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.

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\x06b\x09a\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.

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  aaaaaabbbbbbbba

$ ./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  aaaaaabbbbbbbba

$ ./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.

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) 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.

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.

Compiling

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`.

Testing

Your code will first be measured for **correctness**. Parallelism means nothing if it cannot encode files correctly.

If you pass the correctness tests, your code will be tested 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 > /dev/null

==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.

Tips

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.

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.

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/8bx ptr
```