COMP2006 Assignment Report

Archer Fabling - 20885436

1. Software Solution

The following is a breakdown of all the source code contained in my project, excluding my Makefile.

1.1 Initialisation of Shared Memory

```
/*    main.c    */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include "params.h"
#include "customer.h"
#include "teller.h"

int main (int argc, char** argv) {
  if (argc != 6) {
    perror("Wrong number of arguments");
    return 1;
  }

  /* Create a queue, a teller-totals struct, a logfile */
  Queue* c_queue = createQueue();
  TellerTotals* totals = initTellerTotals();
  LogFile* logfile = openLogFile("r_log");
  /* Get parameters */
  Parameters params = {
    .m = atoi(argv[1]),
    .tc = atoi(argv[2]),
    .tw = atoi(argv[3]),
    .td = atoi(argv[4]),
    .ti = atoi(argv[5]),
    .queue = c_queue,
    .totals = totals,
    .logfile = logfile
  };

  pthread_t customer_th;
  pthread_t teller_th[4];
  int i;

  /* Create threads */
  printf("Starting threads...\n");
  pthread_create(&customer_th, NULL, &customer, &params);
  for (i=0; i<4; i++) {
    pthread_create(&teller_th[i], NULL, &teller, &params);
  }

  /* Join threads */
  pthread_join(customer_th, NULL);
  for (i=0; i<4; i++) {
    pthread_join(teller_th[i], NULL);
  }
  printf("Threads joined...\n");

  /* Free memory */
  freeQueue(c_queue, &free);
  freeTellerTtotals(params.totals);
  closeLogFile(params.logfile);
  return 0;
}
```

`main()` starts off by checking that the correct number of arguments are given, if not, it returns an error (non-zero status code). Then it begins by initialising several structs in the heap that will be used for handling shared memory- `Queue`, `TellerTotals`, and `LogFile`.

`Queue` is a wrapper for `LinkedList` that includes conditions, a mutex, and a boolean that allow synchronisation between threads. These mechanisms will be explained in further detail. It is defined in `queue.h` and `queue.c` as such:

```
/*    queue.h    */

#ifndef QUEUE_H
#define QUEUE_H

#include <pthread.h>
#include "linkedlist.h"

/* Queue is a LinkedList wrapper struct that include pthread mutex and condition types */
typedef struct {
    LinkedList* list;
    pthread_mutex_t lock;
    pthread_cond_t not_full;
    pthread_cond_t new_customer;
    int incoming;
} Queue;

Queue* createQueue();
void freeQueue(Queue* q, listFunc func);

#endif
```

```
/*    queue.c    */

#include <stdlib.h>
#include "queue.h"
#include "macros.h"

Queue* createQueue() {
    /* Initialises Queue on heap */
    Queue* q = (Queue*) malloc(sizeof(Queue));

    /* Create linked list */
    q->list = createLinkedList();

    /* Initailise synchronisation fields */
    pthread_mutex_init(&q->lock, NULL);
    pthread_cond_init(&q->not_full, NULL);
    pthread_cond_init(&q->new_customer, NULL);
    q->incoming = TRUE;

    return q;
}

void freeQueue(Queue* q, listFunc func) {
    /* Queue destructor function */

    /* Free LinkedList */
    freeLinkedList(q->list, func);

    /* Destroy synchronisation variables */
    pthread_mutex_destroy(&q->lock);
    pthread_cond_destroy(&q->not_full);
    pthread_cond_destroy(&q->new_customer);

    free(q);
}
```

`TellerTotals` is a struct that hold data pertaining to the number of active tellers, the number customers they have served, and a mutex for read/write synchronisation between tellers. `LogFile` is a struct that contains a `FILE*` pointer to a file stream and, like `TellerTotals`, includes a mutex for write synchronisation. They are both defined in `params.h` and `params.c` (see Figures 1.1.4, 1.1.5).

