1.0 Mutual Exclusion

This will describe how mutual exclusion is achieved in both parts of the assignment. Access to the SubTotal shared memory structure is the only mutual exclusion achieved. Access to Buffer1 had no mutual exclusion due to the fact that it is written to shared memory before the call to fork and that the children only access Buffer1 with a SHM_RDONLY (shared memory read only) flag.

1.1 Part A

Mutual Exclusion is achieved in part A of the assignment through the use of semaphores. The data structure SubTotal is defined as this:

```
typedef struct {
   sem_t mutex;
   sem_t full;
   sem_t empty;
   int value;
} SubTotal;
```

The SubTotal structure is then written into shared memory. mutex is initialised to the value of 1, full is initialised to 0 and empty is initialised to 1 because SubTotal can only hold one value. Here is how the critical section is accessed by the child, who is the producer, and the parent, who is the consumer:

Child Process:

```
sem_wait(&st->empty); //st is the pointer to SubTotal shared memory
sem_wait(&st->mutex);
/******** Critical section begins *****************/
st->value = subTotal; //Write the subtotal to the shared memory block
/******** End of critical section *****************/
sem_post(&st->mutex);
sem_post(&st->full);
```

Parent Process:

```
for(i = 0; i < k; i++) //where k is the number of processes
{
   sem_wait(&st->full);
   sem_wait(&st->mutex);

   /******** Critical section begins ***********/
   //Read the value in the shared memory block into SubTotal
   total += st->value;
```

```
   /****** Critical section ends ***************/

   sem_post(&st->mutex);
   sem_post(&st->empty);
}
```

This is a solution to the bounded buffer problem where we initialise the size of the buffer to be 1. This ensures that the parent is reading the sub total only when a child process has written one and that no child process can write two sub totals before the parent reads one.

1.2 Part B

Mutual exclusion achieved in Part B is very similar to Part A but with a few differences. The SubTotal data structure is now defined as this:

```
typedef struct {
   sem_t full;
   sem_t empty;
   int value;
} SubTotal;

pthread_mutex_t mutex;
```

The SubTotal structure is defined as a global variable since threads share memory. Here we initialise full to 0 and empty to 1 but now we have a thread mutex instead of a semaphore mutex. This mutex is initialised as the default thread mutex. Here is how the critical section is accessed by the created threads, who are the producers, and the main thread, who is the consumer:

Created threads:

```
sem_wait(&st->empty);
pthread_mutex_lock(&mutex);
/******** Critical section begins *****************/
st->value = subTotal; //Write the value of subtotal to st
/******** End of critical section *****************/
pthread_mutex_unlock(&mutex);
sem_post(&st->full);
```

Main thread:

```
for(i = 0; i < k; i++) //where k is the number of threads
{
   sem_wait(&st->full);
   pthread_mutex_lock(&mutex);
```

```
    /******** Critical section begins ***********/
    //Read the value in the shared memory block into SubTotal
    total += st->value;
    /****** Critical section ends ***************/

    pthread_mutex_unlock(&mutex);
    sem_post(&st->empty);
}
```

This solution to the bounded buffer problem is the same as that shown for Part A but instead of using wait and signal on a mutex semaphore we are using lock and unlock on a thread mutex variable.

2.0 Program Execution

I believe that my programs work perfectly due to the output of my programs after execution. Here is how I test my program and its output using the numbers from the assignment spec.

2.1 Part A

```
[15520564@saeshell01p assignment]$ ./AdderA Numbers 4
Sub-total produced by Processor with ID 24928: 52
Sub-total produced by Processor with ID 24929: 46
Sub-total produced by Processor with ID 24927: 19
Sub-total produced by Processor with ID 24926: 7
Total: 124
[15520564@saeshell01p assignment]$ ./AdderA Numbers 6
Sub-total produced by Processor with ID 24968: 21
Sub-total produced by Processor with ID 24969: 25
Sub-total produced by Processor with ID 24970: 42
Sub-total produced by Processor with ID 24971: 21
Sub-total produced by Processor with ID 24967: 11
Sub-total produced by Processor with ID 24966: 4
Total: 124
[15520564@saeshell01p assignment]$ ./AdderA Numbers 2
Sub-total produced by Processor with ID 24978: 98
Sub-total produced by Processor with ID 24977: 26
Total: 124
```

