به نام خدا

درس سیستم‌های عامل

نیم‌سال دوم ۹۸-۹۹

دانشکدهٔ مهندسی کامپیوتر

دانشگاه صنعتی شریف

مدرس **مهدی خزّازی**

تمرین **دوم فردی**

موضوع **کارگزار پروتکل انتقال اَبَرمتن**

موعد تحویل **ساعت ۲۳:۵۹ دوشنبه ۲۵ فروردین ۱۳۹۹**

۱ مقدمه

امروزه پروتکل انتقال اَبَرمتن[۱] رایج‌ترین پروتکل مورد استفاده در لایه‌ی کاربرد[۲] در سطح اینترنت است. مانند بسیاری از دیگر پروتکل‌های شبکه، این پروتکل هم از مدل کارخواه[۳]-کارگزار[۴] استفاده می‌کند. کارخواه در این پروتکل، یک اتصال شبکه به یک کارگزار ایجاد کرده و سپس یک پیام درخواست[۵] HTTP می‌فرستد. سپس کارگزار با یک پیام پاسخ[۶] که معمولاً شامل منابع خواسته شده توسط کارخواه، اعم از متن، پرونده و ... است به این درخواست جواب می‌دهد.
در این تمرین از شما می‌خواهیم یک کارگزار پروتکل انتقال ابرمتن پیاده‌سازی کنید که بتواند به درخواست‌های از نوع GET در این پروتکل پاسخ دهد. به این منظور کد شما باید سرآیندهای جواب[۷]، کدهای خطا، ساخت لیست پوشه‌ها[۸] با HTML و ساخت پیشکار[۹] HTTP را پیاده‌سازی کند.

۱.۱ راه‌اندازی مقدمات

به ماشین مجازی خود وارد شده و همانند تمرین‌های قبل، یک پوشه متناسب با این تمرین ایجاد کرده و موارد مرتبط با این تمرین را در آن قرار دهید.
همچنین شالوده[۱۰] کد شروع این تمرین را می‌توانید از مخزن مربوط به جزوات درس دریافت کنید.

۲.۱ نحوه تنظیم

ماشین مجازی Vagrant شما به‌گونه‌ای تعبیه شده است که از طریق سیستم‌عامل میزبان خود می‌توانید به شبکه‌ی آن متصل شوید. آدرس آی‌پی[۱۱] ماشین مجازی `192.168.162.162` است.
شما باید بتوانید دستور `ping 192.168.162.162` را از طریق سیستم‌عامل میزبان خود اجرا کرده و پاسخ مناسب را از ماشین مجازی دریافت کنید. اگر این اتفاق نیفتاد، می‌توانید تنظیمات مربوط به بازارسالی درگاه[۱۲] را در ماشین مجازی خود انجام دهید. (برای اطلاعات بیشتر اینجا را ببینید)

۲ پیش‌زمینه

۱.۲ ساختار یک درخواست HTTP

قالب یک درخواست HTTP به صورت زیر است:

- یک خط درخواست HTTP (شامل روش[۱۳]، یک پرسمان[۱۴] به صورت رشته و نسخه[۱۵] پروتکل)
- صفر یا تعداد بیشتری خط از سرآیندهای HTTP
- یک خط خالی (شامل فقط یک نویسه CRLF)

1) Hypertext Transport Protocol - HTTP
2) Application Layer
3) Client
4) Server
5) Request Message
6) Response Message
7) Response Headers
8) Directory Listings
9) Proxy
10) Skeleton
11) IP
12) Port Forwarding
13) Method
14) Query
15) Version

خط آخر یک درخواست، فقط یک نویسه CRLF است که به صورت یک **\n** و **\r** چسبیده به‌هم در زبان C نمایش داده می‌شود.
در زیر یک نمونه از درخواست HTTP که توسط مرورگر Chrome به یک کارگزار وب HTTP که در حال اجرا بر روی درگاه **8000** از سرور محلی[۱۶] (**127.0.0.1**) است را می‌بینید:

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

خطوط سرآیند، اطلاعاتی در مورد درخواست را تعیین می‌کنند[۱۷]. دو نمونه از آنها را در زیر ببینید:

- **Host**: بخشی از آدرس URL که نام میزبان را تعیین می‌کند، مشخص می‌کند (به عنوان مثال ce.sharif.edu).
- **User-Agent**: نوع برنامه‌ی کارخواه را مشخص می‌کند و به شکل **Program-name/x.xx** است که بخش دوم آن، نسخه‌ی برنامه را تعیین می‌کند.

