PROJECT 3 - MULTITHREADED WEB SPIDER
Computer Systems Principles

Emery Berger Mark Corner

October 28, 2010

1 Overview

Goals of this assignment: understanding threads; understanding mutual exclusion; using concurrency to hide I/O latency; using the work-queue model.

For this assignment, you will write a mini web spider. Search engines use web spiders (also called crawlers) to retrieve documents recursively from the Internet.

2 Spider

2.1 Input

Your program, to be called spider, will take three command-line inputs:

- The root URL to start from;
- The maximum depth to crawl;
- The number of worker threads to spawn.

2.1.1 Root URL

The root website will be specified in the following form:
`http://www.cs.umass.edu`
OR
`http://www.cs.umass.edu/`
OR
`http://www.cs.umass.edu/index.html`

See the section on the helper functions for using `parse_single_URL()` to parse this.

2.1.2 Depth

The maximum depth tells your crawler how far to recurse.

A depth of zero means that the root URL should be retrieved, but no others.

A depth of one indicates that the root URL should be retrieved, and all pages that it links to, but no others.

2.1.3 Threads

This is the number of worker threads to spawn for crawling web pages. You will have one additional thread that will do the parsing of pages to find new URLs.

2.2 Crawling Web Pages

Your program will use a work queue style of concurrency, where multiple threads pull work off of a single queue.

Each worker thread will pull a URL off the from of a queue, retrieve that web page into a buffer, then insert that buffer into another queue for parsing.

A separate thread will pull the buffers off of the parsing work queue, parse them for new URLs, insert those URLs back onto the work queue and so on.

The worker thread may not retrieve another web page until the previous page has been processed. This implies an ordering constraint!

No page should be crawled more than once. Your program should track which pages have been visited. A host and file is unique, so if a file is on more than one host, you should crawl each of them.

3 Threading, Retrieving and Parsing Pages

3.1 Retrieving Web Pages

You will be provided with a simple socket library that will make it easy to connect up to a given web server and read the contents of a particular file. The interface to that library is contained in `simplesocket.h`.

The following code provides the basics of retrieving a page:

```
clientsocket sock (host.c_str(), 80, 0, false);
if (sock.connect()){
  sprintf (buf, "GET␣/%s␣HTTP/1.0\r\nHost:␣%s\r\n\r\n", file.c_str(),host.c_str());
  sock.write (buf, strlen(buf));

  int ret;
  int size = 0;
  sock.setTimeout(5);
  while ((ret = sock.read(buf+size, MAX_READ-1-size)) > 0){
    size += ret;
  }
```

```
}
sock.close();
```

This code will timeout after 5 seconds if it fails to retrieve any data. The return value contains the number of bytes have been read from the socket. Multiple reads may be required to retrieve the page up to the `MAX_READ` size. (Note: because we are using HTTP 1.0, the server should close the connection after sending the data)

Your code should not read any more than `MAX_READ` size, so it will only get URLs that are in the first `MAX_READ` bytes of the page.

The code is also set to fail connecting if it doesn't complete after 5 seconds.

3.2 Parsing Web Pages for URLs

We have written a simple URL parser for you (see url.h) that can be called using:

```
parse_URLs(buf, size, urls);
```

where buf is the buffer you read from the web server, the size is the size of the buffer and urls is a set containing `url_t` structs (see url.h).

This isn't the smartest parser ever, so don't expect it to get every URL, just the simpler ones. It should find plenty of URLs in most web pages to crawl.

3.3 Starting Threads

You should get your program to work as a single threaded program first, then make it multi-threaded. This is the hard part.

You will be using the popular pthreads threading package to complete your spider.

The types and functions you should be concerned with are

- mutexes: pthread_mutex_t
- condition variables: pthread_cond_t
- pthread identifiers: pthread_t
- pthread attributes: pthread_attr_t
- initialization for mutexes: pthread_mutex_init
- initialization for condition variables: pthread_cond_init
- thread join: pthread_join
- thread create: pthread_create
- lock: pthread_mutex_lock
- unlock: pthread_mutex_unlock

- cv signal: pthread_cond_signal
- cv broadcast: pthread_cond_broadcast
- cv wait: pthread_cond_wait

3.3.1 Limiting Stack Sizes

For all of your threads, please limit their stack sizes

```
status = pthread_attr_init(&attr);
if (status) {
   cout << "pthread_attr_init␣returned␣" << status << endl;
   exit(1);
}

status = pthread_attr_setstacksize(&attr, 5*1024*1024);
if (status) {
  cout << "pthread_attr_setstacksize␣returned␣" << status << endl;
  exit(1);
}
```

3.3.2 Starting Threads

```
status = pthread_create(&thread_id[i], &attr, (void * (*)(void *)) worker_thread, (void *
```

Assuming that the worker thread is declared as:

```
void worker_thread (void *arg)
```

Note that you can "cheat" and use arg to pass integers: For instance: `int thread_id = (int) arg;`.

3.3.3 Joining Threads

You may need to wait for a thread to complete using join:

```
pthread_join(parse_thread_id, NULL);
```

4 Output

You program should only create two pieces of output. Your output must be identical.

The requester (worker thread) should output this:

```
cout << "requester␣" << thread_id << "␣url␣" << host << "/" << file << endl;
```

right before adding a buffer to the parser's work queue.

The parsing thread should output:

```
cout << "service␣requester␣" << thread << "␣url␣" << url.host << "/" << url.file << endl;
```

after parsing the page, and before adding the new urls to the work queue.

Note that when using cout, you should be carefully about mutual exclusion, as you don't want two pieces of output corrupting one another.

5 Compiling, Testing and Hints

5.1 Compiling

Use the following command to compile your spider:

```
g++ -Wall -g -o spider spider.cc libspider.a -lpthread
```

5.2 Debugging

Notice that your program should be much faster when running with a number of threads than when it runs with just one thread. Verify this by running it with `/usr/bin/time`. Make sure you link your program with -lpthread, or it won't actually spawn any threads (thanks, GNU libc).

You should not be holding any locks when connecting or retrieving a web page.

We have set up a tree structure web page here: `http://www.cs.umass.edu/~mcorner/cs377/root_tree_1000.html`

It has 3 levels, with a branching factor of 10. This might be helpful in debugging your spider.

5.3 Hints

The stack size in for each thread is limited to `STACK_SIZE` in thread.h. You will get odd segfaults if you go over this size, so be careful of creating a huge numbers, or sizes, of stack variables.

The hardest part may be deciding when to quit! One way is to track how many pages are currently in the work queue, plus the number that have been removed from the work queue and are currently being retrieved. If the sum of those two things is zero, and the parser is not parsing anything, the program is done.

6 Handing Project In

Your project will be handed in using the autograding system. The autograder is checking for correctness and performance. Please see the web page for details on how to submit your solution.