Final Report

Gzip 2.0

Prepared by:

Arvind Ramachandran

Yang Zhao

Justin Hong

Shuhao Sun

Nilesh Gupta

Zachary King

Introduction

Background
Gzip is a popular data compression/decompression program, and the default on many Linux systems. The interface for this program is done through the command line and it offers several options for changing functionality. Gzip currently uses the Deflate algorithm to compress or decompress files and does this utilizing a single thread. This algorithm is written by Mark Adler himself, and is made of several files that are part of the current gzip source code.

Zlib is a compression/decompression algorithm library that implements, among others, the deflate algorithm. This library was also written by Mark Adler, and is meant to be a self-contained library for file compression/decompression.

Improvements
Currently, the compression code is handwritten and scattered throughout the program, and it only runs on one thread. Given that many machines nowadays support multiple threads, gzip can be expanded to utilize this. Parallelized compression offers greater performance in regards to speed when compared to a single thread implementation. However, the increased complexity of managing threads and ensuring they can all complete the overarching problem in unison adds a great deal of overhead. This project deals with implementing a multithreaded option to gzip in order optimize performance, while ensuring the rigidity of the program stays strong.

To implement parallelized gzip, we referenced Mark Adler's pigz project, which is his separate implementation of a parallelized compression program.

Another improvement we made is replacing the handwritten compression code that is currently part of the gzip source code with zlib. This way, all of the compression/decompression algorithm code will be self-contained in an outside library, making the gzip source code smaller and easier to read/maintain.

Features Description

In this quarter, we only managed to implement parallelized compression and a rewrite of gzip's compression and decompression with zlib. We added 2 options, the **--threads x** option, which specifies the number of threads to use when compressing in parallel, and **--independent**, which compresses each block of data independently, so that even if one block is corrupted, the rest of the file can still be decompressed and used.

- **Parallelized compression --** in gzip2, an user can use the --**threads x** option to compress their file with x number of threads in parallel, using multiple cores.

- **Rewrite of gzip's compression and decompression with zlib --** in gzip2, we have removed many lines of code in the original gzip that implemented compression and decompression functions and replaced them with short 100 line functions that use the zlib API to do compression and decompression. The compression and decompression functionalities are now also separated out into individual files: deflate.c and inflate.c, so that it's more organized.

Usage

There are two main use cases for a user of gzip2, and they are consistent with the use cases for standard gzip.

1. Compress
2. Decompress

Usage of gzip is often paired, meaning for every compressing there is a corresponding time it will be decompressed. One scenario compression is very useful is during network message transmissions. If the data being sent is compressed before transmitting, there is less strain on the network, freeing up space for other messages.

For heavy users, such kernel distributors that use gzip to compress their kernel images, compression usually takes a very long time. With gzip2, these users will be able to utilize all of the cores on their computers to compress in parallel and decrease the time it takes to compress dramatically.

Update:
We did not have time to implement the **--rsyncable** option to take advantage of parallelized compression, so we kept the old gzip's --rsyncable option.

