Server Assignment

> This lab is based on the Tiny web server by Bryant and O'Hallaron for *Computer Systems: A Programmer's Perspective*, Third Edition

Due: Wednesday, December 5, 11:59pm

In this lab, you'll modify a web server to implement a concurrent server for managing people and the places that they have visited—something like "red pins" on a map. Part of the server's functionality will involve acting as a client to copy pins from other servers that implement the same protocol.

Getting Started

Start by unpacking `servlab-handout.zip`. The only file you will be modifying and handing in is `"redpin.c"`.

You can build and run the initial server, which is a variant of the book's Tiny web server, with

```
$ make
$ ./redpin ‹port›
```

where ‹*port*› is some number between 1024 and 65535. After starting `redpin`, you can connect to it using a web browser or `curl` on the same machine with

`http://localhost:‹port›/`

The initial server replies to any request with a list of two people: `alice` and `bob`. The server also prints to standard output any query parameters that it received through the request, both in the URL and as POST data.

Server Behavior

Your revised server must keep track (in memory) of any number of people and, for each person, any number of places that they have visited. That's equivalent to keeping track of any number of places and, for each place, any number of people who have visited the place.

Each person and each place is identified by an arbitrary string that does not include a space or newline character. A person or place identity is case-sensitive. A person and a place can each have the same name, but a place is distinct from a person with the same name.

Any number of clients should be able to connect to the server (up to typical load limits) and communicate about any number of people and places. The server starts with an empty set of people and places.

Your server must be highly available as described in Availability and Client Constraints, which means that it is robust against slow or misbehaving clients. This availability requirement will require concurrency in your server implementation. For grading, we will test your server by throwing a mixture of clients—fast and slow, behaving and misbehaving—all at the same time.

Finally, your server must not leak memory as long as all connections are well-behaved (i.e., any client errors are confirmed to bad query parameters). We will run your server under Valgrind to check whether it can detect any leaks.

Your server's output to `stdout` and `stderr` will be ignored, so you can use those streams for logging or debugging as you see fit.

Server Queries

Your server starts out with an empty set of people and places. A `pin` operation adds person-at-place combinations to the server, and an `unpin` operation can remove person-at-place combinations. There is no fixed set of people and places; when a person is mentioned in a `pin` operation, the person is added to the set of people tracked by the server, and similarly for places. When an `unpin` operation removes the last pin for a person or place, then the person or place is no longer tracked by the server.

Your server must support several kinds of HTTP GET/POST requests:

- `/counts` — Returns the number of people currently tracked by the server and the number of places tracked by the server. Each of those two numbers is reported on a newline-terminated line as plain text (i.e., `text/plain; charset=utf-8`).

  For example, if this is the first query to a freshly started server, the result content is equivalent to the C string `"0\n0\n"`.

- `/reset` — Removes all pins (i.e., all people and places) and returns new people and place counts in the same format as for a `counts` query.

  Since `reset` clears all people and places, the result content is always equivalent to the C string `"0\n0\n"`.

- `/people` or `/people?place=`‹*place*› — Returns

  - all of the people currently tracked by the server if ‹*place*› is not specified in the query; or
  - all people who have visited ‹*place*› if ‹*place*› is specified.

  Each person is reported on a separate newline-terminated line as plain text (i.e., `text/plain; charset=utf-8`), and the people returned by the server can be in any order.

  The result is empty if no people are tracked or if ‹*place*› is specified and is not tracked by the server (which means that no people have visited that ‹*place*›).

- `/places` or `/places?person=`‹*person*› — Returns

  - all of the places currently tracked by the server if ‹*person*› is not specified in the query; or
  - all places that have visited by ‹*person*› if ‹*person*› is specified.

Each place is reported on a separate newline-terminated line as plain text (i.e., `text/plain; charset=utf-8`), and the places returned by the server can be in any order.

The result is empty if no places are tracked or if ‹*person*› is specified and is not tracked by the server (which means that the ‹*person*› hasn't visited any places).

- `/pin?people=`‹*people*›`&places=`‹*places*› — Adds a pin for each person in ‹*people*› at each place in ‹*places*›. The ‹*people*› list can be a single person or multiple newline-separated person names, optionally ending with a newline character. The ‹*places*› list can be a single place or multiple newline-separated place names, optionally ending with a newline character.