۲.۲ ساختار یک پاسخ HTTP

قالب یک پاسخ HTTP به صورت زیر است:

- یک خط وضعیت پاسخ (شامل نسخه‌ی پروتکل، کد وضعیت و توضیحی برای کد وضعیت)
- صفر یا بیشتر خط از سرآیندها
- یک خط خالی (شامل فقط یک نویسه CRLF)
- محتوای درخواست‌شده توسط پیام درخواست

در زیر یک نمونه از پیام پاسخ HTTP با کد وضعیت **200** که یک پرونده‌ی HTML به آن ضمیمه شده، آمده است:

```
HTTP/1.0 200 OK\r\n
Content-Type: text/html\r\n
Content-Length: 128\r\n
\r\n
<html>\n
<body>\n
<h1>Hello World</h1>\n
<p>\n
Let's see this works\n
</p>\n
</body>\n
</html>\n
```

کدهای رایج وضعیت، **HTTP/1.0 200 OK** ، **HTTP/1.0 404 Not Found** و ... است.
کد وضعیت یک عدد سه‌رقمی است که رقم اول آن دسته‌ی وضعیت را مشخص می‌کند:

- **1xx** نشان‌دهنده فقط یک‌سری اطلاعات
- **2xx** نشان‌دهنده موفقیت
- **3xx** کارخواه را به یک URL دیگر انتقال می‌دهد.

16) localhost

۱۷) برای درک بیشتر، حالت **developer view** را در مرورگر خود فعال کرده و به سرآیندهای ارسالی هنگام درخواست یک وب‌گاه توجه کنید.

- **4xx** نشان‌دهنده خطا در سمت کارخواه
- **5xx** نشان‌دهنده خطا در سمت کارگزار

خط‌های سرآیند اطلاعاتی در خصوص پاسخ مشخص می‌کنند. در زیر دو نمونه از آنها را می‌توانید ببینید:

- **Content-Type**: نوع MIME داده متصل به پاسخ، مانند `text/html` و `text/plain` را مشخص می‌کند.
- **Content-Lenght**: تعداد بایت موجود در بدنه‌ی پاسخ را مشخص می‌کند.

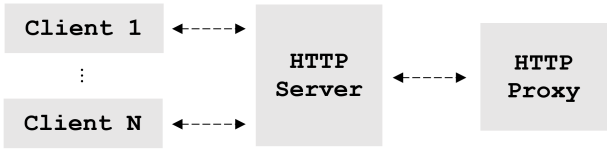
۳ تمرین

۱.۳ طرح کلی از کارگزار وب HTTP

از دید شبکه، کارگزار وب شما باید موارد زیر را پیاده کرده باشد:

۱. یک سوکت[18] بسازد که بر روی یک درگاه گوش می‌کند.
۲. تا اتصال یک کارخواه به این درگاه، صبر کند.
۳. کارخواه را پذیرفته و یک اتصال جدید سوکت فراهم کند.
۴. با خواندن از روی این سوکت، درخواست HTTP را پردازش کند.
۵. با توجه به ورودی‌های برنامه، یکی از دو کار زیر را انجام دهد:

- یک پرونده از سامانه پرونده‌های محلی را به درخواست ارائه کرده یا پیغام `404 Not Found` را برگرداند
- به عنوان یک پیشکار بین این درخواست و یک کارگزار HTTP عمل کند



شکل ۱: هنگام استفاده از یک پیشکار، کارگزار درخواست‌ها را به یک کارگزار بیرونی (پیشکار) فرستاده و سپس پاسخ را گرفته و به سمت کارخواه برمی‌گرداند.

کارگزار در هر لحظه یا در نقش پیشکار عمل می‌کند و یا در نقش ارائه‌دهنده‌ی پرونده و نمی‌تواند همزمان جفت این کارها را انجام دهد.

۶. سرآیندهای مناسب برای پاسخ HTTP را به همراه پرونده /سند مورد درخواست به سمت کارخواه برگرداند (یا یک پیغام خطا بفرستد).