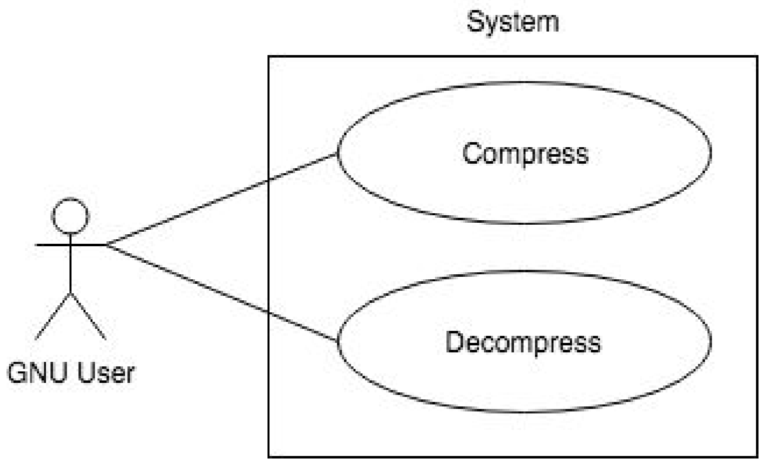


Figure 1. A use case diagram for gzip2

Interaction

The interface for gzip2 is a command line interface. Standardly, there are a variety of options that can currently be set for gzip. In order to add support for multithreading, while maintaining current functionality for single threaded machines, a few extra command line options must be made available to the users. The options are as follows:

Current gzip options		Options added to gzip 2
-c, --stdout, --to-stdout Write on standard output, keep original files unchanged **-d, --decompress, --uncompress** Decompress **-f, --force** Force overwrite of output file and compress links **-h, --help** Print help message describing the options then quit **-k, --keep** Keep (don't delete) input files **-l, --list** List compressed file contents **-L, --license** Display software license **-n, --no-name** Do not save or restore the original name and timestamp **-N, --name** Save or restore the original name and timestamp **-q, --quiet** Suppress all warnings	**-r, --recursive** Operate recursively on directories **--rsyncable** Make rsync-friendly archive **-S, --suffix=suf** Use suffix suf on compressed files **--synchronous** Synchronous output (safer if system crashes, but slower) **-t, --test** Test compressed file integrity **-v, --verbose** Verbose mode **-V, --version** Display version number **-#, --fast, --best** Regulate the speed of compression using the specified digit #, where 1 or fast indicates the fastest compression method (less compression) and 9 or best indicates the slowest compression method (best compression). 0 is no compression. The default is 6.	**-p --threads [n]** Allow n threads (default is the number of *online* processors).

After further examination of the pigz code and a more complete implementation of our gzip2, we have decided that the **--blocksize x** option, which sets the compression block size to x KiB (default 128KiB), is not needed. The code that implements the blocksize option is complicated and spread out inside of pigz, and the option serves little purpose except for special users in specific situations. Given our time constraints and the ratio of effort required vs functionality provided, we have decided to not include the blocksize option in our gzip2 implementation.

Also, the **-i, --independent** option that is present in pigz, which compresses blocks independently for damage recovery and removes the 32Kb preset dictionary that is usually added from the previous block, was not added to our version of gzip2. We did not have time to implement this option.

Finally, although the **--rsyncable** option (compresses the file into a format that is rsync-friendly, meaning that changes are isolated and minimized, so it can be rsync-ed faster) from the original gzip is kept, we were not able to complete our new implementation of the option. Our new implementation was supposed to take advantage of the parallelized compression, which splits the input file into individual compression jobs organically, thereby easily isolating the changes made to a file and minimizing changes to the overall compressed output.

Use Scenarios

1. Code distribution
    a. Computer Science professors must distribute files for programming projects to their students. By compressing those files before sending them to students, the file that students must download becomes smaller. This step is often the bottleneck to them acquiring these files so improvements here help the most.
2. Network packet switching
    a. When packets need to be routed throughout the internet, servers sometimes compress large packets so they can be transferred faster throughout the network and use less bandwidth. However, compression takes time, which adds to the delay user requesting the data experiences. By compressing faster, loading times can be decreased for internet users.
3. Kernel distributors
    a. Kernel images are very large files, and when kernel developers need to send their images to testers or to distribution, they often compress it so it is easier and faster to upload and download. Using gzip2, the kernel developers can compress the massive images faster, saving significant time.
4. Video editors
    a. Users who edit and store lots of video files on their computer will often want to compress those files so they take up less space. Video files are often very large, many are 10 to 20 Gb, so compressing them will take a very long time. Using our gzip2, users with a lot of video files can utilize the multiple cores on their computers (which they probably use for video processing too) to compress faster.
5. Data storage companies

