Jacob Saltzman
23 March 2017
CS 4414: Spring 2017

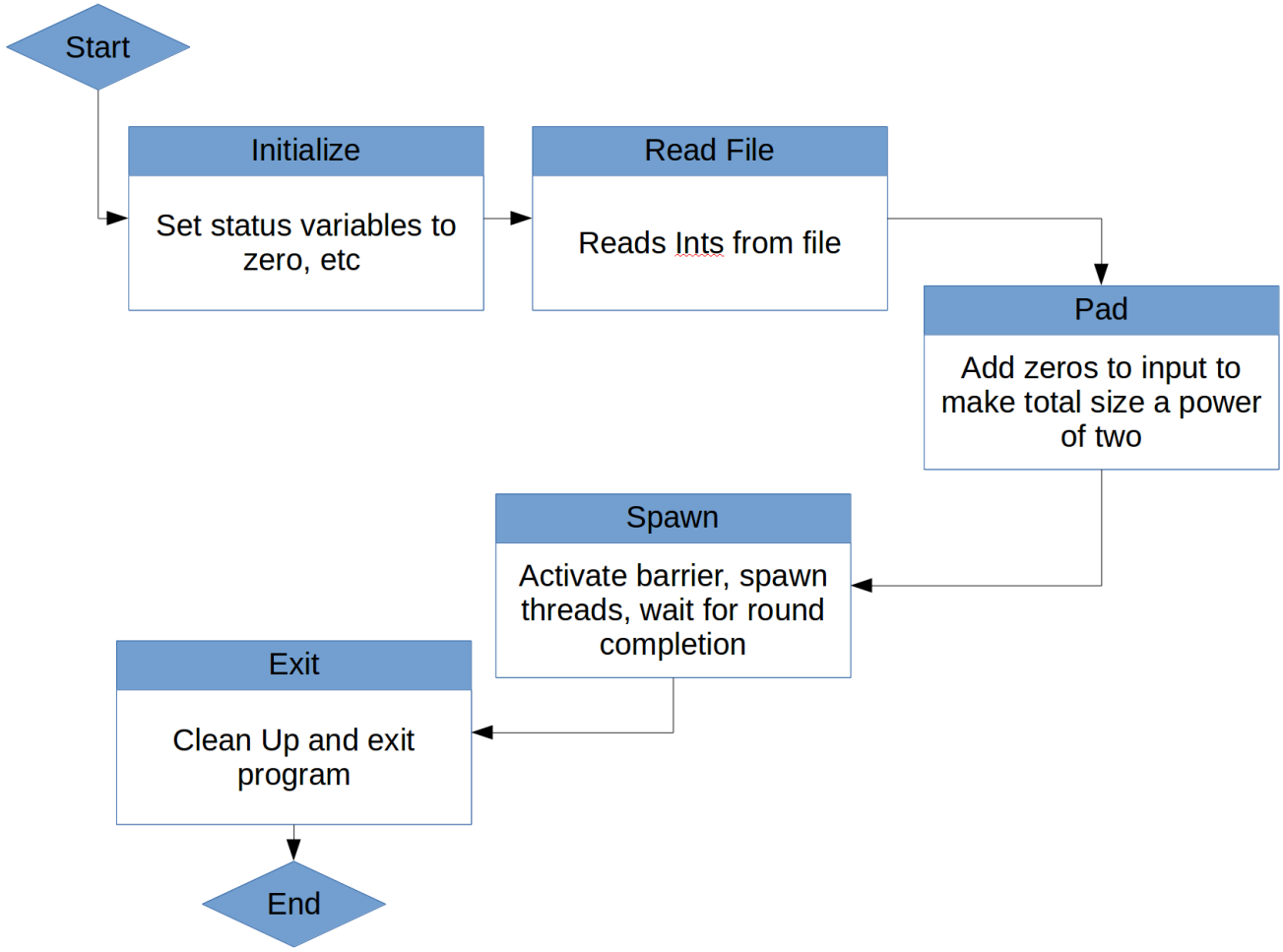
Project 2: Quicksort using Threads

Purpose

The purpose of this lab was to implement the quicksort algorithm using the pthreads library on the Linux operating system.

General Design

Like all good programs, I organized the program as a state machine, whose diagram is shown below:



Since the quicksort algorithm defined by the specification requires the list to be a power of two, the Pad state adds enough zeros to input in order for the total size to be a power of two. This is acceptable since adding zero to a list does not change the max.

Barrier Design

The barrier I used for multithreading is very simple, and is found in barrier.h. It consists of three variables, namely: a polling boolean which is 1 if the barrier is closed and 0 of it is open; a count variable which keeps track of how many threads are created; and a count varibale which is the number of threads the barrier should wait for before opening. It also contained a private pthread mutex to lock the incrementing variable so two threads did not try to modify it simultaneously. A simple API for the barrier was created as follows:

barrier.h

```
#include <pthread.h>

typedef struct {
      int barrier_active;
      int barrier_counter;
      int barrier_max;
      pthread_mutex_t barrier_mutex;
} barrier_t;

void ResetBarrier(barrier_t* b, int max_count);
void IncrementBarrier(barrier_t* b);
int PollBarrier(barrier_t* b);
```

The benefit of the barrier approach is that it abstracts away some of the complexities of multithreading from the viewpoint of the quicksort algorithm. Thus, we can write our algorithm using calls to the barrier versus calls to pthread directly.

Threading Strategy

The strategy for thread spawning I used was fairly simple. For each round, the number of threads to spawn was precomputed based on the nearest power of two and the count of the round. Then, an array of type pthread_t was dynamically allocated. Next, I looped over the threads and passed the appropriate parameters to the threads, which were passed the LargerInt() function. The pseudocode for this is shown below:

```
while(number_of_threads > 1):
      for i in range 0 to number_of_threads – 1:
            params[i] = {i , input_ints[2*i], input_ints[2*i + 1]}
            pthread_create(tids[i], NULL, LargerInt, &params[i])

      while(PollBarrier(&barrier)); // block the master until the barrier is open
      number_of_threads = number_of_threads / 2;
```

The comparison method simply did the comparison and incremented the current barrier count:

```
LargerInt(params)
      int retval;
      if(params.i1 > params.i2)
            retval =  params.i1
      else
            retval = params.i2
      IncrementBarrier(&barrier);
      return retval
```

This is a nice approach since nowhere in quicksort.h are mutexes or semaphores explicitly called.

Challenges Encountered

Debugging the barrier was difficult for a number of reasons. I encountered many challenges using GDB, including the fact that sometimes the program would work in GDB but not outside of it. I figured this was due to a race condition, which GDB might be able to avoid since it runs so much slower than normal execution. However sometimes the reverse occurred, where the program would run in normal execution but hang in GDB. This behavior eventually corrected itself, but I am still not sure what would cause such a problem.

Testing

The algorithm was well tested, and all tests passed. Text files were generated via bash using the seq and shuf commands. The results are summarized below:

Test File	**Test Description**	**Passed?**
empty_test.txt	Empty file w/ whitespace	Yes: Prints out error message
reps_test_55.txt	2034 integers on the interval of 0 and 55	Yes
lotsaints_test_4095.txt	All integers 0-4095, shuffled	Yes
toomany_ints.txt	All integers 0-5000, shuffled	Yes: Prints out error message
all333_test.txt	The number 333, repeated 401 times	Yes

These testing files can all be found in the root project directory.