Homework 3. Multithreaded gzip compression filter

Background

You're working for a web server house that regularly needs to generate some fairly large data streams and compress them using `gzip`. The compression part of this is turning into a CPU bottleneck and the users are starting to complain. Your company can easily afford to throw hardware at the problem, so your boss asks you whether you can reprogram your servers to do the compression in parallel, taking advantage of the fact that your servers are multiprocessor.

You look into the matter, and discover that there's a C implementation of a program called `pigz` that does something along the lines that you want. (For convenience, a copy of a recent version of this program is available on SEASnet in the `/usr/local/cs/src/pigz` directory, and an executable is installed in `/usr/local/cs/bin/pigz`.) The `pigz` program can be used as a filter that reads programs from standard input and writes to standard output. Unfortunately, it has a problem: it's written in C. Your company has standardized on Java, so that it can use just one version of each executable and run it on a wide variety of servers that you have: some are x86-64, some are ARM, and some use secret RISC-V-based hardware whose full nature isn't disclosed even to your group.

You tell this to your boss, who responds, "OK, so do what `pigz` is doing, but do it in Java". Her suggestion is to use standard Java classes and see how well your substitute does, compared to standard `pigz`.

The gzip format lets you partition an input stream, compress each partition separately, and concatenate the compressed versions of each partition; the resulting compressed stream can be decompressed by `pigz`, by `gzip`, or by any other gzip-format-understanding program. Unfortunately, this approach doesn't work for your application, because the data are generated dynamically in in the form of a stream, and you want the data to be compressed and delivered to its destination on the fly. You do not want to generate all the data into a huge file, partition the file, compress each partition separately, and send the concatenation of the compressed partitions.

What you want instead, is to do what `pigz` does: divide the input into fixed-size blocks (with block size equal to 128 KiB), and have P threads that are each busily compressing a block. That is, `pigz` starts by reading P blocks and starting a compression thread on each block. It then waits for the first thread to finish, outputs its result, and then can reuse that thread to compress the (P+1)st block.

For better compression, `pigz` does something else. Instead of compressing each block independently, it uses the last 32 KiB of the previous block to prime the compression dictionary for the next block. That way, each block other than the first is compressed better, in the typical case. You want to do that too.

You search around the net some more to see whether someone has done this, and find that there's a package by Cédrik Lime called MessAdmin that has a similar feature in MessAdmin-Core. It has a lot of code that you don't need, though, and doesn't have a simple standalone application to try out. You'd like a stripped down version that just does `pigz`-style multithreaded compression, so that you compare the two applications' performances.

Assignment

Write a Java program called `Pigzj` that behaves like the C `pigz` implementation, in the sense that it operates with multiple compression threads to improve wall-clock performance. Each compression thread acts on an input data block of size 128 KiB. Each thread uses as its dictionary the last 32 KiB of the previous input data block. Compressed output blocks are generated in the same order that their uncompressed blocks were input. The number of compression threads defaults to the number of available processors, but this can be overridden. Your program may also use a small, fixed number of threads to control the compression threads or to do input/output.

Your implementation can be a simplification of `pigz`, in the following ways:

- `Pigzj` need not decompress; you have to implement only the compression part.
- `Pigzj` needs to support only the `-p` *`processes`* options of `pigz`. The latter option must be spelled with the space between the option and the value: for example, `-p3` (without a space) need not be recognized. `Pigzj` can report an error if any other option syntax is used.
- `Pigzj` always reads from standard input and writes to standard output. It can report an error if you specify a file name.
- `Pigzj`'s behavior is not specified if the input or the output is a terminal. For example, it can unconditionally read standard input and write to standard output without checking whether these streams are connected to a terminal.
- When an error occurs, `Pigzj` need not issue exactly the same diagnostic message as `pigz`, so long as it detects and reports the error to standard error, and exits with nonzero status.