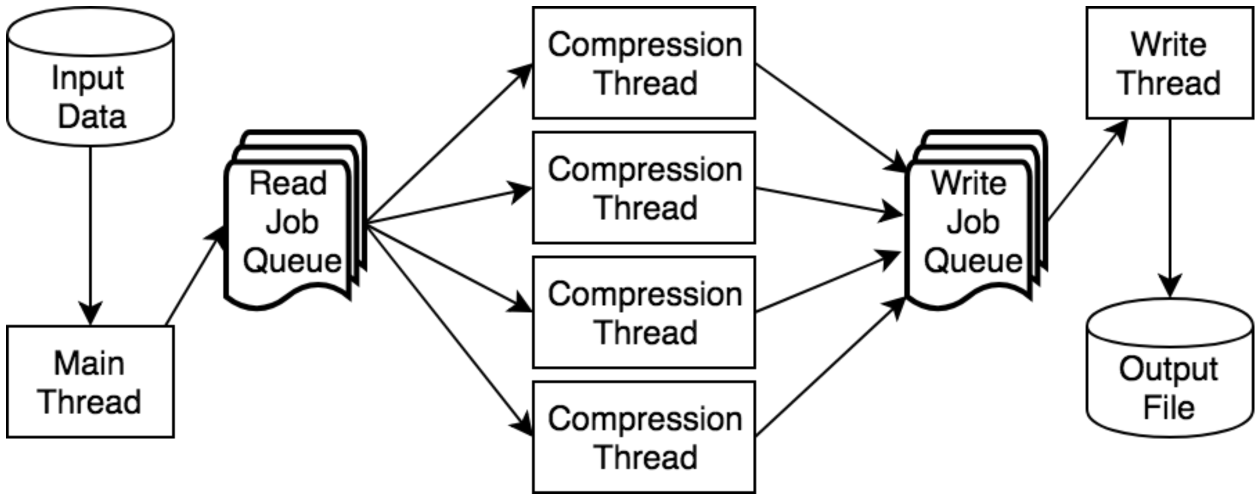
a. Many companies offer online long-term data storage services. In order to save space in their data centers and store more data on less hardware, they often compress the files that are stored on their servers. With a massive amount of data, these companies will save significant time if they are able to use their powerful computers with multiple cores to compress in parallel faster.

Design - Parallelized Architecture

In our midterm report, we planned on making the main thread a thread manager that would assign jobs to each of the compression threads periodically, but after further examination of the pigz code and our architecture, we decided that a thread manager was not needed. We decided on a looser coupling of the threads, based upon the ***Chain of Responsibility*** design pattern. The Sender role is played by both our main thread and our compress threads. To the compress threads, the main thread sends compress jobs for them to handle, and to the write thread, the compress threads send write jobs for it to handle. The different components of our Chain of Responsibility communicate through pushing and popping jobs from our compress and write job queues.

Definitions

- **Job** - A unit of work that a thread performs work on. For the main thread, the work is reading from the input file descriptor and filling in the fields of the job. For compression threads, the work is compressing the data within the job, and for the write thread, it is taking the compressed data and writing it to the output file descriptor in order.
- **Job Queue** - A thread safe data structure that supports a multitude of operations: adding a job to the beginning, add a job to the end, getting a job from the beginning, and getting a job with a desired sequence number.



In order to achieve parallelized compression, the new implementation for gzip utilizes three different types of threads to achieve this goal. The types of threads and their responsibilities in the overarching goal is as follows:

1. Main thread
    a. The main thread is responsible for reading from the input file descriptor, allocating new jobs and filling in their fields with the data received from the input.
    b. Once the job is allocated and filled, the main thread appends it to the end of the read job queue.
2. Compression Thread
    a. The compression threads take jobs from the beginning of the read job queue, and compresses the data for that job using the zlib compression library. The compression threads also calculates a CRC hash from the segment of data, which will be inserted into the write job along with the compressed data.
    b. Once jobs have been compressed by this thread they are added to the beginning of the write job queue.
3. Write Thread
    a. The write threads request jobs from the write job queue in order. For example the write thread first begins looking for job 0, and only once it has processed job 0 does it begin looking for job 1. For each job, the write thread writes its input to the destined file descriptor.

