Operating Systems Assignment
SDS

Kai Li Shi
19157364

2018 Semester 1

Pthreads

1. Mutual Exclusion

Mutual exclusion was achieved by using a total of two mutex variables and two condition variables. The main critical section was reading and writing the `data_buffer` array, which was stored as a global variable.

Mutex `readmutex` was used to ensure mutual exclusion between the reader and writer threads. Readers had priority over writers, so the first readers writers problem was implemented. The first reader to enter the critical section locks the `readmutex` mutex by calling `pthread_mutex_lock(&readmutex)`, and the last reader to exit the critical section unlocks the `readmutex` mutex by calling `pthread_mutex_unlock(&readmutex).` This allowed multiple reader threads to read at the same time. Only a single writer thread could write at a time, given that there were no reader threads holding the `readmutex` mutex.

Mutex `count` was used to ensure mutual exclusion between reader threads when updating the `readcount` variable (critical section).

Condition variable `empty` was used by the reader and writer threads to implement the bounded buffer problem. The writer thread calls `pthread_cond_wait(&empty, &readmutex)` after obtaining the `readmutex` lock, while there is no space in the buffer to write. The condition `emptycount == 0` must be true for this to be called. Once called, this allows the reader thread to obtain the `readmutex` lock and finish reading the `data_buffer.` Once the last reader of each index finishes reading, it increments `emptycount` and calls `pthread_cond_signal(&empty)`, signalling to the writer that there is a free spot in the buffer to write to.

Condition variable `full` was used by the reader to wait for the writer to write more data into the `data_buffer`. It is called while `fullcount == 0`, which indicates that it has finished reading everything in the buffer. The `readmutex` is given to the writer. Once the writer has finished writing a number to the buffer, it increments `fullcount` and signals to the readers that it can continue to read, by calling `pthread_cond_signal(&full).`

Condition variables `full` and `empty` were used to avoid starvation of writers, since priority was given to the readers.

2. Variables Used

- `numDWritten` keeps track of the total number of elements that have been written from `shared_data` to the `data_buffer`.

- `emptycount` keeps track of the number of free spots in the `data_buffer` to which the writer can write. If `emptycount` = 0 then there is no space left and writers must wait for readers to finish reading. Writers decrement `emptycount` and the last (slowest) reader for each index in the `data_buffer` increments `emptycount`. `emptycount` is initialised to `b` (the size of the buffer).

- `fullcount` keeps track of the number of full spots in the `data_buffer` from which the reader can read. If `fullcount` = 0 then all current data in the `data_buffer` has been read in and readers must wait for writers to write. Writers increment `fullcount` after writing each number to the `data_buffer`. Readers decrement `fullcount` after it finishes waiting for the writer to write. `fullcount` is initialised to 0.

- `readcount` keeps track of the number of readers that are currently reading. The `count` mutex variable is used to ensure mutual exclusion.

- `data_buffer` is an array of integers that is read by readers and written to by writers. The `readmutex` mutex variable is used to ensure mutual exclusion.

- `countarray` is an array of integers that keeps track of the number of readers that have read at each index. Each reader decrements the value at `countarray[i]` after reading the i^{th} element of the `data_buffer`. When `countarray[i]` = 0, `emptycount` is incremented and `countarray[i]` is reset to `r`.

- `sharedarray` is an array of integers that stores all numbers in the `shared_data` file. The parent process opens the file and writes all contents to the `sharedarray`, which is then used by the writer threads to write into the `data_buffer`.

3. Testing

There are no known errors in the program. Multiple test cases were conducted on the lab machines in building 314, with different values of r, w, t1 and t2. Memory leaks were found using Valgrind, but are caused by the pthread library.

