Jacob Saltzman

23 March 2017

CS 4414: Spring 2017

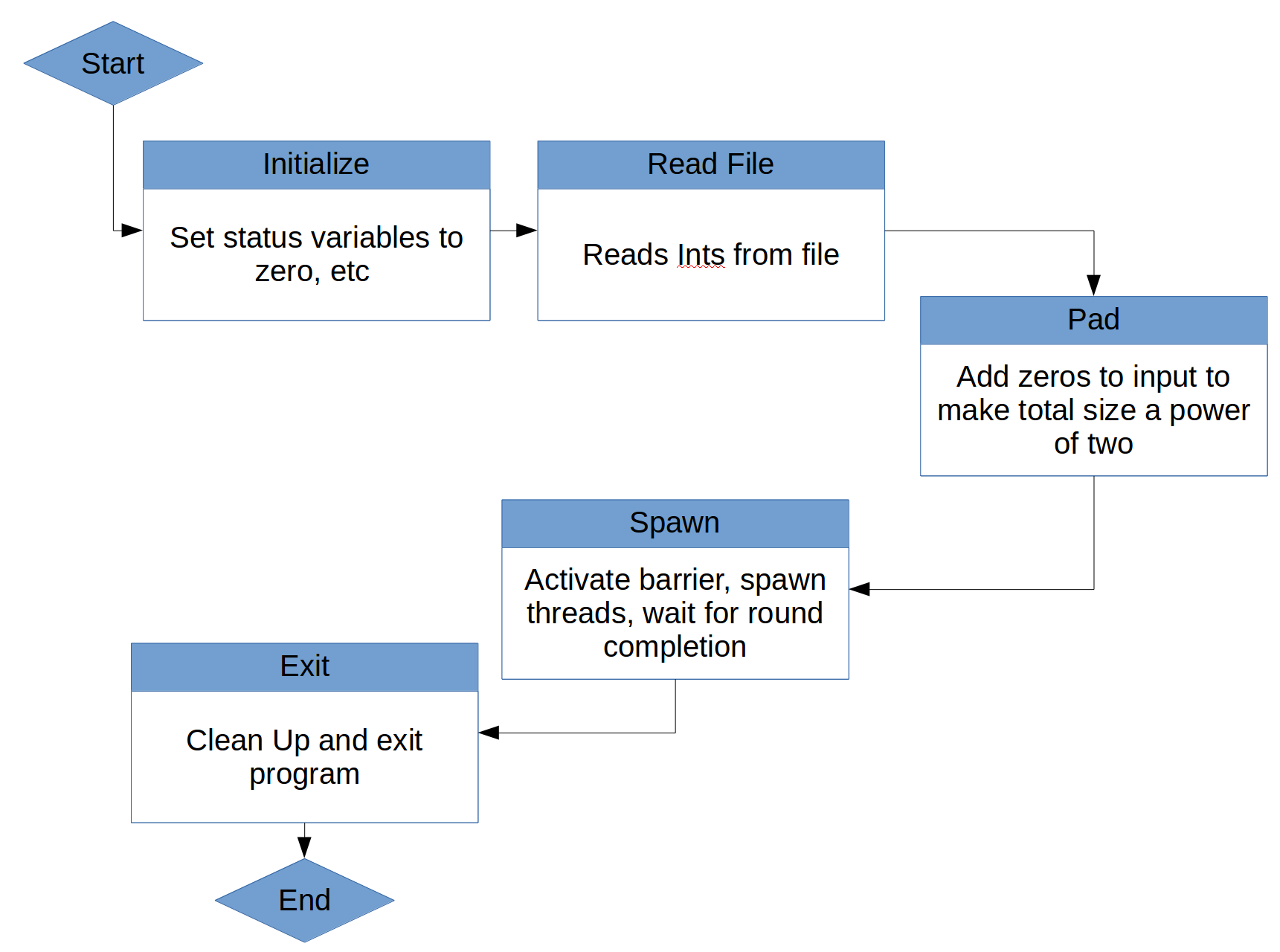
**Project 2: Quicksort using Threads**

**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 DecrementBarrier(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 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;