ECE-C353: Systems Programming
Final Project

Sean Barag
<sjb89@drexel.edu>

December 8, 2011

Contents

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`.

```
int MAX_THREADS;
int number_sleeping = 0;
int all_done = 0;
int spawned = 0;
pthread_t *threads;

pthread_cond_t wake_up;
pthread_mutex_t num_sleeping;
pthread_mutex_t *sleepers;
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.

```
/* structure for thread arguments */
typedef struct thread_args
{
    int threadID;

    queue_t *queue;
    pthread_mutex_t *mutex_queue;

    int *num_occurrences;
    pthread_mutex_t *mutex_count;

    char **argv;

} 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.

```
/* structure for return values
 *
 * these make scripting the output easier */
typedef struct return_type
{
    int count;
    float time;
} 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:

```
/* Function prototypes */
RET_TYPE* search_for_string_serial(char **);
RET_TYPE* search_for_string_mt(char **);
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.

```c
int main(int argc, char** argv)
{
    if(argc < 3)
    {
        printf("Usage: %s <search string> <path> [num threads]\n", argv[0]);
        exit(0);
    }
    if( argc == 4 )
        MAX_THREADS = atoi(argv[3]);
    else
        MAX_THREADS = 8;

    // Perform a serial search of the file system
    RET_TYPE *a = search_for_string_serial(argv);

    // Perform a multi-threaded search of the file system
    RET_TYPE *b = search_for_string_mt(argv);

    printf("\n\n\n");
    printf("                  Threads\tSingle\t\tMulti\t\tMatch\n");
    printf("Scriptable output:\t%d\t%f\t%f\t%s\n", MAX_THREADS, a->time, b->time, (a->count == b->count) ? "true" : "false");
    exit(0);
}
```

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.

```
RET_TYPE* search_for_string_mt(char **argv)
{
    /* allocate space for threads */
    threads = (pthread_t
        *)malloc(sizeof(pthread_t)*MAX_THREADS);

    /* initialize mutexes */
    pthread_mutex_t queue_mutex, count_mutex;
    pthread_mutex_init( &queue_mutex, NULL );
    pthread_mutex_init( &count_mutex, NULL );
    pthread_mutex_init( &num_sleeping, NULL );
    pthread_mutex_init( &done, NULL );

    pthread_cond_init( &wake_up, NULL );

    /* create mutex array */
    sleepers = (pthread_mutex_t *)malloc(
        sizeof(pthread_mutex_t)*MAX_THREADS );
    for( int i = 0; i < MAX_THREADS; i ++ )
        pthread_mutex_init( &sleepers[i], NULL );

    /* initialize count */
    int count = 0;
    int *p_count = &count;

    /* create and fill thread arguments */
    THREAD_ARGS *t_args;

    t_args = (THREAD_ARGS
        *)malloc(sizeof(THREAD_ARGS));

    t_args->threadID = 0;
    t_args->mutex_queue = &queue_mutex;
    t_args->mutex_count = &count_mutex;
    t_args->num_occurrences = p_count;
    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.

```
    /* start timer */
    struct timeval start;
    gettimeofday(&start, NULL);

    /* spawn thread */
    if( pthread_create( &threads[0], NULL,
        worker_thread, (void *)t_args ) != 0 )
    {
        printf("E: could not create thread! 
            Exiting\n");
        exit(-1);
    }
```

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.

```
    /* wait for all threads to return */
    for( int i = 0; i < MAX_THREADS; i++ )
        pthread_join( threads[i], NULL );

    /* stop timer */
    struct timeval stop;
    gettimeofday(&stop, NULL);

    /* create return value */
    RET_TYPE *out = (RET_TYPE
        *)malloc(sizeof(RET_TYPE));
    out->time = (float)(stop.tv_sec - start.tv_sec +
        (stop.tv_usec - start.tv_usec)/(float)1000000);
    out->count = count;

    /* print results */
    printf("\n␣\n␣\n");
    printf("Overall␣execution␣time␣=␣%fs.␣\n",
        out->time);
    printf("The␣string␣%s␣was␣found␣%d␣times␣within␣
        the␣file␣system.␣\n", argv[1], count);
    printf("====================\n\n");

    return out;
}
```

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`.