  Every person in ‹*people*› is recorded as visiting every place in ‹*places*›. If any person in ‹*people*› is already recorded as having visited any place in ‹*places*›, then the `pin` request does not create a redundant entry for that person–place combination.

  The result of the query is new people and place counts in the same format as for a `counts` query.

  Note that a long list of people in ‹*people*› or places in ‹*places*› will require that the `people` part or `places` part of the query is sent as POST data, instead of supplied as part of the URL. The starting code already merges POST query data with URL query arguments for you, so the idea of using POST is only relevant if you want to construct your own extra large tests.

- `/unpin?people=`‹*people*›`&places=`‹*places*› — Removes the pin for each person in ‹*people*› at each place in ‹*places*›. The ‹*people*› list can be a single person or multiple newline-separated person names, optionally ending with a newline character. The ‹*places*› list can be a single place or multiple newline-separated place names, optionally ending with a newline character.

  Every person in ‹*people*› is combined with every place in ‹*places*› to remove the corresponding person–place entry from the server. If any person in ‹*people*› is not recorded as having visited any place in ‹*places*›, then the `unpin` request has no effect for that person–place combination.

  The result of the query is new people and place counts in the same format as for a `counts` query.

  Just like `pin` requests, long ‹*people*› and ‹*places*› lists may be supplied as POST data instead of being part of the URL.

- `/copy?person=`‹*person*›`&as=`‹*person*›`&host=`‹*host*›`&port=`‹*port*› — Contacts a server running on ‹*host*› at ‹*port*› to get all of the places visited by the `person` argument's ‹*person*›, and adds them as places visited by the `as` argument's ‹*person*›.

  The given ‹*host*› and ‹*port*› should refer to some `redpin` server. The server that receives the `copy` request should itself not fail if the server at ‹*host*› and ‹*port*› fails to respond, and it should only import pins if the server reports success with status code 200. As long as the HTTP status code from the server at ‹*host*› and ‹*port*› is 200, then the importing server can assume that the place list is well-formed.

  The given ‹*host*› and ‹*port*› can refer to same server that received the `copy` request. In that case, as long as no other client is accessing the server at the same time, the places of the `person` argument on the same server will be added as places for the `as` argument.

If the ‹*host*› and ‹*port*› server is different from the one that receives the `copy` request, then the ‹*host*› and ‹*port*› server's pin registry is queried but **not changed** by the `copy` request. Pins change only on the server that receives the `copy` request.

A server that receives a `copy` request should not return a result to the client until the copy succeeds. The result sent back to a client that makes a `copy` request is the new people and place counts in the same format as for a `counts` query.

- `/copy?place=`‹*place*›`&as=`‹*place*›`&host=`‹*host*›`&port=`‹*port*› — Contacts a server running on ‹*host*› at ‹*port*› to get all of the people who have visited the `place` argument's ‹*place*›, and adds them as having visited the `as` argument's ‹*place*›.

  The details and result of this query are otherwise the same as for `copy` with a `person` argument instead of a `place` argument.

In the queries described above, arguments to ‹*person*›, ‹*place*›, etc., should all be interpreted as UTF-8 text with the usual encoding within URLs. Basically, that means you don't have to worry about encodings as long as you use the provided parsing functions. The order of query parameters should not matter; for example, `places` might be supplied before `people` in a `pin` request, and the server should treat that the same as `people` before `places`.

If a query is unrecognized or missing a required argument, your server should respond with status 400 (and not crash). If a query has extra arguments that are not listed above, your server can ignore them. If a query has the same argument multiple times, your server can use any one of them; the given header-parsing code will effectively pick one for you.