`Pigzj` should behave like `pigz` in the following respects:

- If you decompress the output with `gzip` or with `pigz`, you get a copy of the input.
- The output is compressed, about as well as `pigz` would compress it.
- The output follows the GZIP file format standard, Internet RFC 1952.
- The output contains just a single member, that is, it does not contain the concatenation of two or more members. For a definition of "member" please see RFC 1952 §2.3. If you have trouble implementing this, then for partial credit you can generate output with multiple members.
- Ideally the output is byte-for-byte identical with the output of `pigz`. If this is not possible, the reason for any discrepancies must be documented.
- `Pigzj` runs faster than `gzip`, when the number of processors is greater than 1. It is competitive in speed with `pigz`.
- The default value for *`processes`* is the number of available processors; see the Java standard library's `availableProcessors` method.
- Read errors and write errors are detected. For example, the command "`pigz </dev/zero >/dev/full`" reports a write error and exits with nonzero exit status, and the same should be true for "`java Pigzj </dev/zero >/dev/full`".
- Out-of-range requests are detected. For example, on the Seasnet Linux servers "`pigz -p 10000000 </dev/zero >/dev/null`" by default reports an error and exits with nonzero status due to lack of virtual memory, and `Pigzj` should do likewise.
- The input and output need not be a regular file; they may be pipes. For example, the command "`cat /etc/passwd | java Pigzj | cat`" should output the same thing as the command "`java Pigzj </etc/passwd`".

Measure the performance of three programs: your `Pigzj`, `/usr/local/cs/bin/pigz`, and `/usr/local/cs/bin/gzip`. For your measurement platform, use a Seasnet Linux server, and specify its configuration well enough so that others outside the class could reproduce your results. Use shell commands like the following to compare the performance of the three implementations:

```
input=/usr/local/cs/jdk-16.0.1/lib/modules
time gzip <$input >gzip.gz
time pigz <$input >pigz.gz
time java Pigzj <$input >Pigzj.gz
ls -l gzip.gz pigz.gz Pigzj.gz

# This checks Pigzj's output.
pigz -d <Pigzj.gz | cmp - $input
```

See what happens if the number of processors is changed to values other than the default, trying this with both `pigz` and `Pigzj`. Run each trial at least three times, and report each instance of real time, user time, and system time; also, report the compression ratio of each command.

Use strace to generate traces of system calls executed by the three programs, and compare and contrast the resulting traces. Do they explain the performance differences that you observe?

Assess your work by writing an after-action report that summarizes your observations. Focus in particular on any problems you foresee as the file size and the number of threads scales up, and which method you expect to work better in general. This report should be a simple ASCII text file that consumes at most 50,000 bytes. (Because this report is more substantial than usual, it's worth more of the homework than usual.)

Your implementation may use ideas taken from MessAdmin, but all the code you submit must be your own. If you use ideas, give MessAdmin's author proper credit for them in your code's comments and in your after-action report.

Your implementation should operate correctly under OpenJDK 16. There is no need to run on older Java versions. Please keep your implementation as simple and short as possible, for the benefit of the reader. Your program should compile cleanly, without any warnings.

If your `PATH` is set correctly to a string starting with "`/usr/local/cs/bin:`", the command "`java -version`" should output the following text:

```
openjdk version "16.0.1" 2021-04-20
OpenJDK Runtime Environment (build 16.0.1+9-24)
OpenJDK 64-Bit Server VM (build 16.0.1+9-24, mixed mode, sharing)
```

the command "`pigz --version`" should output "`pigz 2.6`", and the command "`gzip --version`" should output "`gzip 1.10`" followed by some licensing information.

Submit

To turn in your assignment, submit a single jar file `hw3.jar` that contains both your Java source code and a plain text file `README.txt` that holds your assessment. Do not submit class files. Before submitting `hw3.jar` you should test it using the following shell commands on SEASnet:

```
# Make a fresh directory and change into it.
mkdir testdir
cd testdir

# Extract your program.
jar xf ../hw3.jar

# Make sure you have a README.txt file.
ls -l README.txt

# Build your modified version.
javac $(find . -name '*.java')

# Check your modified version; the output should be empty.
cat ../hw3.jar | java Pigzj | pigz -d | cmp - ../hw3.jar
```

Hints

Here are some library references and tips that may help.

- java.lang.Integer.parseInt
- java.nio
- java.util.zip
- Lesson: Concurrency, *The Java Tutorials* (2013)
- Cédrik Lime, MessAdmin: Notification system and Session administration for Java EE Web Applications (2010)

$Id: hw3.html,v 1.88 2021/04/28 22:55:53 eggert Exp eggert $