CPSC 3500 Computing Systems

Project 3: Parallel Zip

Assigned: Wednesday, 02/09/2022

Due: 11:59PM, Wednesday, 02/16/2022

Note: Students may choose either individual or group projects on this assignment. For a group project, the group size is capped at 3 members and only one submission is required.

1. Assignment Requirement

In project #1, you implemented a simple compression tool, **wzip**, based on run-length encoding. Here, you'll implement something similar, except you'll use threads to make a parallel version of **wzip**. We'll call this version **pzip**.

There are three specific objectives to this assignment:

- To familiarize yourself with the Linux pthreads.
- To learn how to parallelize a program with proper synchronization.
- To learn how to program for high performance.

1.1 Background: wzip

The type of compression used in **wzip** is a simple form of compression called *run-length encoding* (*RLE*). RLE is quite simple: when you encounter **n** characters of the same type in a row, the compression tool **wzip** will turn that into the number **n** and a single instance of the character.

Thus, if we had a file with the following contents:

```
aaaaaaaaaabbbb
```

the tool would turn it (logically) into:

```
10a4b
```

However, the exact format of the compressed file is quite important; here, you will write out a **4-byte integer in binary format** followed by the single character in ASCII. Thus, a compressed file will consist of some number of 5-byte entries, each of which is comprised of a 4-byte integer (the run length) and the single character.

To write out an integer in binary format (not ASCII), you should use **fwrite()**. (other similar library functions can do the same.) Read the man page for more details. For **wzip**, all output should be written to standard output (the **stdout** file stream, which, as with **stdin**, is already open when the program starts running).

Note that typical usage of the **wzip** tool would thus use shell redirection in order to write the compressed output to a file. For example, to compress the file **file.txt** into a (hopefully smaller) **file.z**, you would type:

```
prompt> ./wzip file.txt > file.z
```

The "greater than" sign is a UNIX shell redirection; in this case, it ensures that the output from **wzip** is written to the file **file.z** (instead of being printed to the screen).

1.2 Design Considerations

Doing so effectively and with high performance will require you to address (at least) the following issues:

- **How to parallelize the compression?** Of course, the central challenge of this project is to parallelize the compression process. Think about what can be done in parallel, and what must be done serially by a single thread, and design your parallel zip as appropriate. One interesting issue that the "best" implementations will handle is this: what happens if one thread runs more slowly than another? Does the compression give more work to faster threads? It is worth pointing out that you do not need to split a file among multiple threads for compression.

- **How to determine how many threads to create?** On Linux, this means using interfaces like `get_nprocs()` and `get_nprocs_conf()`; read the man pages for more details. Then, create threads to match the number of CPU resources available. If the server does not have multiple cores (which is pretty unlikely), then the default number of threads is 5.

- **How to efficiently perform each piece of work?** While parallelization will yield speed up, each thread's efficiency in performing the compression is also of critical importance. Thus, making the core compression loop as CPU efficient as possible is needed for high performance.

- **How to access the input file efficiently?** On Linux, there are many ways to read from a file, including C standard library calls like `fread()` and raw system calls like `read()`. One particularly efficient way is to use memory-mapped files, available via `mmap()`. By mapping the input file into the address space, you can then access bytes of the input file via pointers and do so quite efficiently. `mmap()` is highly recommended here.

- **How to coordinate multiple threads?** The outputs from multiple threads must be sequenced in the order of their inputs (in the order of input files). As figure 1 shows, you should follow a producer-consumer synchronization with an unbounded buffer in this project. The main thread acts as a producer and puts tasks into the buffer. The consumer threads, which initially sleep on the thread pool, wake up, compress the files and write outputs to stdout. You should use

semaphores for two purposes: (1) synchronization between the producer and consumer threads, and (2) ordering of outputs to stdout.

- **How to terminate consumer threads in the thread pool?** In your consumer thread function, if you follow the design guidelines, each consumer thread should wait on a semaphore. If all tasks are completed, the consumer threads will be sleeping on "the thread pool". We need to kill them before the process terminates. In order to do so, the main thread may artificially inject some bogus tasks (= # of consumer threads) in the buffer (e.g., the file name is empty), so the consumer thread, after waking up, can remove a bogus task and graciously terminate itself.

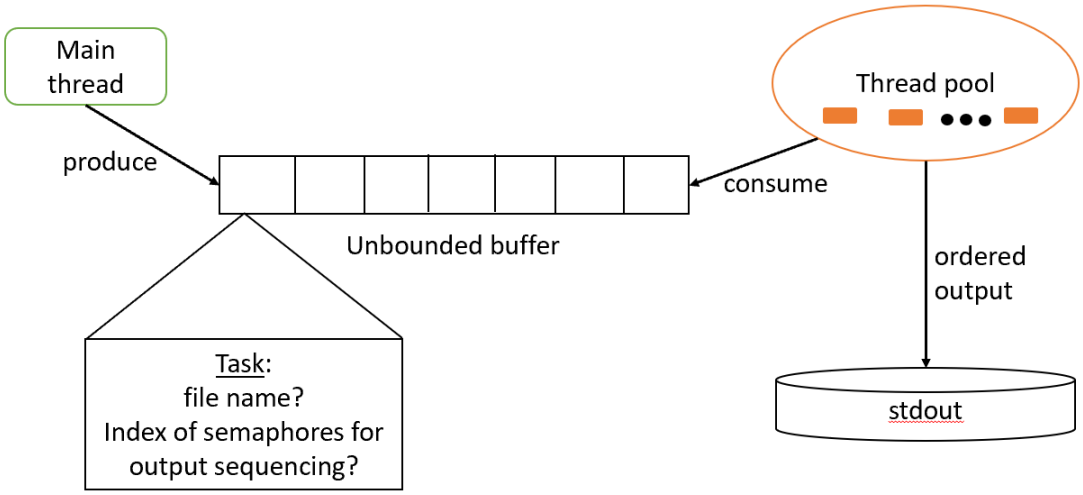

Figure 1. Pzip Synchronization: (1) Producer-consumer synchronization with an unbounded buffer. (2) semaphores for ordering outputs. (3) The consumer threads wait on the thread pool (use sem_wait) and wake up to process tasks on the buffer.

