

# ECE-C353: Systems Programming

## Final Project

Sean Barag  
<sjb89@drexel.edu>

December 4, 2011

## Contents

<b>1</b>	<b>Design</b>	<b>2</b>
1.1	Header File . . . . .	2
1.2	Main File . . . . .	3
1.2.1	Main . . . . .	3
1.2.2	Single-Threaded . . . . .	4
1.2.3	Multi-Threaded . . . . .	4
1.2.4	Worker Thread . . . . .	6
1.2.4.1	Setup . . . . .	6
1.2.4.2	The Loop . . . . .	7
<b>2</b>	<b>Results</b>	<b>11</b>
<b>A</b>	<b>Scripts</b>	<b>12</b>
A.1	Data Acquisition . . . . .	12
A.2	Data Processing . . . . .	12
A.3	Data Calculation . . . . .	13
<b>B</b>	<b>License</b>	<b>14</b>

# 1 Design

The design for this multi-threaded recursive text search was based mostly on provided code, but included heavy modification to prevent deadlocks.

## 1.1 Header File

All global instance, function, and type definitions were placed in the header file `work_crew.h`, to improve the readability of the main source file `work_crew.c`.

```

1 | int MAX_THREADS;
2 | int number_sleeping = 0;
3 | int all_done = 0;
4 | int spawned = 0;
5 | pthread_t *threads;
6 |
7 | pthread_cond_t wake_up;
8 | pthread_mutex_t num_sleeping;
9 | pthread_mutex_t *sleepers;
10| pthread_mutex_t done;

```

These objects were declared globally, with the following intended meanings:

**MAX\_THREADS** Maximum number of threads to possibly create.

**num\_threads** Number of threads that have been spawned at any time.

**number\_sleeping** Number of spawned threads that are currently sleeping.

**all\_done** Boolean-like variable indicating if the entire origin directory has been searched.

**spawned** Boolean-like variable indicating if other threads have been spawned yet.

**threads** Pointer to array of type `pthread`, containing one element for each thread spawned.

**wake\_up** Signal to wake threads up from sleeping.

**num\_sleeping** Mutex protecting `number_sleeping`.

**sleepers** Pointer to array of mutexes protecting `number_sleeping`.

**done** Mutex protecting `all_done`.

In order to provide each thread with the appropriate information, a custom datatype was created as shown below.

```

12 | /* structure for thread arguments */
13 | typedef struct thread_args
14 | {
15 |     int threadID;
16 |
17 |     queue_t *queue;
18 |     pthread_mutex_t *mutex_queue;
19 |
20 |     int *num_occurrences;
21 |     pthread_mutex_t *mutex_count;
22 |
23 |     char **argv;
24 |
25 | } THREAD_ARGS;

```

Similar to previous assignments, the following pieces of information were passed to each thread:

**threadID** Integer identifier for the thread. The first thread spawned will receive ID 0, and subsequent threads will receive 1, 2, ..., NUM\_THREADS - 1.

**queue** Pointer to the queue of files and directories to search.

**mutex\_queue** Pointer to the mutex protecting the work queue.

**num\_occurrences** Pointer to the integer representing the number of occurrences of the target string.

**mutex\_count** Pointer to the mutex protecting num\_occurrences.

Additionally, a custom datatype was created to ease the process of generating scriptable output.

```
27 | /* structure for return values
28 | *
29 | * these make scripting the output easier */
30 | typedef struct return_type
31 | {
32 |     int count;
33 |     float time;
34 | } RET_TYPE;
```

While these instance variables are fairly self-explanatory, they were created with the following intended meanings:

**count** The number of occurrences of the target string found by the returning function.

**time** Amount of time required to execute the returning function.

Finally, the non-main functions were declared:

```
36 | /* Function prototypes */
37 | RET_TYPE* search_for_string_serial(char **);
38 | RET_TYPE* search_for_string_mt(char **);
39 | void *worker_thread( void * );
```

in which `search_for_string_serial` is the single-threaded implementation, `search_for_string_mt` is the multi-threaded implementation, and `worker_thread` is the function executed by each thread.

