Operating Systems Assignment

Due on 6 May, 2019

Powell, Liam

Contents

Shared Resources and Mutual Exclusion

File IO

The log file is shared by all threads. Mutual exclusion is not needed for file IO since all POSIX functions which have a `FILE*` parameter call `flockfile`, or act as if they had and all my log functions use a single `fprintf` call each.

`http://pubs.opengroup.org/onlinepubs/9699919799/functions/flockfile.html`

Statistics

The statistics struct is shared by all cpu threads. Mutual exclusion is achieved by guarding access to the struct with a mutex.

Queue

All functions that can access data inside the queue struct hold a mutex whenever they are running.

Testing

The program will not work correctly if the numbers in the input file can not be parsed as `unsigned int`, this is due to using `fscanf` rather than a more robust function.

The program does not check for overflow when updating statistics.

Apart from these issues the program is working correctly.

Examples

I have not included `simulation_log` outputs in most of the examples below because they take up a lot of space, I have included gantt charts instead.

Example 1

Queue size: 7

Input

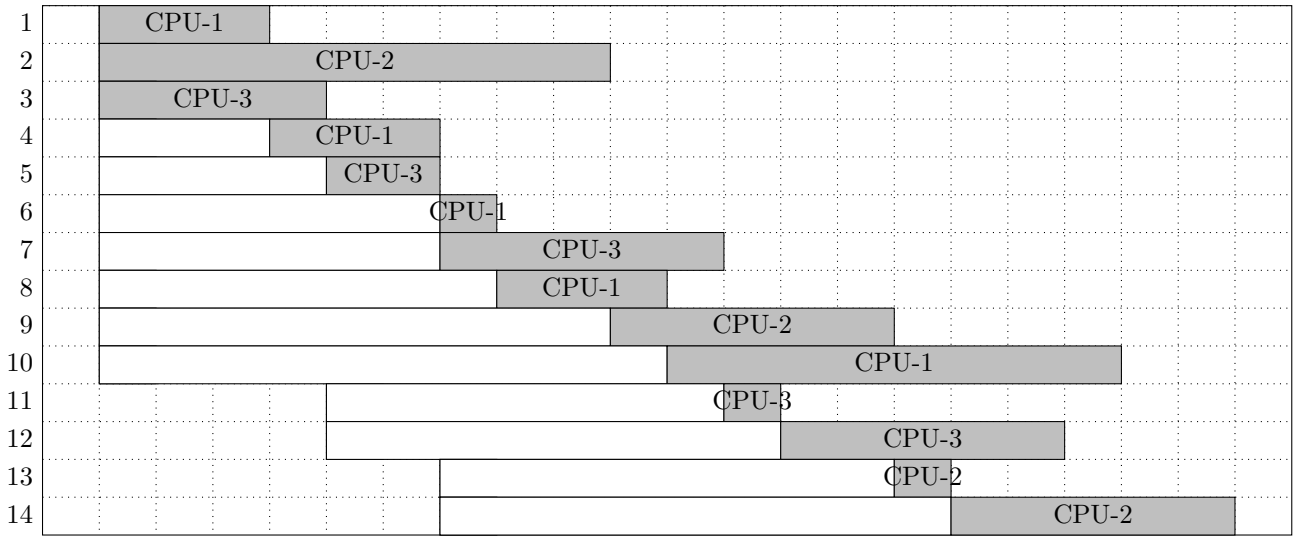
```
1 3
2 9
3 4
4 3
```

```
5 2
6 1
7 5
8 3
```

```
9 5
10 8
11 1
12 5
```

```
13 1
14 5
```

Chart



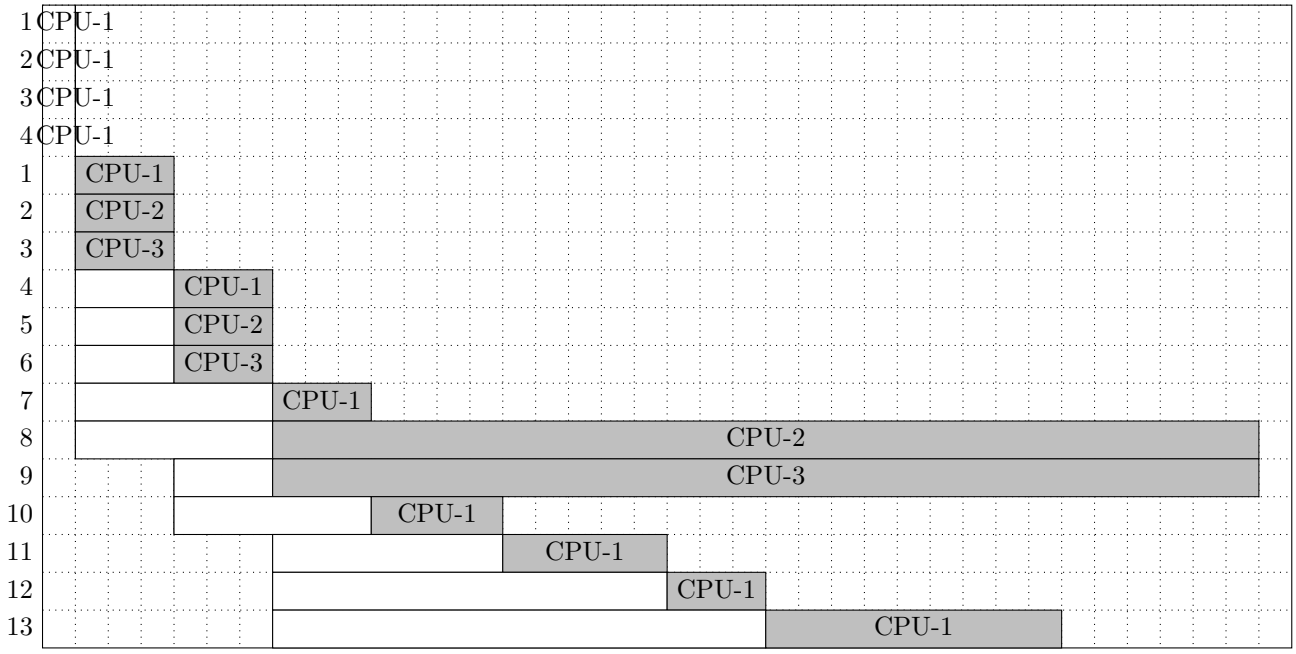
Example 2

Queue size: 5

Input

```
1 0
2 0
3 0
4 0
1 3
2 3
3 3
4 3
5 3
6 3
7 3
8 30
9 30
10 4
11 5
12 3
13 9
```