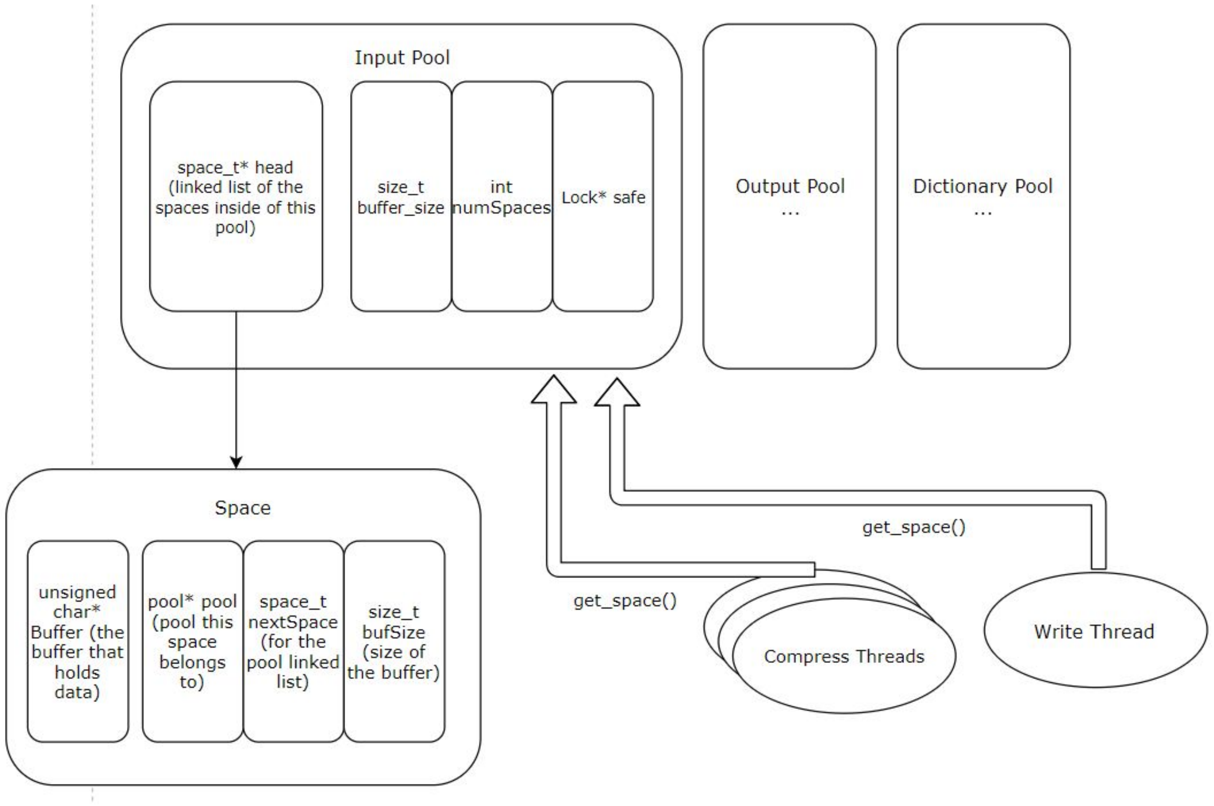
There is only one Main Thread and Write Thread, the parallelization happens in the Compress Threads. The user can specify the number of compress threads they want to use via the -p command line option. The thread safe property of the job queue ensures that each job is only received and compressed by a single compress thread, while the implementation of the write thread ensures that the work done by these threads eventually ends up in the correct ordering when placed into the output file descriptor.