3.1 Different number of readers and writer

- **More readers than writers**

```
[19157364@lab232-b04 pthreads]$ ./sds 5 2 1 1
```

sim_out file contents

```
1 reader--1464580352 has finished reading 100 pieces of data from the data_buffer
2 writer--1514936576 has finished writing 50 pieces of data to the data_buffer
3 writer--1506543872 has finished writing 50 pieces of data to the data_buffer
4 reader--1498151168 has finished reading 100 pieces of data from the data_buffer
5 reader--1472973056 has finished reading 100 pieces of data from the data_buffer
6 reader--1481365760 has finished reading 100 pieces of data from the data_buffer
7 reader--1489758464 has finished reading 100 pieces of data from the data_buffer
```

- **More writers than readers**

```
[19157364@lab232-b04 pthreads]$ ./sds 2 5 1 1
```

sim_out file contents:

```
1 writer-688240384 has finished writing 20 pieces of data to the data_buffer
2 writer-671454976 has finished writing 20 pieces of data to the data_buffer
3 writer-654669568 has finished writing 20 pieces of data to the data_buffer
4 writer-663062272 has finished writing 20 pieces of data to the data_buffer
5 writer-679847680 has finished writing 20 pieces of data to the data_buffer
6 reader-696633088 has finished reading 100 pieces of data from the data_buffer
7 reader-705025792 has finished reading 100 pieces of data from the data_buffer
```

- **Equal readers and writers**

```
[19157364@lab219-c03 pthreads]$ ./sds 2 2 1 1
```

sim_out file contents:

```
1 reader--506390784 has finished reading 100 pieces of data from the data_buffer
2 reader--497998080 has finished reading 100 pieces of data from the data_buffer
3 writer--523176192 has finished writing 50 pieces of data to the data_buffer
4 writer--514783488 has finished writing 50 pieces of data to the data_buffer
```

3.2 Different sleep times

- **`t1` and `t2` are 0**

```
[19157364@lab232-b04 pthreads]$ ./sds 5 2 0 0
```

sim_out file contents:

```
1 reader--1443428608 has finished reading 100 pieces of data from the data_buffer
2 reader--1451821312 has finished reading 100 pieces of data from the data_buffer
3 reader--1460214016 has finished reading 100 pieces of data from the data_buffer
4 reader--1435035904 has finished reading 100 pieces of data from the data_buffer
5 reader--1426643200 has finished reading 100 pieces of data from the data_buffer
6 writer--1468606720 has finished writing 50 pieces of data to the data_buffer
7 writer--1476999424 has finished writing 50 pieces of data to the data_buffer
```

- **`t1` and `t2` are more than 0**

```
[19157364@lab232-b04 pthreads]$ ./sds 5 2 3 4
```

sim_out file contents:

```
1 writer--1520359680 has finished writing 50 pieces of data to the data_buffer
2 writer--1511966976 has finished writing 50 pieces of data to the data_buffer
3 reader--1503574272 has finished reading 100 pieces of data from the data_buffer
4 reader--1478396160 has finished reading 100 pieces of data from the data_buffer
5 reader--1486788864 has finished reading 100 pieces of data from the data_buffer
6 reader--1470003456 has finished reading 100 pieces of data from the data_buffer
7 reader--1495181568 has finished reading 100 pieces of data from the data_buffer
```

3.3 Memory Leaks

Memory leaks are caused by libpthread.

```
==19727== HEAP SUMMARY:
==19727==     in use at exit: 1,576 bytes in 4 blocks
==19727==   total heap usage: 17 allocs, 13 frees, 8,408 bytes allocated
==19727==
==19727== 21 bytes in 1 blocks are still reachable in loss record 1 of 4
==19727==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==19727==    by 0x4019EF9: strdup (in /usr/lib64/ld-2.17.so)
==19727==    by 0x4017143: _dl_load_cache_lookup (in /usr/lib64/ld-2.17.so)
==19727==    by 0x4008A29: _dl_map_object (in /usr/lib64/ld-2.17.so)
==19727==    by 0x40143A3: dl_open_worker (in /usr/lib64/ld-2.17.so)
==19727==    by 0x400F8D3: _dl_catch_error (in /usr/lib64/ld-2.17.so)
==19727==    by 0x4013C8A: _dl_open (in /usr/lib64/ld-2.17.so)
==19727==    by 0x518F011: do_dlopen (in /usr/lib64/libc-2.17.so)
==19727==    by 0x400F8D3: _dl_catch_error (in /usr/lib64/ld-2.17.so)
==19727==    by 0x518F0D1: __libc_dlopen_mode (in /usr/lib64/libc-2.17.so)
==19727==    by 0x4E45E02: pthread_cancel_init (in /usr/lib64/libpthread-2.
17.so)
==19727==    by 0x4E45FCB: _Unwind_ForcedUnwind (in /usr/lib64/libpthread-2
.17.so)
```

