Project 4a: Scalable Web Crawler

Objectives

There are three objectives to this assignment:

- Learn how to use mutexes and condition variables effectively.
- Learn how web crawlers utilize links to explore the online graph.
- Build a library that uses callbacks.

Background

Search engines (e.g., Google) perform several tasks internally. First, they **download** pages from the Internet. Second, they **parse** the downloaded content, looking for links to new, unknown content. Third, they use link counts and other factors to determine what a page is about and how important it is. Fourth, they handle user queries, returning content that is relevant and important. The first two tasks (downloading and parsing) are performed by a web crawler.

In this assignment, you will be developing a real, working **web crawler.** To simplify this project, your crawler will parse web pages in a simplified format (not HTML). In order to achieve high performance, your crawler will use multiple threads, for two reasons. First, the latency of fetching a web page can be on the order of 100s of milliseconds. Thus, a single threaded crawler might only fetch 10 pages per second. By issuing many concurrent requests from multiple threads, you will avoid this bottleneck. Second, most modern computers have multiple CPU cores. By splitting parsing across multiple threads, can fully utilize your computer's CPU.

To help you get started, we have provide some skeleton code:

```
wget http://pages.cs.wisc.edu/~harter/537/p4/p4a.tar.gz
```

OR

```
cp ~harter/public/html/537/p4/p4a.tar.gz .; tar -xf p4a.tar.gz
```

Thread Pools

Your crawler will have two groups (or pools) of threads. The **downloaders** will be responsible for fetching new pages from the Internet. The **parsers** will scan the downloaded pages for new links.

Note the circular interactions between the two groups. When a downloader fetches a new page, it creates more work for a parser. Similarly, when parser finds a new link, it creates more work for a downloader.

Queues

The downloaders and the parsers will send work to each other via two queues.

First, you will implement a fixed-size queue for parsers to send work to downloaders. Parsers will push new links onto the queue, and downloaders will pop them off. A parser must wait if the queue is full, and a downloader must wait if the queue is empty. Note that this is the canonical producer/consumer problem, so you will need one mutex and two condition variables.

Second, you will implement an unbounded queue for downloaders to send work to parsers. Unbounded simply means that you malloc memory for each new entry, so you can always insert another entry into the queue (unless, of course, you run out of memory and malloc fails; in that case, it is ok to exit). Note that with an unbounded buffer no thread will ever need to wait because the queue is full. Thus, while this queue represents a producer/consumer problem, you may not need the standard code with two condition variables and a mutex.

Callbacks

Your code will deployed as a library so that different applications that need to do web crawling may use it. Your library will use callbacks to provide the most flexibility.

In particular, a program using your crawler library will pass two function pointers to the library. You crawler will call the provided fetch() function to retrieve web-page content. Using the callback enables you to work with multiple protocols. Generally, fetch() will issue HTTP requests to web servers, but alternate implementations may fetch data over FTP or may even fetch data from the local file system (our tests will do the latter).

When your crawler finds a new link in a page, it will do two things: (1) add the link to the downloaders' work queue, and (2) return the link (or edge) to the program using your library. You will return the edge by calling the provided edge() function.

Format

HTML links generally look like this:

```
<a href="http://pages.cs.wisc.edu/~remzi/Classes/537/Spring2016/">
Fun, Free Projects!
</a>
```

You're free to write a parser to find such URLs, but we'll be sharing websites for you to crawl that follow a simpler format (all tests will follow the simple format). In particular, a link to addr will be prefixed by link: and have a following space (or come at the end of the document):

```
link:addr
```

Library Specifications

Your shared library must provide the following function:

```
int crawl(
char *start_url,
int download_workers,
int parse_workers,
int queue_size,
char * (*fetch_fn)(char *link),
void (*edge_fn)(char *from, char *to)
);
```

crawl() will return 0 on success and -1 on failure.

The arguments are as follows:

- start_url: the first link to visit. All other links will be discovered by following links out recursively from this first page.
- download_workers: the number of worker threads in the download pool.
- parse_workers: the number of workers threads in the parse pool.
- queue_size: the size of the links queue (the downloaders pop off this queue to get more work).
- fetch_fn: a callback to ask the program using the library to fetch the content to which the link refers. The function will return NULL if the content isn't reachable (a broken link). Otherwise, the function will malloc space for the return value, so you must free it when you're done.
- edge_fn: this is to notify the program that the library encountered a link on the from page to the to page.

Shared Library

You must provide these routines in a shared library named "libcrawler.so". Placing the routines in a shared library instead of a simple object file makes it easier for other programmers to link with your code. There are further advantages to shared (dynamic) libraries over static libraries. When you link with a static library, the code for the entire library is merged with your object code to create your executable; if you link to many static libraries, your executable will be enormous. However, when you link to a shared library, the library's code is not merged with your program's object code; instead, a small amount of stub code is inserted into your object code and the stub code finds and invokes the library code when you execute the program. Therefore, shared libraries have two advantages: they lead to smaller executables and they enable users to use the most recent version of the library at run-time. To create a shared library named libcrawler.so, use the following commands (assuming your library code is in a single file "crawler.c"):

```
gcc -c -fpic crawler.c -Wall -Werror
gcc -shared -o libcrawler.so crawler.o
```

To link with this library, you simply specify the base name of the library with "-lcrawler" and the path so that the linker can find the library "-L.".

```
gcc -lcrawler -L. -o myprogram mymain.c -Wall -Werror
```

Of course, these commands should be placed in a Makefile. Before you run "myprogram", you will need to set the environment variable, LD_LIBRARY_PATH, so that the system can find your library at run-time. Assuming you always run myprogram from this same directory, you can use the command:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

Implementation Hints

Your library should use pthreads. You should take an especially close look at pthread_create, pthread_cond_wait, pthread_cond_signal, pthread_mutex_lock, and pthread_mutex_lock. An implementation that spins instead of using condition variables will receive a poor grade (even if it passes the tests).

Your crawl() function should return when the entire graph has been visited, but we don't require you to cleanly exit from the thread pools.

If you want to use strtok() for parsing, you should use strtok_r() instead. The strtok_r() is reentrant, or thread safe (i.e., multiple threads can call it at the same time and nothing breaks). The strtok() function is not thread safe.

There are many cycles in the web graph (e.g., A links to B, and B links to A). You need to make sure not to end up in an infinite loop. One way to avoid this is to use a hash table or hash set to keep track of already-processed links. You may copy a hash function from online (e.g., fletcher32), as long as a comment in your code references the source URL.