## 1.2 Main File

### 1.2.1 Main

The main entry point for the search test is, as is expected, the `main` function, shown below.

```

24 int main(int argc, char** argv)
25 {
26     if(argc < 3)
27     {
28         printf("Usage: %s <search_string> <path> [num_
                threads]\n", argv[0]);
29         exit(0);
30     }
31     if( argc == 4 )
32         MAX_THREADS = atoi(argv[3]);
33     else
34         MAX_THREADS = 8;
35
36     // Perform a serial search of the file system
37     RET_TYPE *a = search_for_string_serial(argv);
38
39     // Perform a multi-threaded search of the file
        system
40     RET_TYPE *b = search_for_string_mt(argv);
41
42     printf("\n\n");
43     printf("Threads\tSingle\tMulti\tMatch\n");
44     printf("Scriptable output: %d\t%f\t%f\t%s\n",
        MAX_THREADS, a->time, b->time, (a->count ==
        b->count) ? "true" : "false");
45     exit(0);
46 }

```

The function first checks the command line arguments to ensure that it has sufficient information to run — a string and a target directory. If this information is not provided, the usage syntax is printed to the screen and the application exists. Additionally, if the number of threads is not provided, it defaults to eight. Once this initial validation is complete, the single- and multi-threaded implementations are executed, and a script-safe output in the format of the following example:

	Threads	Single	Multi	Match
Scriptable output:	8	3.292550	3.163284	true

This provides the number of threads, execution time for both the single- and multi-threaded implementations, and whether or not the number of occurrences for the two implementations matches in an easily-parsable format. While this was not a requirement for the assignment, it made the process of acquiring and analyzing the resulting data much easier.

### 1.2.2 Single-Threaded

There were no appreciable changes made to the single-threaded implementation. While the return type was changed to `RET_TYPE` and there were slight modifications to the post-search code, the searching portions of the function remained unmodified from the provided version.

### 1.2.3 Multi-Threaded

The multi-threaded approach to string search begins by allocating space for the threads array and declaring and initializing the relevant mutexes.

```

194 RET_TYPE* search_for_string_mt(char **argv)
195 {
196     /* allocate space for threads */
197     threads = (pthread_t
198                *)malloc(sizeof(pthread_t)*MAX_THREADS);
199
200     /* initialize mutexes */
201     pthread_mutex_t queue_mutex, count_mutex;
202     pthread_mutex_init( &queue_mutex, NULL );
203     pthread_mutex_init( &count_mutex, NULL );
204     pthread_mutex_init( &num_sleeping, NULL );
205     pthread_mutex_init( &done, NULL );
206
207     pthread_cond_init( &wake_up, NULL );
208
209     /* create mutex array */
210     sleepers = (pthread_mutex_t *)malloc(
211                 sizeof(pthread_mutex_t)*MAX_THREADS );
212     for( int i = 0; i < MAX_THREADS; i ++ )
213         pthread_mutex_init( &sleepers[i], NULL );
214
215     /* initialize count */
216     int count = 0;
217     int *p_count = &count;
218
219     /* create and fill thread arguments */
220     THREAD_ARGS *t_args;
221     t_args = (THREAD_ARGS
222               *)malloc(sizeof(THREAD_ARGS));
223
224     t_args->threadID = 0;
225     t_args->mutex_queue = &queue_mutex;
226     t_args->mutex_count = &count_mutex;
227     t_args->num_occurrences = p_count;
228     t_args->argv = argv;

```

Next, an array of mutexes for sleeping threads is created, as well as a series of counters. Finally, an instance of `THREAD_ARGS` is created and filled with the appropriate values.

