Lecture 15: Introduction to Networking

Principles of Computer Systems

Winter 2020

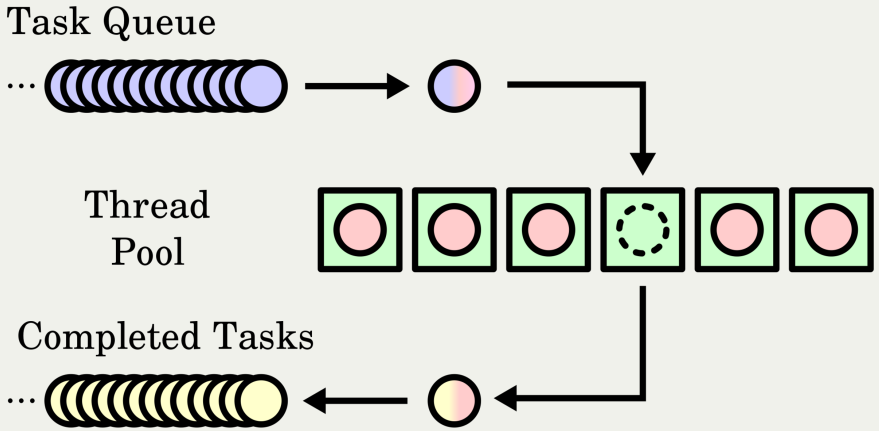
Stanford University

Computer Science Department

Instructors: Chris Gregg and Nick Troccoli

PDF of this presentation

Lecture 15: Assignment 6: ThreadPool



A sample thread pool (green boxes) with waiting tasks (blue) and completed tasks (yellow)

Source: https://en.wikipedia.org/wiki/Thread_pool#/media/File:Thread_pool.svg

- So far, we have been working with multiple threads that we launch when we need them. This is often fine, but there is an associated cost (in time) to start a thread -- there is setup and launching that we might like to avoid.
- Instead of launching threads on-demand, we can leverage a *thread pool* (which is what you are building for assignment 6). A thread pool preemptively launches a number of threads that each utilize a semaphore or condition variable to wait until they are needed. They are given another function to run, and when they are signaled, they run the function immediately and avoid the on-demand setup time.
- The primary benefit of using a thread pool is for increased performance by decreasing the *latency* for each unit of work that needs to get done.
- Thread pools can be set up to increase or decrease the number of threads running, based on need. For example, if a web server is using a Thread Pool and suddenly gets many requests at once, more threads can be launched to handle the load, and they can be ended if the load eventually diminishes.
- We will see a networking use of thread pools using networking later in the lecture.

Lecture 15: Introduction to Networking

- Networking is simply communicating between two computers connected on a network. You can actually set up a network connection on a single computer, as well.
- A network requires one computer to act as the *server*, waiting patiently for an incoming connection from another computer, the *client*.
- Server-side applications set up a *socket* that listens to a particular port. The server socket is an integer identifier associated with a local IP address, and a the port number is a 16-bit integer with up to 65535 allowable ports.
  - You can think of a port number as a virtual process ID that the host associates with the true pid of the server application.
  - You can see some of the ports your computer is listening to with the **`netstat`** command:

```
cgregg@myth59: $ netstat -plnt
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:587           0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      -
tcp6       0      0 :::22                   :::*                    LISTEN      -
tcp6       0      0 ::1:631                 :::*                    LISTEN      -
```

Lecture 15: Introduction to Networking

```
cgregg@myth59: $ netstat -plnt
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:587           0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      -
tcp6       0      0 :::22                   :::*                    LISTEN      -
tcp6       0      0 ::1:631                 :::*                    LISTEN      -
```

- Some common ports are listed above. You can see a full list here and here.
  - Ports 25 and 587 are the SMTP (Simple Mail Transfer Protocol), for sending and receiving email.
  - Port 53 is the DNS (Domain Name Service) port, for associating names with IP addresses.
  - Port 22 is the port for SSH (Secure Shell)
  - Port 631 is for IPP (internet printing protocol)
- For your own programs, generally try to stay away from port numbers listed in the links above, but otherwise, ports are up for grabs to any program that wants one.