Chart



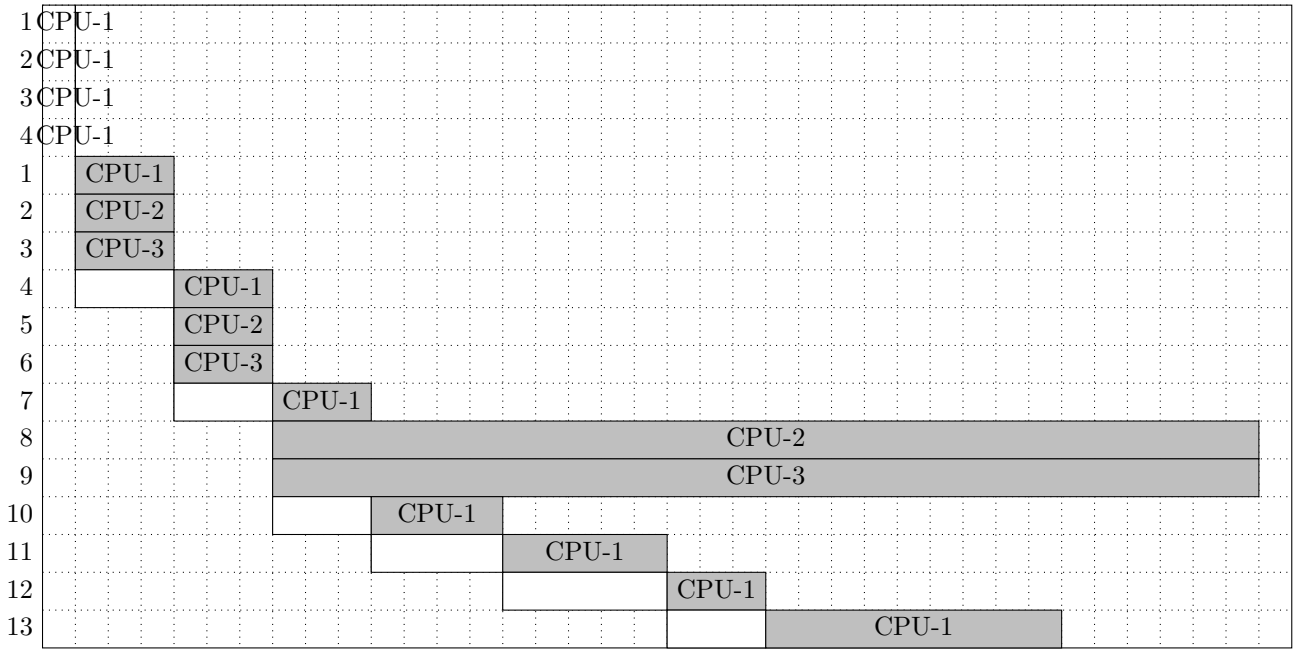
Example 3

Queue size: 1

Input

```
1 0
2 0
3 0
4 0
1 3
2 3
3 3
4 3
5 3
6 3
7 3
8 30
9 30
10 4
11 5
12 3
13 9
```

Chart



With Log

Queue size: 4

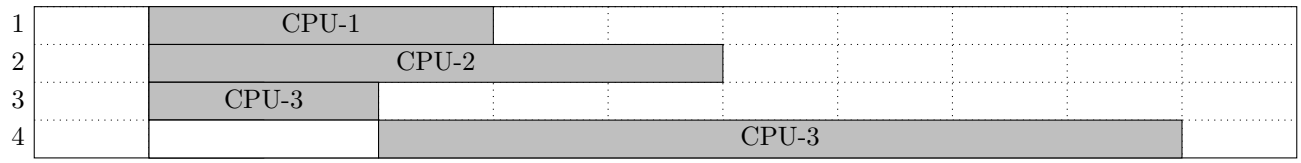
Input

```
1 3
2 5
3 2
4 7
```

Log

```
1: 3
Arrival time: 22:58:54

2: 5
Arrival time: 22:58:54

3: 2
Arrival time: 22:58:54

4: 7
Arrival time: 22:58:54

Number of tasks put into Ready-Queue: 4
Terminates at time: 22:58:54

Statistics for CPU 1:
Job #1
Arrival time: 22:58:54
Service time: 22:58:54

Statistics for CPU 2:
Job #2
Arrival time: 22:58:54
Service time: 22:58:54

Statistics for CPU 3:
Job #3
Arrival time: 22:58:54
Service time: 22:58:54

Statistics for CPU 3:
Job #3
Arrival time: 22:58:54
Completion time: 22:58:56

Statistics for CPU 3:
Job #4
Arrival time: 22:58:54
Service time: 22:58:56

Statistics for CPU 1:
Job #1
Arrival time: 22:58:54
Completion time: 22:58:57

CPU-1 terminates after servicing 1 tasks

Statistics for CPU 2:
Job #2
Arrival time: 22:58:54
Completion time: 22:58:59

CPU-2 terminates after servicing 1 tasks

Statistics for CPU 3:
Job #4
Arrival time: 22:58:54
Completion time: 22:59:03

CPU-3 terminates after servicing 2 tasks

Number of tasks: 4
Average waiting time: 0 seconds
Average turn around time: 4 seconds
```

Chart



Source Code

README.org

* Compiling

Compile =scheduler= by running =make= in this directory.

* Usage

=./scheduler [job file] [queue size]=

End of README.org

Makefile

```
.POSIX:
CC      = gcc
CFLAGS  = -std=c11 -Wall -g
LDFLAGS =
LDLIBS  = -lpthread

OBJS = build/cpu.o build/error.o build/log.o build/main.o build/task.o \
       build/tsqueue.o

scheduler: $(OBJS)
        $(CC) $(OBJS) $(LDFLAGS) $(LDLIBS) -o $@

build/cpu.o: src/cpu.c src/cpu.h src/tsqueue.h src/job.h src/log.h
        @mkdir -p build
        $(CC) $(CFLAGS) -c $< -o $@

build/error.o: src/error.c src/error.h
        @mkdir -p build
        $(CC) $(CFLAGS) -c $< -o $@

build/log.o: src/log.c src/log.h src/job.h src/config.h
        @mkdir -p build
        $(CC) $(CFLAGS) -c $< -o $@

build/main.o: src/main.c src/config.h src/cpu.h src/tsqueue.h src/task.h \
              src/error.h src/job.h
        @mkdir -p build
        $(CC) $(CFLAGS) -c $< -o $@

build/task.o: src/task.c src/task.h src/tsqueue.h src/job.h src/log.h
        @mkdir -p build
        $(CC) $(CFLAGS) -c $< -o $@

build/tsqueue.o: src/tsqueue.c src/tsqueue.h
        @mkdir -p build
        $(CC) $(CFLAGS) -c $< -o $@

clean:
        rm -rf build scheduler
```

