# Lab 3: Encoder

**Note: the due date of this lab has been extended to Tuesday, November 21 to accommodate our class progress.**

## 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 content:

```
aaaaaabbbbbbbbba
```

would be encoded (logically, as the numbers would be in binary format instead of ASCII) 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**. See the example below for the actual content in the output file.

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 should use `xxd` to inspect a binary file since not all characters are printable. `xxd` dumps the file content in hexadecimal.

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
```

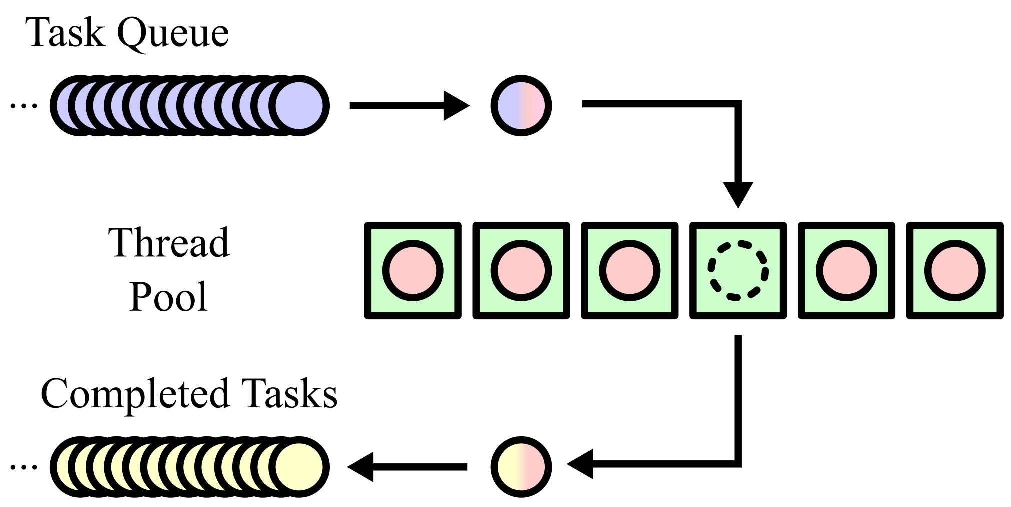
You can see the difference in running time between the sequential version and the parallel version. (Note: redirecting to `/dev/null` discards all output, so the time won't be affected by I/O.)

## 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 worker threads is specified by the command-line argument `-j jobs`.

The main thread should divide the input data logically 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 in an x86_64 Rocky Linux 8 container on Gradescope. We will compile your program using `gcc` 12.1.1 with the C17 standard and GNU extensions.

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 `LDFLAGS=-pthread` to your `Makefile`. *(Refer to Lab 1 for an example of the `Makefile`.)*

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.

Please note that **your computer needs to have at least as many cores as the number of threads you use in order to benefit from parallelization**. Consider running your program on the CIMS compute servers if you are using a single- or dual-core machine.

### 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 > /dev/r
==12345== Helgrind, a thread error detector
==12345== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==12345== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==12345== Command: ./nyuenc -j 3 file.txt
==12345==
==12345==
==12345== Use --history-level=approx or =none to gain increased speed, at
==12345== the cost of reduced accuracy of conflicting-access information
==12345== For lists of detected and suppressed errors, rerun with: -s
==12345== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 240 from 89)
```

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/null
==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 info
==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 11)
```

Note that Valgrind is very slow; so you should only test it with small files. Also note that Valgrind may also have *false negatives*, meaning that a report of `0 errors` does not guarantee your program is *actually* free of errors.

## Submission

You must submit a `.zip` archive containing all files needed to compile `nyuenc` in the root of the archive. You can create the archive file with the following command in the Docker container:

```
$ zip nyuenc.zip Makefile *.h *.c
```

Note that other file formats (*e.g.*, `rar`) will **not** be accepted.

You need to **upload the `.zip` archive to Gradescope**. If you need to acknowledge any influences per our academic integrity policy, write them as comments in your source code.

## Rubric

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

- **Compile successfully and encode the example file correctly.** (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 one minute on Gradescope 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 7x of Prof. Tang's.
- You will get 1 point if your program's running time is within 10x 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 inside the Docker container and follow the instructions in the `README` file. *(Refer to Lab 1 for how to extract a `.tar.xz` 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 on Gradescope are different from the ones provided and will not be disclosed. Do not try to hack or exploit the autograder.

## Tips

### Don't procrastinate!

This lab requires **significant programming and debugging effort**. Therefore, **start as early as possible!** Don't wait until the last week.

## 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, here is the cheat code:*

Click to reveal spoiler

## 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`.

## My program passed all numbered test cases but failed some lettered test cases. Why?

All numbered test cases (Cases 1–7) on Gradescope are the same as those in the sample autograder, while lettered test cases (Cases *a*–*e*) are "hidden" test cases that will not be disclosed. If your program passed the former but failed the latter, please double-check if it can handle all corner cases correctly.

Although we cannot disclose the lettered test cases, you can find Prof. Tang's implementation of `nyuenc` in the sample autograder (under `aarch64/` or `x86_64/`, depending on your computer's architecture). Therefore, you can try to generate some test cases yourself and compare your program's output with Prof. Tang's program's output.

Python is handy in generating test cases. For example, the following command generates a file named `my_awesome_test_case` that consists of 255 occurrences of 0, 255 occurrences of 1, 255 occurrences of 2, and so on:

```
[root@... cs202]# python3 -c "for i in range(17): print(chr(i) * 255, end='')'
```

You can then run the programs and compare the outputs:

```
[root@... cs202]# ./nyuenc my_awesome_test_case > my_output
[root@... cs202]# path/to/prof/tangs/nyuenc my_awesome_test_case > prof_tangs_
[root@... cs202]# diff -y -W139 <(xxd my_output) <(xxd prof_tangs_output)
```

## Gradescope says “your program does not use a thread pool correctly.” Why?

The most likely reason is that some parts of your program are executed serially while they should have been executed in parallel. In particular, your main thread should run concurrently with your worker threads:

1. In the task-submission phase, worker threads should run while the main thread is submitting tasks. As soon as a task is submitted to the task queue, a worker thread should process it.
2. In the result-collection phase, the main thread should run while the worker threads are processing tasks. As soon as a task is complete, the main thread should collect the result.

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