As an example, suppose that your server has just started running on `localhost` at port `8090`:

```
$ curl "http://localhost:8090/counts"
0
0
$ curl "http://localhost:8090/pin?people=alice&places=warnock"
1
1
$ curl "http://localhost:8090/places?person=alice"
warnock
$ curl "http://localhost:8090/pin?people=bob%0Acarol&places=warnock%0Amerrill"
3
2
$ curl "http://localhost:8090/people"
carol
bob
alice
$ curl "http://localhost:8090/places"
warnock
merrill
$ curl "http://localhost:8090/places&person=bob"
warnock
merrill
$ curl "http://localhost:8090/places&person=carol"
warnock
merrill
$ curl "http://localhost:8090/unpin?people=alice%0Abob&places=warnock"
2
2
$ curl "http://localhost:8090/people"
carol
bob
```

```
$ curl "http://localhost:8090/places?person=bob"
merrill
$ curl "http://localhost:8090/places?person=carol"
warnock
merrill
$ curl "http://localhost:8090/copy?host=localhost&port=8090&person=carol&as=alice"
3
2
$ curl "http://localhost:8090/people"
carol
bob
alice
$ curl "http://localhost:8090/places?person=alic"
$ curl "http://localhost:8090/places?person=alice"
warnock
merrill
$ curl "http://localhost:8090/places?person=dan"
$ curl "http://localhost:8090/people?place=warnock"
carol
alice
$ curl "http://localhost:8090/people?place=union"
$
```

In the responses above, the order of people or places in a response can be different than the shown order.

Availability and Client Constraints

You must make essentially no assumptions about clients of your server. It's ok to limit the initial request line and header lines to `MAXLINE` characters. Otherwise, as long as clients follow the HTTP protocol and as long as machine resources are not exhausted (including memory or allowed TCP connections), your server should continue to respond to new requests in a timely manner. Your server must not run out of resources as a result of failing to free unneeded memory or close finished connections.

You should make a good effort to report errors to clients, but it's also ok to just drop a client that makes an invalid request. It's ok if communication errors cause the error-checking `csapp.[ch]` functions to print errors; the starting server includes `exit_on_error(0)` so that discovered errors are printed and the function returns, instead of exiting the server process.

As long as a client is well behaved, the server should not drop a client's pin addition, even if multiple clients are adding places for the same person list at the same time. For example, if three clients each add 1000 different places for a person concurrently, the result will be a person with 3000 places.

There is no *a priori* limit on the length of user names or time that the server must stay running. If your server runs out of memory because the given data is too large, that's ok. If your server crashes because it didn't expect a user name to have 1 character or to have 1,753 characters, that's not ok. If your server runs out of memory because it has a leak and cannot deal with thousands of sequential requests to access a person's place list, that's also not ok.

We expect that your server exhibits good asymptotic performance. Adding N places for one person should take $O(N)$ time, and adding N people for one place should take $O(N)$ time. Adding N places for M people can take $O(NM)$ time. Removing one person from one place should take $O(1)$ time. Use the dictionary routines that we have provided to store people and

places, and that should provide the right asymptotic performance. Even though we may not directly probe asymptotic performance of your server, we expect large tests to work.

We will send arbitrary people and place names through the query interface to make sure that they are is reported back intact, and we'll try perverse people and place names such as `&` or `"`. The `"more_string.[ch]"` library provides relevant encoding and decoding functions.

Your server is free to support additional queries other than the ones listed in Server Queries.

Support Libraries

In addition to `"csapp.[ch]"`, the starting code in `servlab-handout.zip` includes `"dictionary.[ch]"` and `"more_string.[ch]"`.

The `"dictionary.[ch]"` library provides an implementation of dictionaries with case-sensitive or case-insensitive keys. When you add to the dictionary, the given string key is copied, but the given value pointer is added as-is. When creating a dictionary, you supply a function that is used to destroy values in the dictionary when they are no longer needed. For example, if data values are allocated with `malloc`, supply `free` to be used to destroy data values when `dictionary_free` is called or when the value is removed or replaced with a different one. See `"dictionary.h"` for more information.

The `"more_string.[ch]"` library provides string helpers and functions for some basic parsing and encoding functions. See `"more_string.h"` for details.

The starting code in `servlab-handout.zip` includes some tests:

- The `"basic.rkt"` script exercises all query forms and checks server results. If your `redpin` server is running `localhost` at port `8090`, run `"basic.rkt"` as

  ```
  $ racket basic.rkt localhost 8090
  ```

  The `"basic.rkt"` test involves some randomness, but you can make it deterministic by supplying a seed with `--seed`:

  ```
  $ racket basic.rkt --seed 42 localhost 8090
  ```

  The `"basic.rkt"` script also accepts a `--keep-going` flag to make it keep running if a test fails, but beware that a failing test may leave the server out-of-sync with later tests.