End of Makefile

tsqueue.h

```
/**
 * @file   tsqueue.h
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  Thread safe single-producer, multi-consumer FIFO queue.
 *
 * This is a thread safe single-producer, multi-consumer FIFO queue with the
 * following properties:
 * - Uses memory provided by the user for the queue itself, allowing the user
 *   to ensure any alignment requirements. This memory may already contain
 *   elements.
 * - Arbitrary queue element sizes.
 * - Supports pushing and popping multiple elements atomically.
 * - Leaves elements in the user provided memory after destroying the queue in
 *   the order they would have been popped.
 */

#ifndef TSQUEUE_H
#define TSQUEUE_H

#include <stdbool.h>
#include <stddef.h>
#include <limits.h>

/** Thread safe single-producer, multi-consumer FIFO queue. */
typedef struct tsqueue tsqueue;

/**
 * @brief Create a new tsqueue.
 *
 * @param[out] queue The tsqueue, will be NULL if creation fails.
 * @param[in] data Array used to store queue elements, must not be used until
 *                 tsqueue_destroy() is called.
 * @param capacity The number of elements that @p data can hold.
 * @param elem_size The size of an element in @p data.
 * @param used The number of items already in @p data, these elements will be
 *             accessible via tsqueue_pop(). The first element to be popped
 *             will be the first element of @p data.
 *
 * @return Zero if the function succeeds, else a POSIX error number.
 */
int tsqueue_create(tsqueue **queue, void *data, size_t capacity,
                   size_t elem_size, size_t used);

/**
 * @brief Forces all tsqueue_put() and tsqueue_pop() functions using this
 *        queue to exit and return TSQUEUE_CLOSED.
 *
 * Will block until all calls have exited. This function is intended to be
 * used before tsqueue_destroy as a way to signal to other threads that they
 * should stop reading from the queue. All future calls to tsqueue_put() and
 * tsqueue_pop() will also return TSQUEUE_CLOSED.
 *
 * @param[in] queue The tsqueue to close.
 */
```

```
void tsqueue_close(tsqueue *queue);

/**
 * @brief Calls tsqueue_close() before freeing resources allocated by tsqueue
 *        functions.
 *
 * Using a tsqueue while it is being destroyed or after it is destroyed will
 * cause undefined behaviour. tsqueue_close() and tsqueue_set_done() can be
 * useful for indicating to other threads that they should stop using the
 * queue.
 *
 * @param[in] queue The tsqueue to destroy.
 * @param[out] used The number of queue items left in the array provided to
 *                  tsqueue_create(). The first array element would have been
 *                  popped next. Can be NULL.
 */
void tsqueue_destroy(tsqueue *queue, size_t *used);

/**
 * @brief Returns the capacity of the queue.
 *
 * @param[in] queue The tsqueue.
 *
 * @return The capacity of @p queue.
 */
size_t tsqueue_capacity(tsqueue *queue);

/**
 * @brief Blocks until there are @p n_elems free spaces in the queue.
 *
 * @param[in] queue The tsqueue.
 * @param n_elems The number of elements to wait for.
 *
 * @return Zero if the function is successful.
 *
 *         TSQUEUE_CLOSED if the queue is closed.
 *
 *         TSQUEUE_TOO_MANY if @p n_elems is greater than the queue's
 *         capacity.
 *
 *         TSQUEUE_SINGLE_PRODUCER if a tsqueue_put() or
 *         tsqueue_wait_for_space() call is already running.
 */
int tsqueue_wait_for_space(tsqueue *queue, size_t n_elems);

/**
 * @brief Waits until there are @p n_elem free slots in the queue and then
 *        adds all elements to the end of the queue.
 *
 * The first element in @p in will be popped first, after all elements already
 * in the queue.
 *
 * @param[in] queue The tsqueue.
 * @param n_elems The number of elements in @p in.
 * @param[in] in The items to insert in to the queue.
 *
 * @return Zero if the function is successful.
 *
```

```
 *         TSQUEUE_CLOSED if the queue is closed.
 *
 *         TSQUEUE_TOO_MANY if @p n_elems is greater than the queue's
 *         capacity.
 *
 *         TSQUEUE_SINGLE_PRODUCER if a a tsqueue_put() or
 *         tsqueue_wait_for_space() call is already running.
 */
int tsqueue_put(tsqueue *queue, size_t n_elems, void *in);

/**
 * @brief Retrieve elements from a tsqueue.
 *
 * Will block until there is n_elems elements unless the queue is closed or
 * `tsqueue_set_done(queue, true)` has been called.
 *
 * @param[in] queue The tsqueue.
 * @param[in,out] n_elems The maximum number of elements to place in @p
 *                        out. Will be set to the actual number of elements
 *                        retrieved. Will be set to zero if the queue is
 *                        closed or `tsqueue_set_done(queue, true)` has been
 *                        called.
 * @param[out] out Buffer to place the elements in. The first element was
 *                 first in the queue.
 *
 * @return Zero if the function is successful, including when zero elements
 *         are retrieved.
 *
 *         TSQUEUE_CLOSED if the queue is closed.
 */
int tsqueue_pop(tsqueue *queue, size_t *n_elems, void *out);

/**
 * @brief Indicate that no more items will be placed in the queue. Can be
 *        reversed.
 *
 * @param[in] queue The tsqueue.
 * @param done True to indicate that no more items will be placed in the
 *             queue, false to reset to the normal state.
 */
void tsqueue_set_done(tsqueue *queue, bool done);

enum
{
    // The reason we use negative numbers is that POSIX errno
    // values are always positive. This allows us to return a POSIX error
    // value or a custom error value and differentiate between them.

    /** The queue was closed. */
    TSQUEUE_CLOSED = INT_MIN,

    /** tsqueue_put() was called with more items than the queue can hold. */
    TSQUEUE_TOO_MANY,

    /** A tsqueue_put() or tsqueue_wait_for_space() call was made while one
     * was already running. */
    TSQUEUE_SINGLE_PRODUCER
};
```

```
#endif /* TSQUEUE_H */
```

