Multi-threaded Web Crawler Built with C

Possible Files of Interest for Code Sample Review

- `findpng2.c` is the main driver program and is about 350 lines of code
- `stack.c` and `hash.c` (300 lines combined) define two memory-safe data structures
- Please feel free to look at any of the files you feel are appropriate. I included descriptions of each file below.
- Please also feel free to build using `make`

Project context

Purpose

Starting from a seed URL, recursively crawl to all other URLs linked on the web page of the seed URL, searching for PNGs. Stop when we find a specified number of PNGs or run out of URLs to visit. The program outputs all the pngs found in a `png_urls.txt` file at the end of the program. The user can also specify to output all the unique URLs the program crawled to a log file.

To speed up the search, the program launches multiple threads. The user may specify the number of runner threads that the program will launch. Each thread can process a different URL concurrent with other threads. The user may also specify the number of PNGs to find before stopping.

We use multiple threads primarily because network downloads can be slow compared to the CPU. Each thread synchronously waits for the network download of a page. The CPU can process other threads while this thread is blocked on the network download. If the program runs on one thread, the program cannot make progress while the one thread waits for its network download to complete.

Files

- `findpng2.c` : main driver program
    - takes input
    - runs the specified number of threads
    - the threads perform the crawl
    - handles synchronization between threads
    - handles clean-up of threads at the end of the program
- `stack.c` :
    - a memory-safe dynamic stack that holds strings
    - used for holding URLs to crawl (the `frontier`)
- `p_stack.c` :
    - a memory-safe dynamic stack that holds pointers
    - used in `hash.c` for holding pointers to memory for later deallocation
- `hash.c` :
    - a memory-safe hash set that holds strings as keys
    - used for holding visited URLs to prevent cycles in the crawling process
- `curl_xml.c` :
    - utility functions for downloading web pages using cURL
    - utility functions for processing XML pages using libxml
    - utility function for checking if a png file is a valid png
    - used for downloading web pages, searching for URLs listed on the pages, and checking the validity of found pngs

External libraries used

- cURL (https://curl.se/libcurl/)
- libxml (https://github.com/GNOME/libxml2)

Notes

- I wrote this program for the course, ECE 252 (Systems Programming and Concurrency).
    - I received 100% on this project (marked based on functionality, speed, lack of deadlocks, and proper memory management).
    - I received 100% on the course (60% final exam, 40% projects).
- The program is named `findpng2` because I have a few other versions that perform the same task with different techniques (e.g. using processes and interprocess communication for synchronization; or using asynchronous cURL).

Building

- run `make` in this directory

Usage

```
findpng2 [OPTION]... [ROOT_URL]
```

- options:
    - -t=NUM - the program will create NUM threads to crawl the web (default: 1)
    - -m=NUM - the program will find up to NUM unique PNG URLs (default: 50)
    - -v=LOGFILE - if specified, program will log the unique URLs visited in a file named LOGFILE
- output:
    - on terminal, `findpng2 execution time: S seconds`

    - the program will create a `png_urls.txt` file containing all the valid PNG URLs found
    - if a log file is specified, the program will create a `<LOGFILE>` file containing all unique URLs visited
- for example, `./findpng2 -t 1 -m 1 -v test.txt https://www.cleanpng.com/static/img/logo.png` will launch 1 thread to find 1 png starting from the URL `https://www.cleanpng.com/static/img/logo.png` and output all visited URLs into `test.txt`. Note that since the seed URL is a png itself, the crawl will end immediately after the first visit.

Code Best Practice Notes

- `findpng2.c` uses global variables, which is not good practice; however, the course instructors recommended them for simplicity for the sake of the small project.
    - I could instead pass to the thread runner a pointer to a struct containing all the global variables.
- `findpng2.c`'s thread runner function exits with a `return` rather than `pthread_exit`, which is the recommended way to exit a thread. However, valgrind will report a small number of reachable blocks for threads that use `pthread_exit`. This is not a real issue in most applications. However, since a part of the marking scheme for this project was to achieve 0 reachable blocks at the end of program, my only option was to use a `return`.