`main()` then defines an instance of the `Parameters` struct which contains pointers to the aforementioned structs and integer parameters parsed from the command line. A reference to the `Parameters` struct can then be passed to the thread routines.

```
/*    params.h    */

#ifndef PARAMS_H
#define PARAMS_H
#include <stdio.h>
#include "queue.h"

/* Contains teller total data and a mutex lock */
typedef struct {
  int num_tellers;
    int tallies[4];
```

```
    pthread_mutex_t lock;
} TellerTotals;

/* Constructor and destructor functions */
TellerTotals* initTellerTotals(void);
void freeTellerTotals(TellerTotals* totals);

/* Wrapper for file stream with a mutex lock*/
typedef struct {
    FILE* fd;
    pthread_mutex_t lock;
} LogFile;

/* Constructor and destructor functions */
LogFile* openLogFile(char* filename);
void closeLogFile(LogFile* log);

/* Parameter data and pointed shared memory that is passed to all threads */
typedef struct {
    const int m;
    const int tc;
    const int td;
    const int tw;
    const int ti;
    Queue* queue;
    TellerTotals* totals;
    LogFile* logfile;
} Parameters;

#endif
```

```
/*    params.c    */

#include <stdlib.h>
#include <time.h>
#include "params.h"

TellerTotals* initTellerTotals() {
    /* Constructor function */
    TellerTotals* totals = (TellerTotals*) malloc(sizeof(TellerTotals));
    int i;

    totals->num_tellers = 0;

    /* Initialise array of teller totals to [0, 0, 0, 0] */
    for (i=0; i<4; i++) {
        totals->tallies[i] = 0;
    }

    pthread_mutex_init(&totals->lock, NULL);

    return totals;
}

void freeTellerTotals(TellerTotals* totals) {
    /* Destructor function */
    pthread_mutex_destroy(&totals->lock);
    free(totals);
}

LogFile* openLogFile(char* filename) {
    /* LogFile constructor function */
    LogFile* logfile = (LogFile*) malloc(sizeof(LogFile));

    /* Opens logfile in write mode without buffering */
    logfile->fd = fopen(filename, "w+");
    setbuf(logfile->fd, NULL);

    pthread_mutex_init(&logfile->lock, NULL);

    return logfile;
}

void closeLogFile(LogFile* logfile) {
    /* Destructor function */
    fclose(logfile->fd);

    pthread_mutex_destroy(&logfile->lock);

    free(logfile);
}
```

Once the parameters and shared memory have been initialised, `main()` starts one thread running `customer()` and four running `teller()`, using `pthread_create()`. The `customer()` thread's purpose is to read from `c_file.txt` and push customers to the queue so the `teller()` threads can pop them from the queue and serve them. All threads are given a void pointer to the `Parameters` struct defined earlier.

1.2 Customer() Thread

Information about customers, including number, type, and time data, are stored in a `customer_t` struct, defined in `customer.h`:

```
/*    customer.h    */

#ifndef CUSTOMER_H
#define CUSTOMER_H
#include "queue.h"
#include <time.h>

/* Customer data */
typedef struct {
    int n;
    char type;
    struct tm arrival;
    struct tm response;
    struct tm finish;
} customer_t;

/* Customer methods */
customer_t* create_customer();
void print_customer(void* customer);

/* Thread function */
void* customer(void* queue);

#endif
```

The main thread routine, `customer()`, is defined in `customer.c` as such:

```
/*    customer.c    */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "macros.h"
#include "params.h"
#include "customer.h"

void* customer(void* arg) {
  /* Thread routine to add customers to queue */
  Parameters* params = (Parameters*) arg;
  Queue* queue = params->queue;
  customer_t* customer = NULL;
  int push_sucess = FALSE;
  const char* linebreak = "----------------------------------------------------------------------\n";
  time_t ltime;

  /* Open customer file */
  FILE* file = fopen(C_FILE, "r");

  if (file == NULL ) {
    perror("Error reading file.\n");
  } else {
    while(!feof(file)) {
      /* For every line in the file, read into a Customer struct */
      customer = create_customer();
      fscanf(file, "%i %c ", &customer->n, &customer->type);

      /* Sleep for specified time interval */
      sleep(params->tc);

      /* Push customer to Queue */
      push_sucess = FALSE;
      pthread_mutex_lock(&queue->lock);
      while (!push_sucess) {
        if (queue->list->length < params->m) { /* Check Queue has room for one more */
          insertStart(queue->list, customer);
          ltime = time(NULL);
          localtime_r(&ltime, &customer->arrival);
          push_sucess = TRUE;
```

```
          /* Print arrival time string to logfile */
          LOG(params->logfile, "%s%i: %c\nArrival time: %02d:%02d:%02d\n%s",
            linebreak,
            customer->n, customer->type,
            customer->arrival.tm_hour, customer->arrival.tm_min, customer->arrival.tm_sec,
            linebreak);

        } else {
          /* Block thread if there is no space */
          pthread_cond_wait(&queue->not_full, &queue->lock);
        }
      }
      pthread_mutex_unlock(&queue->lock);

      /* Unblock a teller to serve the customer by signalling */
      pthread_cond_signal(&queue->new_customer);

    }
    /* End of file, no more incoming customer */
    pthread_mutex_lock(&queue->lock);
    queue->incoming = FALSE; /* Set incoming flag to false so tellers terminate */
    pthread_mutex_unlock(&queue->lock);

    fclose(file);
  }

  return NULL;
}

customer_t* create_customer() {
  /* Constructor function */
  customer_t* customer = (customer_t*) malloc(sizeof(customer_t));

  /* Set default values */
  customer->n = 0;
  customer->type = ' ';

  return customer;
}

void print_customer(void* customer) {
  /* Standard printer function used by printLinkedList */
  customer_t* c = (customer_t*)customer;
  printf("#%i: '%c'\n", c->n, c->type);

  return;
}
```