At this point in the execution, a timer is created and started, and an initial thread is spawned.

```

228     /* start timer */
229     struct timeval start;
230     gettimeofday(&start, NULL);
231
232     /* spawn thread */
233     if( pthread_create( &threads[0], NULL,
234                        worker_thread, (void *)t_args ) != 0 )
235     {
236         printf("E: could not create thread!\n");
237         Exiting\n");
238         exit(-1);
239     }

```

After spawning the first thread, the multi-threaded control function simply waits for all threads to return, at which point it stops the timer, calculates the execution time, and prints its final search results and execution time.

```

239     /* wait for all threads to return */
240     for( int i = 0; i < MAX_THREADS; i++ )
241         pthread_join( threads[i], NULL );
242
243     /* stop timer */
244     struct timeval stop;
245     gettimeofday(&stop, NULL);
246
247     /* create return value */
248     RET_TYPE *out = (RET_TYPE
249         *)malloc(sizeof(RET_TYPE));
250     out->time = (float)(stop.tv_sec - start.tv_sec +
251         (stop.tv_usec - start.tv_usec)/(float)1000000);
252     out->count = count;
253
254     /* print results */
255     printf("\n\n");
256     printf("Overall execution time = %fs.\n",
257         out->time);
258     printf("The string %s was found %d times within\n",
259         the_file_system, argv[1], count);
260     printf("=====\n\n");
261
262     return out;
263 }

```

## 1.2.4 Worker Thread

**1.2.4.1 Setup** Each thread begins simply by casting the arguments of `worker_thread` from void (as required by the pthread libraries) back to type `THREAD_ARGS`.

```

263     THREAD_ARGS *l_args = (THREAD_ARGS *) args;
264
265     queue_element_t *element, *new_element;
266     struct stat file_stats;
267     int status;
268     DIR *directory = NULL;
269     struct dirent *result = NULL;
270     /* allocate memory for the directory structure */
271     struct dirent *entry = (struct dirent
272         *)malloc(sizeof(struct dirent) + MAX_LENGTH);

```

As in the single-threaded case, two queue elements are first declared as well as the relevant objects for any directory information that will be encountered.

If the calling thread is the first one spawned (i.e. with a `threadID` of zero), the shared work queue is allocated and initialized.

```

274     if( l_args->threadID == 0 )
275     {
276         /* Create and initialize the queue data
277            structure. */
278         l_args->queue = create_queue();
279         element = (queue_element_t
280                  *)malloc(sizeof(queue_element_t));
281         if(element == NULL)
282         {
283             printf("E: Error allocating memory.
284                Exiting.\n");
285             exit(-1);
286         }
287         strcpy(element->path_name, l_args->argv[2]);
288         element->next = NULL;
289         /* Insert the initial path name into the queue
290            */
291         insert_in_queue(l_args->queue, element);
292     }

```

While this could have also been performed before any threads were spawned, it was included here in the interest of clarity. Testing for the thread's ID happens once per thread and requires only  $O(1)$  time, so this causes no significant slowdown.

**1.2.4.2 The Loop** While the implementation of the searching algorithm required modifications in order to prevent deadlocking and data corruption, its overall flow and operation remained largely unchanged. For this reason, only the changes will be discussed here.

Each thread begins by removing the first item of the queue. Because all threads access and modify the same queue, the mutex referenced by `mutex_queue` must be locked prior to access and unlocked after in order to prevent conflicts.

```

291     while( 1 )
292     {
293         pthread_mutex_lock( l_args->mutex_queue );
294         /* While there is work in the queue, process
295            it. */
296         if(l_args->queue->head != NULL)
297         {
298             queue_element_t *element =
299                 remove_from_queue(l_args->queue);
300             pthread_mutex_unlock( l_args->mutex_queue );

```

While the single-threaded implementation tested the state of the queue in the `while` loop's condition, the test was moved to the first operation inside the loop so that the queue could be properly protected by its mutexes.