2.1 Part B

```
[15520564@saeshell01p assignment]$ ./AdderB Numbers 4
Sub-total produced by Thread with ID-326052096: 46
Sub-total produced by Thread with ID-315562240: 52
```

```
Sub-total produced by Thread with ID-305072384: 19
Sub-total produced by Thread with ID-294582528: 7
Total: 124
[15520564@saeshell01p assignment]$ ./AdderB Numbers 6
Sub-total produced by Thread with ID1744553728: 25
Sub-total produced by Thread with ID1734063872: 42
Sub-total produced by Thread with ID1723574016: 21
Sub-total produced by Thread with ID1755043584: 21
Sub-total produced by Thread with ID1765533440: 11
Sub-total produced by Thread with ID1776023296: 4
Total: 124
[15520564@saeshell01p assignment]$ ./AdderB Numbers 2
Sub-total produced by Thread with ID-1750706432: 98
Sub-total produced by Thread with ID-1740216576: 26
Total: 124
```

3.0 Source Code

AdderA.h

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <math.h>
#include <signal.h>

typedef struct Node {
    int number;
    struct Node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    int size;
} LinkedList;
```

```
typedef struct {
   sem_t mutex;
   sem_t full;
   sem_t empty;
   int value;
} SubTotal;


void readValues(LinkedList* values, char* filename);
void calcSum(LinkedList* values, int processes);
void fillBuffer(LinkedList* values);
void childProcess(int index, int size, int k);
int parentConsumer(int k);
void destroySM();
void initValues(int size);

//Global variables for accessing shared memory
int shmidBUF1;
int shmidST;
```

AdderA.c

```
/*******************************************\
File: AdderA.c
Author: Liam Pilling (15520564)

This file is responsible for PartA of the OS
assignment which involves adding up the numbers
in a file using processes.
\*******************************************/

#include "AdderA.h"

key_t BUFFER1 = 1390;
key_t SUBTOTAL = 2740;

int main(int argc, char* argv[])
{
   LinkedList* values;
   values = (LinkedList*)malloc(sizeof(LinkedList));
   values->size = 0;
   values->head = NULL;
```

```
    values->tail = NULL;

    //If we don't have three command line strings then prompt user to
    //start again
    if(argc == 3)
    {
      readValues(values, argv[1]);
      calcSum(values, atoi(argv[2]));
    }
    else
    {
      printf("Incorrect command line values. Must be of format:\n");
      printf("./AdderA <InputFile> <NumberOfProcesses>\n");
    }

    //free all allocated memory
    Node *currentNode, *node;
    currentNode = values->head;
    while(currentNode != NULL)
    {
      node = currentNode;
      currentNode = currentNode->next;
      free(node);
    }
    free(values);
}

void readValues(LinkedList* values, char* filename)
{
    FILE* file;
    Node* newValue;
    int value;

    file = fopen(filename, "r");
    if(file == NULL)
      perror("Error opening file\n");
    else
    {
      while(fscanf(file, "%d,", &value) == 1)
      {
      //Here we read the values in the file into a linked list
      newValue = (Node*)malloc(sizeof(Node));
      newValue->next = NULL;
```

```
      newValue->number = value;
      if(values->size == 0)
            values->head = newValue;
      else
            values->tail->next = newValue;
      values->tail = newValue;
      values->size++;
      }
      fclose(file);
   }
}

void calcSum(LinkedList* values, int processes)
{
   int i, size, total;
   total = 0;
   pid_t pid;

   //Initialise the shared memory and semaphores
   initValues(values->size);

   //Fill BUFFER1 with the values
   fillBuffer(values);

   //Parent produces k children
   for(i = 0; i < processes; i++)
   {
     pid = fork();
     if(pid < 0)
     {
     perror("Fork failed\n");
     exit(1);
     }
     else if(pid == 0)
     {
     //Here the child process will read the values and add
     //them up to put into the subtotal shared memory block
     childProcess(i, values->size, processes);
     }
   }
   //Parent consumes the values and returns the total
   total = parentConsumer(processes);
```

```
   printf("Total: %d\n", total);

   //Destroy the shared memory segments
   destroySM();
}

void fillBuffer(LinkedList* values)
{
   int j;
   Node* current;
   void* ptr;
   int* array;
   j = 0;

   //Bind the shared memory block to our array
   if((array = shmat(shmidBUF1, NULL, 0)) == (int*)-1)
   {
     perror("Error binding shared memory in fillBuffer()\n");
     exit(1);
   }

   //Fill buffer with our values
   current = values->head;
   while(current != NULL)
   {
     //Writing the values for our processes to the Buffer block
     array[j] = current->number;
     j++;
     current = current->next;
   }
}

void childProcess(int index, int size, int k)
{
   int i, subTotal, assignedSize, tempSize, tempK;
   int tempAssignedSize, amount;
   void* ptr;
   int* array;
   SubTotal* st;
   tempSize = size;
   tempK = k;
   assignedSize = (int)ceil((float)size/(float)k);
   i = subTotal = amount = 0;
```