3.4 Test Input Files

- **contents of the shared_data file that was used in testing**

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
```

4. Readme File

4.1 Purpose

The purpose of this program is to simulate the operation of the readers-writers and bounded buffer problem using threads.

4.2 Known bugs

There are no known bugs in the program. Testing was conducted on various lab machines in building 314.

4.3 Files

- sds.c
- sds.h
- makefile
- shared_data

4.4 To compile and run the program:

1. Navigate to the pthreads directory

```
cd Documents/OS/assignment/pthreads
```

2. Run make

```
make
```

3. Execute the program with the arguments [r] for the number of readers, [w] for the number of writers, [t1] for the reader sleep time, [t2] for the writer sleep time

```
./sds r w t1 t2
(example: ./sds 5 2 1 1)
```

4.5 To remove all object files:

1. Run make clean

```
make clean
```

5. Code

5.1 sds.c

```c
/*****************************************************
 * FILE: sds.c
 * DATE: 4/5/2018
 * UNIT: OS, 2018 S1
 * AUTHOR: Kai Li Shi 19157364
 * PURPOSE: Implements the Readers Writers problem for
 *          Simple Data Sharing using threads
 *****************************************************/

#include "sds.h"

int main (int argc, char * argv[])
{
    // check that all arguments are entered
    if (argc != 5)
    {
        printf("Usage: ./sds r w, t1 t2\n");
        return -1;
    }

    // rename argument names
    r = atoi(argv[1]);
    w = atoi(argv[2]);
    t1 = atoi(argv[3]);
    t2 = atoi(argv[4]);

    // validate arguments
    if (r < 1 || w < 1 || t1 < 0 || t2 < 0)
    {
        printf("Error. Invalid command line arguments\n");
        return -1;
    }

    // open shared_data file and add all elements to the sharedarray
    // for writers to access
    FILE * fp = fopen("shared_data", "r");
    if (fp == NULL)
    {
       printf("Error opening shared_data file\n");
       return -1;
    }
    for (int i = 0; i < d; i++)
    {
       int number;
       fscanf(fp, "%d,", &number);
      sharedarray[i] = number;
    }
    fclose(fp);
    fclose(fopen("sim_out","w"));

    // initialise countarray
    for (int i = 0; i < b; i++)
    {
        countarray[i] = r;
    }

    // launch threads. array of reader and writer threads
    pthread_t tidReader[r];
    pthread_t tidWriter[w];

    // create reader threads
    for (int i = 0; i < r; i++)
    {
        pthread_create(&tidReader[i], NULL, reader, NULL);
    }

    // create writer threads
    for (int i = 0; i < w; i++)
```

```
    {
        pthread_create(&tidWriter[i], NULL, writer, NULL);
    }

    // wait for reader threads
    for (int i = 0; i < r; i++)
    {
        pthread_join(tidReader[i], NULL);
    }

    // wait for writer threads
    for (int i = 0; i < w; i++)
    {
        pthread_join(tidWriter[i], NULL);
    }

    // free all memory
    pthread_mutex_destroy(&readmutex);
    pthread_mutex_destroy(&count);
    pthread_cond_destroy(&empty);
    pthread_cond_destroy(&full);

    return 0;
}

/* METHOD writer
 * IMPORTS ptr (void*)
 * EXPORTS void*
 * PURPOSE allows writers to write to the data_buffer one at a time */
void * writer ()
{
    int counter = 0;
    int exit = FALSE;
    while (!exit)
    {
       // lock the read mutex
        pthread_mutex_lock(&readmutex);

       // no space in the buffer to write
       // writer waits for the readers to finish reading
       while (emptycount == 0)
       {
           pthread_cond_wait(&empty, &readmutex);
       }

       // writer writes 1 number to the data_buffer
       if (numDWritten < d )
       {
           data_buffer[numDWritten%b] = sharedarray[numDWritten];
           numDWritten++;
           counter++;
           fullcount++;
           emptycount--;
       }
       else
       {
           exit = TRUE;
       }

       // signals to readers that they can read another number
       pthread_cond_signal(&full);
       //unlocks the read mutex
       pthread_mutex_unlock(&readmutex);
      sleep(t2);
    }

     // writes out message to sim_out
     char message[300];
     sprintf(message, "writer-%d has finished writing %d pieces of data to the
data_buffer\n", (int)pthread_self(), counter);
    writeMessage(message);
    pthread_exit(0);
}
```