End of tsqueue.h

tsqueue.c

```c
/**
 * @file   tsqueue.c
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  Implementation of tsqueue.
 */

#include "tsqueue.h"
#include <pthread.h>
#include <stdbool.h>
#include <stddef.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

/** The internal structure of tsqueue. */
struct tsqueue
{
    /** The lock for the data in this struct. Must be held before reading or
     * writing any values in this struct. */
    pthread_mutex_t lock;

    /** The capacity of the queue. */
    size_t capacity;

    /** The number of used elements in the queue. */
    size_t used;

    /** The size of an element in the queue. */
    size_t elem_size;

    /** The actual data provided by the user. Currently this is just an array
     * where new elements are inserted at the end and everything is shifted
     * when elements are removed. */
    void *data;

    /** The number of unused elements that a producer is waiting for, or zero
     * if no producer is waiting. */
    size_t producer_n_elems;

    /** Used to signal the waiting producer to be called by consumers if they
     * find there is enough free elements after consuming some. */
    pthread_cond_t producer_wakeup;

    /** The number of waiting consumers. */
    size_t n_consumers_waiting;

    /** Used to signal waiting consumers. */
    pthread_cond_t consumer_wakeup;

    /** Used to signal to tsqueue_destroy() that all consumers and producers
     * have exited. */
    pthread_cond_t all_dead;

    /** Indicates that consumers should not wait for more items to be
```

```
     * inserted. */
    bool producers_done;

    /** Indicates that the queue is about to be destroyed and all functions
     * should return an error indicator. */
    bool die;
};

/**
 * @brief Wait for space in the queue, the queue lock must be held when
 *        calling this function.
 *
 * @param queue The queue.
 * @param n_elems Number of unused elements to wait for.
 *
 * @return Zero is successful, TSQUEUE_CLOSED if the queue is closed,
 *         else a POSIX error number.
 */
static int wait_for_space_internal(struct tsqueue *queue, size_t n_elems);

/**
 * @brief Signals queue.all_dead if there are no producers or consumers
 *        waiting, the queue lock must be held when calling this function.
 *
 * @param queue The queue.
 */
static void signal_if_all_dead(struct tsqueue *queue);

int tsqueue_create(struct tsqueue **queue, void *data, size_t capacity,
                   size_t elem_size, size_t used)
{
    int retval = 0;

    *queue = malloc(sizeof(**queue));

    if (queue == NULL) {
        retval = errno;
    }

    if (retval == 0) {
        **queue = (struct tsqueue){
            .capacity = capacity,
            .elem_size = elem_size,
            .data = data,
            .used = used
        };
    }

    int steps_done = 0;
    if (retval == 0)
    {
        retval = pthread_mutex_init(&(*queue)->lock, NULL);
    }

    if (retval == 0)
    {
        steps_done = 1;
        retval = pthread_cond_init(&(*queue)->producer_wakeup, NULL);
```

```
    }

    if (retval == 0)
    {
        steps_done = 2;
        retval = pthread_cond_init(&(*queue)->consumer_wakeup, NULL);
    }

    if (retval == 0)
    {
        steps_done = 3;
        retval = pthread_cond_init(&(*queue)->all_dead, NULL);
    }

    if (retval == 0)
    {
        steps_done = 4;
    }

    if  (retval != 0)
    {
        switch (steps_done)
        {
        case 4:
            pthread_cond_destroy(&(*queue)->all_dead);
            /* FALL THROUGH */
        case 3:
            pthread_cond_destroy(&(*queue)->consumer_wakeup);
            /* FALL THROUGH */
        case 2:
            pthread_cond_destroy(&(*queue)->producer_wakeup);
            /* FALL THROUGH */
        case 1:
            pthread_mutex_destroy(&(*queue)->lock);
            /* FALL THROUGH */
        default:
            break;
        }

        free(*queue);
    }

    return retval;
}

void tsqueue_close(struct tsqueue *queue)
{
    pthread_mutex_lock(&queue->lock);

    queue->die = true;

    if (queue->producer_n_elems != 0)
    {
        pthread_cond_signal(&queue->producer_wakeup);
    }

    for (size_t i = 0; i < queue->n_consumers_waiting; ++i)
    {
```

```
        pthread_cond_signal(&queue->consumer_wakeup);
    }

    while (queue->producer_n_elems != 0 && queue->n_consumers_waiting != 0)
    {
        pthread_cond_wait(&queue->all_dead, &queue->lock);
    }

    pthread_mutex_unlock(&queue->lock);
}

void tsqueue_destroy(struct tsqueue *queue, size_t *used)
{
    tsqueue_close(queue);

    if (used != NULL)
    {
        *used = queue->used;
    }

    pthread_mutex_destroy(&queue->lock);
    pthread_cond_destroy(&queue->producer_wakeup);
    pthread_cond_destroy(&queue->consumer_wakeup);
    pthread_cond_destroy(&queue->all_dead);
    free(queue);
}

size_t tsqueue_capacity(struct tsqueue *queue)
{
    pthread_mutex_lock(&queue->lock);
    size_t capacity = queue->capacity;
    pthread_mutex_unlock(&queue->lock);
    return capacity;
}

int tsqueue_wait_for_space(struct tsqueue *queue, size_t n_elems)
{
    pthread_mutex_lock(&queue->lock);
    int retval = wait_for_space_internal(queue, n_elems);
    pthread_mutex_unlock(&queue->lock);
    return retval;
}

int tsqueue_put(struct tsqueue *queue, size_t n_elems, void *in)
{
    int retval = 0;

    pthread_mutex_lock(&queue->lock);

    retval = wait_for_space_internal(queue, n_elems);

    if (retval == 0)
    {
        memcpy((char *)queue->data + (queue->used * queue->elem_size),
               in, n_elems * queue->elem_size);
        if (n_elems != 0 && queue->n_consumers_waiting != 0)
        {
            pthread_cond_signal(&queue->consumer_wakeup);
```

```
        }

        queue->used += n_elems;
    }

    pthread_mutex_unlock(&queue->lock);

    return retval;
}

int tsqueue_pop(struct tsqueue *queue, size_t *n_elems, void *out)
{
    int retval = 0;

    pthread_mutex_lock(&queue->lock);

    if (*n_elems > queue->capacity)
    {
        retval = TSQUEUE_TOO_MANY;
    }

    if (retval == 0)
    {
        ++queue->n_consumers_waiting;
        while (!queue->producers_done && !queue->die && queue->used < *n_elems)
        {
            pthread_cond_wait(&queue->consumer_wakeup, &queue->lock);
        }
        --queue->n_consumers_waiting;
    }

    if (queue->die)
    {
        retval = TSQUEUE_CLOSED;
        *n_elems = 0;
        signal_if_all_dead(queue);
    }

    if (retval == 0)
    {
        if (queue->used < *n_elems)
        {
            *n_elems = queue->used;
        }
        memcpy(out, queue->data, *n_elems * queue->elem_size);
        queue->used -= *n_elems;
        memmove(queue->data,
                (char *)queue->data + (*n_elems * queue->elem_size),
                queue->used * queue->elem_size);

        if (queue->producer_n_elems != 0
            && queue->producer_n_elems <= (queue->capacity - queue->used))
        {
            pthread_cond_signal(&queue->producer_wakeup);
        }

        if (queue->used != 0 && queue->n_consumers_waiting != 0)
        {
```

```
            pthread_cond_signal(&queue->consumer_wakeup);
        }
    }

    pthread_mutex_unlock(&queue->lock);

    return retval;
}

void tsqueue_set_done(struct tsqueue *queue, bool done)
{
    pthread_mutex_lock(&queue->lock);

    queue->producers_done = done;
    for (size_t i = 0; i < queue->n_consumers_waiting; ++i)
    {
        pthread_cond_signal(&queue->consumer_wakeup);
    }

    pthread_mutex_unlock(&queue->lock);
}

static int wait_for_space_internal(struct tsqueue *queue, size_t n_elems)
{
    int retval = 0;

    if (n_elems > queue->capacity)
    {
        retval = TSQUEUE_TOO_MANY;
    }

    if (queue->producer_n_elems != 0)
    {
        retval = TSQUEUE_SINGLE_PRODUCER;
    }

    if (retval == 0)
    {
        queue->producer_n_elems = n_elems;
        while (!queue->die && (queue->capacity - queue->used) < n_elems)
        {
            pthread_cond_wait(&queue->producer_wakeup, &queue->lock);
        }
        queue->producer_n_elems = 0;
    }

    if (queue->die)
    {
        retval = TSQUEUE_CLOSED;
        signal_if_all_dead(queue);
    }

    return retval;
}

static void signal_if_all_dead(struct tsqueue *queue)
{
    if (queue->producer_n_elems == 0 && queue->n_consumers_waiting == 0)
```

```
    {
        pthread_cond_signal(&queue->all_dead);
    }
}
```