`customer()` starts by casting the arguemnt it was given from `void*` to `Parameter*`, defining some variables it will use later on, and opening `c_file.txt` (Note: `C_FILE` is defined in `macros.h`, see appendix A). It then enters a while loop that will only exit after reading every line of the file. `customer()` will repeatedly create a file and read its number of type from `file` with `fscanf()`. It will then sleep for the specified customer_interval, `params->tc`, before attempting to push the customer to the queue.

To push a customer to the queue, it first must obtain a lock on the queue, which it does with `pthread_mutex_lock(&queue->lock);`. Once it has a lock on the queue it can check if it is full. If the queue is full, it will block and `pthread_cond_wait()` for the queue to not be full using the `queue->not_full` condition, which will be signalled every time a teller thread sucessfully pops a customer from the queue. If there is room in the queue, however, the customer threads proceed to `insertStart()` a customer to the back of the queue (see appendix B for the `LinkedList` code). It then sets the customer's arrival time using `localtime_r()` and finally logs the arrival of the customer to the log file using the `LOG()` variadic macro (see appendix A).

Once the customer has been pushed to the queue, the `customer()` unlocks `&queue->lock`

and signals on condition `new_customer` to unblock a waiting teller. It then proceeds to the top of the `while` loop to parse the next customer, unless it has reached the end of the file, in which case it will set the boolean `queue->incoming` to FALSE, close the file and exit.