شالوده داده شده به شما گام‌های ۱ تا ۴ را پیاده‌سازی کرده است. شما باید گام‌های ۵ و ۶ و یک استخر ریسه[19] را برای پشتیبانی از چند درخواست همزمان پیاده‌سازی کنید. `httplib.c/h` به شما برای گام‌های ۵ و ۶ و `wq.c/h` برای استخر ریسه کمک خواهد کرد.

۲.۳ استفاده از `./httpserver`

در زیر چگونگی استفاده از `httpserver` آمده ست. برای راحتی شما، کد پردازش ورودی‌ها از قبل پیاده‌سازی شده است:

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

18) Socket
19) Thread Pool

گزینه‌های موجود به شرح زیر است:

- **--files** مسیر یک پوشه را می‌گیرد تا پرونده‌های آن را ارائه کند. شما باید پرونده‌ها 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 پیشکار انتخاب می‌کند. آدرس وارد شده به‌عنوان پیشکار می‌تواند یک شماره درگاه هم بعد از نویسه `:` داشته باشد (برای مثال `ce.sharif.edu:80` - شماره درگاه پیش‌فرض **80** است).
- **--port** درگاهی که کارگزار روی آن برای اتصالات پیش‌رو گوش می‌دهد را مشخص می‌کند. در هر دو حالت پیشکار و پرونده از این گزینه استفاده می‌شود.
- **--num-threads** تعداد ریسه‌های استخر ریسه را مشخص می‌کند. این ورودی در ابتدا بدون استفاده است و تصمیم استفاده کردن یا نکردن از آن برعهده شماست.

شما نباید همزمان از هر دو گزینهٔ **--files** و **--proxy** استفاده کنید وگرنه گزینه دوم، اولی را بازنویسی می‌کند. گزینه‌ی **--proxy** همزمان یک آدرس آی‌پی هم می‌گیرد.
مشکلی ندارد اگر کاکرد تک ریسه‌ای را حذف و استفاده از گزینه **--num-threads** را اجباری کنید.
اگر می‌خواهید از شماره درگاهی بین ۰ تا ۱۰۲۳ استفاده کنید، باید برنامه را به دسترسی کاربر ریشه اجرا کنید. چون این درگاه‌ها از پیش رزرو شده‌اند و تنها کاربر ریشه می‌تواند از آنها استفاده کند. به این منظور باید در ابتدای دستور خود عبارت **sudo** را قرار دهید. مثلا: `sudo ./httpserver --files files/` .

۳.۳ دسترسی به کارگزار HTTP

با رفتن به آدرس `http://192.168.162.162:8000` می‌توانید چک کنید که کارگزار درست کار می‌کند.
یا می‌توانید درخواست‌های HTTP خود را با برنامه **curl** که بر روی ماشین مجازیتان نصب شده است بفرستید. مثالی از نحوه استفاده این ابزار:

```
$ curl -v http://192.168.162.162:8000/
$ curl -v http://192.168.162.162:8000/index.html
$ curl -v http://192.168.162.162:8000/path/to/file
```

همچنین می‌توانید مستقیماً یک ارتباط بر روی سوکت شبکه با استفاده از نت‌کت[20] (**nc**) ایجاد کرده و سپس درخواست‌های HTTP خود را وارد کنید یا از یک پرونده آنها را بخوانید:

```
$ nc -v 192.168.162.162 8000
Connection to 192.168.162.162 8000 port [tcp/*] succeeded!
(Now, type out your HTTP request here.)
```

۴.۳ پیغام‌های خطای رایج

۱.۴.۳ Failed to bind on socket: Address already in use

این پیام به این معناست که شما یک کارگزار HTTP دیگر در حال اجرا در پس‌زمینه دارید. به عبارت دیگر، برنامهٔ دیگری در حال استفاده از درگاه موردنظر شماست. این می‌تواند زمانی اتفاق بیفتد که برنامهٔ شما بخواهد پردازه‌هایی که سوکت را در اختیار دارند را مورد استفاده قرار دهد یا اینکه از ماشین مجازی خود خارج شده اما کارگزار HTTP خود را متوقف نکرده باشید. شما می‌توانید این خطا را با اجرای دستور `pkill -9 httpserver` رفع کنید. اگر این دستور خطا را برطرف نکرد، باید درگاه دیگری را برای اجرای کارگزار خود مشخص کرده (`httpserver --files files/ --port 8001`) یا اینکه ماشین مجازی خود را مجدداً بارگیری کنید (`vagrant reload`).