The next change occurs when the end of a directory is reached, in which the initial thread spawns its siblings.

```

329         if(result == NULL)
330         {
331             if( l_args->threadID == 0 &&
332                !spawned )
333             {
334                 spawned = 1;
335                 THREAD_ARGS *t_args;
336                 for( int i = 1; i <
337                    MAX_THREADS; i++ )
338                 {
339                     t_args = (THREAD_ARGS
340                               *)malloc(sizeof(THREAD_ARGS));
341                     t_args->threadID = i;
342                     t_args->queue =
343                         l_args->queue;
344                     t_args->mutex_queue =
345                         l_args->mutex_queue;
346                     t_args->mutex_count =
347                         l_args->mutex_count;
348                     t_args->num_occurrences
349                         =
350                         l_args->num_occurrences;
351                     /* NULL argv implies
352                        it's a worker
353                        thread */
354                     t_args->argv =
355                         l_args->argv;
356
357                     if( pthread_create(
358                         &threads[i], NULL,
359                         worker_thread,
360                         (void *)t_args )
361                        != 0 )
362                     {
363                         printf("E: could
364                                not create
365                                thread %d!
366                                Exiting\n");
367                         exit(-1);
368                     }
369                 }
370             }
371
372             break; // End of directory
373         }

```

Each thread tests its ID and whether or not the extra threads have been spawned. If it is thread zero and the sibling threads haven't been created (as indicated by the value of the `spawned` variable), then an instance of `THREAD_ARGS` is created and filled prior to each thread's creation.

For each new element reached by a thread, the queue must be locked and subsequently unlocked in order to contribute to the work queue.



```

370         /* Construct the full path name
           for the directory item stored
           in entry. */
371         strcpy(new_element->path_name,
372               element->path_name);
373         strcat(new_element->path_name,
374               "/");
375         strcat(new_element->path_name,
376               entry->d_name);
377
378         /* safely add to queue */
379         pthread_mutex_lock(
380             l_args->mutex_queue );
381         insert_in_queue(l_args->queue,
382             new_element);
383         pthread_mutex_unlock(
384             l_args->mutex_queue );
385
386         /* wake up others
387          * man, pthread_broadcast never
388          works for me */
389         for( int i = 0; i < MAX_THREADS-1;
390             i++ )
391             pthread_cond_signal( &wake_up
392                                 );

```

Once the newly constructed queue element has been added and the queue unlocked, all remaining threads are woken up with an iterated call to `pthread_cond.signal`. While `pthread_cond.broadcast` should be used to wake multiple threads that are waiting on one signal, it does not work in this case — this is most likely caused by the fact that each thread locks a different mutex prior to calling `pthread_cond.wait`.

In the event that the queue is empty, a thread simply unlocks the queue's mutex and locks the mutexes protecting the number of sleeping threads and the completion status.

```

434     else
435     {
436         pthread_mutex_unlock( l_args->mutex_queue
437                               );
438         pthread_mutex_lock( &num_sleeping );
439         pthread_mutex_lock( &done );
440
441         if( all_done == 1 )
442         {
443             pthread_mutex_unlock( &done );
444             pthread_mutex_unlock( &num_sleeping );
445             pthread_exit(0);
446         }

```

If the process is complete, the two mutexes are unlocked (in the reverse order of locking to prevent deadlocks) and the thread exits. If the process is not complete but the currently-executing thread is the only remaining thread awake, it sets the completion status to true before unlocking the two previously locked mutexes.

```

446         else if( number_sleeping == (MAX_THREADS -
447             1) )
448         {
449             all_done = 1;
450             pthread_mutex_unlock( &done );
451             pthread_mutex_unlock( &num_sleeping );
452
453             /* wake everyone up
454              * man, pthread_broadcast never works
455              * for me */
456             for( int i = 0; i < MAX_THREADS-1; i++ )
457                 pthread_cond_signal( &wake_up );
458             pthread_exit( 0 );
459         }

```

As it is the only thread not sleeping, the executing thread serially wakes the remaining threads before exiting.

