Lab 3: Encoder

The autograder was updated on November 1st at 11:25 p.m. Please re-download it if you had an earlier version. Thanks!

Introduction

Data compression is the process of encoding information using fewer bits than the original representation. **Run-length encoding (RLE)** is a simple yet effective compression algorithm: repeated data are stored as a single data and the count. In this lab, you will build a parallel run-length encoder called **Not Your Usual ENCoder**, or `nyuenc` for short.

Objectives

Through this lab, you will:

- Familiarize yourself with multithreaded programming using POSIX threads.
- Learn how to implement a thread pool using mutexes and condition variables.
- Learn how to use a thread pool to parallelize a program.
- Get a better understanding of key IPC concepts and issues.
- Be a better C programmer and be better prepared for your future technical job interviews. In particular, the data encoding technique that you will practice in this lab frequently appears in interview questions.

Run-length encoding

Run-length encoding (RLE) is quite simple. When you encounter *n* characters of the same type in a row, the encoder (`nyuenc`) will turn that into a single instance of the character followed by the count *n*.

For example, a file with the following contents:

```
aaaaaabbbbbbbbba
```

would be encoded (logically, as the numbers would be in binary format) as:

```
a6b9a1
```

Note that the exact format of the encoded file is important. You will store the character in ASCII and the count as a **1-byte unsigned integer in binary format**. In other words, the output is actually `"a\x06""b\x09""a\x01"`.

In this example, the original file is 16 bytes, and the encoded file is 6 bytes.

For simplicity, you can assume that no character will appear more than 255 times in a row. In other words, you can safely store the count in one byte.

Milestone 1: sequential RLE

You will first implement `nyuenc` as a single-threaded program. The encoder reads from one or more files specified as command-line arguments and writes to `STDOUT`. Thus, the typical usage of `nyuenc` would use shell redirection to write the encoded output to a file.

Note that you can use `xxd` to inspect a file in binary format since not all characters are printable.

For example, let's encode the aforementioned file.

```
$ echo -n "aaaaaabbbbbbbbba" > file.txt
$ xxd file.txt
0000000: 6161 6161 6161 6262 6262 6262 6262 6261  aaaaaabbbbbbbbba
$ ./nyuenc file.txt > file.enc
$ xxd file.enc
0000000: 6106 6209 6101                           a.b.a.
```

If multiple files are passed to `nyuenc`, they will be **concatenated** and encoded into a single compressed output. For example:

```
$ echo -n "aaaaaabbbbbbbbba" > file.txt
$ xxd file.txt
0000000: 6161 6161 6161 6262 6262 6262 6262 6261  aaaaaabbbbbbbbba
$ ./nyuenc file.txt file.txt > file2.enc
$ xxd file2.enc
0000000: 6106 6209 6107 6209 6101                 a.b.a.b.a.
```

Note that the last `a` in the first file and the leading `a`'s in the second file are merged.

For simplicity, you can assume that there are no more than 100 files, and the total size of all files is no more than 1GB.

Milestone 2: parallel RLE

Next, you will parallelize the encoding using POSIX threads. In particular, you will implement a thread pool for executing encoding tasks.

You should use mutexes, condition variables, or semaphores to realize proper synchronization among threads. **Your code must be free of race conditions. You must not perform busy waiting, and you must not use `sleep()`, `usleep()`, or `nanosleep()`.**

Your `nyuenc` will take an optional command-line option `-j jobs`, which specifies the number of worker threads. (If no such option is provided, it runs sequentially.)

For example:

```
$ time ./nyuenc file.txt > /dev/null
real    0m0.527s
user    0m0.475s
sys     0m0.233s
$ time ./nyuenc -j 3 file.txt > /dev/null
real    0m0.191s
user    0m0.443s
sys     0m0.179s
```

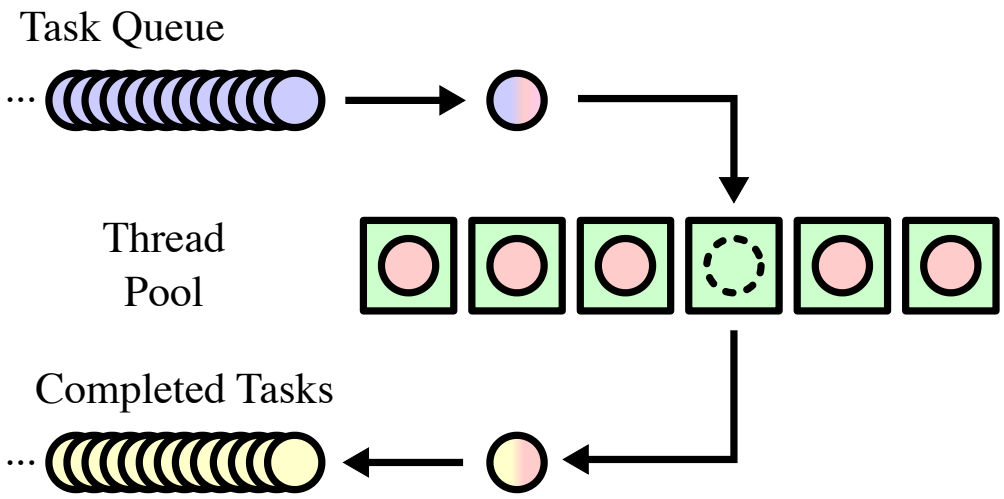
You can see the difference in running time between the sequential version and the parallel version. (Note: redirecting to `/dev/null` discards all output, so the time won't be affected by I/O.)