20) netcat

۲.۴.۳ Failed to bind on socket: Permission denied

اگر شماره درگاهی کمتر از ۱۰۲۴ انتخاب کرده باشید احتمالاً با این خطا روبرو می‌شوید. تنها کاربر ریشه می‌تواند به این درگاه‌ها دسترسی داشته باشد. برای رفع این مشکل باید شماره درگاه بزرگ‌تری انتخاب کنید (۱۰۲۴ تا ۶۵۵۳۵).

۵.۳ وظیفه شما

۱. برای رسیدگی به درخواست‌های GET برای پرونده‌ها، **`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 ورودی **`files--`** که شامل مسیری است که پرونده‌ها در آن قرار دارند، استفاده کنید (این مسیر در متغیر سراسری **`server_files_directory*`** ذخیره شده است).
- اگر مسیر درخواست HTTP ، مربوط به یک پرونده باشد، با پیغام **`200 OK`** و محتوای آن پرونده پاسخ دهید (مثال: **`/index.html`** مورد درخواست باشد و پرونده‌ای با نام `index.html` در مسیر پرونده‌ها موجود باشد). همچنین باید قادر باشید به درخواست پرونده‌هایی که در زیرپوشهٔ مسیر پرونده‌ها قرار دارند هم پاسخ مناسب دهید.
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`** را به درستی مقداردهی کرده‌اید. تابعی سودمند برای این کار در **`libhttp.h`** موجود است که نوع MIME پرونده را بر می‌گرداند (این تنها سرآیندی است که نیاز دارید برای نشان دادن پرونده‌ها/سندها استفاده کنید).
  - همچنین اطمینان یابید که سرآیند **`Content-Length`** را به درستی مقداردهی کرده‌اید. مقدار این سرآیند برابر با اندازهٔ بدنهٔ پاسخ بر حسب بایت خواهد بود. برای مثال **`Content-Length: 7810`** [۲۱].
  - مسیرهای درخواست HTTP همیشه با **`/`** آغاز می‌شوند؛ حتی اگر صفحهٔ 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 پرونده **`index.html`** باشد، با یک پیغام **`200 OK`** و محتوای پرونده **`index.html`** پاسخ دهید (حواستان باشد که فرض نکنید مسیر درخواست‌ها همیشه با **`/`** خاتمه می‌یابند).
  - برای فرق گذاشتن بین پرونده‌ها و مسیرها احتمالاً از تابع **`stash()`** و ماکرو[۲۲]های **`S_ISDIR`** یا **`S_ISREG`** استفاده خواهید کرد.
  - نیازی نیست به اشیاء دیگر فایل‌سیستم[۲۳] غیر از پرونده‌ها و مسیرها رسیدگی کنید.
  - سعی کنید بخش‌هایی از کدتان را که زیاد تکرار می‌شوند حتماً به صورت تابع در بیاورید تا راحت‌تر خطایابی شود.
- اگر درخواست مربوط به مسیری بود که شامل پرونده **`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 پدرش به شکل **`<a href="../">`** **`Parent directory</a>`** در می‌آیند).
راهنمایی:
  - برای لیست‌کردن محتوای یک مسیر توابع **`opendir()`** و **`readdir()`** مفیدند.
  - مسیرهای لینک‌ها می‌توانند مطلق یا نسبی باشند (مثل اینکه دستور **`cd usr/`** و **`cd /usr/`** دو کار متفاوت انجام می‌دهند).
  - نیازی نیست نگران **`/`** های اضافی در لینک‌هایتان باشید (مثال: **`//files///a.jpg`**) هم فایل‌سیستم و هم مرورگر این را تحمل می‌کنند.
  - فراموش نکنید سرآیند **`Content-Type`** را مقداردهی کنید.

---

۲۱) 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.