```
THREAD_ARGS *l_args = (THREAD_ARGS *) args;

queue_element_t *element, *new_element;
struct stat file_stats;
int status;
DIR *directory = NULL;
struct dirent *result = NULL;
/* allocate memory for the directory structure */
struct dirent *entry = (struct dirent
    *)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.

```c
/* initialize queue if on thread zero */
if( l_args->threadID == 0 )
{
    /* Create and initialize the queue data structure. */
    l_args->queue = create_queue();
    element = (queue_element_t *)malloc(sizeof(queue_element_t));
    if(element == NULL)
    {
        printf("E: Error allocating memory. Exiting. \n");
        exit(-1);
    }
    strcpy(element->path_name, l_args->argv[2]);
    element->next = NULL;
    /* Insert the initial path name into the queue */
    insert_in_queue(l_args->queue, element);
}
```

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 nd 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.

```c
while( 1 )
{
    pthread_mutex_lock( l_args->mutex_queue );
    /* While there is work in the queue, process it. */
    if(l_args->queue->head != NULL)
    {
        queue_element_t *element = remove_from_queue(l_args->queue);
        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.

```
if(result == NULL)
{
    if( l_args->threadID == 0 && !spawned )
    {
        spawned = 1;
        THREAD_ARGS *t_args;
        for( int i = 1; i < MAX_THREADS; i++ )
        {
            t_args = (THREAD_ARGS
                *)malloc(sizeof(THREAD_ARGS));
            t_args->threadID = i;
            t_args->queue = l_args->queue;
            t_args->mutex_queue = l_args->mutex_queue;
            t_args->mutex_count = l_args->mutex_count;
            t_args->num_occurrences =
                l_args->num_occurrences;
            /* NULL argv implies it's a worker thread
                */
            t_args->argv = l_args->argv;

            if( pthread_create( &threads[i], NULL,
                worker_thread, (void *)t_args ) != 0 )
            {
                printf("E:␣could␣not␣create␣thread␣%d!␣
                    Exiting\n");
                exit(-1);
            }
        }
    }

    break; // End of directory
}
```

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.

```
/* Construct the full path name for the directory item
    stored in entry. */
strcpy(new_element->path_name, element->path_name);
strcat(new_element->path_name, "/");
strcat(new_element->path_name, entry->d_name);

/* safely add to queue */
pthread_mutex_lock( l_args->mutex_queue );
insert_in_queue(l_args->queue, new_element);
pthread_mutex_unlock( l_args->mutex_queue );
```

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.

```
else
{
    pthread_mutex_unlock( l_args->mutex_queue );
    pthread_mutex_lock( &num_sleeping );
    pthread_mutex_lock( &done );

    if( all_done == 1 )
    {
        pthread_mutex_unlock( &done );
        pthread_mutex_unlock( &num_sleeping );
        pthread_exit(0);
    }
```

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.

```
else if( number_sleeping == (MAX_THREADS - 1) )
{
    all_done = 1;
    pthread_mutex_unlock( &done );
    pthread_mutex_unlock( &num_sleeping );

    /* wake everyone up
     * man, pthread_broadcast never works for me */
    for( int i = 0; i < MAX_THREADS-1; i++ )
        pthread_cond_signal( &wake_up );
    pthread_exit( 0 );
}
```

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.

```
else
{
    pthread_mutex_unlock( &done );
    /* add to number of sleeping */
    number_sleeping ++;
    pthread_mutex_unlock( &num_sleeping );