How to parallelize the encoding

Think about what can be done in parallel and what must be done serially by a single thread.

Also, think about how to divide the encoding task into smaller pieces. Note that the input files may vary greatly in size. Will all the worker threads be fully utilized?

Here is an illustration of a thread pool from Wikipedia:



At the beginning of your program, you should create a pool of worker threads (the green boxes in the figure above). The number of threads is specified by the command-line argument `-j jobs`.

You should divide the input data into fixed-size 4KB (*i.e.*, 4,096-byte) chunks and submit the tasks (the blue circles in the figure above) to the task queue, where each task would encode a chunk. Whenever a worker thread becomes available, it would execute the next task in the task queue. (Note: it's okay if the last chunk of a file is smaller than 4KB.)

For simplicity, you can assume that the task queue is unbounded. In other words, you can submit all tasks at once without being blocked.

After submitting all tasks, the main thread should collect the results (the yellow circles in the figure above) and write them to `STDOUT`. Note that you may need to stitch the chunk boundaries. For example, if the previous chunk ends with `aaaaa`, and the next chunk starts with `aaa`, instead of writing `a5a3`, you should write `a8`.

It is important that you synchronize the threads properly so that there are no deadlocks or race conditions. In particular, there are two things that you need to consider carefully:

- The worker thread should wait until there is a task to do.
- The main thread should wait until a task has been completed so that it can collect the result. Keep in mind that the tasks might not complete in the same order as they were submitted.

Compilation

We will grade your submission on one of the compute servers, which runs CentOS Linux release 7.9.2009. We will compile your program using `gcc` 9.2.0. You need to run the following command to load it:

```
$ module load gcc-9.2
```

You must provide a `Makefile`, and by running `make`, it should generate an executable file named `nyuenc` in the current working directory. Note that you need to add the compiler option `-pthread`.

During debugging, you may want to disable compiler optimizations. However, for the final submission, it is recommended that you enable compiler optimizations (`-O`) for better performance.

*You are **not** allowed to use thread libraries other than pthread.*

Evaluation

Your code will first be tested for **correctness**. Parallelism means nothing if your program crashes or cannot encode files correctly.

If you pass the correctness tests, your code will be measured for **performance**. Higher performance will lead to better scores.

strace

We will use `strace` to examine the system calls you have invoked. You will suffer from **severe score penalties** if you call `clone()` too many times, which indicates that you do not use a thread pool properly, or if you call `nanosleep()`, which indicates that you do not have proper synchronization.

On the other hand, you can expect to find many `futex()` invocations in the `strace` log. They are used for synchronization.

You can use the following command to run your program under `strace` and dump the log to `strace.txt`:

```
$ strace ./nyuenc -j 3 file.txt > /dev/null 2> strace.txt
```

Valgrind

We will use two Valgrind tools, namely Helgrind and DRD, to detect thread errors in your code. **Both should report 0 errors from 0 contexts**.

You can use the following command to run your program under Helgrind:

```
$ valgrind --tool=helgrind --read-var-info=yes ./nyuenc -j 3 file.txt >
==12345== Helgrind, a thread error detector
==12345== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et a
==12345== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
==12345== Command: ./nyuenc -j 3 file.txt
==12345==
==12345==
==12345== Use --history-level=approx or =none to gain increased speed,
==12345== the cost of reduced accuracy of conflicting-access informatio
==12345== For lists of detected and suppressed errors, rerun with: -s
==12345== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 240 from
```

You can use the following command to run your program under DRD:

```
$ valgrind --tool=drd --read-var-info=yes ./nyuenc -j 3 file.txt > /dev
==12345== drd, a thread error detector
==12345== Copyright (C) 2006-2017, and GNU GPL'd, by Bart Van Assche.
==12345== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
==12345== Command: ./nyuenc -j 3 file.txt
==12345==
==12345==
==12345== For lists of detected and suppressed errors, rerun with: -s
==12345== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 139 from
```

Note that Valgrind is very slow; so you should only test it with small files.