22) Macro

23) File System

- در غیر این صورت با پیغام `404 Not Found` پاسخ دهید (بدنهٔ HTTP اختیاری است). بسیاری از چیزها ممکن است در درخواست HTTP به خطا بینجامند ولی ما فقط انتظار داریم که از دستور خطای `404 Not Found` برای پرونده ناموجود پشتیبانی کنید.
- در حالت ارائه‌کردن پرونده، لازم است تنها یک درخواست یا پاسخ را به‌ازای هر اتصال، پشتیبانی کنید. نیازی نیست اتصال **keep-alive** یا **pipelining** را برای این قسمت پیاده‌سازی کنید.

۲. یک استخر ریسه به‌اندازهٔ ثابت پیاده‌سازی کنید که به درخواست‌های کارخواه‌ها به طور همزمان رسیدگی کند.

- برای این کار از کتابخانهٔ **pthread** استفاده کنید.
- استخر ریسه شما باید قادر باشد به دقیقاً و نه بیشتر **--num-threads** کارخواه به طور همزمان رسیدگی کنید. توجه کنید که ما معمولاً از **--num-threads + 1** ریسه در برنامه‌مان استفاده می‌کنیم: ریسهٔ اصلی مسئول پذیرفتن (**()accept**) اتصالات کارخواه‌ها در یک حلقه بی‌نهایت و توزیع درخواست‌ها به استخر ریسه‌هاست تا ریسه‌های دیگر به آنها رسیدگی کنند.
- با مشاهده 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 پرونده[۲۴] 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 پرونده سوکت کارخواه رسیدگی می‌کنند.
  - بیشتر عملکرد صف کارها[۲۵] در **wq.c** پیاده‌سازی شده است. اما شالوده پیاده‌سازی **wq_pop** بدون انسداد[۲۶] است (در حالی که باید این ویژگی را داشته باشد) و **()wq_pop** و **()wq_push** هیچکدام امن-ریسه[۲۷] نیستند. شما باید این مشکل را برطرف کنید.
- علاوه بر پیاده‌سازی صف کارهای مسدودشونده، شما نیاز دارید به تعداد **--num-threads** ریسهٔ جدید بسازید که در یک حلقه، کارهای زیر را انجام دهند:
  - **برای توصیف‌گر پرونده بعدی کارخواه، فراخوانی‌های مسدودشونده‌ای به wq_pop انجام دهد.**
  - **بعد از pop کردن موفقیت‌آمیز یک توصیف‌گر پرونده کارخواهی که باید خدمت‌رسانی شود، handler request مناسب را برای رسیدگی به درخواست کارخواه فراخوانی کنید.**

راهنمایی‌ها:

- صفحهٔ 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 هردوی اینها نیاز خواهید داشت.

۳. تابع **handle_proxy_request(int fd)** را برای پیشکاری درخواست‌های HTTP به یک کارگزار HTTP دیگر پیاده‌سازی کنید. درحال 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 کارهایی است که انجام داده‌ایم:

- از مقدار ورودی **--proxy** که شامل آدرس و شماره درگاه کارگزار HTTP بالادست است استفاده کرده‌ایم (این دو مقدار در متغیرهای سراسری `char *server_proxy_hostname` و `int server_proxy_port` ذخیره شده‌اند).
- یک جست‌وجوی DNS برای **server_proxy_hostname** انجام دادیم تا آدرس آی‌پی آن را بیابیم (به تابع **()gethostbyname2** نگاه بیندازید).
- یک سوکت با آدرس به‌دست آمده در قسمت قبل ساختیم. توابع **()socket** و **()connect** را چک کنید.