Algorithm overview

findpng2.c

Global data structures

- `frontier`: a stack of discovered URLs that we have yet to process, shared by all threads
- `pngs`: a stack of all pngs found so far
- `visited`: a hash set of URLs we have visited (so we don't crawl repeat URLs)

Synchronization

- `frontier_mutex`: a lock for accessing `frontier` and other accounting variables related to the state of the crawl (whether if the crawl is done; number of threads waiting for the frontier to be non-empty; number of threads processing a URL)
    - used for the `frontier_empty` condition variable as well
- `pngs_mutex`: a lock for accessing `pngs`
- `visited_mutex`: a lock for accessing `visited`
- `frontier_empty`: a condition variable that threads will wait on when `frontier` is empty
    - when another thread adds to `frontier`, it will broadcast to wake up the sleeping threads
    - alternatively, a thread may broadcast when the program is finished (no more URLs we can recursively crawl or we have found `num_pngs_to_find` pngs) so that sleeping threads can wake up and exit

Thread runner

- The main function launches `t` number of these thread runners.
- The runner does the following in an infinite loop.
    - With lock `frontier_mutex`:
        - Check if our crawl is finished. If so, signal all threads to exit.
            - If `frontier` is empty and there are no threads currently processing a URL, we don't and won't have any more URLs to crawl: our crawl is finished.
            - If our crawl is finished, broadcast `frontier_empty` to wake up sleeping threads so they can exit.
        - Wait until there is a URL to crawl or our crawl is finished.
            - The thread does this by waiting on the condition variable `frontier_empty`.
            - The signal to wake up will come from a thread pushing a URL to `frontier` or a thread signalling our crawl is finished.
        - If our crawl is finished, exit loop.
        - Else, grab the first URL on `frontier` for processing.
        - With lock `visited_mutex`:
            - Check if we've already visited the URL.
            - If we have, don't progress further and return to the start of the loop.
            - If we haven't, add it to the `visited` hash set and continue on.
    - Download the URL's contents
        - If it is a HTML file, grab all URLs that it links to.
        - If it is a PNG file, determine if it is a valid png.
    - (If HTML file) With lock `frontier_mutex`:
        - Push all URLs found onto `frontier`.
        - If there are any sleeping threads waiting for a non-empty frontier, broadcast on `frontier_empty`.
    - (If PNG file) With lock `pngs_mutex`:
        - Push URL into `pngs`.
        - If we hit `num_pngs_to_find`:
            - With lock `frontier_mutex`:
                - Set a global variable indicating our crawl is finished, and broadcast on `frontier_empty`.
- When the runner exits the infinite loop, clean up any data structures used (memory deallocation, network libraries).

stack.c and p_stack.c

- Memory-safe stacks
- Resizing is done by allocating a larger chunk of memory, moving the old items over, and deallocating the old memory.

hash.c

- A hash set that holds strings as keys

Data structures

- `hmap` : `hsearch_data` struct used as the underlying hashmap
- `elements` : an array of pointers to keys currently in the hash set
  - used for linear access to keys, moving keys over on resize, and for deallocating the keys at destruction
- `ps` : a stack of pointers to keys that were used when searching `hmap`
  - used for deallocating the strings at destruction
  - note we don't deallocate the keys/strings immediately after the search since `hsearch_data` may hold on to the pointer to the key and use it in future operations after the search

Adding a key

- If full, resize.
- Create a copy of the passed-in key/string.
- Add the key (copy) to `hmap` .
- Add the key (copy) to `elements` .

Searching for a string

- Create a copy of the passed-in key/string.
- Search for the key (copy) in `hmap` .
- If we found the key:
  - Add the key (copy) to `ps` .
- Else:
  - Free memory for the key (copy).
- Why we add the key to `ps` :
  - If the key is found in `hmap` , `hmap` may use the memory of the passed-in string for future references instead of the string used when the key was initially added to `hmap` .
  - `hsearch` does not deallocate memory upon destruction. Thus, we keep track of `ps` : the strings to deallocate at destruction.

Resizing

- Destroy and reinitialize `hmap` with a larger size.
- Allocate a larger `elements` .
- Transfer keys from the old `elements` to the new `hmap` and `elements` .
- Deallocate old `elements` .