COMP 345 **Operating Systems**

Project 3

Assigned: March 18 *Due: April 10*

IMPORTANT: For both parts of this project, if your code does not compile, you get, at most, a D+. I am happy to help with compilation errors.

1 Overview

This project is based on a project by Jon Shidal designed for the *Operating Systems* course at Washington University in St. Louis.

Like the first project, this project will have 2 parts. In the first part you'll get experience with multi-threaded programming in Linux. In the second, you'll modify `xv6` to include a thread library, allowing multi-threaded programs to run in `xv6`.

You may work in groups of size 1, 2, or 3 for this project. Note, however, that doing one part per person will not be effective; both (or all 3) people should work together on both parts to finish fastest.

2 Part A: A Concurrent Web Crawler

2.1 Goals

1. Learn about web crawlers and the producer/consumer model by creating a multi-threaded web crawler.
2. Gain experience with mutexes and condition variables (or semaphores) to write thread-safe code.
3. Learn to create a library using callbacks to increase flexibility.

2.2 Background

Search engines like Google do many things. One task they perform is discovering new content on the web, which is done using a web crawler. A web crawler designed to look for new content downloads web pages and scans them for links to other pages. Those new links are then downloaded and scanned, until there are no new links. Once downloaded by the crawler, the search engine generally classifies a page somehow and stores information about it. This information can then be used to respond to user queries with the most relevant pages.

In this project, you will develop a basic web crawler. Web crawlers generally parse HTML, but to simplify the parsing part of this project we will work with a simplified format described below. Fetching a web page may take many milliseconds, meaning a single-threaded crawler may only fetch a few pages per second. For performance, you will be required to create a multi-threaded crawler. This will allow many page requests to be made at once, alleviating this bottleneck. Also, using multiple CPUs will allow for several web pages to be parsed at once.

2.3 Starting Point

Some starter code is given on the department server in `/home/comp345/proj3a.zip`. I have generously implemented

1. A working `Makefile`
2. A sequential, non-threaded crawler, in `seq.c`
3. Several utility data structures, including queues and a set, in `util/`

for you.

2.4 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.

2.5 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, you should print an error message and 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 (though you will still need some synchronization...).

2.6 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: `fetch()` and `edge`.

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) notify the program using the library about the link (or edge). Your crawler will do this by calling the provided `edge()` function.

2.7 Link Format

You're free to write a parser to find such HTML 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:`, then contain the address, and then have a following space (or come at the end of the document):

> This link:addr comes in the middle of a sentence.

2.8 Library Specifications

Your shared library must provide the following function:

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

`crawl()` should 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.
- `queue_size`: the size of the links queue (the downloaders pop off this queue to get more work).
- `download_workers`: the number of worker threads in the download pool.
- `parse_workers`: the number of workers threads in the parse pool.
- `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.

Your `crawl` function must find all possible links, but can NOT fetch content from a link more than once. In other words, once a page like `pageA.txt` has been fetched and processed, you must be careful not to fetch and parse it again.

2.9 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 runtime.

This works just like creating our `malloc` shared library from project 2. To create a shared library named `libcrawler.so`, use the following commands (assuming your library code is in a single file `crawler.c`):

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

Of course, your life will be a lot easier if you place these commands 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 runtime. Assuming you always run `myprogram` from this same directory, you can use the command:

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

2.10 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`. Also note that I have `pthread` wrapper functions defined in `util/threads.h`. An implementation that spins instead of using condition variables will not receive full points (even if it passes the tests).

If you want to use `strtok()` for parsing, you should use `strtok_r()` instead. This `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. I have provided a simple hash set in `util/hash.h`, but it is not multi-threaded! You will encounter race conditions if you try to run this with multiple threads. I suggest copying this file and adding whatever code is necessary to make the set thread-safe. You could use a list instead for this purpose, but either way you'll have to make it thread-safe.

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. In other words, your crawler should find all possible links, but it can keep running forever (press `Ctrl-c` to kill it).

If you'd like a significant amount of bonus points (perhaps to make up for project 2?), try to implement clean termination. Hint 1: have the parser workers keep track of how many links have been found but not yet parsed. Hint 2: Once all the found links have been parsed, push a special "poison pill" onto the download workers' queue, 1 for each worker. Any downloader that gets on of these immediately quits.

3 Part B: xv6 Kernel Threads

3.1 Overview

Now that you've learned about kernel threads and multi-threaded programming, it's time to add threads to `xv6`. How? you'll do two things:

1. Define a new system call to create a kernel thread, called `clone()` , as well as one to wait for a thread called `join()`.

2. Then, you'll use `clone()` to build a thread library, with `thread_create()`, `thread_join()`, `lock_acquire()` and `lock_release()` functions.

3.2 Details

Your clone system call should look like this:

```
int clone(void(*fcn)(void*), void *arg, void* stack)
```

This call creates a new kernel thread which shares the calling process's address space. File descriptors are copied as in `fork()`. The new process uses `stack` as its user stack, which is passed the given argument `arg` and uses a fake return PC (`0xffffffff`). The stack should be one page in size.

The new thread starts executing at the address specified by `fcn`. As with `fork()`, the PID of the new thread is returned to the parent.

The other new system call is `int join(void **stack)`. This call waits for a child thread that shares the address space with the calling process. It returns the PID of the waited-for child or -1 if none. The location of the child's user stack is copied into the argument `stack`.

You also need to think about the semantics of a couple of existing system calls. For example, `int wait()` should wait for a child process that does **not** share the address space with this process. It should also free the address space **if** this is last reference to it. Finally, `exit()` should work as before but for both processes and threads; most solutions to this project will require little (if any) change here, but it is something should think about.

Your thread library will be built on top of this, and just have a simple `thread_create(void (*start_routine)(void*), void *arg)` routine. This routine should call `malloc()` to create a new user stack, use `clone()` to create the child thread and get it running. A `thread_join()` call should also be used, which calls the underlying `join()` system call, frees the user stack, and then returns.

Your thread library should also have a simple spin lock. There should be a type `lock_t` that one uses to declare a lock, and two routines: `lock_acquire(lock_t *)` and `lock_release(lock_t *)` which acquire and release the lock. The spin lock should use x86 atomic exchange to implement the spin lock (see the xv6 kernel for an example of something close to what you need to do). One last routine, `lock_init(lock_t *)`, is used to initialize a lock.

3.3 Hints

You should probably start with a clean copy of `xv6`.

One thing you need to be careful with is when an address space is grown by a thread in a multi-threaded process. Trace this code path carefully and see where a new lock is needed and what else needs to be updated to grow an address space in a multi-threaded process correctly.

It might be a good idea to read the `xv6` book (`https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf`), particularly the section on locking.

4 Submitting

Create a README file for your project. In it, list each member in your group's name and briefly describe:

1. Part 1: what each part of your library does. Include a description of how your `crawler.c` works and each concurrent data structure that you used.
2. Part 2: the changes you made to `xv6`, including a description of the test program(s) you wrote.

To submit, first use `git add <files>` on all source files that you modified or created. Then use `git diff --staged > threads.patch` . At this point you can use `handin` to submit all your files (all part 1 files + `threads.patch`).