Announcements

- **Assignment 5 and 6.**
  - Assignment 5 due tonight, Assignment 6 out right now and due next Friday.
  - Assignment 6 is nontrivial, but made much easier because you paid it forward by implementing your **`ThreadPool`** this past week.
- **Today's Agenda**
  - Implement **`createClientSocket`**.
  - Understand the data structures used to model IPv4 and IPv6 addresses and ports.
  - Review classic server idiom, cover implementation of **`createServerSocket`**.
  - Review introduction of threading and **`detach`** method to get computation off main thread.

This Slide Deck's Larger Example

- **Imitation of Lexical Word Finder**
  - Assumes existence of standalone **`scrabble-word-finder`**.
  - Code contributing to **`scrabble-word-finder`**, which has no idea it might contribute to a server, is right here.
    - Implemented using straightforward procedural recursion with pruning.
    - Hardly optimized to be fast—no caching, makes use of only the most obvious pruning strategies.
- **We want to implement a server to share what `scrabble-word-finder` is capable of.**
  - Approach: allow URL to specify rack of letters.
  - **`http://myth4.stanford.edu:13133/ieclxal`** should produce all words that can be formed from **`ieclxal`**.

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

Today's Larger Example (continued)

- **Computation relevant to server already exists.**
    - Reimplementing is bad, and reinventing the wheel is wasteful and time consuming.
    - **`scrabble-word-finder`**, as an executable, already outputs the core of what we'd like to serve as plain text, as with:

```
myth4> ./scrabble-word-finder ieclxal
ace
lex
lexica
lexical
li
lice
lie
lilac
xi
myth4>
```

- **Can we write a server that leverages existing functionality and packages it differently?**
    - Of course we can, else I wouldn't be asking.

```
FILE *popen(const char *command, const char *mode); // mode must be either "r" or "w"
int pclose(FILE *stream);
```

- Requires the use of **`popen`** and **`pclose`**, the prototypes of which are supplied above.
    - **`popen`** is similar to the **`subprocess`** we covered int lecture, except that it returns a single **`FILE *`** instead of two file descriptors.
    - With **`popen`**, you get access to the subprocess's output stream (**`"r"`**) or its input stream (**`"w"`**), but not both. You specify which one you want when you call **`popen`**.
    - **`pclose`** closes the process—presumably a zombie process at the time it's called—and returns the process status as surfaced by **`waitpid`** (which you know must be involved in the implementation of **`pclose`** if zombies and status codes are involved).

Today's Larger Example (continued)

- **Each request is handled by a detached, dedicated thread.**
  - Thread routine uses **`popen`** and **`pclose`** to marshal plain text output of **`scrabble-word-finder`** into JSON, and publishes that JSON as the payload of the HTTP response.
  - Here's the core of the server-side computation:

```
static void publishScrabbleWords(int clientSocket) {
  sockbuf sb(clientSocket);
  iosockstream ss(&sb);
  string letters = getLetters(ss); // extracts tail of path from GET <path> <protocol>
  skipHeaders(ss); // skips everything else
  string command = "./scrabble-word-finder \"" + letters + "\"";
  FILE *infileFromProcess = popen(command.c_str(), "r");
  vector<string> formableWords;
  pullFormableWords(formableWords, infileFromProcess);
  int status = pclose(infileFromProcess);
  ostringstream payload;
  constructPayload(status, formableWords, payload); // posts JSON to payload
  sendResponse(ss, payload.str()); // publishes HTTP response to ss out of payload
}
```

  - Helper functions are omitted, but included as part of the full code base, which in addition to the core functionality, also includes some caching to improve server response time.