1.3 Teller() Thread

```
/*    teller.h    */

#ifndef TELLER_H
#define TELLER_H

#include "params.h"
#include "customer.h"
```

```
/* Thread function */
void* teller (void* arg);
void teller_serve(Parameters* params, int teller_id, customer_t* customer);
void increment_tallies(TellerTotals* totals, int teller_id);

#endif
```

```
/*    teller.c   */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include "macros.h"
#include "params.h"
#include "teller.h"

void* teller(void* arg) {
    /* Thread routine to serve customers in the queue */
    Parameters* params = (Parameters*) arg;
    Queue* queue = params->queue;
    int teller_id, total_served, i;
    struct tm start_time;
    struct tm finish_time;
    time_t ltime; /* Temp time value holder */

    pthread_mutex_lock(&params->totals->lock);
    if (params->totals->num_tellers < 4) {
        /* Set teller_id and update number of tellers*/
        params->totals->num_tellers++;
        teller_id = params->totals->num_tellers;
        pthread_mutex_unlock(&params->totals->lock);

        /* Store start time */
        ltime = time(NULL);
        localtime_r(&ltime, &start_time);
    } else {
        /* Terminate if there are already four tellers */
        pthread_mutex_unlock(&params->totals->lock);
        printf("Too many tellers already.\n");
        return NULL;
    }

    pthread_mutex_lock(&queue->lock);
    while (TRUE) {
        /* If customers still incoming, wait for signal and then recheck length of queue */
        if (queue->incoming) {
            pthread_cond_wait(&queue->new_customer, &queue->lock);
            if (queue->list->length == 0) break;
        } else { /* Else keep serving customers until the queue it empty */
            if (queue->list->length == 0) break;
        }

        /* Pop last customer from queue and signal not_full */
        customer_t* customer = (customer_t*) removeLast(queue->list);
        ltime = time(NULL);
        localtime_r(&ltime, &customer->response);
        pthread_cond_signal(&queue->not_full);
        pthread_mutex_unlock(&queue->lock);

        /* Log repsonse time of customer */
        LOG(params->logfile, "Teller: %i\nCustomer: %i\nArrival time: %02d:%02d:%02d\nResponse Time: %02d:%02d:%02d\n",
            teller_id,
            customer->n,
            customer->arrival.tm_hour, customer->arrival.tm_min, customer->arrival.tm_sec,
            customer->response.tm_hour, customer->response.tm_min, customer->response.tm_sec);

        /* Serve customer */
        teller_serve(params, teller_id, customer);

        /* Store finish time */
        ltime = time(NULL);
        localtime_r(&ltime, &customer->finish);

        /* Log finish time of customer */
        LOG(params->logfile, "Teller: %i\nCustomer: %i\nArrival time: %02d:%02d:%02d\nFinish Time: %02d:%02d:%02d\n",
            teller_id,
            customer->n,
            customer->arrival.tm_hour, customer->arrival.tm_min, customer->arrival.tm_sec,
            customer->finish.tm_hour, customer->finish.tm_min, customer->finish.tm_sec);

        free(customer);
```

```
        /* Update total customers served */
        increment_tallies(params->totals, teller_id);

        pthread_mutex_lock(&queue->lock);
    }
    /* Terminate if there are no more incoming customers */
    pthread_mutex_unlock(&queue->lock);
    pthread_cond_signal(&queue->new_customer); /* Unblock other tellers */

    /* Store finish time */
    ltime = time(NULL);
    localtime_r(&ltime, &finish_time);

    pthread_mutex_lock(&params->totals->lock);

    /* Log final stats of teller */
    LOG(params->logfile, "Termination: teller-%i\n#served customers: %i\nStart time: %02d:%02d:%02d\nFinish Time: %02d:%02d:%02d\n",
        teller_id,
        params->totals->tallies[teller_id-1],
        start_time.tm_hour, start_time.tm_min, start_time.tm_sec,
        finish_time.tm_hour, finish_time.tm_min, finish_time.tm_sec);

    params->totals->num_tellers -= 1;

    /* If last teller, log final stats of all tellers */
    if (params->totals->num_tellers == 0) {
        LOG(params->logfile, "\nTeller Statistics\n");
        total_served = 0;
        for (i = 0; i<4; i++) {
            LOG(params->logfile, "Teller-%i served %i customers.\n", i+1, params->totals->tallies[i]);
            total_served += params->totals->tallies[i];
        }
        LOG(params->logfile, "\nTotal number of customers: %i customers.\n", total_served);
    }

    pthread_mutex_unlock(&params->totals->lock);

    return NULL;
}

void teller_serve(Parameters* params, int teller_id, customer_t* customer) {
    /* Sleeps for the desired duration depending on customer type */
    switch (customer->type) {
        case 'W':
            sleep(params->tw);
            break;
        case 'D':
            sleep(params->td);
            break;
        case 'I':
            sleep(params->ti);
            break;
        default:
            break;
    }
}

void increment_tallies(TellerTotals* totals, int teller_id) {
    /* Increments the teller's total customers served */
    pthread_mutex_lock(&totals->lock);

    totals->tallies[teller_id - 1] += 1;

    pthread_mutex_unlock(&totals->lock);
}
```

Like the customer thread, a teller thread starts by casting `arg` from `void*` to `Parameter*` and defining some primitive variables. It then must ascertain which teller and assign it to `teller_id`. It does this by first obtaining a lock to the `TellerTotals` struct, incrementing `num_tellers`, setting this as its `teller_id`, and releasing the lock. It also stores store its start time in start_time. If there are already 4 tellers, the thread will release the lock and exit.

Once this is done, the teller may start serving customers. It first locks the customer queue, and, if the `queue->incoming` flag is set, it wait for the customer thread to signal on `new_customer`. Once it resumes, or in the case where incoming was set to false, it breaks out of the loop if the queue is empty. Otherwise, it proceeds to pop a customer from the queue with `removeLast()` (see Appendix B), record their response time and release the lock. Before the lock is realeased, the teller signals on `queue->not_full`, which awakens the customer thread in the case that it was blocking due to the queue being full. Once the lock is released, the teller thread writes to the log file and calls `teller_serve()` which sleeps the thread for the desired amount of

time. After serving, the customer's finish time is recorded, logged, and the customer is freed. The teller then increments its tally before obtained the queue lock and repeating the while loop.