24) File Descriptor
25) Work Queue
26) Block
27) Thread Safe

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

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

- برای داده‌ی جدید روی هر دو سوکت‌ها منتظر بمانید (هم **fd** کارخواه و هم **fd** کارگزار بالادست). وقتی داده رسید شما باید سریعاً آن را از داخل یک میان‌گیر[۲۸] بخوانید و سپس آن را روی یک سوکت دیگر بنویسید. باید یک ارتباط دوطرفه بین کارخواه و کارگزار بالادست برقرار سازید. پیشکار شما باید از چندین درخواست/پاسخ پشتیبانی کند.
راهنمایی:
  - این کار از نوشتن داخل یک پرونده یا خواندن از **stdin** سخت‌تر است؛ زیرا شما نمی‌دانید کدام طرف جریان دو طرف ابتدا داده را می‌نویسد یا داده‌ی بیشتری می‌خواهند بعد از دریافت پاسخ بفرستند. در حالت پیشکار، شما با این مواجه می‌شوید که برخلاف کارگزارتان که فقط نیاز دارد از یک درخواست/پاسخ به‌ازای هر ارتباط پشتیبانی کند، چندین درخواست/پاسخ روی ارتباط یکسان فرستاده می‌شوند.
  - برای این قسمت نیز نیاز دارید از **pthread** استفاده کنید. در نظر بگیرید که از دو ریسه برای تسهیل کردن ارتباط دوطرفه استفاده کنید؛ یکی از A به B و دیگری از B به A. تا زمانی که پیاده‌سازی شما دقیقا به **--num-thread** کارخواه خدمت‌رسانی کند، مشکلی ندارد اگر از چندین ریسه 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 می‌تواند گیج‌کننده باشد.
- اگر یکی از سوکت‌ها بسته شد، اتصال دیگر برقرار نمی‌ماند. در نتیجه شما باید سوکت دیگر را ببندید و از پروندهٔ فرزند خارج شوید.

۶.۳ طریقه ارسال

برای ارسال و نمره‌دهی ابتدا تغییرات خود را مطابق زیر کامیت کنید:

```
git push personal master
```

سپس می‌توانید نمره‌ی خود را در پرونده **grade.txt** دریافت کنید.

۷.۳ تحویل‌دادنی‌ها

تنها تحویل‌دادنی این تمرین یک **httpserver** است که می‌بایست قابلیت‌های مطرح‌شده در قسمت‌های قبل را فراهم کند. ۴ قابلیت مطرح شده را می‌بایست به صورت مجزا کامیت[۲۹] کرده باشید و خطاهایی که با آنها مواجه می‌شوید را به صورت issue مشخص کنید.

۴ ضمیمه: مرجع توابع httplib

ما تعدادی تابع سودمند فراهم کردیم که راحت‌تر بتوانید با جزئیات پروتکل HTTP کنار بیایید. آنها در پرونده‌های **httplib.c** و **httplib.h** موجودند. این توابع بخش کوچکی از پروتکل را پیاده می‌کنند اما برای این تمرین کافی هستند.

۱.۴ مثالی از چگونگی استفاده

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

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

28) Buffer
29) Commit

فرستادن یک پاسخ HTTP یک فرایند چندمرحله‌ای است. اول باید خط وضعیت HTTP با استفاده از تابع **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 نشده‌باشد). 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.

```
http_start_response(fd, 200);
http_send_header(fd, "Content-Type", http_get_mime_type("index.html"));
http_send_header(fd, "Server", "httpserver/1.0");
http_end_headers(fd);
http_send_string(fd, "<html><body><a href='/'>Home</a></body></html>");
http_send_data(fd, "<html><body><a href='/'>Home</a></body></html>", 47);
close(fd);
```

۲.۴ شیء درخواست

یک اشاره‌گر به ساختار[۳۰] **http_request** توسط **http_request_parse** برگردانده می‌شود. این ساختار از دو عضو زیر تشکیل شده است:

```
struct http_request {
    char *method;
    char *path;
};
```

۳.۴ توابع

- **`struct http_request *http_request_parse(int fd)`**
  یک اشاره‌گر به **`struct http_request`** بر می‌گرداند که شامل روش HTTP و مسیری که درخواست از سوکت خوانده است، می‌شود. در صورت معتبر نبودن درخواست این تابع **NULL** بر می‌گرداند. 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 را در **fd** می‌تویسد که نشان‌دهندهٔ 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** می‌نیوسد. اگر داده طولانی‌تر از آن بود که یک‌جا نوشته شود، تابع **write()** را دی یک حلقه فرا می‌خواند 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 پرونده **file_name** را بر می‌گرداند.

30) Struct

در صورت داشتن هرگونه سوال در رابطه با درس و تمرینات، سوال خود را به لیست ایمیلی درس به آدرس **ce424@lists.sharif.ir** ارسال کنید. همچنین اگر در استفاده از سامانه طرشت به مشکلی برخوردید، آن را از طریق ایمیل **hossein.moghaddas26@student.sharif.edu** مطرح کنید.