Lecture 22: API Servers, Threads, Processes

- I want to implement an API server that's architecturally in line with the way Google, Twitter, Facebook, and LinkedIn architect their own API servers.
- This example is inspired by a website called Lexical Word Finder.
  - Our implementation assumes we have a standard Unix executable called `scrabble-word-finder`. The source code for this executable—completely unaware it'll be used in a larger networked application—can be found right here.
  - `scrabble-word-finder` is implemented using only CS106B techniques—standard file I/O and procedural recursion with simple pruning.
  - Here are two abbreviated sample runs:

```
poohbear@myth61:$ ./scrabble-word-finder lexical
ace
// many lines omitted for brevity
lei
lex
lexical
li
lice
lie
lilac
xi
poohbear@myth61:~
```
```
poohbear@myth61:$ ./scrabble-word-finder network
en
// many lines omitted for brevity
wonk
wont
wore
work
worn
wort
wot
wren
wrote
poohbear@myth61:$
```
Lecture 22: API Servers, Threads, Processes

- I want to implement an API service using HTTP to replicate what **scrabble-word-finder** is capable of.
  - We'll expect the API call to come in the form of a URL, and we'll expect that URL to include the rack of letters.
  - Assuming our API server is running on **myth54:13133**, we expect
    
    http://myth54:13133/lexical and http://myth54:13133/network to generate the following payloads:

```json
{
  time: 0.223399,
  cached: false,
  possibilities: [
    'ace',
    // several words omitted
    'lei',
    'lex',
    'lexica',
    'lexical',
    'li',
    'lice',
    'lie',
    'lilac',
    'xi'
  ]
}

{
  time: 0.223399,
  cached: false,
  possibilities: [
    'en',
    // several words omitted
    'wonk',
    'wont',
    'wore',
    'work',
    'worn',
    'wort',
    'wot',
    'wren',
    'wrote'
  ]
}
```
Lecture 22: API Servers, Threads, Processes

- One might think to cannibalize the code within `scrabble-word-finder.cc` to build the core of `scrabble-word-finder-server.cc`.
  - Reimplementing from scratch is wasteful, time-consuming, and unnecessary.
  - `scrabble-word-finder` already outputs the primary content we need for our payload. We're packaging the payload as JSON instead of plain text, but we can still tap `scrabble-word-finder` to generate the collection of formable words.
  - Can we implement a server that leverages existing functionality? Of course we can!
  - We can just leverage our `subprocess_t` type and `subprocess` function from Assignment 3.

```c
struct subprocess_t {
    pid_t pid;
    int supplyfd;
    int ingestfd;
};
subprocess_t subprocess(char *argv[],
                        bool supplyChildInput, bool ingestChildOutput,
                        const set<int>& openfds = set<int>());
```
Lecture 22: API Servers, Threads, Processes

- Here is the core of the `main` function implementing our server:

```cpp
int main(int argc, char *argv[]) {
    unsigned short port = extractPort(argv[1]);
    int server = createServerSocket(port);
    cout << "Server listening on port " << port << "." << endl;
    ThreadPool pool(16);
    map<string, vector<string>> cache;
    set<int> openfds;
    mutex cacheLock, openfdsLock;
    while (true) {
        struct sockaddr_in address;  // used to surface IP address of client
        socklen_t size = sizeof(address);  // also used to surface client IP address
        bzero(&address, size);
        int client = accept(server, (struct sockaddr *) &address, &size);
        char str[INET_ADDRSTRLEN];
        cout << "Received a connection request from "
             << inet_ntop(AF_INET, &address.sin_addr, str, INET_ADDRSTRLEN) << "." << endl;
        pool.schedule([client, &cache, &cacheLock, &openfds, &openfdsLock] {
            publishScrabbleWords(client, cache, cacheLock, openfds, openfdsLock);
        });
    }
    return 0;
}
```
Lecture 22: API Servers, Threads, Processes