- The `"trouble.rkt"` script helps check how well a server responds to misbehaved clients, and it requires the server to handle concurrent connections. If a server is running on `localhost` at port `8090`, then

  ```
  $ racket trouble.rkt localhost 8090
  ```

  reports its status. If the script doesn't finish in 5-10 seconds, then something has gone wrong.

  The `"trouble.rkt"` script may cause your server to report many connection errors. That's fine, and you may want to redirect output to `/dev/null` when running this test. There's only a problem if the `"trouble.rkt"` script itself reports errors.

- The `"stress.rkt"` script throws lots of concurrent queries at a server and uses the `copy` query. If a fresh server is running on `localhost` at port `8090`, then

  ```
  $ racket stress.rkt localhost 8090
  ```

  runs the stress test.

  The `"stress.rkt"` script assumes that the server state is fresh (i.e., no people or places registered). If your server prints logging information to `stdout` or `stderr`, you'll probably want to redirect it to `/dev/null` when running this test.

Evaluation

We will grade your submission based on the following functionality thresholds:

- Implementing the server enough to handle the `"basic.rkt"` test is worth 60 points.
- Adding concurrency well enough to handle the `"trouble.rkt"` test is worth 80 points.
- Passing the `"stress.rkt"` test while limited via `ulimit -v 20971520` (to disallow extreme leaks) is worth 100 points.

  In `bash`, you can run `redpin` with the memory limit like this:

  ```
  $ (ulimit -v 20971520 ; ./redpin 8090 > /dev/null)
  ```

  The equivalent command in `csh` is

  ```
  % (limit vmemoryuse 20971520 ; ./redpin 8090 > /dev/null)
  ```

- Failing the memory-leak test *deducts* 15 points, independent of the level of completion. We will check for memory leaks by starting your server with

  ```
  $ valgrind ./redpin ‹port›
  ```

  running `"basic.rkt"`, terminating the server with Ctl-C, and checking whether Valgrind reports any memory leaks. As long as the output includes

  ```
  All heap blocks were freed -- no leaks are possible
  ```

  or

  ```
  LEAK SUMMARY:
     definitely lost: 0 bytes in 0 blocks
  ```

  then Valgrind found no leak.

Tips

- Pay attention to the ownership rules for values added to or extracted from a dictionary. See Support Libraries and the comments in `"dictionary.h"`.
- If you use a dictionary to map keys to `NULL` values, then the dictionary effectively implements a set. Note that you can specify `NULL` instead of a value-destroying function

when creating a dictionary to indicate that values do not need to be destroyed.

- Don’t try to store a set of people or places as one string. If you represent a set of places as a dictionary, and if you also map each person to a set of places using a dictionary, then you’ll have a dictionary of dictionaries—which is a good idea.

- People and places are symmetric in `redpin`. Consider using one dictionary that maps people to places plus another dictionary that maps places to people. Although that strategy that keeps all information in two ways, it makes all operations have a suitable asymtoptic complexity.

- Since people and places are symmetric in `redpin`, you may be tempted to copy code that works on people to make it work on places or vice versa. When you do that, remember that you’re copying all the bugs, too. A better strategy is to write one piece of code that is parameterzied over people or places.

- If you get seg faults or if values seem to be changing out from under you, don’t forget to try tools like Valgrind and `gdb`.

- Use functions like `query_encode` to ensure that an unusual user name or user string doesn’t create trouble when embedded in a URL.

- More generally, you don’t want to be in the business of encoding, decoding, or parsing strings. If you find yourself having to parse, encode, or decode a string, check again whether functions in `"more_string.[ch]"` could be used for the job—maybe with a slightly different approach to the communication pattern.

- You’ll need to use `Pthread_create` to make your server handle multiple clients concurrently, but it makes sense to add concurrency as a last step.

- When you do add concurrency, you’ll need to make sure that uses of your people and places tables are properly synchronized. The `"stress.rkt"` script mostly tries to check your server’s synchronization.

- If your server leaks when exercised via `"basic.rkt"`, try using `--leak-check=full` with Valgrind (just like that Valgrind output suggests) to get more information about the source of the leak.