For the case that a thread is not the last one awake and the search process is not complete, it (safely) increments the number of sleeping threads. It next locks the mutex protecting its sleeping state and safely checks the completion state to ensure that the search hasn't completed since its last check. This prevents a thread from entering an eternal sleep.

```

458         else
459         {
460             pthread_mutex_unlock( &done );
461             /* add to number of sleeping */
462             number_sleeping ++;
463             pthread_mutex_unlock( &num_sleeping );
464
465             /* sleep */
466             pthread_mutex_lock(
467                 &sleepers[l_args->threadID] );
468             pthread_mutex_lock( &done ); // check
469             done status just before sleeping
470             if( all_done == 1 )
471             {
472                 pthread_mutex_unlock( &done );
473                 pthread_mutex_unlock(
474                     &sleepers[l_args->threadID] );
475                 pthread_exit(0);
476             }
477             pthread_mutex_unlock( &done );
478             pthread_cond_wait( &wake_up,
479                 &sleepers[l_args->threadID] );
480             pthread_mutex_unlock(
481                 &sleepers[l_args->threadID] );
482             if( all_done == 1 )
483                 pthread_exit(0);
484             else
485             {
486                 pthread_mutex_lock( &num_sleeping);
487                 number_sleeping --;
488                 pthread_mutex_unlock(
489                     &num_sleeping );
490                 continue;
491             }
492         }

```

If the search still has not completed then the thread waits for a `wake_up` signal, essentially putting it to sleep until another thread wakes it. Upon waking, a thread will either exit if the search is complete or decrement the number of sleeping threads and continue from the beginning of the loop once again.

## 2 Results

Initial testing of the single- and multi-threaded implementations was tested on my local development machine, but the final data was acquired through the College of Engineering's Linux cluster. A recursive search beginning at `/home/DREXEL/nk78` for the string "Kandasamy" was performed one hundred times for maximum thread counts of one, two, four, and eight. By acquiring the data at roughly 2:00 PM on Thanksgiving Day, the number of processes contending for the cluster's CPU's was minimized. The resulting average, minimum, and maximum execution times are plotted in Figure 1 and tabulated in Table 1.

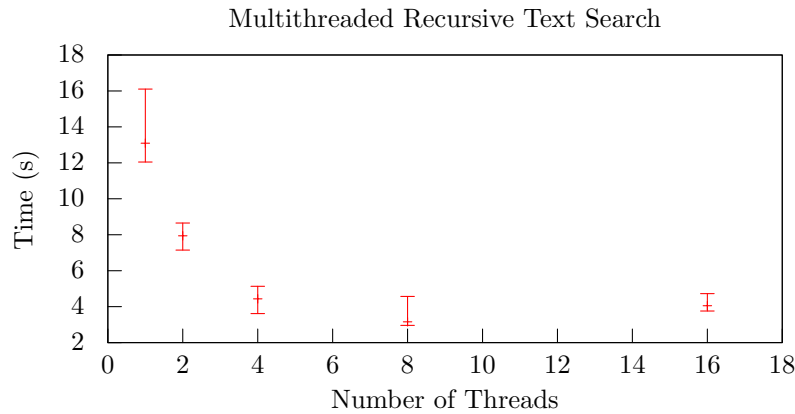


Figure 1: Mean, minimum, and maximum execution times from a set of 100 tests executed on `xunil-01`.

As is shown, there is a nearly exponential decay in mean execution time over the range from one to eight threads, causing the average value to decrease from 13.09s to 3.15s. This falls very much in line with one's expectations about parallelizing an algorithm such as the one used here. Single threaded, it takes roughly  $O(n)$  time; by adding  $i$  additional threads, this drops to  $O(\frac{n}{i+1})$ , as each thread can contribute  $\frac{1}{i+1}$  of the work in a well-balanced program.

hline Number of Threads	Average (s)	Minimum (s)	Maximum (s)
1	13.09095947	12.047517	16.104431
2	7.94123965	7.145596	8.650321
4	4.43541181	3.615055	5.133428
8	3.1560362	2.960557	4.570848
16	4.05127182	3.758859	4.725392

Table 1: Data used to generate the plot in Figure 1.

