به نام خدا

درس سیستم‌های عامل

نیم‌سال دوم ۰۳-۰۲

دانشکدهٔ مهندسی کامپیوتر

دانشگاه صنعتی شریف

مدرس **محمدعلی میرزایی، محدثه میربیگی**

تمرین **دوم عملی**

مسئول تمرین **امیرمهدی کوششی**

موضوع **کارگزار پروتکل انتقال اَبَرمتن**

موعد تحویل **ساعت ۲۳:۵۹ جمعه ۱ دی ۱۴۰۲**

با سپاس از **دکتر مهدی خرازی، محمد حدادیان، مهرانه نجفی و هومان کشوری**

اقتباس شده از **CS162 در بهار ۲۰۲۰ در دانشگاه کالیفرنیا، برکلی**

۱ مقدمه

امروزه پروتکل انتقال اَبَرمتن[۱] رایج‌ترین پروتکل مورد استفاده در لایهٔ 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 سرتیترهای جواب[۷]، کدهای خطا، ساخت لیست پوشه‌ها[۸] با HTML و ساخت پروکسی[۹] HTTP را پیاده‌سازی کند.

۱.۱ راه‌اندازی مقدمات

همانند تمرین قبل، برای این تمرین نیز به یک محیط لینوکسی نیاز دارید. پیشنهاد ما Ubuntu v۲۰ می‌باشد. همچنین فایل‌های مورد نیاز این تمرین در یک فایل زیپ قرار داده شده است.

۲ پیش‌زمینه

۱.۲ ساختار یک درخواست HTTP

قالب یک درخواست HTTP به صورت زیر است:

- یک خط درخواست HTTP (شامل روش[۱۰]، یک پرسمان[۱۱] به صورت رشته و نسخهٔ پروتکل)
- صفر یا تعداد بیشتری خط از سرآیندهای HTTP
- یک خط خالی (شامل فقط یک نویسهٔ CRLF)

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 (`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
```

[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] Method
[11] Query

خطوط سرآیند، اطلاعاتی در مورد درخواست را تعیین می‌کنند[۱۲]. 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 برنامهٔ کارخواه را مشخص می‌کند و به شکل `Program-name/x.xx` است که بخش دوم آن، نسخهٔ برنامه را تعیین می‌کند.

۲.۲ ساختار یک پاسخ HTTP

قالب یک پاسخ HTTP به صورت زیر است:

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

در زیر یک نمونه از پیام پاسخ HTTP با کد وضعیت `200` که یک پروندهٔ 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 سه رقمی است که رقم اول آن دستهٔ وضعیت را مشخص می‌کند:

- **1xx** نشان‌دهندهٔ فقط یک سری اطلاعات
- **2xx** نشان‌دهندهٔ موفقیت
- **3xx** کارخواه را به یک URL دیگر انتقال می‌دهد.
- **4xx** نشان‌دهندهٔ خطا در سمت کارخواه
- **5xx** نشان‌دهندهٔ 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 دادهٔ متصل به پاسخ، مانند `text/html` و `text/plain` را مشخص می‌کند.
- **Content-Lenght**: تعداد بایت موجود در بدنهٔ پاسخ را مشخص می‌کند.

[۱۲] برای درک بیشتر، حالت `developer view` را در مرورگر خود فعال کرده و به سرآیندهای ارسالی هنگام درخواست یک وب‌سایت توجه کنید.

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[۱۳] بسازد که بر روی یک درگاه گوش می‌کند.

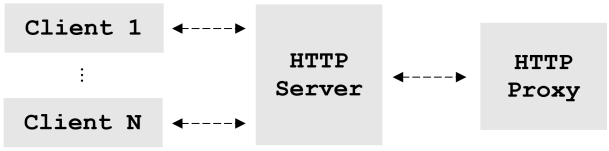
۲. تا اتصال یک کارخواه به این درگاه، صبر کند.

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 پروکسی بین این درخواست و یک کارگزار HTTP عمل کند.



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

کارگزار در هر لحظه یا در نقش پروکسی عمل می‌کند و یا در نقش ارائه‌دهندهٔ 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).

شالودهٔ داده شده به شما گام‌های ۱ تا ۴ را پیاده‌سازی کرده است. شما باید گام‌های ۵ و ۶ و یک استخر ریسه[۱۴] 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 است. 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]
```

گزینه‌های موجود به شرح زیر است:

- **`--files`** مسیر یک پوشه را می‌گیرد تا پرونده‌های آن را ارائه کند. شما باید پرونده‌ها را از پوشهٔ `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 وارد شده به‌عنوان پروکسی می‌تواند یک شماره درگاه هم بعد از نویسهٔ `:` 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 پروکسی و پرونده از این گزینه استفاده می‌شود.