    /* sleep */
    pthread_mutex_lock( &sleepers[l_args->threadID] );
    pthread_mutex_lock( &done ); // check done status just before sleeping
    if( all_done == 1 )
    {
        pthread_mutex_unlock( &done );
        pthread_mutex_unlock( &sleepers[l_args->threadID] );
        pthread_exit(0);
    }
    pthread_mutex_unlock( &done );
    pthread_cond_wait( &wake_up, &sleepers[l_args->threadID] );
    pthread_mutex_unlock( &sleepers[l_args->threadID] );
    if( all_done == 1 )
        pthread_exit(0);
    else
    {
        pthread_mutex_lock( &num_sleeping);
        number_sleeping --;
        pthread_mutex_unlock( &num_sleeping );
        continue;
    }
}
```

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.

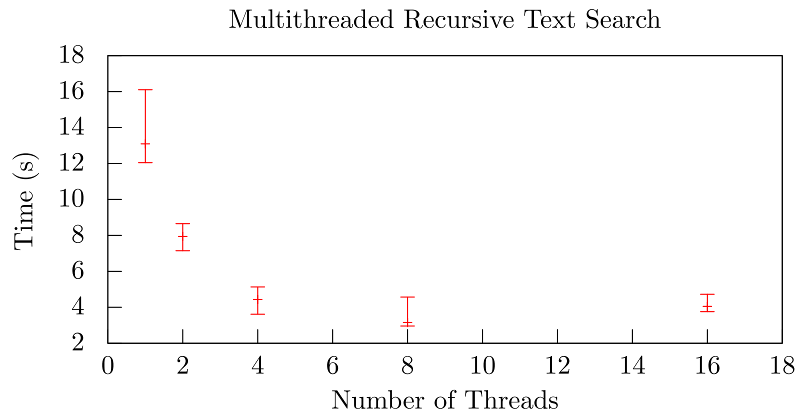

Figure 1: Mean, minimum, and maximum execution times from a set of 100 tests executed on `xunil-01`.

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 13.09 s to 3.15 s. 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.

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

```
#!/bin/bash
#
# Gets data for the ECE-C353 final project
# Author: Sean Barag <sjb89@drexel.edu>
SIG_TEXT="Scriptable␣output:"
STRING="kandasamy"
DIR="/home/DREXEL/nk78"
NUM_RUNS=100
OUT="data_100.txt"

> $OUT
for loop in {1..$NUM_RUNS}
do
    echo "Running...␣($loop/$NUM_RUNS)"
    for threads in 2 4 8 16
    do
        out/work_crew $STRING $DIR $threads | grep "$SIG_TEXT" | sed
            "s/$SIG_TEXT\t//" >> $OUT
    done
done
echo "All␣done!"
```

A.2 Data Processing

```
#!/bin/bash
#
# Re-orders data so that it can be easily plotted by gnuplot
# Author: Sean Barag <sjb89@drexel.edu>
IN="data_100.txt"
OUT="data_processed.txt"
OUT2="data_calculated.txt"

# split data into separate files
for i in 2 4 8 16
do
    cat $IN | grep "^$i" | awk '{print $3}' > /tmp/pData$i.txt
done

# paste files together
paste /tmp/pData2.txt /tmp/pData4.txt /tmp/pData8.txt /tmp/pData16.txt > $OUT

# delete temporary files
rm /tmp/pData{2,4,8,16}.txt

# calculate min, max, and avg
python calcData.py > $OUT2
```

A.3 Data Calculation

```
#!/usr/bin/python
#
# Performs calculations on data to get the mean, min, and max
# Author: Sean Barag <sjb89@drexel.edu>

# open the data file and read it
lines = []
f = open('data_processed.txt', 'r')
for line in f:
    lines.append(line.strip().split('\t'))
f.close()

# transpose it.  Gotta love list comprehensions
cols = []
for i in range(4):
    cols.append([row[i] for row in lines])


# open the other data file, because I forgot to pull the single-threaded data
# out in the previous script.
f = open('data_100.txt', 'r')
slines = []
for line in f:
    if line != '\n':
        slines.append(line.strip().split('\t'))
f.close()

# add the single-threaded data
cols.insert(0, [row[1] for row in slines])

# calculate and print
print "#num threads\taverage\t\tmin\t\tmax"
for i in range(len(cols)):
    print 2**(i), "\t\t", sum(float(v) for v in cols[i])/len(cols[i]), "\t",
        min(cols[i]), "\t", max(cols[i])
```

B License

Source code for this project, including this document, is available at `http://github.com/sjbarag/`.