```
//This helps us to find where we are up to and to
//evenly distribute values among the processes
for(i=0;i <= index; i++)
{
  tempAssignedSize = (int)ceil((float)tempSize/(float)tempK);
  tempK--;
  tempSize -= tempAssignedSize;

  if(i > 0)
  amount += assignedSize;
  if(tempAssignedSize < assignedSize)
  {
  assignedSize = tempAssignedSize;
  }

}

//Bind Buffer1 to our array to add up
if((array = shmat(shmidBUF1, NULL, SHM_RDONLY)) == (int*)-1)
{
  perror("Error binding BUFFER1 to child\n");
  exit(1);
}

//Bind the SubTotal block to our value
if((st = shmat(shmidST, NULL, 0)) == (SubTotal*)-1)
{
  perror("Error binding SUBTOTAL to child\n");
  exit(1);
}

//Reading the values in Buffer1 and adding them up to retrieve the
//sub-total
for(i = amount; i < (amount+assignedSize); i++)
{
  if(i < size)
  {
  subTotal += array[i];
  }
}
```

```
   printf("Sub-total produced by Processor with ID %d: %d\n",
getpid(),
           subTotal);

   sem_wait(&st->empty);
   sem_wait(&st->mutex);
/******** Critical section begins ***********/
   //Write the subtotal to the shared memory block
   st->value = subTotal;
/******** End of critical section *****************/
   sem_post(&st->mutex);
   sem_post(&st->full);

   exit(0);
}

int parentConsumer(int k)
{
   int total, i;
   total = 0;
   SubTotal *st;

   //Bind the SubTotal block to our value
   if((st = shmat(shmidST, NULL, 0)) == (SubTotal*)-1)
   {
     perror("Error binding shared memory in parentConsumer()\n");
     exit(1);
   }

   for(i = 0; i < k; i++)
   {
     sem_wait(&st->full);
     sem_wait(&st->mutex);

   /******** Critical section begins ***********/
     //Read the value in the shared memory block into SubTotal
     total += st->value;
   /****** Critical section ends ***************/

     sem_post(&st->mutex);
     sem_post(&st->empty);
   }
   return total;
```

```
}

void initValues(int size)
{
   SubTotal *st;

   //Initialise the Buffer1 shared memory block
   if((shmidBUF1 = shmget(BUFFER1, size*sizeof(int), IPC_CREAT |
0666)) <
   0)
   {
     perror("Error creating shared memory BUFFER1 in
initValues()\n");
     exit(1);
   }

   //Initialise the SubTotal shared memory block
   if((shmidST = shmget(SUBTOTAL, sizeof(SubTotal), IPC_CREAT |
0666)) <
   0)
   {
     perror("Error creating shared memory SUBTOTAL in
initValues()\n");
     exit(1);
   }

   //Bind the SubTotal block to our value
   if((st = shmat(shmidST, NULL, 0)) == (SubTotal*)-1)
   {
     perror("Error binding shared memory to SubTotal in
initValues()\n");
     exit(1);
   }

   //Initialise the semaphores
   sem_init(&st->mutex, 1, 1);
   sem_init(&st->full, 1, 0);
   sem_init(&st->empty, 1, 1);
}

void destroySM()
{
   //Destroy the shared memory
```

```
   shmctl(shmidBUF1, IPC_RMID, NULL);
   shmctl(shmidST, IPC_RMID, NULL);
}
```

AdderB.h

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <pthread.h>
#include <math.h>
#include <semaphore.h>

typedef struct Node {
    int number;
    struct Node *next;
} Node;

typedef struct {
    Node *head;
    Node *tail;
    int size;
} LinkedList;

typedef struct {
   sem_t full;
   sem_t empty;
   int value;
} SubTotal;

typedef struct {
   int index;
   int size;
   int k;
} ThreadInfo;

void readValues(LinkedList *values, char *filename);
void calcSum(LinkedList *values, int threads);
void *threadAdd(void *information);
void initGlobals();
void initBuffer(LinkedList *values);
int parentConsumer(int k);
```

```
//Global variables for the buffer and sub total
int *buffer;
SubTotal *st;

pthread_mutex_t mutex;
```