Design - Buffer Pools

Another new architectural component that we added to our design are the buffer pools. With multiple threads operating in parallel, it’s very easy to generate and append too many blocks into the job queue, reading most of the input file into memory and taking up lots of memory.
To solve this problem, we keep track of the number of buffers being used and limit the maximum number of buffers created using buffer pools.

Definitions

- **Space -** a data structure that implements a buffer that can hold various types of data. The data structure also includes the necessary values and variables
- **Pool -** a collection of spaces, used to organize buffers into categories, such as buffers for input, output, and dictionaries used for compressing. Pools also contain the necessary locks and values that allow spaces to be acquired and returned in a thread-safe manner.

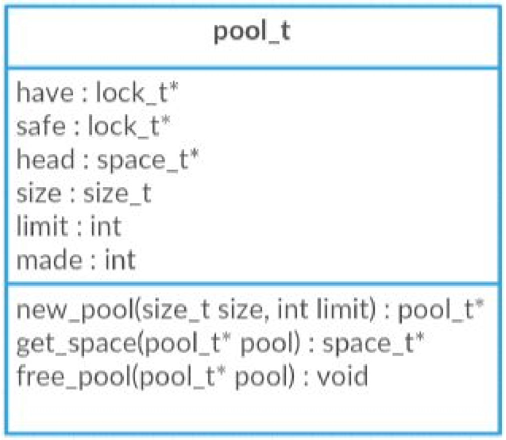

Input Pool
space_t* head (linked list of the spaces inside of this pool)
size_t buffer_size
int numSpaces
Lock* safe
Output Pool ...
Dictionary Pool ...
Space
unsigned char* Buffer (the buffer that holds data)
pool* pool (pool this space belongs to)
space_t nextSpace (for the pool linked list)
size_t bufSize (size of the buffer)
get_space()
get_space()
Compress Threads
Write Thread

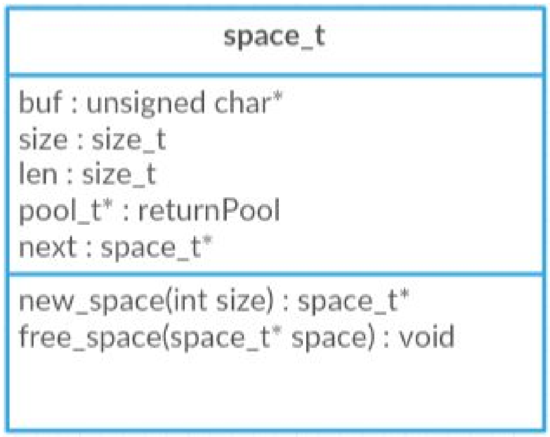
There are three pools in our implementation, one to hold the input spaces, one for the output spaces, and one for the dictionary spaces.

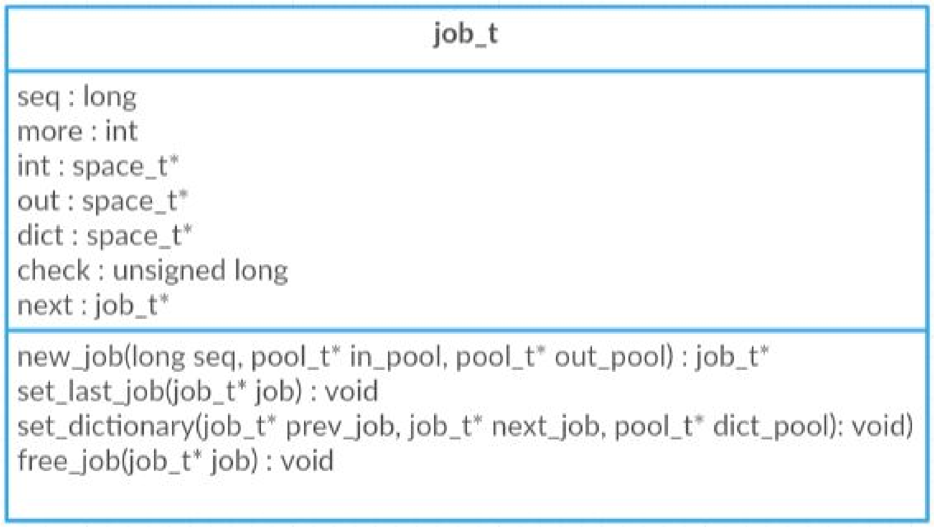
Each pool contains vital data about the pool itself and its spaces, such as the size and number of spaces. The most important part of the pool are the locks. These lock coordinate lock getting and returning between the threads, since multiple compress or write threads will be trying to get spaces at the same time.