[13] Socket
[14] Thread Pool

- **--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 گزینهٔ دوم، اولی را بازنویسی می‌کند. گزینهٔ **--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 با 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 توجه به اینکه شما در چه محیطی برنامه خود را اجرا می‌کنید، می‌توانید با رفتن به آدرس آن، خروجی خود را ببینید. برای مثال اگر در localhost این برنامه را اجرا می‌کنید، باید بتوانید با رفتن به آدرس http://127.0.0.1 : 8000 بتوانید صفحه موردنظر را ببینید. در غیر اینصورت با رفتن به آدرس شبکه‌ای که کد شما در آن‌جا در حال اجرا است، می‌توانید صفحه موردنظر را ببینید. برای مثال http://192.168.162.162 : 8000
یا می‌توانید درخواست‌های HTTP خود را با برنامه **curl** به کد خود بفرستید. مثالی از نحوهٔ استفادهٔ این ابزار:

```
$ curl -v http://127.0.0.1:8000/
$ curl -v http://127.0.0.1:8000/index.html
$ curl -v http://127.0.0.1:8000/path/to/file
```

همچنین می‌توانید مستقیماً یک ارتباط بر روی سوکت شبکه با استفاده از نت‌کت[15] (**nc**) ایجاد کرده و سپس درخواست‌های HTTP خود را وارد کنید یا از یک پرونده آنها را بخوانید:

```
$ nc -v 127.0.0.1 8000
Connection to 127.0.0.1 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).

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 وظیفهٔ 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:

[15] netcat

- 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 نحوهٔ 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** [۱۶].
  - 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[۱۷]های **S_ISDIR** یا **S_ISREG** استفاده خواهید کرد.
  - نیازی نیست به اشیاء دیگر فایل‌سیستم[۱۸] 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 **Parent directory** 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.
- 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.

[۱۶] 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.

[17] Macro

[18] File System

- 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[19] 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[20] در **wq.c** پیاده‌سازی شده است. اما شالوده پیاده‌سازی **wq_pop** بدون انسداد[21] است (در حالی که باید این ویژگی را داشته باشد) و **wq_pop()** و **wq_push()** هیچکدام امن-ریسه[22] 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 درخواست‌های پروکسی 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.
- 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 دادهٔ 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 بافر بخوانید و سپس آن را روی یک سوکت دیگر بنویسید. باید یک ارتباط دوطرفه بین کارخواه و کارگزار بالادست برقرار سازید. پروکسی 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 دادهٔ بیشتری می‌خواهند بعد از دریافت پاسخ بفرستند. در حالت پروکسی، 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.

[19] File Descriptor
[20] Work Queue
[21] Block
[22] Thread Safe

- 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 (به این مورد توجه ویژه کنید که خیلی از اشکالات دانشجویان در پاس نشدن تست‌ها مربوط به این مورد بوده است).

۶.۳ تحویل‌دادنی‌ها

فایل‌های داده شده در skeleton را به صورت یک فایل زیپ در کوئرا آپلود 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);
```

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، **http_end_headers()** باید فراخوانده شود (حتی اگر هیچ سرآیندی فرستاده نشده باشد). در آخر از **http_send_string** (برای رشته‌های null-terminated در C) یا **http_send_data()** (برای دادهٔ دودویی) برای ارسال داده می‌توان 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[۲۳] **http_request** توسط **http_request_parse** برگردانده می‌شود. این ساختار از دو عضو زیر تشکیل شده است:

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

[23] Struct

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 دادهٔ **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** می‌نویسد. 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.