1.3 Correctness and Performance

Your code should compile (and should be compiled) with the following flags: `-Wall -Werror -pthread -O`. The last one is important: it turns on the optimizer! In fact, for fun, try timing your code with and without -O and marvel at the difference.

Your code will first be measured for correctness, ensuring that it zips input files correctly. How do you know whether your pzip works correctly? It is surprisingly easy. Just simply compare the output of `wzip` (project 1) and that of `pzip`.

If you pass the correctness tests, your code will be tested for performance; higher performance will lead to better scores. if you simply want to measure the CPU and/or wall time for your entire program, you

don't really need to change your code for that. Just write `time` before what you would usually write to run your program from the terminal command line. Then, when your program is done executing, the measured times will show up on the screen. Like this:

```
$ time ./MyProgram
Result: 2.00000000000000000000

real 0m5.931s
user 0m5.926s
sys 0m0.005s
```

2. Additional Requirement

Your submission should include a Makefile. The included Makefile controls the compilation of your code and produces the pzip executable.

Your submission should include a README. The included README should state the following: (1) Briefly answer the questions in Section 1.2 Design Considerations. For the last question "**How to coordinate multiple threads?**", you should state the semaphores and/or mutex locks used in your code as well as their respective purposes. (2) Briefly state strength and weakness of your pzip. (3) If it is a group project, the README should list team members and their respective contributions.

Your code should not have obvious memory leak. Your code should deallocate all semaphores and mutex locks used.

Your code should be well-commented. You do not have to comment every single line. Important segments of code should be commented for being readable.

Your code should not produce any debugging and testing messages. I understand that you may find debugging and testing messages helpful at some point. But be sure to turn them off in your assignment submission. Otherwise, penalty can be imposed on those unsolicited messages.

The program should be compiled and executed. If I cannot compile your code with the provided Makefile, your submission receives zero point. If the program immediately aborts abnormally once it is started, your submission receives zero point.

3. Summary

Pzip is a multi-threaded program that compresses a list of input files specified in the command line arguments in the listed order to stdout as described earlier. You should implement a producer-consumer synchronization with an unbounded buffer. You should also use semaphores for output ordering. Correctness is the top priority. Performance is also emphasized. Pzip should use proper synchronization to ensure files are compressed in the listed order and prevent deadlocks.

To run pzip, it must use the command-line arguments for input files:

./pzip input1.dat input2.dat input3.dat > output.dat

The goal of this assignment is to further your understanding in multithreading and synchronization. The secondary goal is to expose you to code optimization for better performance.

Last but not the least, **no unsolicited messages are allowed**. When your program output is redirected to a file, everything written to stdout will become part of that file.

4. Optional Requirement

Some of you will find the performance bottleneck due to sequencing outputs. This can be fixed if the consumer threads compress data into memory first and then write the memory to stdout in bulks (this may further introduce another producer-consumer synchronization). I intentionally leave this out due to complexity of memory management. However, I do not mind if you give a try after you finish this project.

5. Submission

Before submission, you should make sure that your code has been compiled and executed on `cs1.seattleu.edu` and your submission include all required files!

The following files must be included in your submission:

- `README`
- `*.h: all .h files`
- `*.c/cpp: all .c/cpp files`
- `Makefile`

You should create a package p3.tar including the required files as specified above, by running the command:

```
tar  -cvf   p3.tar  README  *.h  *.c  Makefile
```

Then, use the following command to submit p1.tar:

```
/home/fac/zhuy/zhuy/class/SubmitHW/submit3500   p3   p3.tar
```

If submission succeeds, you will see a message similar to the following one on your screen:

```
======================Copyright(C)Yingwu Zhu=========================
Mon Jan 13 12:43:34 PST 2020
Welcome testzhuy!
You are submitting hello.cpp for assignment p2.
Transferring file.....
```

Congrats! You have successfully submitted your assigment! Thank you!
Email: zhuy@seattleu.edu
==========

You can submit your assignment multiple times before the deadline. Only the most recent copy is saved.

The assignment submission will close automatically once the deadline passes. You need to contact me for instructions on your late submission. Do NOT email me your submission!

6. Grading Criteria

Label	Notes
a. Meeting submission requirements (2 pts)	• Your Makefile works properly. • All required files are included in your submission. If incomplete submission prevents your program from being executed, you may receive zero point on this assignment.
b. README & well-commented code (2 pts)	• README contains the required items • Code is commented so that your code is easy to follow
c. Functionality (10 pts)	Pzip should behave as specified. • Works with multiple input files • Uses multithreading • Use proper synchronization • Correctness • Performance
d. Resource management & Outputs (2 pts)	• No memory leaks (if applicable). • Deallocate semaphores and/or mutex locks used • No debugging/testing messages. • No messy messages scrambling on screen
e. Overriding policy	If the code cannot be compiled or can barely executed (segmentation faults on it starts, for instance), it results in zero point on this assignment.
f. Late submission	Please refer to the late submission policy on Syllabus.