The reasons for the pool and space structures are one, to have a uniform and abstracted interface for thread-safe buffer getting and returning. And two, to keep track of the number of buffers used, so that the main thread does not fill the job queue with too many jobs or threads do not allocate too many buffers and use too much memory.

Design - UML Class Diagrams for Key Structs


pool_t
have : lock_t*
safe : lock_t*
head : space_t*
size : size_t
limit : int
made : int
new_pool(size_t size, int limit) : pool_t*
get_space(pool_t* pool) : space_t*
free_pool(pool_t* pool) : void




job_t
seq : long
more : int
int : space_t*
out : space_t*
dict : space_t*
check : unsigned long
next : job_t*
new_job(long seq, pool_t* in_pool, pool_t* out_pool) : job_t*
set_last_job(job_t* job) : void
set_dictionary(job_t* prev_job, job_t* next_job, pool_t* dict_pool): void)
free_job(job_t* job) : void

Design - External Libraries Used

Zlib

- *int deflateInit2(z_streamp strm, int level, int method, int windowBits, int memLevel, int strategy)* -- used to initialize the deflate stream so that files can be compressed
- *int deflate(z_streamp strm, int flush)* -- used to compress data passed in through the deflate stream
- *int deflateEnd(z_streamp strm)* -- finishes the deflation and frees all allocated data structures for the particular deflate stream
- *int inflateInit(z_streamp strm)* -- used to initialize the inflate stream so that files can be decompressed
- *int inflate(z_streamp strm, int flush)* -- used to decompress the data passed in through the stream

- *int inflateEnd(z_streamp strm)* -- used to finish the inflation and frees all allocated data structures
- *int deflateSetDictionary(z_stream strm, const Bytef* dictionary, uInt dictLength)* -- sets the preset dictionary for the block for use in compression
- *ulong crc32(uLong crc, const Bytef* buf, uInt len)* -- calculates the CRC-32 hash value for the data in the buffer

Pthreads

- *int sem_init(sem_t* sem, int pshared, unsigned int value)* -- initializes a semaphore
- *int sem_wait(sem_t* sem)* -- waits on a semaphore
- *int sem_post(sem_t* sem)* -- increments the value of a semaphore
- *int sem_destroy(sem_t* sem)* -- destroys a semaphore

Testing

Link to tests directory with README:
https://github.com/arra1997/parallelized-gzip/tree/master/tests

Future Plans and Changes

In the future, we plan to completely rewrite gzip from scratch. The current code for gzip is scattered, outdated, and buggy. There are many improvements to be made in a complete rewrite of the program. In this rewrite, we would also implement the unfinished **--rsyncable** option and the --**blocksize** option, and also fix the **--test** option. We will also add extensive testing, make the program into the GNU coding style, and add extensive documentation, so that the program can be inducted into the official GNU repository as the new gzip2.0.