Lecture 15: Introduction to Networking

- Let's create our first server (entire program here):

```
int main(int argc, char *argv[]) {
  int server = createServerSocket(12345);
  while (true) {
    int client = accept(server, NULL, NULL); // the two NULLs could instead be used to
                                             // surface the IP address of the client
    publishTime(client);
  }
  return 0;
}
```

- **`accept`** (found in **`sys/socket.h`**) returns a descriptor that can be written to and read from. Whatever's written is sent to the client, and whatever the client sends back is readable here.
  - This descriptor is one end of a bidirectional pipe bridging two processes—on different machines!

Lecture 15: Introduction to Networking

- The **`publishTime`** function is straightforward:

```
static void publishTime(int client) {
  time_t rawtime;
  time(&rawtime);
  struct tm *ptm = gmtime(&rawtime);
  char timestr[128]; // more than big enough
  /* size_t len = */ strftime(timestr, sizeof(timestr), "%c\n", ptm);
  size_t numBytesWritten = 0, numBytesToWrite = strlen(timestr);
  while (numBytesWritten < numBytesToWrite) {
    numBytesWritten += write(client,
                             timestr + numBytesWritten,
                             numBytesToWrite - numBytesWritten);
  }
  close(client);
}
```

- The first five lines here produce the full time string that should be published.
  - Let these five lines represent more generally the server-side computation needed for the service to produce output.
  - Here,the payload is the current time, but it could have been a static HTML page, a Google search result, an RSS document, or a movie on Netflix.
- The remaining lines publish the time string to the client socket using the raw, low-level I/O we've seen before.

Lecture 15: Introduction to Networking

- Note that the **`while`** loop for writing bytes is a bit more important now that we are networking: we are more likely to need to write multiple times on a network.
  - The socket descriptor is bound to a network driver that may have a limited amount of space
  - That means **`write`**'s return value could very well be less than what was supplied by the third argument.
- Ideally, we'd rely on either C streams (e.g. the **`FILE *`**) or C++ streams (e.g. the **`iostream`** class hierarchy) to layer over data buffers and manage the **`while`** loop around exposed **`write`** calls for us.
- Fortunately, there's a stable, easy-to-use third-party library—one called **`socket++`** that provides exactly this.
  - **`socket++`** provides iostream subclasses that respond to **`operator<<`**, **`operator>>`**, **`getline`**, **`endl`**, and so forth, just like **`cin`**, **`cout`**, and file streams do.
  - We are going to operate as if this third-party library was just part of standard C++.
- The next slide shows a prettier version of **`publishTime`**.

Lecture 15: Introduction to Networking

- Here's the new implementation of **`publishTime`**:

```
static void publishTime(int client) {
  time_t rawtime;
  time(&rawtime);
  struct tm *ptm = gmtime(&rawtime);
  char timestr[128]; // more than big enough
  /* size_t len = */ strftime(timestr, sizeof(timestr), "%c", ptm);
  sockbuf sb(client);
  iosockstream ss(&sb);
  ss << timestr << endl;
} // sockbuf destructor closes client
```

- We rely on the same C library functions to generate the time string.
- This time, however, we insert that string into an **`iosockstream`** that itself layers over the client socket.
- Note that the intermediary **`sockbuf`** class takes ownership of the socket and closes it when its destructor is called.