- The second and third arguments to `accept` are used to surface the IP address of the client. Ignore the details around how I use `address`, `size`, and the `inet_ntop` function until this coming Wednesday, when I'll talk more about them. Right now, it's a neat-to-see!
- Each request is handled by a dedicated worker thread within a `ThreadPool` of size 16.
- The thread routine called `publishScrabbleWords` will rely on our `subprocess` function to marshal plain text output of `scrabble-word-finder` into JSON and publish that JSON as the payload of the HTTP response.
- The next several slides include the full implementation of `publishScrabbleWords` and some of its helper functions.
- Most of the complexity comes around the fact that I've elected to maintain a cache of previously processed letter racks and that I absolutely need to maintain a set of open `ingestfd`s so overlapping calls to `subprocess`—that is, parallel calls to `subprocess`—work properly and without race conditions.
- I will present a cache-free, `openfds`-ignoring version of `publishScrabbleWords` in lecture, just so you understand the overarching workflow of how the server gets its job done.
Lecture 22: API Servers, Threads, Processes

- Here is the first 70% or so of `publishScrabbleWords`:

```cpp
static void publishScrabbleWords(int client, map<string, vector<string>>& cache, mutex& cacheLock, 
    set<int>& openfds, mutex& openfdsLock) {
    sockbuf sb(client);
    iosockstream ss(&sb);
    string letters = getLetters(ss);
    sort(letters.begin(), letters.end());
    skipHeaders(ss);
    struct timeval start;
    gettimeofday(&start, NULL);  // start the clock
    cacheLock.lock();
    auto found = cache.find(letters);
    cacheLock.unlock();  // release lock immediately, iterator won't be invalidated by competing find calls
    bool cached = found != cache.end();
    vector<string> formableWords;
    if (cached) {
        formableWords = found->second;
    } else {
        const char *command[] = {"./scrabble-word-finder", letters.c_str(), NULL};
        openfdsLock.lock();
        subprocess_t sp = subprocess(const_cast<char *>(command), false, true);
        openfds.insert(sp.ingestfd);
        openfdsLock.unlock();
        pullFormableWords(formableWords, sp.ingestfd, openfds, openfdsLock);  // function exits with lock on openfdsLock
        openfdsLock.unlock();
        waitpid(sp.pid, NULL, 0);
        lock_guard<mutex> lg(cacheLock);
        cache[letters] = formableWords;
    }
}
```
Lecture 22: API Servers, Threads, Processes

- Here's the rest of `publishScrabbleWords` along with the `pullFormableWords` and `sendResponse` helper functions.

```c
struct timeval end, duration;
gmtimeofday(&end, NULL); // stop the clock, server-computation of formableWords is complete
timersub(&end, &start, &duration);
ostringstream payload;
constructPayload(formableWords, cached, /* time = */ duration.tv_sec + duration.tv_usec/1000000.0, payload);
sendResponse(ss, payload.str());
}

static void pullFormableWords(vector<string>& formableWords, int ingestfd, set<int>& openfds, mutex& openfdsLock) {
    stdio_filebuf<char> inbuf(ingestfd, ios::in);
    istream is(&inbuf);
    while (true) {
        string word;
        getline(is, word);
        if (is.fail()) break;
        formableWords.push_back(word);
    }
    openfdsLock.lock(); // acquire lock on mutex before inbuf is destroyed, leave mutex is locked state as we exit
    openfds.erase(ingestfd);
}

static void sendResponse(iosockstream & ss, const string& payload) {
    ss << "HTTP/1.1 200 OK\r\n";
    ss << "Content-Type: text/javascript; charset=UTF-8\r\n";
    ss << "Content-Length: " << payload.size() << "\r\n";
    ss << "\r\n";
    ss << payload << flush;
}
```
Lecture 22: API Servers, Threads, Processes

- Finally, here are the `getLetters` and the `constructPayload` helper functions. I omit the implementation of `skipHeaders`—you saw it with `web-get`—and `constructJSONArray`, which you're welcome to view right here.

```cpp
static string getLetters(iosockstream& ss) {
    string method, path, protocol;
    ss >> method >> path >> protocol;
    string rest;
    getline(ss, rest);
    size_t pos = path.rfind("/");
    return pos == string::npos ? path : path.substr(pos + 1);
}

static void constructPayload(const vector<string>& formableWords, bool cached, double time, ostringstream& payload) {
    payload << '{' << endl;
    payload << "  time: " << time << ",” << endl;
    payload << "  cached: " << boolalpha << cached << ",” << endl;
    payload << "  possibilities: " << constructJSONArray(formableWords, 2) << endl;
    payload << "}'" << endl;
}
```

- Our `scrabble-word-finder-server` provided a single API call that resembles the types of API calls afforded by Google, Twitter, or Facebook to access search, tweet, or friend-graph data.