Submission

You will submit an archive containing all files needed to compile `nyuenc`. You can do so with the following command:

```
$ tar cvJf nyuenc-Your_NetID.tar.xz Makefile *.h *.c
```

Rubric

The total of this lab is **100 points**, mapped to **15% of your final grade** of this course.

- **Meaningful submission that compiles successfully.** (40 points)
- **Milestone 1.**
  - **Correctly encode one file.** (10 points)
  - **Correctly encode multiple files.** (10 points)

- **Milestone 2.**
  - **Correctness.** (20 points)
    Your program should produce the correct output within 30 seconds without crashing.
  - **Free of thread errors (*e.g.*, deadlocks, race conditions).** (15 points)
    For each error context reported by Helgrind or DRD (we will take the smaller of the two), there will be 5 points deduction.
  - **Performance.** (5 points)
    Your program will not be evaluated for performance if it fails any previous test.
    If your program is correct and free of thread errors, its performance will be compared against Prof. Tang's implementation on a large multi-threaded workload.
    - You will get 5 points if your program's running time is within 2x of Prof. Tang's.
    - You will get 4 points if your program's running time is within 3x of Prof. Tang's.
    - You will get 3 points if your program's running time is within 5x of Prof. Tang's.
    - You will get 2 points if your program's running time is within 10x of Prof. Tang's.
    - You will get 1 point if your program's running time is within 20x of Prof. Tang's.

You will lose all points for Milestone 2 if your implementation is not based on the thread pool, if you invoke `nanosleep` (directly or indirectly), or if you use thread libraries other than pthread.

The autograder

We are providing a sample autograder with a few test cases. Please extract them on a CIMS compute server and follow the instructions in the `README` file.

Note that the sample autograder only checks if your program has produced the correct output. You need to run valgrind and measure the performance yourself. Also note that these test cases are **not** exhaustive. The test cases for final grading will be different from the ones provided and will not be shared. Do not try to hack or exploit the autograder or the CIMS computer servers.

Tips

Don't procrastinate!

This lab requires significant programming and debugging effort. Therefore, **start as early as possible!**

How to parse command-line options?

The `getopt()` function is useful for parsing command-line options such as `-j jobs`. Read its man page and example.

If you still can't figure out how to use `getopt()` after reading the man pages, feel free to parse `argv[]` yourself.

How to access the input file efficiently?

You may have used `read()` or `fread()` before, but one particularly efficient way is to use `mmap()`, which maps a file into the memory address space. Then, you can efficiently access each byte of the input file via pointers. Read its man page and example.

If you still can't figure out how to call `mmap()` after reading the man pages, select (highlight) the following text to reveal the cheat code:
`mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0)`

How to represent and write binary data?

Store your data as a `char[]` or `unsigned char[]` and use `write()` or `fwrite()` to write them to `STDOUT`. Don't use `printf()` or attempt to convert them to any human-readable format.

Also keep in mind that you should never use string functions (*e.g.*, `strcpy()` or `strlen()`) on binary data as they may contain null bytes.

Which synchronization mechanisms should I use?

I personally find mutexes and condition variables easier to work with than semaphores, but it's up to you.

How to debug?

Debugging multithreaded code can be frustrating, especially when it hangs. If you are using `gdb`, you can press `Ctrl-C` to stop the running program and use the following command to show what each thread is doing:

```
(gdb) thread apply all backtrace
```

Suppose you want to switch to Thread 2. You can type:

```
(gdb) thread 2
```

Then, you can use `up` and `down` to navigate through the stack frames and use `info locals` or `print` to examine variables.

A particularly useful command is `x`, which can show a portion of memory in binary format. For example, to examine the next 8 bytes pointed by `ptr`, you can use:

```
(gdb) x/8xb ptr
```

Lastly, we are aware that a startup called Undo provides free educational licenses for its UDB time travel debugger. You can use `udb` in place of `gdb`. Here is a quick reference guide. You can also read a tutorial on debugging race conditions using `udb`.

My program is slow! Why?

First of all, the encoding algorithm needs to iterate through the input data **only once**. This means you should never copy the input data. If you find yourself iterating through the data more than once, that's an issue. (Using `mmap()` is efficient here: by default, it does not load the data from disk to memory unless you access the mapped address. Think of it as *lazy evaluation*.)

Second, make sure all threads are working **in parallel** without constantly contending for a lock. (Otherwise, they're just encoding data serially.) You can run your program under `gdb` and press `Ctrl-C` to stop the execution. Examine what each thread is doing. Most, if not all, worker threads should be doing the encoding. If you find many threads blocked, that's an issue.

Finally, you can run your program under a profiler to see where the bottleneck is. Follow these steps:

1. Run `perf record` followed by your command.
2. Run `perf report`.

This lab has borrowed some ideas from Prof. Arpaci-Dusseau.