Julie Muzina
2713300
jumuzina
CIS 545 Project 1

```
jmuzina@jmuzina-ubuntu:~/source/repos/CSU/2023_Fall/CIS_545_ARCH_OS/cis_545_project_1$ make atomic; ./thr_atomic 8 16384
gcc thr_atomic.c -pthread -lm -o thr_atomic
Thread 0 finished calculating 2048 quadruple roots from 1 to 2048 with a sum of 11024.840697.
Thread 2 finished calculating 2048 quadruple roots from 4097 to 6144 with a sum of 17302.530491.
Thread 5 finished calculating 2048 quadruple roots from 10241 to 12288 with a sum of 21093.349127.
Thread 3 finished calculating 2048 quadruple roots from 6145 to 8192 with a sum of 18832.528312.
Thread 4 finished calculating 2048 quadruple roots from 8193 to 10240 with a sum of 20058.728158.
Thread 1 finished calculating 2048 quadruple roots from 2049 to 4096 with a sum of 15193.238893.
Thread 6 finished calculating 2048 quadruple roots from 12289 to 14336 with a sum of 21994.517317.
Thread 7 finished calculating 2048 quadruple roots from 14337 to 16384 with a sum of 22796.643460.
The sum of fourth roots from 1 to 16384 is 148296.376455.
```

Figure 1: Output of thr_atomic.c for 8 threads up to 16384.

```c
// Prepare semaphores to ensure mutex
sem_init(&dispatcher, 0, 1);
sem_init(&worker, 0, 0);

for (int threadNum = 0; threadNum < numThreads; ++threadNum) {
    pthread_t threadId;
    struct RootCalculationThreadArgs* args = malloc(sizeof (struct RootCalculationThreadArgs));
    args->threadNum = threadNum;
    args->numThreads = numThreads;
    args->upperBound = upperBound;
    const int err = pthread_create(&threadId, NULL, &calculateRoot, args);
    if (err) {
        printf("Failed to create thread %d", threadNum);
        exit(1);
    }
}

/** Signal all worker threads created; threads begin adding their partial sums to the shared total */
for (int threadNum = 0; threadNum < numThreads; ++threadNum) {
    sem_post(&dispatcher);
}

/*** Wait for all worker threads to be finished adding their partial sums to the shared total */
for (int threadNum = 0; threadNum < numThreads; ++threadNum) {
    sem_wait(&worker);
}

// All worker threads have added their partial sums to `total`. Print the result!
printf("The sum of fourth roots from %d to %d is %f.\n", 1, upperBound, total);

// Semaphore cleanup
sem_destroy(&dispatcher);
sem_destroy(&worker);
```

Figure 2: thr_atomic.c: Main thread's thread creation logic, and semaphore management

```c
void * calculateRoot(void *_args) {
    // Args pre-processing
    struct RootCalculationThreadArgs *args = (struct RootCalculationThreadArgs *) _args;

    // Define bounds of the computation
    const int lowerLoopBound = ((args->threadNum) * (args->upperBound / args->numThreads)) + 1;
    const int upperLoopBound = ((args->threadNum + 1) * (args->upperBound / args->numThreads));

    assert(upperLoopBound >= lowerLoopBound);

    /** Running sum of quadruple roots from lowerLoopBound to upperLoopBound */
    double threadTotal = 0;
      You, 5 hours ago • initial
    // Calculate the quadruple roots and accumulate resulting sum in `threadTotal`.
    for (int i = lowerLoopBound; i <= upperLoopBound; ++i) {
        threadTotal += pow(i, 1.0/4);
    }

    printf("Thread %d finished calculating %d quadruple roots from %d to %d with a sum of %f.\n", args->threadNum

    // Wait until no other worker threads are in their critical section and all threads have been created
    sem_wait(&dispatcher);
    // Enter the critical section; atomically update the total
    total += threadTotal;
    // End critical section
    // Allow another worker thread to enter their critical section
    sem_post(&dispatcher);
    // Inform the main thread that this worker has left its critical section
    sem_post(&worker);

    // Thread cleanup
    free(args);
    pthread_exit(NULL);
    return NULL;
}
```

Figure 3: thr_atomic.c: Thread function

```
jmuzina@jmuzina-ubuntu:~/source/repos/CSU/2023_Fall/CIS_545_ARCH_OS/cis_545_project_1$ make reduce; ./thr_reduce 8 16384
gcc thr_reduce.c -pthread -lm -o thr_reduce
The sum of fourth roots from 1 to 16384 is 148296.376455.
```

Figure 4: Output of thr_reduce.c for 8 threads up to 16384.

```c
void * calculateRoot(void *_args) {
    // Args pre-processing
    struct RootCalculationThreadArgs *args = (struct RootCalculationThreadArgs *) _args;

    // Define bounds of the computation
    const int lowerLoopBound = ((args->threadNum) * (args->upperBound / args->numThreads)) + 1;
    const int upperLoopBound = ((args->threadNum + 1) * (args->upperBound / args->numThreads));

    assert(upperLoopBound >= lowerLoopBound);

    /** Running sum of quadruple roots from lowerLoopBound to upperLoopBound */
    double threadTotal = 0;

    // Calculate the quadruple roots and accumulate resulting sum in `threadTotal`.
    for (int i = lowerLoopBound; i <= upperLoopBound; ++i) {
        threadTotal += pow(i, 1.0/4);
    }

    // store our partial total in the partialSums arr.
    partialSums[args->threadNum] = threadTotal;

    // Thread cleanup
    free(args);
    pthread_exit(NULL);
    return NULL;
}
```

