت مجزا commit کرده باشید و خطاهایی که با آنها مواجه می شوید را به صورت issue مشخص کنید.

۵. مرجع توابع http-lib

ما تعدادی تابع سودمند فراهم کردیم که راحت‌تر بتوانید با جزئیات پروتکل HTTP کنار بیایید. آن‌ها در فایل‌های `http-lib.c` و `libhttp.h` موجودند. این توابع بخش کوچکی از پروتکل HTTP را پیاده می‌کنند اما برای این تمرین کافی هستند.

۱.۵. مثالی از چگونگی استفاده

خواندن یک درخواست HTTP از سوکت `fd` فقط به فراخوانی یک تابع نیازمند است.

```
1 // Returns NULL if an error was encountered.
2 struct http_request *request = http_request_parse(fd);
```

فرستادن یک پاسخ HTTP یک فرایند چند مرحله‌ایست. اول باید خط وضعیت HTTP با استفاده از `http_start_response()` فرستاده شود. سپس می‌توانید هر تعداد سرآیند را با `http_send_header()` بفرستید. بعد از اینکه تمامی سرآیندها فرستاده شدند، باید `http_end_headers()` فراخوانده شود (حتی اگر هیچ سرآیندی فرستاده نشده باشد). در آخر می‌توان از `http_send_string()` (برای رشته‌های `null-terminated` در C) یا `http_send_data()` (برای داده‌ی دودویی) برای ارسال داده استفاده کرد.

```
1 http_start_response(fd, 200);
2 http_send_header(fd, "Content-type", http_get_mime_type("index.html"));
3 http_send_header(fd, "Server", "\r{httpserver/1.0}");
4 http_end_headers(fd);
5 http_send_string(fd, "<html><body><a href='/'>Home</a></body></html>");
6 http_send_data(fd, "<html><body><a href='/'>Home</a></body></html>", 47);
7 close(fd);
```

۲.۵. شیء درخواست

یک اشاره‌گر به `http_request` struct توسط `http_request_parse` بازگردانده می‌شود. این struct دو عضو را شامل می‌شود:

```
1 struct http_request {
2     char *method;
3     char *path;
4 };
```

۳.۵. توابع

- `struct http_request *http_request_parse(int fd)`
یک اشاره‌گر به `struct http_request` برمی‌گرداند که شامل متد HTTP و مسیری که درخواست از سوکت خوانده می‌شود است. در صورت معتبر نبود درخواست این تابع `NULL` برمی‌گرداند. این تابع تا زمانی که داده‌ی `fd` در دسترس باشد مسدود می‌شود.
- `http_start_response(int fd, int status_code)`
خط وضعیت HTTP را در `fd` می‌نویسد تا پاسخ HTTP شروع شود. برای مثال هنگامی که `status_code` برابر ۲۰۰ است، تابع `http/1.0 200 OK\r\n` را تولید می‌کند.
- `void http_send_header(int fd, char *key, char *value)`
سرآیند پاسخ HTTP را در `fd` می‌نویسد. برای مثال اگر کلید برابر "Content-Type" و مقدارش برابر "text/html" باشد این تابع `Content-Type: text/html\r\n` را می‌نویسد.
- `void http_end_headers(int fd)`
CRLF (`\r\n`) را در `fd` می‌نویسد که نشان‌دهنده‌ی پایان سرآیندهای پاسخ HTTP است.
- `void http_send_string(int fd, char *data)`
نام مستعاری برای `http_send_data(fd, data, strlen(data))` است.
- `void http_send_data(int fd, char *data, size_t size)`
`data` را در `fd` می‌نویسد. اگر طولانی‌تر از آن بود که یکجا نوشته شود، تابع `write()` را در یک حلقه فرامی‌خواند تا در هربار فراخوانی حلقه قسمتی از داده نوشته شود.
- `char *http_get_mime_type(char *file_name)`
یک رشته برابر مقدار صحیح Content-Type بر اساس فایل `file_name` را برمی‌گرداند.