End of tsqueue.c

task.h

```
/**
 * @file   task.h
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  The task() function described by the assignment spec.
 */

#ifndef TASK_H
#define TASK_H

#include "tsqueue.h"
#include <stdio.h>

/** Parameters to pass to task(). */
struct task_params
{
    /** The ready-queue, the task() thread only acts as a producer. */
    tsqueue *queue;

    /** The file to  get jobs from. */
    FILE *job_file;

    /** The buffer to store jobs in before placing them in the queue. */
    struct job_struct *job_buffer;

    /** The length of job_buffer. */
    size_t job_buffer_length;

    /** The file to write log messages to. */
    FILE *log_file;

    /** The return value of the task() call. task() will set this before
     * exiting. Zero if successful, otherwise can be passed to
     * errno_or_ae_to_str(). */
    int retval;
};

/**
 * @brief Places jobs in the provided queue until the end of the file is
 *        reached or an error occurs.
 *
 * @param data See task_params for details.
 *
 * @return NULL, see task_params for return value.
 */
void *task(void *data);

#endif /* TASK_H */
```

End of task.h

task.c

```c
/**
 * @file   task.c
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  Implementation of task().
 */

#define _POSIX_C_SOURCE 200809L

#include "task.h"
#include "job.h"
#include "log.h"
#include "error.h"
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>

/**
 * @brief Fill @p buffer with jobs from @p job_file.
 *
 * The file should contain "<job id> <job time in seconds> <job id> <job time
 * in seconds> ..." separated by whitespace.
 *
 * @param[in,out] job_file The file to read jobs from.
 * @param length The maximum number of jobs to read.
 * @param[out] buffer The buffer to fill with at most @p length jobs.
 * @param[out] used The number of jobs placed in the buffer. Zero if end of
 *                  file is reached.
 *
 * @return Zero if the function succeeds, else an error code that can be
 *         passed to errno_or_ae_to_str().
 */
static int fill_job_buffer(FILE *job_file, size_t length,
                           struct job_struct *buffer, size_t *used);

void *task(void *ptr)
{
    int retval = 0;

    // Input arguments
    struct task_params *params = ptr;
    tsqueue *queue = params->queue;
    FILE *job_file = params->job_file;
    struct job_struct *job_buffer = params->job_buffer;
    size_t job_buffer_length = params->job_buffer_length;
    FILE *log_file = params->log_file;

    // Total number of jobs processed
    unsigned long n_jobs = 0;

    if (tsqueue_capacity(queue) < job_buffer_length)
    {
        job_buffer_length = tsqueue_capacity(queue);
```

```
    }


    int queue_retval = 0;
    while (retval == 0 && !feof(job_file) && queue_retval == 0)
    {
        size_t jobs_in_buffer = 0;
        retval = fill_job_buffer(job_file, job_buffer_length, job_buffer,
                                 &jobs_in_buffer);
        if (retval == 0)
        {
            queue_retval = tsqueue_wait_for_space(queue, jobs_in_buffer);
        }

        if (retval == 0 && queue_retval == 0)
        {
            for (size_t i = 0; i < jobs_in_buffer; ++i)
            {
                clock_gettime(CLOCK_MONOTONIC, &job_buffer[i].arrival_mono);
                clock_gettime(CLOCK_REALTIME, &job_buffer[i].arrival_real);
            }
            queue_retval = tsqueue_put(queue, jobs_in_buffer, job_buffer);
            n_jobs += jobs_in_buffer;
        }

        if (retval == 0 && queue_retval == 0)
        {
            for (size_t i = 0; i < jobs_in_buffer; ++i)
            {
                retval = log_arrival(log_file, &job_buffer[i]);
                if (retval != 0)
                {
                    break;
                }
            }
        }
    }

    tsqueue_set_done(queue, true);

    if (retval == 0)
    {
        struct timespec time = {0};
        clock_gettime(CLOCK_REALTIME, &time);
        retval = log_task_done(log_file, time, n_jobs);
    }

    params->retval = retval;
    return NULL;
}

static int fill_job_buffer(FILE *job_file, size_t length,
                           struct job_struct *buffer, size_t *used)
{
    int retval = 0;

    *used = 0;
    while (!feof(job_file) && retval == 0 && *used < length)
```

```
    {
        // I have used fscanf rather than a more robust function here to keep
        // this function simple.
        struct job_struct *job = &buffer[*used];
        unsigned int tmp_time;
        errno = 0;
        if (fscanf(job_file, " %u %u ", &job->id, &tmp_time) != 2)
        {
            retval = (errno != 0) ? errno : AE_BAD_FILE;
        }
        else
        {
            job->cpu_burst = (time_t)tmp_time;
            ++*used;
        }
    }

    return retval;
}
```

End of task.c

cpu.h

```
/**
 * @file   cpu.h
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  The cpu() function described by the assignment spec.
 */

#ifndef CPU_H
#define CPU_H

#include "tsqueue.h"
#include <stdio.h>
#include <time.h>
#include <pthread.h>

/** Parameters to pass to cpu(). */
struct cpu_params
{
    /** Statistics shared between all cpu() threads. */
    struct cpu_shared_stats *stats;

    /** The ready-queue, cpu() threads only act as consumers. */
    tsqueue *queue;

    /** The id of the cpu to be used for logging. */
    unsigned id;

    /** The file to write log messages to. */
    FILE *log_file;

    /** The return value of the cpu() call. cpu() will set this before
     * exiting. Zero is successful, otherwise can be passed to
     * errno_or_ae_to_str(). */
    int retval;
};

/** Statistics shared between all cpu() calls. */
struct cpu_shared_stats
{
    /** Lock for the data in this struct. All cpu() threads must hold this
     * lock when reading or writing any of the values in this struct. */
    pthread_mutex_t lock;

    /** Total amount of time spent waiting by jobs in the ready queue. */
    time_t total_waiting_time;

    /** Total amount of time spent by jobs waiting in the queue or
     * running. */
    time_t total_turnaround_time;

    /** Number of jobs which have been inserted in to the queue. */
    unsigned long num_tasks;
};

/**
```

```
 * @brief Runs jobs from the provided queue until the queue is empty or an
 *        error occurs.
 *
 * @param data See cpu_params for details.
 *
 * @return NULL, see cpu_params for return value.
 */
void *cpu(void *data);

#endif /* CPU_H */
```