Individual Contributions

Arvind Ramachandran

- Wrote original 'small gzip' to test zlib API
- Wrote lock_t data structure and functions: new_lock(), get_lock(), release_lock(), increment_lock() and free_lock()
- Designed and implemented condition_t data structure and functions: new_condition(), free_condition(), reset_condition(), wait_condition() and signal_condition(), which lead to gzip being able to almost match pigz's compression time (within 2% on my tests)
- Wrote space_t data structure and functions: new_space() and free_space()

- Designed two-layer semaphore structure for pool_t data structure and implemented its functions: new_pool(), get_space(), drop_space(), free_pool(), so that threads looking for a new space would sleep until it becomes available
- Wrote job_t data structure and functions: new_job(), set_last_job(), set_dictionary(), load_job(), finish_processing() and free_job()
- Helped rewrite and debug job_queue's get_job_seq() function and implemented condition variable structure to prevent write_thread from spinning, and to allow compiler optimization; helped debug write_thread()
- Designed and implemented job_queue's 2 layer semaphore structure (similar to space_t), and added atomic countdown through num_threads so that new_job() returns NULL when queue is closed and empty and implemented close_job_queue()
- Added utils.h & utils.c functions that allow for commonly used functionality to be safe, by throwing exceptions in case of failure, and added many assert() cases throughout the code that lead to discovery of several bugs and race conditions.
- Fixed bug that resulted in overuse of memory and computing resources using valgrind massif and memcheck tools, and optimized memory usage & compression speed to make it similar to pigz
- Made write_opts and compress_opts structures and their associated initialization and free functions
- Modified allocations so that most of them happen in the beginning and failures occur earlier rather than later
- Implemented deflate_file_parallel function that allocates buffer pools, initializes jobs, sets dictionaries and creates worker threads
- Implemented --independent option, debugged many segfaults, race conditions and deadlocks
- SRS, Midterm Report, and Readme

Shuhao Sun

- Wrote inflation part of gzip.
- Achieve concatenation property of decompression of gzip.
- Wrote original version of job related functions and job queue structure.
- Simulate the of process partial compression and assembly of blocks of data and prove the validity of compressed file.
- Wrote most part of crc check value calculation.
- Wrote a big part of the compress thread in parallel compression which uses raw deflation method of zlib.
- Debug the pure function bug which causes over optimization.
- SRS, Midterm Report

Yang Zhao

- Fixed deflate_file's bug of only compressing part of the file, now using correct flush flag

- Updated and debugged Makefile.am with new dependencies, linked with zlib, so the correct Makefile is generated
- Wrote and debugged get_job_seq() function first version
- Wrote write_thread() function so that compressed jobs can be written to output file
- Wrote functions for manually writing headers and trailers of the output file (put(), writen(), put_header(), put_trailer())
- Wrote functions for initializing and finishing buffer pools
- Debugged compiler errors in main thread and simple_parallel branches
- Debugged deadlock and run-time errors in simple_parallel branch to generate correct output
- Midterm & Final: Presentation, Report
- Test cases, README
- SRS

Zach King

- Implementation of main logic for compress_thread() function
- Fixed several segmentation faults discovered when integrating thread-safe structures into compress thread functionality
- Debugged deadlock in main thread related to buffer spaces failing to properly get deallocated during usage.
- Debugging and testing of parallel integration into core gzip system
- Midterm & Final: Presentation, Report

Justin Hong

- Modified deflate function in ‘small’ gzip to use zlib’s init2 initialization function to create a gzip header and trailer so a correct gzip file is created
- Added --license, --version, --help, --quiet, --verbose, --suffix options to ‘small’ gzip and reformatted all our code using GNU coding standards
- Incorporated our zlib based deflate and inflate functions into gzip 1.9 and fixed issues with headers to accomplish our first goal of using zlib for compression and decompression in gzip
- Added display of compression ratio when deflating and inflating using zlib
- Debugged error checking in zlib-based inflate from standard input so many more test cases are passed
- Conducted performance testing of parallel gzip on files of various sizes and types
- Report, Presentation, and SRS

Nilesh Gupta

- Initial draft of SRS
- Midterm presentation & report