Lecture 15: Introduction to Networking

- If you've been working on assignment 5, you've already seen an example (**`aggregate`**) where multithreading can significantly improve the performance of networked applications.
- Our time server can benefit from multithreading as well. The work a server needs to do in order to meet the client's request might be time consuming—so time consuming, in fact, that the server is slow to iterate and accept new client connections.
- As soon as **`accept`** returns a socket descriptor, spawn a child thread—or reuse an existing one within a **`ThreadPool`**—to get any intense, time consuming computation off of the main thread. The child thread can make use of a second processor or a second core, and the main thread can quickly move on to its next **`accept`** call.
- Here's a new version of our time server, which uses a **`ThreadPool`** to get the computation off the main thread.

```
int main(int argc, char *argv[]) {
    int server = createServerSocket(12345);
    ThreadPool pool(4);
    while (true) {
        int client = accept(server, NULL, NULL); // the two NULLs could instead be used
                                                 // to surface the IP address of the client
        pool.schedule([client] { publishTime(client); });
    }
    return 0;
}
```

Lecture 15: Introduction to Networking

- The implementation of **`publishTime`** needs to change just a little if it's to be thread safe.
- The change is simple but important: we need to call a different version of **`gmtime`**.
- **`gmtime`** returns a pointer to a single, statically allocated global that's used by all calls.
- If two threads make competing calls to it, then both threads race to pull time information from the shared, statically allocated record.
- Of course, one solution would be to use a **`mutex`** to ensure that a thread can call **`gmtime`** without competition and subsequently extract the data from the global into local copy.
- Another solution—one that doesn't require locking and one I think is better—makes use of a second version of the same function called **`gmtime_r`**. This second, reentrant version just requires that space for a dedicated return value be passed in.
- A function is reentrant if a call to it can be interrupted in the middle of its execution and called a second time before the first call has completed.
- While not all reentrant functions are thread-safe, **`gmtime`**_r itself is, since it doesn't depend on any shared resources.
- The thread-safe version of **`publishTime`** is presented on the next slide.

Lecture 15: Introduction to Networking

- Here's the updated version of **`publishTime`**:

```
static void publishTime(int client) {
    time_t rawtime;
    time(&rawtime);
    struct tm tm;
    gmtime_r(&rawtime, &tm);
    char timestr[128]; // more than big enough
    /* size_t len = */ strftime(timestr, sizeof(timestr), "%c", &tm);
    sockbuf sb(client); // destructor closes socket
    iosockstream ss(&sb);
    ss << timestr << endl;
}
```

Lecture 15: Introduction to Networking

- Implementing your first client! (code here)
  - The protocol—that's the set of rules both client and server must follow if they're to speak with one another—is very simple.
    - The client connects to a specific server and port number. The server responds to the connection by publishing the current time into its own end of the connection and then hanging up. The client ingests the single line of text and then itself hangs up.

```
int main(int argc, char *argv[]) {
    int clientSocket = createClientSocket("myth64.stanford.edu", 12345);
    assert(clientSocket >= 0);
    sockbuf sb(clientSocket);
    iosockstream ss(&sb);
    string timeline;
    getline(ss, timeline);
    cout << timeline << endl;
    return 0;
}
```

    - We'll soon discuss the implementation of **`createClientSocket`**. For now, view it as a built-in that sets up a bidirectional pipe between a client and a server running on the specified host (e.g. **`myth64`**) and bound to the specified port number (e.g. 12345).

Lecture 15: Introduction to Networking

- Emulation of wget
  - **`wget`** is a command line utility that, given its URL, downloads a single document (HTML document, image, video, etc.) and saves a copy of it to the current working directory.
  - Without being concerned so much about error checking and robustness, we can write a simple program to emulate the **`wget`**'s most basic functionality.
  - To get us started, here are the **`main`** and **`parseUrl`** functions.
- **`parseUrl`** dissects the supplied URL to surface the host and pathname components.

```
static const string kProtocolPrefix = "http://";
static const string kDefaultPath = "/";
static pair<string, string> parseURL(string url) {
    if (startsWith(url, kProtocolPrefix))
        url = url.substr(kProtocolPrefix.size());
    size_t found = url.find('/');
    if (found == string::npos)
        return make_pair(url, kDefaultPath);
    string host = url.substr(0, found);
    string path = url.substr(found);
    return make_pair(host, path);
}

int main(int argc, char *argv[]) {
    pullContent(parseURL(argv[1]));
    return 0;
}
```

