Department of Computer Science and Engineering
Tandon School of Engineering
New York University

Homework 4 : Concurrency

CS-UY 3224, Intro to Operating Systems

Abstract

Threads are commonly used in programs for multiple purposes. In this homework, you will be using threads, locks, and conditional variables to perform concurrent actions.
This homework does NOT require XV6.

Academic Integrity

All work submitted for this assignment must be your own. Cheating and plagiarism will not be tolerated. NYU Tandon's policy on academic misconduct[1] applies to this assignment. If you have questions about appropriate conduct with regard to homework that is not answered by the syllabus or Tandon's code of conduct, please contact your professor.

Part 1: Run Length Encoding (RLE)

Introduction

The goal of Part 1 is to implement a string encoding algorithm, *Run Length Encoding* (RLE).

Run Length Encoding is a simple and popular data compression algorithm. It is based on the idea of replacing a long sequence of the same symbol with a shorter sequence. More specifically, **assuming the input sequence is alphabetical**, it replaces a long sequence of the same symbol with:

1) head count of the symbol in decimal form;
2) the symbol.

The output sequence will then be **alphanumeric**. Here *run* means a consecutive sequence, and *length* means the count of that run. One exception is if the count of the run is `1`, then we do not place the count `1` before the symbol. The rationale is that placing `1` inserts more bytes to represent the same information, playing against the purpose of compression.

Consider a sequence of alphabetical ASCII characters of length 14, `"aaaaaaaaaabbbb"`, which normally would require 14 bytes to store. This can be compressed to the form `"10a4b"` which takes 5 bytes only. There are more effective variations of RLE but let's keep it simple for now.

Requirements

- Create the repository on Anubis, and work on the file `rle.c`; implement the algorithm here. We are not using xv6 this time, so you are free to use the C standard library.
- `rle` program should accept one string from a file or the standard input. You may assume that the input has only one line, and it is **NOT** ended with a newline character (`\n`). The string from the input will be alphabetical, i.e, no special symbols or spaces. The implementation should be case-sensitive.

[1] NYU Tandon Student Code of Conduct

- Do not assume the length of the input.
- `rle` should write the output to standard output.

Expected Output

Like many programs we have seen in previous exercises, no surprise that the `rle` program will be used in this way:

```
./rle <filename>
```

You can also use pipes as follows:

```
cat <filename> | ./rle
```

Or a con**cat**enation of files:

```
cat <filename1> <filename2> <filename3> | ./rle
```

A non-exhaustive list of examples is given below:

- `aaaaaaaaaAbbbb` $\longrightarrow$ `9aA4b`
- `a` $\longrightarrow$ `a`
- `ab` $\longrightarrow$ `ab`
- `aa` $\longrightarrow$ `2a`
- `aab` $\longrightarrow$ `2ab`
- `aabbb` $\longrightarrow$ `2a3b`
- `aardvark` $\longrightarrow$ `2ardvark`
- `Aaardvark` $\longrightarrow$ `A2ardvark`

Part 2: Parallel RLE

Introduction

The single-threaded version works, but we can spice it up with multiple threads. For Part 2, you will implement **prle**, which concurrently compresses the given string. You have to split up the input and process the workloads with multiple threads.

We do not use xv6 for this assignment because it does not support multiple threads of execution. Instead, you need to get familiar with pthreads.

In particular, you need to use `pthread_create` to create new threads and `pthread_join` to wait for them. Do note that while we aim for better performance, optimization with multithreading does not always yield better results. That is because not all parts of the tasks are parallelizable and sometimes we are bottlenecked by I/O, the number of cores, etc.

Requirements

- The parallel implementation should produce exactly the same output as the single-threaded version does.
- The RLE algorithm needs to be run in multiple threads. Your solution has to work for inputs of any size. However, it is fine to create only one thread when the input size is too small to be split.
- The maximum number of threads that you can create is 8 (not including the original thread).
- The implementation should take advantage of parallelism by splitting up the work reasonably. For example, one way to partition 8192 bytes of input will be evenly splitting it up for each thread.
- For simplicity, you may assume that the input will be finite. In other words, you are allowed to read the entire input into memory before starting to process it. Do note that the input might still be substantial even if it is finite.
- Your implementation should be free of race condition.

The output of **prle** should stay the same as in Part 1.

Hints

- Things to think about:

  How to compress the strings parallelly?

  How many threads to create?

  What should each thread work on?

  What are some edge cases to consider?
- We provide a past assignment with the reference solution that might help. You can find them on brightspace under the **Content/Resources** section.
- Locks and conditional variables are not required unless your implementation needs them to avoid race conditions.

Grading

Part	Key points	Points
1	Handle file input	10 points
1	Handle pipes	10 points
1	RLE Algorithm implementation	20 points
2	Multithreaded implementation	40 points
2	Handle edge cases	10 points
/	Style, safety, comments	10 points

Various parts of your program may not be completely covered by this rubric. Bugs and other non-functional elements should be documented in your comments. A documented bug with a note of what was done to try to fix it will be penalized less than bugs discovered during grading.