When a teller thread exits the while loop it will release `queue->lock` and signal on `queue->new_customer`. This means in the case where the `customer()` thread sets `incoming` to false and exits while a teller is still waiting on condition `new_customer`, the teller can be unblocked by another teller thread exiting and signalling on `new_customer`. This now-unblocked teller can then see that the queue is empty and exit.

Before a teller exits, it records its finish time, locks `totals->lock`, logs its final statistics, decrements `num_tellers`, and releases the lock. If it is the last teller to exit, it will also log the final statistics of all the tellers before releasing the lock.

After the last teller exits, `main()` resumes and cleans up memory before exiting sucessfully.

2. Discussion

2.1 Edge Cases

There are several edge cases where `cq` works as normal, but a few will cause a deadlock. For example, when all the time parameters are set to 0, but the queue length is greater than the number of customers in c_file, r_log output is normal, however, when the queue length is resticted a deadlock occures and the program freezes.

It may also be possible for a bottleneck to occur when:

- Three tellers are waiting on `new_customer`
- The customer() thread has exited and `incoming = FALSE`
- There are still several customers in the queue.

This would cause the one teller to serve all the remaining customers before it exiting and unblokcing the remaining tellers. However, **I was not able to produce this edge case in practice**, so I can not be certain the aforementioned criteria can ever occur all at once.

2.2 Sample Inputs and Outputs

A sample `c_file.txt` with 150 customers is included in the source code, as well as its corresponding output, `r_log`. `c_file.txt` was generated with the following command:

```
python3 -c "import random; [print(f\"{i+1} {random.choice(['W', 'D', 'I'])}\") for i in range(150)]" > c_file.txt
```

2.2.1 Sample results

Below is an example of a short 10 customer `c_file` (truncation of included `c_file`) and its corresponding `r_log` output, assuming the paramters `./cq 15 3 6 4 2`.

```
/*    c_file.txt    */

1 W
2 W
3 W
4 D
5 I
6 D
7 D
8 W
9 W
10 D
```

```
/*    r_log    */

---------------------------------------------------------------------
1: W
Arrival time: 15:27:10
---------------------------------------------------------------------
Teller: 1
Customer: 1
Arrival time: 15:27:10
Response Time: 15:27:10
```

```
------------------------------------------------------------------------
2: W
Arrival time: 15:27:14
------------------------------------------------------------------------
Teller: 2
Customer: 2
Arrival time: 15:27:14
Response Time: 15:27:14
Teller: 1
Customer: 1
Arrival time: 15:27:10
Finish Time: 15:27:16
------------------------------------------------------------------------
3: W
Arrival time: 15:27:18
------------------------------------------------------------------------
Teller: 3
Customer: 3
Arrival time: 15:27:18
Response Time: 15:27:18

...

Teller: 1
Customer: 9
Arrival time: 15:27:42
Finish Time: 15:27:48
Termination: teller-1
#served customers: 3
Start time: 15:27:06
Finish Time: 15:27:48
Termination: teller-3
#served customers: 2
Start time: 15:27:06
Finish Time: 15:27:48
Termination: teller-4
#served customers: 2
Start time: 15:27:06
Finish Time: 15:27:48
Teller: 2
Customer: 10
Arrival time: 15:27:46
Finish Time: 15:27:50
Termination: teller-2
#served customers: 3
Start time: 15:27:06
Finish Time: 15:27:50

Teller Statistics
Teller-1 served 3 customers.
Teller-2 served 3 customers.
Teller-3 served 2 customers.
Teller-4 served 2 customers.

Total number of customers: 10 customers.
```

Appendix A - `macros.h`

The following is the source code for `macros.h`, a common header file naming convention for storing preprocessor definitions. In this case I have define the booleans, TRUE and FALSE, the name of the customer file (for easier testing), and a variadic macro, `LOG()`, that is used in both the teller and customer threads to write to the log file. It starts by obtaining a lock to the logfile, printing to the file with `fprintf()`, flushing the file write buffer (so the changes are made instantly and can be seen as the program runs), and releasing the lock. The call to `fflush()` enables the use of `& tail -f r_log`, as mentioned in the REAME.

```
/*    macros.h    */

#ifndef MACROS_H
#define MACROS_H

#define TRUE 1
#define FALSE !TRUE

#define C_FILE "c_file.txt"

/* Variadic macros to write to LogFile */
#define LOG(LOGFILE, ...) {                 \
    pthread_mutex_lock(&LOGFILE->lock);     \
    fprintf(LOGFILE->fd, __VA_ARGS__);      \
```

```
    fflush(LOGFILE->fd);                    \
    pthread_mutex_unlock(&LOGFILE->lock);   \
}

#endif
```