۵ برای مطالعه بیشتر

۱.۵ epoll

می‌دانیم که در یک server http می‌توان به دو روش به درخواست‌ها پاسخ داد :

- ساخت ریسه
- برنامه نویسی آسنکرون

حال در این برنامه از شما خواسته شده بود به روش اول پاسخ دهید اما نکته قابل توجه این است که ساخت ریسه سربار قابل توجهی برای برنامه شما دارد.
حال برای این که از این سربار کم کنیم و همچنین قابلیت پاسخ‌گویی موازی به درخواست‌ها را داشته باشیم از توابع آسنکرون استفاده می‌کنیم که به برنامه این قابلیت را می‌دهند که بدون ساخت ریسه، عمل موازی سازی را انجام دهد.
چالش ما در استفاده از توابع اسنکرون این است که باید تعداد کانکشن‌ها، ورودی‌ها و خروجی‌ها، در هنگام وصل و قطع شدن مشخص باشند.
epoll یک مکانیزم اطلاع رسانی ورودی خروجی است که در لینوکس به کار می‌رود و عملا عملیات تقسیم‌بندی[۲۴] ورودی خروجی‌ها را انجام می‌دهد.
این مکانیز به این صورت عمل می‌کند که چندین توصیح کننده پرونده را مانیتور می‌کند تا در هر لحظه یکی از آنها قابل استفاده بود از آن استفاده کند.
دلیل این که می‌توان از همچین قابلیتی برای کنترل کارگزار استفاده کرد این است که ارتباط شبکه مانند بسیاری از ویژگی‌های دیگر با پرونده کنترل می‌شود و نیز برای این کار نیاز به تعدادی توصیف‌کننده پرونده وجود دارد .
فراخوان سیستمی epoll به این صورت عمل می‌کند که در صورتی که پردازه‌ای نیاز به کار با یک توصیف‌کننده پرونده داشته باشد، توصیف کننده را در اختیار پردازه قرار می‌دهد. پس از اینکه پردازه توصیف‌کننده را در اختیار گرفت منتظر اتمام کار آن می‌شود.

[24] Multiplexing

حال با این سیستم می‌توان به صورت موازی با تعدادی توصیف‌کننده پرونده کار کرد به صورتی که هر کدام به صورت اسنکرون در حال انجام کار بخصوصی باشند و نیز در صورت اتمام کار نیز با آزاد شدن توصیف‌کننده پرونده بتوان یک کانکشن دیگر برقرار کرد.

Cache ۲.۵

این قسمت نمره اضافی ندارد و صرفا برای علاقه‌مندان این حوزه است.**
در تمرینی که آن را انجام دادیم، یک نکته در نظر گرفته نشد و آن هم این که در دنیای واقعی و به صورت روزمره، امکان دارد هزاران درخواست برای یک کارگزار برود و پاسخگویی به این تعداد درخواست می‌تواند منابع و هزینه بالایی برای کارگزار داشته باشد. نکته قابل توجه این است که در کارگزار واقعی تقریبا ۸۰ درصد درخواست‌ها در یک روز مشابهت دارند پس به صورت منطقی نباید برای پاسخگویی به یک درخواست یکسان که در طی زمان مشخص چندین مرتبه ثبت شده در هر مرتبه از منبع اصلی پاسخ داد بلکه می‌توانیم در قسمتی از کارگزار آنها را کش کنیم و در این صورت اگر درخواست مشابه دیگری بیاید، از کش پاسخ دهیم.
برای این کار کش ما نیاز دارد یک تناظر[۲۵] از یو آر ال به ساختار کلی سایت داشته باشد و همچنین برای اطمینان از صحت زمانی سایت باید یک حد زمانی داشته باشد که پس از طی شدن آن، ورودی کش را پاک کند.
نکته دیگر در ساخت این کارگزار این است که اگر چندین نفر همزمان درخواست گرفتن یک سایت مشخص را بدهند، باید درخواست یکی از آنها بررسی شده و باقی درخواست‌ها منتظر اتمام بررسی دیگر درخواست‌ها بمانند و اگر چند ریسه برای یو آر ال‌های متفاوت درخواست بدهند، هیچ ریسه‌ای نباید متوقف شود.
در پیاده‌سازی این کش متغیرهای شرطی[۲۶] به کار می‌آیند.

[25] Mapping
[26] Condvar