CS 214: Systems Programming, Fall 2016
Assignment 2: Procs vs Threads (round 0)

0. Introduction

In this assignment, you will get some practice with multithreading and multiprocessing, as well as some simple file system calls. For this program, you will write both a multithreaded and a multiprocessed multipart compression program.

The base compression you will be using is fairly simple. Run Length Encoding (RLE) is a compression method that finds repeated characters and replaces the run with a number representing the number of adjacent copies.

Unencoded:	Encoded:
jjjjjjjjoooaaaaanrr	8j3o5a1n2r

As you can see, RLE has some inefficiencies. Firstly, converting the character 'n' to the symbol '1n' is increasing size by adding unnecessary information. Secondly, converting 'rr' to '2r' is no savings in space, but costs computation time to determine, write and read later. So, for this assignment you will implement a modification of RLE; Length Of Long Sequence compression, or LOLS. LOLS is like RLE except that single characters and pairs are not encoded:

Unencoded:	Encoded:
jjjjjjjjoooaaaaanrr	8j3o5anrr

1. Design

Besides simply compressing a file with LOLS, you will need to provide the ability to do multipart compression. That is, a single uncompressed file is compressed, and that compressed file is broken into chunks that are each made individual files. This results in multiple much smaller files.

Unencoded:	RLE encoded:	LOLS encoded:	Multipart LOLS encoded:
bigfile.txt	bigfile_txt_RLE	bigfile_txt_LOLS	bigfile_txt_LOLS0
jjjjjjjjoooaaaaanrr	8j3o5a1n2r	8j3o5anrr	8j2o
			bigfile_txt_LOLS1
			o5anrr

In order to be maximally efficient, your code should spawn a worker element for each part it writes. In the case of your process version, your code should generate a child process per part to read, and your thread version should spawn child threads per part. Each worker element should access the uncompressed file simultaneously. Your parent/manager process/thread should not do any work with the uncompressed file (although it may check it). Each worker element should open its own handle to the file and read from the segment of the file it needs to encode:

Unencoded:
bigfile.txt

```
        v-- worker element 0 starts here
        jjjjjjjjoooaaaaanrr
              ^-- worker element 1 starts here
```

2. Implementation

While only two parts are shown, your code should support splitting the uncompressed file into potentially any number of multiparts. Your code should be robust in checking for errors, especially filesystem errors. Namely, does the uncompressed file exist, do you have read access for it, do compressed files for that uncompressed files already exist, and so on. Be sure your compression program wait()s on child processes and join()s spawned threads.

Your code should take the form of:

compressT_LOLS(<file to compress>, <number of parts>)
compressR_LOLS(<file to compress>, <number of parts>)

.. for thread and process versions, respectively.

Your code should automatically name the compressed file just as illustrated above. It should replace the period before the file extension with an underbar '_' and postpend '_LOLS' and the part number, if there is one. For files compressed without multiparts, there should be no part number:

compressT_LOLS("./stuff.txt", 1):
compresses "./stuff.txt" into "./stuff_txt_LOLS" using threads

compressR_LOLS("./things.txt", 3):
compresses "./things.txt" into 3 files; "./things_txt_LOLS0", "./things_txt_LOLS1" and "./things_txt_LOLS2" using processes

Your code should put the newly-generated compressed files into the same directory as the uncompressed file.

When calculating char lengths to read, please round up toward the earliest compressed file part. For example, if you have an uncompressed file that consists of 13 characters and you must split it into 2 files, the first compressed part ought to have the encoding for the first 7 characters, and the 2nd compressed part the encoding for the remaining 6.

Unencoded:	Encoded:
hhhhhhhiiiiii	part0: 7h
	part1: 6i

Your code should shut down responsibly in the event of an error. All threads should be exited. All processes should be wait()ed on. All file handles should be closed and memory freed.

3. Performance Analysis and Testing

You should determine a good series of tests for your code. Try to exploit corner cases, like parts breaking a sequence that would have been encoded into a sequence that is too short to be encoded, and making sure your code does the right thing (i.e. "oooo" being broken into "oo" and "oo" across two files). Be sure to document your test plan.

You should also determine which is faster/better for this job, processes or threads. Document the way you test this, as well as your results. Keep in mind system details you have no control over (at minimum, other students) will affect your results, so you should run a fairly large number of your tests. Be sure to document your findings.

Be sure to explore all the variables at your disposal: the length of the unencoded file, the number of parts, the amount of replication in the file and process vs thread modes, etc. In order to compare your results across all variables, it would be useful to think of a similar way to score the results. Something like seconds of runtime per byte of unencoded file length is one possibility, but it may leave out the impact of multipart creation. Think of what dimensions are similar and what you can control. You may want to use more than one metric.

4. Submission

compressT_LOLS.c - project source code (threads)
compressR_LOLS.c - project source code (process manager/parent)
compressR_worker_LOLS.c – project source code (process worker/child)
readme.pdf - project documentation
testplan.txt - your testing parameters to bug check your code and be sure it handles all likely errors well.
timetests.pdf - a description of how you tested procs vs threads, the values you logged, how many times you ran your tests, and what your conclusions are.

Your grade will be based on:
• Correctness (how well your code is working),
• Quality of your evaluation (did you use reasonable tests/metrics),
• Quality of your code (how well-written your code is, including modularity and comments),
• Testing thoroughness (quality of your test cases).

5. Extra:

++: Test user threads vs kernel threads and processes as well
++: Implement a better parallel compression algorithm – be sure to do performance testing with it as well and demonstrate that it is 'better', and also test its thread/process performance