Lab Handout 7: proxy Redux

The lab checkoff sheet can be found right here. There is no lab7 folder, so no need to `git clone` anything this week.

Problem 1: `proxy` Thought Questions

- Some students suggested that your Assignment 7 `proxy` implementation open one persistent connection to the secondary proxy when a secondary proxy is used, and that all requests be forwarded over that one connection. Explain why this would interfere with the second proxy's ability to handle the primary proxy's forwarded requests.
- **Long polling** is a technique where new data may be pushed from server to client by relying on a long-standing HTTP request with a protracted, never-quite-finished HTTP response. Long polling can be used to provide live updates as they happen (e.g. push notifications to your smartphone) or stream large files (e.g. videos via Netflix, youtube, vimeo). Unfortunately, the long polling concept doesn't play well with your `proxy` implementation. Why is that?
- HTTP pipelining is a technique where a client issues multiple HTTP requests over a single persistent connection instead of just one. The server can process the requests in parallel and issue its responses in the same order the requests were received. Relying on your understanding of HTTP, briefly explain why `GET` and `HEAD` requests can be safely pipelined but `POST` requests can't be.

- When implementing `proxy`, we could have relied on multiprocessing instead of multithreading to support concurrent transactions. Very briefly describe one advantage of the multiprocessing approach over the multithreading approach, and briefly describe one disadvantage.

- We interact with socket descriptors more or less the same way we interact with traditional file descriptors. Identify one thing you can't do with socket descriptors that you can do with traditional file descriptors, and briefly explain why not.

- The `createClientSocket` function for the `proxy` assignment has a call to `gethostbyname_r`, which has the prototype listed below. This is a reentrant version that is thread-safe, because the client shares the location of a *locally* allocated `struct` `hostent` via argument 2 where the return value can be placed, thereby circumventing the caller's dependence on shared, statically allocated, global data. Note, however, that the client is expected to pass in a large character buffer (as with a locally declared `char` `buffer[1 << 16]`) and its size via arguments 3 and 4 (e.g. `buffer` and `sizeof(buffer)`). What purpose does this buffer serve?

```
struct hostent {
  char *h_name;        // real canonical host name
  char **h_aliases;    // NULL-terminated list of host name aliases
  int h_addrtype;      // result's address type, typically AF_INET
  int length;          // length of the addresses in bytes(typically 4, for IPv4)
  char **h_addr_list   // NULL-terminated list of host's IP addresses
};

int gethostbyname_r(const char *name, struct hostent *ret,
                    char *buf, size_t buflen,
                    struct hostent **result, int *h_errnop);
```

- Part of the `man` page on the `bind` system call is below. `bind` is used to assign an IP address to a socket, but it is generic enough to be able to assign IPv4 or IPv6 addresses:

```
NAME
       bind - bind a name to a socket

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int bind(int sockfd, const struct sockaddr *addr,
                socklen_t addrlen);

DESCRIPTION
```

```
       When  a socket is created with socket(2), it exists in a name space
       (address family) but has no address assigned to it.  bind() assigns
       the address specified by addr to the socket referred to by the file
       descriptor sockfd.  addrlen specifies the size, in  bytes,  of  the
       address  structure  pointed to by addr.  Traditionally, this opera-
       tion is called “assigning a name to a socket”.

       It is normally necessary to assign a  local  address  using  bind()
       before   a   SOCK_STREAM   socket   may  receive  connections  (see
       accept(2)).
```

- There are actually three different siblings of the `sockaddr` record family, as shown below (also see the slides here: https://slides.com/tofergregg/sockaddr/live#/). The first one is a generic socket address structure, the second is specific to traditional IPv4 addresses (e.g. `171.64.64.131`), and the third is specific to IPv6 addresses (e.g. `4371:f0dd:1023:5::259`), which aren't in widespread use just yet. The addresses of socket address structures like those above are cast to (`struct sockaddr `) when passed to all of the various socket-oriented system calls (e.g. `accept`, `connect`, and so forth). How can these system calls tell what the true socket address record type really is—after all, it needs to know how to populate it with data—if everything is expressed as a generic `struct` `sockaddr` `?`

```
struct sockaddr {
  short sa_family;
  char sa_data[14];
};

struct sockaddr_in {
  short sin_family;
  short sin_port;
  struct in_addr sin_addr;
  char sin_zero[8];
};

struct sockaddr_in6 {
  short sin6_family;
  short sin6_port;
  // other fields;
};
```