Figure 5: thr_reduce.c: Thread function. Identical to the function for thr_atomic, but without semaphores, and the partial sum is stored in a global array.

```c
/** The number of times the partial sums array has been reduced in half */
int numReductions = 0;

// Loop over `partialSums`, changing it in-place by adding two elements and reducing its size in half each time.
while (numThreads > 1) {
    // Loop over the remainder of `partialSums`.
    // We start by picking a thread at the start of the array and one at the middle, and work our way to the end.
    for (int threadNum = 0; threadNum < numThreads / 2; ++threadNum) {
        const int partnerThreadNum = (numThreads / 2) + threadNum;
        // Only wait for the threads to finish if this is the first reduction.
        // After the first reduction, all threads have completed.
        if (!numReductions) {
            pthread_join(threadIds[threadNum], NULL);
            pthread_join(threadIds[partnerThreadNum], NULL);
        }
        // Store the sum of these two elements in `partialSums`, overwriting the previous value.
        partialSums[threadNum] = partialSums[threadNum] + partialSums[partnerThreadNum];
    }
    ++numReductions;
    numThreads /= 2;
    // Shrink all thread-related heap data accordingly
    if (numThreads > 0) {
        partialSums = realloc(partialSums, numThreads * sizeof(double));
        threadIds = realloc(threadIds, numThreads * sizeof(pthread_t));
    }
}

// At the end, there is only one element in the array, and it is the total.
const double total = partialSums[0];

// All worker threads have added their partial sums to `total`. Print the result!
printf("The sum of fourth roots from %d to %d is %f.\n", 1, upperBound, total);

free(partialSums);
free(threadIds);
```

Figure 6: thr_reduce: Partial sum reduction logic

Code Description

Input validation

For both programs, I begin with some input validation. The following conditions are all enforced on the command-line arguments:

- Correct number of arguments
- At least one thread
- Number of threads and upper bound are powers of 2
- Number of threads is less than or equal to the upper bound
    - If this is not true, then each thread will not have at least 1 integer to compute the 4^{th} root of.

Thread creation

I create *m* threads where *m* is the second command-line argument after the program name. Each one of these threads is supplied the following information in the thread function arguments:

- Thread number
- Total number of threads
- Calculation upper bound

The thread identifiers created by pthreads are then stored for later usage.

Atomic

Semaphores

The C library semaphore.h s used to provide mutual exclusion to thread critical sections. For the most part, the threads can run completely asynchronously. They can compute their partial root sums without waiting for their turn. However, when it comes time to atomically add to the global total, mutual exclusion is needed. Two semaphores are used:

- Dispatcher (initial value: 1):
  - Ensures:
    - Atomic reads and writes for the global total.
    - No worker threads enter their critical sections before all worker threads have been created.
  - wait() calls
    - Once by each worker thread, before reading and modifying the global total.
  - post() calls
    - By the main thread: once per each worker thread, after creating all worker threads.
    - By each worker thread: once, after updating the global total.
- Worker (initial value: 0)
  - Ensures:
    - Main thread waits for all worker threads to complete before printing final output.
  - wait() calls
    - By the main thread: once per each worker thread, after signaling all worker threads to start.
  - post() calls
    - By each worker thread: once, after updating the global total.

After all worker threads have signaled their completion, the main thread reads and displays the value of the global total.

Reduce

The reduce program is very similar to the atomic program, but each thread stores its partial sum in a globally shared array. Thus, there is no critical section in the thread functions (no volatile memory is being overwritten or used in a calculation), and semaphores are not needed. Threads store their partial sum in the global array using their thread identifier as an index.

The partial sums array is then systematically reduced in half until it contains just one element, the final sum. In each reduction, pairs of partial sums are selected, added, and re-stored in the partial sums array. On the first reduction, pthread_join() is used to wait for the two threads to complete their partial sum calculation. When the array has been reduced to just one element, the final answer is in the first element.

Debugging/Testing experience

This was my first notable hands-on C experience in quite some time so I spent some time first getting familiar with the basics of input validation, setting up a Makefile, and memory management. I then set about creating threads and passing values into them. I found that if I wanted to pass anything more than a single value into a thread, I needed to create a dynamically allocated structure to pass in.

The most time-consuming part of debugging was figuring out exactly where to place the wait() and post() calls for the atomic program's semaphores, as well as what to set their initial values to. I found the semaphore.h library to be very intuitive and useful, I just needed a bit more experience with implementing mutex with semaphores which this project gave me!

Code

Full source code is included with this submission in a .zip file, as well as a README file.