Project 2 Stage 2

In this stage you will add threads to Stage 1. (No longer requires using a thread pool. But if you've finished the stage, adding the pool is easy.) The key to this stage is to divide and conquer. To divide, you'll use `mmap` so as to reading a file from multiple spots. To conquer, you will `join` the threads and merge the results properly. (In all prior examples, we "returned" nothing from the threads.)

When to Parallelize the Task

If a file to compress is greater than 4096 bytes in size, you will use **3** threads to compress it. Otherwise, use the main thread only.

So if the input has three big files, you should expect 9 thread creations (we are not using a fixed-size thread pool).

If the input is `./wzip tests/1.in`, no thread creation (not counting the main thread). If the input is `./wzip tests/1.in tests/6.in`, 3 thread creations for `tests/6.in`. If the input is `./wzip tests/6.in tests/1.in tests/6.in`, 6 thread creations for `tests/6.in`.

How to Parallelize the Task

When the file is large, we divide and conquer. To get the size of a file, `man fstat`.

How to Divide

We can use `mmap` to map the file content to a contiguous chunk of memory. The 3 threads then can start reading from 3 different spots, e.g. one starts from `src`, the second from `src + some_offset`, the third from `src + some_more_offset`.

```
char *src = mmap(0, fsize, PROT_READ, MAP_PRIVATE, fd, 0);
```

Here `fsize` is the size of the file, `src` is the start address of the memory, `fd` is the file descriptor. Keep the others the same in your program.

How to Conquer: Merging

You shall use `pthread_join()` to wait for the completion of all the threads for each large file. You also need to merge results from different files. For the threads, you can update the arguments that you pass in from the threads. (It is also possible to return a pointer to a value as the "return value" from a thread.)

Your design shall include a C structure, call it `arg_t`. Note you don't need nested structures. I am using two structures to indicate that there could be multiple fields.

```
typedef struct __arg {
  struct arg_val_t arg_val; // information passed to the thread

  struct ret_val_t ret_val; // information passed back to the main thread
} arg_t;
```

```
// in the main thread; for a large file (> 4096 bytes)
pthread_t t;  // May use an array in the actual code
arg_t arg;
// initialize the arg, e.g. a pointer (with offset) to the mmapped file
init_arg(arg.arg_val, ...);

// worker will compress the file and write information to arg.ret_val
pthread_create(&t, NULL, worker, (void *) &arg);

pthread_join(t, NULL); // join them all AFTER creating them all

// now arg.ret_val has the information
// merge arg.retval with other threads

// done for a large file. Will need to merge the results from different files too
```

Remember you cannot simply concatenate the results, you need to check if the end of the first is the same letter as the start of the second.

Grading

If you submit a stage 1 solution, i.e. no attempt to use multiple threads at all, you may lose additional points at the instructor's discretion. Submissions will be manually inspected and adjustments may be made.

Special Tests:

- `valgrind` tests. Did you free heap memory properly?
- Tests to show that you create 3 threads for per large file (no threads for smaller files)
- Performance tests.

Regular (Data) Tests:

- Same tests in stage 1 plus test 7 and 8 which are just concatenations of 6.in

You will NOT get any points if you cannot pass all the regular tests first. The total is 100, evenly distributed to all special tests. Some erroneous submissions may escape the Autograder, in which case I may manually adjust the points.

About Performance Test

Your threaded program is expected to be faster than the single-thread one. It depends on the actual machine for how much faster it can be. For example, when my single-thread one spent 9 seconds to compress a large file on my old iMac, my multi-thread one runs under 3 seconds. However, the threaded one is only about 30% faster on Gradescope.

Both my single-thread and multi-thread solutions can finish a 120MB file within 1.3 seconds. I tried some of your submissions. Some will need more than 5 seconds to finish test 6 (~10MB). The test will be conducted by using the `time` command (getting the "real" time, which is not too scientific), e.g. `time ./wzip tests/6.in > /dev/null`.

The performance tests are conducted in steps. For example, "finish test 6 within 5 seconds" and then "finish test 6 within 3 seconds" and so on. The faster your wzip is, the more performance points you will have.