AdderB.c

```
/*******************************************\
File: AdderB.c
Author: Liam Pilling (15520564)

This file is responsible for PartB of the OS
assignment which involves adding up the numbers
in a file using threads.
\*******************************************/

#include "AdderB.h"

int main(int argc, char *argv[])
{
   LinkedList* values;
   values = (LinkedList*)malloc(sizeof(LinkedList));
   values->size = 0;
   values->head = NULL;
   values->tail = NULL;

   //If we don't have three command line strings then prompt user to
   //start again
   if(argc == 3)
   {
     readValues(values, argv[1]);
     calcSum(values, atoi(argv[2]));
   }
   else
   {
     printf("Incorrect command line values. Must be of format:\n");
     printf("./AdderA <InputFile> <NumberOfThreads>\n");
   }

   //free the allocated memory
   Node *currentNode, *node;
```

```
   currentNode = values->head;
   while(currentNode != NULL)
   {
     node = currentNode;
     currentNode = currentNode->next;
     free(node);
   }
   free(values);
}

void readValues(LinkedList *values, char *filename)
{
   FILE* file;
   Node* newValue;
   int value;

   file = fopen(filename, "r");
   if(file == NULL)
     perror("Error opening file\n");
   else
   {
     while(fscanf(file, "%d,", &value) == 1)
     {
     //Here we read the values in the file into a linked list
     newValue = (Node*)malloc(sizeof(Node));
     newValue->next = NULL;
     newValue->number = value;
     if(values->size == 0)
          values->head = newValue;
     else
          values->tail->next = newValue;
     values->tail = newValue;
     values->size++;
     }
     fclose(file);
   }
}

void calcSum(LinkedList *values, int threads)
{
   int i, rc, total;
   ThreadInfo *info;
   void *status;
```

```
   pthread_t thread[threads];
   Node *currentNode;

   info = (ThreadInfo*)malloc(threads*sizeof(ThreadInfo));
   //Initialise the global variables and buffer
   initGlobals();
   initBuffer(values);

   //Create all the information that each thread needs
   for(i = 0; i < threads; i++)
   {
     info[i].index = i;
     info[i].size = values->size;
     info[i].k = threads;
   }

   //Create n number of threads
   for(i = 0; i < threads; i++)
   {
     rc = pthread_create(&thread[i], NULL, threadAdd,
(void*)&info[i]);
   }
   total = parentConsumer(threads);

   printf("Total: %d\n", total);
   pthread_mutex_destroy(&mutex);
   pthread_exit(NULL);
}

void *threadAdd(void *information)
{
   int i, subTotal, assignedSize, tempSize, tempK;
   int tempAssignedSize, amount, index, size, k;
   void* ptr;
   ThreadInfo *info = information;

   pthread_t thisThread;
   thisThread = pthread_self();

   k = info->k;
   size = info->size;
   index = info->index;
   tempSize = size;
```

```
   tempK = k;
   assignedSize = (int)ceil((float)size/(float)k);
   i = subTotal = amount = 0;

   //This helps us to find where we are up to and to
   //evenly distribute values among the threads
   for(i=0;i <= index; i++)
   {
     tempAssignedSize = (int)ceil((float)tempSize/(float)tempK);
     tempK--;
     tempSize -= tempAssignedSize;

     if(i > 0)
     amount += assignedSize;
     if(tempAssignedSize < assignedSize)
     {
     assignedSize = tempAssignedSize;
     }
   }

   //Calculate the sub total
   for(i = amount; i < (amount+assignedSize); i++)
   {
     if(i < size)
     {
     subTotal += buffer[i];
     }
   }

   printf("Sub-total produced by Thread with ID%d: %d\n",
           (int)thisThread, subTotal);

   sem_wait(&st->empty);
   pthread_mutex_lock(&mutex);
/******** Critical section begins ***********/
   //Write the subtotal to the global variable
   st->value = subTotal;
/******** End of critical section ****************/
   pthread_mutex_unlock(&mutex);
   sem_post(&st->full);

   pthread_exit(NULL);
}
```

```
int parentConsumer(int k)
{
   int total, i;
   total = 0;

   for(i = 0; i < k; i++)
   {
     sem_wait(&st->full);
     pthread_mutex_lock(&mutex);

   /******** Critical section begins ***********/
     //Read the value in the shared memory block into SubTotal
     total += st->value;
   /****** Critical section ends ***************/

     pthread_mutex_unlock(&mutex);
     sem_post(&st->empty);
   }
   return total;
}

void initGlobals()
{
   //Initialise the values that we need
   st = (SubTotal*)malloc(sizeof(SubTotal));
   sem_init(&st->empty, 1, 1);
   sem_init(&st->full, 1, 0);
   pthread_mutex_init(&mutex, NULL);
}

void initBuffer(LinkedList *values)
{
   int i;
   i = 0;
   Node *currentNode;

   buffer = (int*)malloc(values->size * sizeof(int));

   currentNode = values->head;
   while(currentNode != NULL)
   {
     buffer[i] = currentNode->number;
```

```
      i++;
      currentNode = currentNode->next;
   }
}
```