Appendix B - `LinkedList`

The following is the source code for my custom implementation of a singly-linked list.

```
/*    linkedlist.h    */

#ifndef LINKEDLIST_H
#define LINKEDLIST_H

typedef struct LinkedListNode {
    void* pData;
    struct LinkedListNode* pNext;
} LinkedListNode;

typedef struct LinkedList {
    LinkedListNode* pHead;
    LinkedListNode* pTail;
    int length;
} LinkedList;

typedef void (*listFunc)(void* data);

LinkedList* createLinkedList();
LinkedListNode* getLinkedListNode(LinkedList* list, int n);
void insertStart(LinkedList* list, void* entry);
void* removeStart(LinkedList* list);
void insertLast(LinkedList* list, void* entry);
void* removeLast(LinkedList* list);
void printLinkedList(LinkedList* list, listFunc func);
void freeLinkedList(LinkedList* list, listFunc func);

#endif
```

```
/*    linkedlist.c    */

#include <stdlib.h>
#include "linkedlist.h"

LinkedList* createLinkedList() {
    /* Create empty linked list */

    LinkedList* list = (LinkedList*) malloc(sizeof(LinkedList));

    list->pHead = NULL;
    list->pTail = NULL;
    list->length = 0;

    return list;
}

LinkedListNode* getLinkedListNode(LinkedList* list, int n) {
    /* Gets the nth element of a linked list */
    int i = 0;
    LinkedListNode* pointer = NULL;

    if (n >= 0 && n < list->length) {
        pointer = list->pHead;
        while (i < n && pointer != list->pTail) {
            pointer = pointer->pNext;
            i++;
        }
    }

    return pointer;
}

void insertStart(LinkedList* list, void* entry) {
    /* Inserts a node at the start of a linked list */
    LinkedListNode* oldHead = list->pHead;
    list->pHead = (LinkedListNode*) malloc(sizeof(LinkedListNode));

    list->pHead->pData = entry;
    list->pHead->pNext = oldHead;
```

```
    list->length += 1;

    if (list->length == 1) {
        list->pTail = list->pHead;
    }
}

void* removeStart(LinkedList* list) {
    /* Frees the first node and returns its data */
    LinkedListNode* start = list->pHead;
    void* data = start->pData;

    list->pHead = start->pNext;
    start->pNext = NULL;

    list->length -= 1;
    free(start);

    return data;
}

void insertLast(LinkedList* list, void* entry) {
    /* Inserts a node at the end of a linked list */
    if (list->length == 0) {
        insertStart(list, entry);
    } else {
        LinkedListNode* newTail = (LinkedListNode*)malloc(sizeof(LinkedListNode));
        list->pTail->pNext = newTail;
        list->pTail = newTail;

        newTail->pData = entry;
        newTail->pNext = NULL;

        list->length += 1;
    }

}

void* removeLast(LinkedList* list) {
    /* Frees the last node and returns its data */
    if (list->length == 0) {
        return NULL;
    } else if (list->length == 1) {
        return removeStart(list);
    } else {
        void* data = list->pTail->pData;
        LinkedListNode* pointer = list->pHead;

        while (pointer->pNext->pNext != NULL) {
            pointer = pointer->pNext;
        }

        free(list->pTail);

        list->pTail = pointer;
        list->pTail->pNext = NULL;
        list->length -= 1;

        return data;
    }
}

void printLinkedList(LinkedList* list, listFunc func) {
    /* Uses a callback function to print the data of each node */
    LinkedListNode* pointer = list->pHead;
    while (pointer != NULL) {
        (*func)(pointer->pData);
        pointer = pointer->pNext;
    }

}

void freeLinkedList(LinkedList* list, listFunc func) {
    /* Uses a callback function to free the data of each node and the list */
    LinkedListNode* pointer = list->pHead;
    LinkedListNode* temp = NULL;
    while(pointer != NULL) {
        (*func)(pointer->pData);
        temp = pointer;
        pointer = pointer->pNext;
        free(temp);
        list->length -= 1;
        list->pHead = pointer;
    }
```

```
    free(list);
}
```