End of cpu.h

cpu.c

```c
/**
 * @file   cpu.c
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  Implementation of cpu().
 */

#define _POSIX_C_SOURCE 200809L

#include "cpu.h"
#include "job.h"
#include "log.h"
#include <pthread.h>
#include <time.h>
#include <errno.h>
#include <stdio.h>

/**
 * @brief Logs service time, waits for @p job.cpu_burst seconds, then logs
 *        completion time. Also increments all values in @p stats.
 *
 * Calls log_service() before waiting and log_completion() after.
 *
 * @param[in] job The job to handle.
 * @param[in,out] log_file The file to log to.
 * @param cpu_id The id to be used in log messages.
 * @param[in,out] stats Stats to modify.
 *
 * @return Zero if the function succeeds, else an error code that can be
 *         passed to errno_or_ae_to_str().
 */
static int handle_job(struct job_struct *job, FILE *log_file, unsigned cpu_id,
                      struct cpu_shared_stats *stats);

void *cpu(void *ptr)
{
    int retval = 0;

    // Input arguments
    struct cpu_params *params = ptr;
    struct cpu_shared_stats *stats = params->stats;
    tsqueue *queue = params->queue;
    unsigned cpu_id = params->id;
    FILE *log_file = params->log_file;

    // Total number of jobs inserted
    unsigned long n_jobs = 0;

    size_t jobs_from_queue = 1;
    // The only time this will be non-zero is if the queue is closed due to an
    // error occurring elsewhere.
    int queue_retval;
    do
    {
        struct job_struct job;
```

```
        queue_retval = tsqueue_pop(queue, &jobs_from_queue, &job);
        if (jobs_from_queue == 1 && queue_retval == 0)
        {
            ++n_jobs;
            retval = handle_job(&job, log_file, cpu_id, stats);
        }
    } while (retval == 0 && jobs_from_queue == 1 && queue_retval == 0);

    if (retval == 0 && queue_retval == 0)
    {
        retval = log_cpu_done(log_file, cpu_id, n_jobs);
    }

    params->retval = retval;
    return NULL;
}



static void run_job(const struct job_struct *job)
{
    struct timespec ts = {.tv_sec = job->cpu_burst};
    // I have chosen to use clock_nanosleep instead of sleep here because it
    // allows us to use a monotonic clock explicitly.
    while (clock_nanosleep(CLOCK_MONOTONIC, 0, &ts, &ts) != 0)
    {
    }
}

static int handle_job(struct job_struct *job, FILE *log_file, unsigned cpu_id,
                      struct cpu_shared_stats *stats)
{
    int retval = 0;

    clock_gettime(CLOCK_MONOTONIC, &job->service_mono);
    clock_gettime(CLOCK_REALTIME, &job->service_real);

    pthread_mutex_lock(&stats->lock);
    ++stats->num_tasks;
    stats->total_waiting_time +=
        job->service_mono.tv_sec - job->arrival_mono.tv_sec;
    pthread_mutex_unlock(&stats->lock);

    retval = log_service(log_file, cpu_id, job);
    if (retval == 0)
    {
        run_job(job);
        clock_gettime(CLOCK_MONOTONIC, &job->completion_mono);
        clock_gettime(CLOCK_REALTIME, &job->completion_real);

        pthread_mutex_lock(&stats->lock);
        stats->total_turnaround_time +=
            job->completion_mono.tv_sec - job->arrival_mono.tv_sec;
        pthread_mutex_unlock(&stats->lock);

        retval = log_completion(log_file, cpu_id, job);
    }
```

```
    return retval;
}
```

End of cpu.c

main.c

```
/**
 * @file   main.c
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  User interaction and initialisation for other functions.
 */


#include "config.h"
#include "cpu.h"
#include "task.h"
#include "error.h"
#include "tsqueue.h"
#include "job.h"
#include "log.h"
#include <errno.h>
#include <stdint.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <inttypes.h>

/**
 * @brief Returns errno if @p ptr is NULL, else returns 0. Used to make main()
 *        a little bit cleaner.
 *
 * @param ptr The pointer to check.
 *
 * @return errno if @p ptr is NULL, else zero.
 */
static int errno_if_null(void *ptr);

int main(int argc, char **argv)
{
    // This function is very long but most of it is just braces and
    // whitespace.

    int retval = 0;

    FILE *log_file = NULL;
    FILE *input_file = NULL;
    pthread_t *cpu_threads = NULL;
    pthread_t task_thread;
    struct cpu_params *cpu_params = NULL;
    struct cpu_shared_stats stats = {0};
    struct task_params task_params = {0};
    struct job_struct *queue_data = NULL;
    tsqueue *queue = NULL;
    bool shared_is_initialised = false;
    size_t queue_length = 0;

    if (argc != 3)
    {
        retval = AE_WRONG_NUM_ARGS;
```

```
}

/*****************************************************/
/* BEGINNING OF PARSING AND RESOURCE ALLOCATION CODE */
/*****************************************************/

if (retval == 0)
{
    char *end;
    errno = 0;
    uintmax_t tmp = strtoumax(argv[2], &end, 10);
    if (errno)
    {
        retval = errno;
    }
    else if (tmp < QUEUE_SIZE_MIN || tmp > QUEUE_SIZE_MAX)
    {
        retval = EINVAL;
    }
    else
    {
        queue_length = (size_t)tmp;
    }
}

if (retval == 0)
{
    retval = pthread_mutex_init(&stats.lock, NULL);
    if (retval == 0)
    {
        shared_is_initialised = true;
    }
}

if (retval == 0)
{
    retval = errno_if_null(log_file = fopen(LOG_FILE_PATH, "a"));
}

if (retval == 0)
{
    retval = errno_if_null(input_file = fopen(argv[1], "r"));
}

if (retval == 0)
{
    retval =
        errno_if_null(cpu_threads = malloc(sizeof(*cpu_threads) * CPU_COUNT));
}

if (retval == 0)
{
    retval =
        errno_if_null(cpu_params = malloc(sizeof(*cpu_params) * CPU_COUNT));
}

if (retval == 0)
{
```

```
        retval =
            errno_if_null(queue_data = malloc(sizeof(*queue_data) * queue_length));
    }

    if (retval == 0)
    {
        retval = tsqueue_create(&queue, queue_data, queue_length,
                                sizeof(*queue_data), 0);
    }

    if (retval == 0)
    {
        for (unsigned int i = 0; i < CPU_COUNT; ++i)
        {
            cpu_params[i] = (struct cpu_params){
                .stats = &stats,
                .queue = queue,
                .id = i + 1,
                .log_file = log_file
            };
        }

        task_params = (struct task_params){
            .queue = queue,
            .job_file = input_file,
            .job_buffer_length = TASK_JOB_BUFFER_LENGTH,
            .log_file = log_file
        };
        retval = errno_if_null(task_params.job_buffer =
                                   malloc(sizeof(*task_params.job_buffer)
                                          * task_params.job_buffer_length));
    }

    /*****************************************/
    /* END OF PARSING AND RESOURCE ALLOCATION */
    /*****************************************/

    if (retval == 0)
    {
        retval = pthread_create(&task_thread, NULL, &task, &task_params);
    }

    if (retval == 0)
    {
        size_t i = 0;
        while (retval == 0 && i < CPU_COUNT)
        {
            retval =
                pthread_create(&cpu_threads[i], NULL, &cpu, &cpu_params[i]);
            ++i;
        }

        if (retval != 0)
        {
            tsqueue_close(queue);
            --i;
        }
```

```
        pthread_join(task_thread, NULL);
        for (size_t j = 0; j < i; ++j)
        {
            pthread_join(cpu_threads[j], NULL);
            if (retval == 0)
            {
                retval = cpu_params[j].retval;
            }
        }

        if (retval == 0)
        {
            retval = task_params.retval;
        }

    }

    if (retval == 0)
    {
        retval = log_main_done(log_file, &stats);
    }

    /*****************************/
    /* BEGINNING OF TEARDOWN CODE */
    /*****************************/

    if (retval != 0)
    {
        fprintf(stderr, "%s\nUsage: %s [job file] [queue size]\n",
                errno_or_ae_to_str(retval), argv[0]);
    }

    if (queue != NULL)
    {
        tsqueue_destroy(queue, NULL);
    }

    if (input_file != NULL)
    {
        fclose(input_file);
    }

    if (log_file != NULL)
    {
        fclose(log_file);
    }

    if (shared_is_initialised)
    {
        pthread_mutex_destroy(&stats.lock);
    }

    free(task_params.job_buffer);
    free(cpu_params);
    free(cpu_threads);
    free(queue_data);

    /***********************/
```

```
    /* END OF TEARDOWN CODE */
    /************************/

    return retval;
}

static int errno_if_null(void *ptr)
{
    return (ptr == NULL) ? errno : 0;
}
```