Lecture 15: Introduction to Networking

Emulation of **`wget`** (continued)

- **`pullContent`**, of course, needs to manage everything, including the networking.

```
static const unsigned short kDefaultHTTPPort = 80;
static void pullContent(const pair<string, string>& components) {
    int client = createClientSocket(components.first, kDefaultHTTPPort);
    if (client == kClientSocketError) {
        cerr << "Could not connect to host named \"" << components.first << "\"." << endl;
        return;
    }
    sockbuf sb(client);
    iosockstream ss(&sb);
    issueRequest(ss, components.first, components.second);
    skipHeader(ss);
    savePayload(ss, getFileName(components.second));
}
```

- We've already used this **`createClientSocket`** function for our **`time-client`**. This time, we're connecting to real but arbitrary web servers that speak HTTP.
- The implementations of **`issueRequest`**, **`skipHeader`**, and **`savePayload`** subdivide the client-server conversation into manageable chunks.
  - The implementations of these three functions have little to do with network connections, but they have much to do with the protocol that guides any and all HTTP conversations.

Lecture 15: Introduction to Networking

Emulation of **`wget`** (continued)

Here's the implementation of **`issueRequest`**, which generates the smallest legitimate HTTP request possible and sends it over to the server.

```
static void issueRequest(iosockstream& ss, const string& host, const string& path) {
    ss << "GET " << path << " HTTP/1.0\r\n";
    ss << "Host: " << host << "\r\n";
    ss << "\r\n";
    ss.flush();
}
```

- It's standard HTTP-protocol practice that each line, including the blank line that marks the end of the request, end in CRLF (short for carriage-return-line-feed), which is **`'\r'`** following by **`'\n'`**.
- The **`flush`** is necessary to ensure all character data is pressed over the wire and consumable at the other end.
- After the **`flush`**, the client transitions from supply to ingest mode. Remember, the **`iosockstream`** is read/write, because the socket descriptor backing it is bidirectional.

Lecture 15: Introduction to Networking

- **`skipHeader`** reads through and discards all of the HTTP response header lines until it encounters either a blank line or one that contains nothing other than a **`'\r'`**. The blank line is, indeed, supposed to be **`"\r\n"`**, but some servers—often hand-rolled ones—are sloppy, so we treat the **`'\r'`** as optional. Recall that getline chews up the **`'\n'`**, but it won't chew up the **`'\r'`**.

```
static void skipHeader(iosockstream& ss) {
    string line;
    do {
        getline(ss, line);
    } while (!line.empty() && line != "\r");
}
```

- In practice, a true HTTP client—in particular, something as HTTP-compliant as the **`wget`** we're imitating—would ingest all of the lines of the response header into a data structure and allow it to influence how it treats payload.
  - For instance, the payload might be compressed and should be expanded before saved to disk.
  - I'll assume that doesn't happen, since our request didn't ask for compressed data.

Lecture 15: Introduction to Networking

- Everything beyond the response header and that blank line is considered payload—that's the timestamp, the JSON, the HTML, the image, or the cat video.
  - Every single byte that comes through should be saved to a local copy.

```
static string getFileName(const string& path) {
    if (path.empty() || path[path.size() - 1] == '/') return "index.html";
    size_t found = path.rfind('/');
    return path.substr(found + 1);
}

static void savePayload(iosockstream& ss, const string& filename) {
    ofstream output(filename, ios::binary); // don't assume it's text
    size_t totalBytes = 0;
    while (!ss.fail()) {
        char buffer[2014] = {'\0'};
        ss.read(buffer, sizeof(buffer));
        totalBytes += ss.gcount();
        output.write(buffer, ss.gcount());
    }
    cout << "Total number of bytes fetched: " << totalBytes << endl;
}
```

- HTTP dictates that everything beyond that blank line is payload, and that once the server publishes that payload, it closes its end of the connection. That server-side close is the client-side's **`EOF`**, and we write everything we read.