There is however a discontinuity to this nearly exponential decay, located where the search was performed with 16 threads. Intuition states that this test case would perform *better* than any of the cases with fewer threads, but this is clearly not the case. Such a slow down is most likely caused by the added contention for mutexes introduced by multi-threading the process. All of the threads share a mutex protecting the completion status, one protecting the number of sleeping threads, and most importantly, a single work queue. As a result, only one thread can read from or post to the work queue at a time (similarly with checking or setting the completion status or the number of sleeping threads). As the number of threads increases, the contention for these limited resources becomes a bottleneck on the overall performance, eventually resulting in threads that spend more time waiting for shared resources than actually doing work.

## A Scripts

Data for this report was acquired and manipulated through a series of Bash and Python scripts, which have been included here for transparency.

### A.1 Data Acquisition

```
1 #!/bin/bash
2 #
3 # Gets data for the ECE-C353 final project
4 # Author: Sean Barag <sjb89@drexel.edu>
5 SIG_TEXT="Scriptable_output:"
6 STRING="kandasamy"
7 DIR="/home/DREXEL/nk78"
8 NUM_RUNS=100
9 OUT="data_100.txt"
10
11 > $OUT
12 for loop in {1..$NUM_RUNS}
13 do
14     echo "Running... ($loop/$NUM_RUNS)"
15     for threads in 2 4 8 16
16     do
17         out/work_crew $STRING $DIR $threads | grep "$SIG_TEXT" | sed
18             "s/$SIG_TEXT\t//" >> $OUT
19     done
20 done
21 echo "All done!"
```

### A.2 Data Processing

```
1 #!/bin/bash
2 #
3 # Re-orders data so that it can be easily plotted by gnuplot
4 # Author: Sean Barag <sjb89@drexel.edu>
5 IN="data_100.txt"
6 OUT="data_processed.txt"
7 OUT2="data_calculated.txt"
8
9 # split data into separate files
10 for i in 2 4 8 16
11 do
12     cat $IN | grep "^$i" | awk '{print $3}' > /tmp/pData$i.txt
13 done
14
15 # paste files together
16 paste /tmp/pData2.txt /tmp/pData4.txt /tmp/pData8.txt /tmp/pData16.txt > $OUT
17
18 # delete temporary files
19 rm /tmp/pData{2,4,8,16}.txt
20
21 # calculate min, max, and avg
22 python calcData.py > $OUT2
```

### A.3 Data Calculation

```
1 #!/usr/bin/python
2 #
3 # Performs calculations on data to get the mean, min, and max
4 # Author: Sean Barag <sjb89@drexel.edu>
5
6 # open the data file and read it
7 lines = []
8 f = open('data_processed.txt', 'r')
9 for line in f:
10     lines.append(line.strip().split('\t'))
11 f.close()
12
13 # transpose it. Gotta love list comprehensions
14 cols = []
15 for i in range(4):
16     cols.append([row[i] for row in lines])
17
18
19 # open the other data file, because I forgot to pull the single-threaded data
20 # out in the previous script.
21 f = open('data_100.txt', 'r')
22 slines = []
23 for line in f:
24     if line != '\n':
25         slines.append(line.strip().split('\t'))
26 f.close()
27
28 # add the single-threaded data
29 cols.insert(0, [row[1] for row in slines])
30
31 # calculate and print
32 print "#num_threads\taverage\t\tmin\t\tmax"
33 for i in range(len(cols)):
34     print 2**(i), "\t\t", sum(float(v) for v in cols[i])/len(cols[i]), "\t",
35         min(cols[i]), "\t", max(cols[i])
```

## B License

Copyright © 2011, Sean Barag. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the owner nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Source code for this document is available at <http://github.com/sjbarag/>.