End of main.c

log.h

```
/**
 * @file   log.h
 * @author Liam Powell
 * @date   2019-04-28
 *
 * @brief  Logging functions used throughout the program.
 */


#ifndef LOG_H
#define LOG_H

#include "job.h"
#include "cpu.h"
#include <stdio.h>
#include <time.h>

/**
 * @brief Log the service of @p j at @p time to the file @p log_file.
 *
 * Uses the format:
 *
 *     Statistics for CPU-<cpu_id>:
 *     Job #<j.id>
 *     Arrival time: <j.arrival>
 *     Service time: <j.end>
 *
 * @param[in,out] log_file The file to write to.
 * @param cpu_id The id of the cpu.
 * @param[in] job The job to be logged.
 *
 * @return Zero if the function succeeds, else a POSIX error number.
 */
int log_service(FILE *log_file, unsigned cpu_id, const struct job_struct *job);

/**
 * @brief Log the completion of @p j at @p time to the file @p log_file.
 *
 * Uses the format:
 *
 *     Statistics for CPU-<cpu_id>:
 *     Job #<j.id>
 *     Arrival time: <j.arrival>
 *     Completion time: <j.end>
 *
 * @param[in,out] log_file The file to write to.
 * @param cpu_id The id of the cpu.
 * @param[in] job The job to be logged.
 *
 * @return Zero if the function succeeds, else a POSIX error number.
 */
int log_completion(FILE *log_file, unsigned cpu_id, const struct job_struct *job);

/**
 * @brief Log the total number of jobs executed by a cpu thread.
 *
```

```
 * Uses the format:
 *
 *     CPU-<cpu_id> terminates after servicing <n_jobs> tasks
 *
 * @param[in,out] log_file The file to write to.
 * @param cpu_id The id of the cpu.
 * @param n_jobs The number of jobs.
 *
 * @return Zero if the function succeeds, else a POSIX error number.
 */
int log_cpu_done(FILE *log_file, unsigned cpu_id, unsigned long n_jobs);

/**
 * @brief Log the arrival of a job.
 *
 * Uses the format:
 *
 *     <j.id>: <j.cpu_burst>
 *     Arrival time: <j.arrival>
 *
 * @param[in,out] log_file The file to write to.
 * @param[in] job The job to log.
 *
 * @return Zero if the function succeeds, else a POSIX error number.
 */
int log_arrival(FILE *log_file, const struct job_struct *job);

/**
 * @brief Log the total number of jobs put in to the queue by task().
 *
 * @param[in,out] log_file The file to write to.
 * @param time The time to add to the log.
 * @param n_jobs The number of jobs to add to the log.
 *
 * Uses the format:
 *
 *     Number of tasks put into Ready-Queue: <n_jobs>
 *     Terminates at time: <time>
 *
 * @return Zero if the function succeeds, else an error code that can be
 *         passed to errno_or_ae_to_str().
 */
int log_task_done(FILE *log_file, struct timespec time, unsigned long n_jobs);

/**
 * @brief Log statistics after all tasks are finished.
 *
 * Uses the format:
 * @verbatim
 * Number of tasks: #
 * Average waiting time: # seconds
 * Average turn around time: # seconds
 * @endverbatim
 *
 * @param log_file The file to write to.
 * @param stats The stats to get data from.
 *
 * @return Zero if the function succeeds, else an error code that can be
```

```
 *          passed to errno_or_ae_to_str().
 */
int log_main_done(FILE *log_file, struct cpu_shared_stats *stats);

#endif /* LOG_H */
```

