```
/*****************************************************************************
 * Lab 4 - Multi-threading
 *
 * This program will accept a user input (a filename) and spawn a thread
 * that will simulate a file lookup.
 *
 * As per directions, the total number of requests serviced are reported
 * as well as the average file lookup times that occur on all threads.
 *
 *
 * @author Ron Rounsifer
 * @version 0.05
 *****************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <string>
#include <iostream>
#include <fcntl.h>
#include <mutex>
#define MAX 1000

using namespace std; // not safe, but it's just a lab

/* method the thread will execute */
void *retrieve_file(void *arg);

/* signal handler for when ^C is passed */
void sig_handler(int);

/* global variables for exiting server and counting requests */
int serving = true;

/* counter for number of requests received */
int total_num_requests = 0;

/* counter for number of threads that are currently spawned */
int num_threads = 0;

/* total execution time of threads */
double total_thread_time;

/* avg execution time of threads */
double avg_thread_time;

/* mutual exclusion for timing threads */
mutex m;


/*****************************************************************************
 * Main entry point of program.
 *****************************************************************************/
int main()
{
  void *result;

  /* set the stdin to be blocking */
  int saved_flags = fcntl(0, F_GETFL, 0);
  saved_flags &= ~O_NONBLOCK;
  fcntl(0, F_SETFL, saved_flags);

  /* status of the thread spawned */
  int status;
  /* the actual thread */
  pthread_t thread;
  /* all threads */
  pthread_t all_threads[MAX];
  /* file name entered by user */
  string file;
```

```
  /* temp file location that is sent to the thread */
  string temp_file;
  /* buffer that can handle 1000 threads */
  string file_buffer[MAX];
  /* counter of files that have been processed */
  int file_count = 0;

  // parent thread that handles incoming requests
  // exits when ^C is passed.
  while(serving)
  {
    signal(SIGINT, sig_handler);
    cout << "\nfilename to access: ";
    getline(cin, file);
    file_buffer[file_count] = file;
    if (strlen(file.c_str()) > 0)
    {
      // create and execute retrieve_file function in its own thread
      if ((status = pthread_create(&thread, NULL, retrieve_file, &file_buffer[file_c
ount])) != 0)
      {
        fprintf(stderr, "thread creation error %d: %s\n", status, strerror(status));
        exit(1);
      } else {
        all_threads[file_count] = thread;
      }

      // used to track file buffer index
      file_count++;
      if (file_count == MAX)
      {
        file_count = 0;
      }
    }
  }

  // join all the threads
  for (int i = 0; i < file_count; ++i)
  {
   if ((status = pthread_join(all_threads[i], &result)) != 0)
   {
     cerr << "join error: " << strerror(status) << endl;
     exit(1);
   }
  }
  // make sure stdin is set to blocking
  fcntl(0, F_SETFL, saved_flags);
  return 0;
}


/**********************************************************************
 * Function ran when a thread is spawned.
 * Simulates a file lookup.
 *
 * @params void* - list of arguments thread needs
 **********************************************************************/
void *retrieve_file(void *arg)
{
  srand(time(0));
  string *filename = (string *)arg;
  total_num_requests++;
  // there is an 80% chance to sleep for 1 second,
  // 20% chance to sleep for 7-10 seconds
  int time_key = (rand() % 10) + 1;

  time_t start = time(0);
  if (time_key <= 8)
  {
    sleep(1);
  } else {
    int time_to_sleep = (rand() % 3) + 7;
    sleep(time_to_sleep);
```

```
  }
  time_t end = time(0);

  // lock code when modifying shared variables
  m.lock();
  total_thread_time += difftime(end, start);
  cout << "\nFile accessed: " << *filename << endl;
  num_threads--;
  m.unlock();

  return NULL;
}


/*****************************************************************************
 * Used to catch the interrupt signal (^C) send by the user.
 *
 * Reports the total number of requests, total thread time, and average
 * thread time.
 *
 * Changes the parents running status to false.
 *
 * Sets the stdin to be non-blocking (so you do not need to enter newline
 * to actually exit parents loop, due to readline())
 *
 * @params int - the interrupt signal number
 *****************************************************************************/
void sig_handler(int sigNum)
{
  // this catches the ^C interrupt
  if (sigNum == SIGINT)
  {
    // calculate avg thread time
    avg_thread_time = total_thread_time / (double) total_num_requests;

    cout << "\nNumber of requests served: " << total_num_requests << endl;
    cout << "Total file access time: " << total_thread_time << " seconds" << endl;
    cout << "Average file access time: " << avg_thread_time << " seconds" << endl;
    // breaks out of loop for parent thread
    serving = false;
    int saved_flags = fcntl(0, F_GETFL, 0);
    saved_flags |= O_NONBLOCK;
    // set the stdin to non-blocking
    fcntl(0, F_SETFL, saved_flags);
  }
}
```