End of log.h

log.c

```c
/**
 * @file   log.c
 * @author Liam Powell
 * @date   2019-04-28
 *
 * @brief  Implementation of log.h.
 */


#define _POSIX_C_SOURCE 200809L

#include "log.h"
#include "job.h"
#include "config.h"
#include "cpu.h"
#include <stdio.h>
#include <time.h>
#include <errno.h>
#include <stdint.h>

/**
 * @brief Log the job @p j and the event @p event at @p time to @p log_file.
 *
 * Uses the format:
 *
 *     Statistics for CPU-<cpu_id>:
 *     Job #<j.id>
 *     Arrival time: <j.arrival>
 *     <time> time: <j.end>
 *
 * @param[in,out] log_file The file to write to.
 * @param cpu_id The id of the cpu.
 * @param[in] job The job to be logged.
 * @param time The time for the event.
 * @param[in] event The event to log.
 *
 * @return Zero if the function succeeds, else an error code that can be
 *         passed to errno_or_app_errnum_to_str().
 */
static int log_cpu_event(FILE *log_file, unsigned cpu_id,
                         const struct job_struct *job, struct timespec time,
                         const char *event)
{
    int retval = 0;

    struct tm arrival_tm;
    struct tm event_tm;
    if (localtime_r(&job->arrival_real.tv_sec, &arrival_tm) == NULL
        || localtime_r(&time.tv_sec, &event_tm) == NULL)
    {
        retval = errno;
    }
    else
    {
        int res = fprintf(log_file,
                          "Statistics for CPU %u:\n"
```

```
                            "Job #%u\n"
                            "Arrival time: %02d:%02d:%02d\n"
                            "%s time: %02d:%02d:%02d\n\n",
                            cpu_id, job->id, arrival_tm.tm_hour,
                            arrival_tm.tm_min, arrival_tm.tm_sec, event,
                            event_tm.tm_hour, event_tm.tm_min, event_tm.tm_sec);
        if (res < 0)
        {
            retval = errno;
        }
    }

    return retval;
}

int log_service(FILE *log_file, unsigned cpu_id, const struct job_struct *job)
{
    return log_cpu_event(log_file, cpu_id, job, job->service_real, "Service");
}

int log_completion(FILE *log_file, unsigned cpu_id, const struct job_struct *job)
{
    // This is used to make the gantt charts in my report.
#ifdef CONFIG_STDOUT_PGFGANTT
    printf("%jd.%09lu "
           "\\ganttset{bar/.append style={fill=white}} "
           "\\ganttbar{%u}{%jd}{%jd} "
           "\\ganttbar[inline]{}{%jd}{%jd} "
           "\\ganttset{bar/.append style={fill=lightgray}} "
           "\\ganttbar[inline]{CPU-%u}{%jd}{%jd}\\\\\n",
           (intmax_t)job->arrival_mono.tv_sec,
           job->arrival_mono.tv_nsec, job->id,
           (intmax_t)job->arrival_mono.tv_sec,
           (intmax_t)job->arrival_mono.tv_sec - 1,
           (intmax_t)job->arrival_mono.tv_sec,
           (intmax_t)job->service_mono.tv_sec - 1, cpu_id,
           (intmax_t)job->service_mono.tv_sec,
           (intmax_t)job->completion_mono.tv_sec - 1);
#endif
    return log_cpu_event(log_file, cpu_id, job, job->completion_real,
                         "Completion");
}

int log_cpu_done(FILE *log_file, unsigned cpu_id, unsigned long n_jobs)
{
    int retval = 0;

    int res = fprintf(log_file,
                      "CPU-%u terminates after servicing %lu tasks\n\n",
                      cpu_id, n_jobs);
    if (res < 0)
    {
        retval = res;
    }

    return retval;
}
```

```
int log_task_done(FILE *log_file, struct timespec time, unsigned long n_jobs)
{
    int retval = 0;

    struct tm tm;
    if (localtime_r(&time.tv_sec, &tm) == NULL)
    {
        retval = errno;
    }
    else
    {
        int res = fprintf(log_file,
                          "Number of tasks put into Ready-Queue: %lu\n"
                          "Terminates at time: %02d:%02d:%02d\n\n",
                          n_jobs, tm.tm_hour, tm.tm_min,
                          tm.tm_sec);
        if (res < 0)
        {
            retval = res;
        }
    }

    return retval;
}

int log_arrival(FILE *log_file, const struct job_struct *job)
{
    int retval = 0;

    struct tm tm;
    if (localtime_r(&job->arrival_real.tv_sec, &tm) == NULL)
    {
        retval = errno;
    }
    else
    {
        int res = fprintf(log_file,
                          "%u: %jd\n"
                          "Arrival time: %02d:%02d:%02d\n\n",
                          job->id, (intmax_t)job->cpu_burst, tm.tm_hour,
                          tm.tm_min, tm.tm_sec);
        if (res < 0)
        {
            retval = errno;
        }
    }

    return retval;
}

int log_main_done(FILE *log_file, struct cpu_shared_stats *stats)
{
    uintmax_t avg_wait = 0;
    uintmax_t avg_turn = 0;
    if (stats->num_tasks != 0)
    {
        avg_wait = (uintmax_t)stats->total_waiting_time / stats->num_tasks;
        avg_turn = (uintmax_t)stats->total_turnaround_time / stats->num_tasks;
```

```
    }
    int retval =
        fprintf(log_file,
                "Number of tasks: %lu\n"
                "Average waiting time: %ju seconds\n"
                "Average turn around time: %ju seconds\n\n",
                stats->num_tasks, avg_wait, avg_turn);
    return (retval < 0) ? errno : 0;
}
```

End of log.c

config.h

```
/**
 * @file   config.h
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  Constants used to configure the rest of the program.
 */

#ifndef CONFIG_H
#define CONFIG_H

#include <stddef.h>

/** The number of cpu function threads to spawn. */
static const unsigned int CPU_COUNT = 3;

/** The minimum size of the job queue. */
static const size_t QUEUE_SIZE_MIN = 1;

/** The maximum size of the job queue. */
static const size_t QUEUE_SIZE_MAX = 10;

/** The number of jobs for the task function to buffer before inserting in to
 * the queue. */
static const size_t TASK_JOB_BUFFER_LENGTH = 2;

/** The path of the simulation log to write to. */
static const char *const LOG_FILE_PATH = "simulation_log";

#endif /* CONFIG_H */
```

End of config.h

job.h

```
/**
 * @file   job.h
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  Data structure for representing jobs.
 */

#ifndef JOB_H
#define JOB_H

#include <time.h>

/** Information about a job. All values are set by task(). */
struct job_struct {
    /** ID of the job. */
    unsigned id;

    /** Time required for the job in seconds. */
    time_t cpu_burst;

    /** Arrival time of the job from CLOCK_MONOTONIC. Used for statistics,
     * CLOCK_REALTIME is not appropriate for this as it can change
     * dramatically for various reasons (such as switching to daylight savings
     * time). */
    struct timespec arrival_mono;

    /** Arrival time of the job from CLOCK_REALTIME. Used in logs. */
    struct timespec arrival_real;

    /** Service time of the job from CLOCK_MONOTONIC. Used for statistics. */
    struct timespec service_mono;

    /** Service time of the job from CLOCK_REALTIME. Used for logs. */
    struct timespec service_real;

    /** Completion time of the job from CLOCK_MONOTONIC. Used for statistics. */
    struct timespec completion_mono;

    /** Completion time of the job from CLOCK_REALTIME. Used for logs. */
    struct timespec completion_real;
};

#endif /* JOB_H */
```

End of job.h

error.h

```
/**
 * @file   error.h
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  Error reporting functions.
 */

#ifndef ERROR_H
#define ERROR_H

#include <limits.h>

// AE stands for application errno

enum ae_codes {
    // The reason we use negative numbers is that POSIX errno values are
    // always positive. This allows us to return a POSIX error value or a
    // custom error value and differentiate between them.

    /** Argument did not represent a number. */
    AE_STR_NOT_A_NUMBER = INT_MIN,

    /** The program was called with the wrong number of arguments. */
    AE_WRONG_NUM_ARGS,

    /** The file could not be parsed. */
    AE_BAD_FILE
};

/**
 * @brief Converts an application error value or an errno value to a
 *        string. Not thread safe.
 *
 * @param err The error number to convert.
 *
 * @return String relevant to the error.
 */
char *errno_or_ae_to_str(int err);

#endif /* ERROR_H */
```

End of error.h

error.c

```c
/**
 * @file   error.c
 * @author Liam Powell
 * @date   2019-04-25
 *
 * @brief  Implementation of error reporting functions.
 */

#include "error.h"
#include <string.h>

char *errno_or_ae_to_str(int err)
{
    char *retval = "Unknown error.";

    if (err >= 0)
    {
        retval = strerror(err);
    }
    else
    {
        switch ((enum ae_codes)err)
        {
        case AE_STR_NOT_A_NUMBER:
            retval = "Argument was not a number.";
            break;
        case AE_WRONG_NUM_ARGS:
            retval = "Wrong number of arguments.";
            break;
        case AE_BAD_FILE:
            retval = "File could not be parsed.";
        }
    }

    return retval;
}
```

End of error.c