```c
/* METHOD reader
 * IMPORTS none
 * EXPORTS void*
 * PURPOSE reads elements from the data_buffer while there are elements to read */
void * reader()
{
    int counter=0;

    while (counter < d)
    {
        // lock the count mutex, to update readcount
        pthread_mutex_lock(&count);
        readcount++;
        if (readcount == 1) // if it is the first reader then lock resource from writer
        {
            pthread_mutex_lock(&readmutex);
        }

        // reader has nothing to read. waits for the writer to write
        while (fullcount == 0)
        {
            pthread_cond_wait(&full, &readmutex);
            fullcount--;
        }

        pthread_mutex_unlock(&count);

        // continues reading while there are unread elements
        while (counter < numDWritten)
        {
            // read the element at data_buffer[counter%b]'

            countarray[counter%b]--; // decrease countarray
            // the last reader will reset the countarray for the index
            // signals to the writer that they can write
            if (countarray[counter%b] == 0)
            {
                countarray[counter%b] = r;
                emptycount++;
                pthread_cond_signal(&empty);
            }
            counter++;
        }

        // lock count mutex to update readcount
        pthread_mutex_lock(&count);
        readcount--;
        if (readcount == 0) // if it is the last reader then unlock resource from writer
        {
            pthread_mutex_unlock(&readmutex);
        }
        pthread_mutex_unlock(&count);
        sleep(t1);
    }

      // writes message to sim_out
      char message[300];
      sprintf(message, "reader-%d has finished reading %d pieces of data from the
data_buffer\n", (int)pthread_self(), counter);
     writeMessage(message);
     pthread_exit(0);
}


/* METHOD writeMessage
 * IMPORTS message (string)
 * EXPORTS none
 * PURPOSE writes message string to the "sim_out" file */
void writeMessage(char * message)
{
    FILE* fp = fopen("sim_out", "a");
    if (fp == NULL)
    {
        printf("error opening sim_out: %s\n", message);
```

```
        fclose(fp);
    }
    else
    {
        int error = fprintf(fp, message);
        if (error == EOF)
        {
            printf("error opening sim_out: %s\n", message);
        }
        fclose(fp);
    }
}
```

5.2 sds.h

```
/*******************************************************
 * FILE: sds.h
 * DATE: 5/5/2018
 * UNIT: OS, 2018 S1
 * AUTHOR: Kai Li Shi 19157364
 * PURPOSE: header file for sds.c
 *          contains method declarations and global variables.
 *******************************************************/

#ifndef ASSIGNMENT_SDS_H
#define ASSIGNMENT_SDS_H

#define FALSE 0
#define TRUE 1
#define d 100
#define b 20

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

int r, w, t1, t2;
int numDWritten = 0;
int emptycount= b;
int fullcount = 0;
int readcount = 0;
pthread_mutex_t readmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t count = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
int data_buffer[b];
int countarray[b];
int sharedarray[d];

void * writer ();
void * reader();
void writeMessage(char *message);

#endif //ASSIGNMENT_SDS_H
```

5.3 makefile

```
EXEC = sds
OBJ = sds.o
CFLAGS = -Wall -g -std=c99 -pthread -D_XOPEN_SOURCE=500 -lrt -Wno-deprecated -Wu>>nused-
variable
CC = gcc

$(EXEC) : $(OBJ)
    $(CC) $(OBJ) -o $(EXEC) $(CFLAGS)

sds.o : sds.c sds.h
    $(CC) -c sds.c $(CFLAGS)
```

```
clean:
	rm -f $(EXEC) $(OBJ)
```

Process

1. Mutual Exclusion

Mutual exclusion was achieved by using a total of four semaphores. The main critical section was reading and writing the `data_buffer` array, which was stored in shared memory.

Semaphore `read` was used to ensure mutual exclusion between the reader and writer processes. Readers had priority over writers, so the first readers writers problem was implemented. The first reader to enter the critical section would decrement `read` by calling `sem_wait(&locks->read)`, and the last reader to exit the critical section would increment `read` by calling `sem_post(&locks->read)`. This allowed multiple reader processes to read at the same time. Only a single writer process could write at a time, given that there were no reader processes holding the `read` lock.

Semaphore `count` was used to ensure mutual exclusion between reader processes when updating the `readcount` variable (critical section).

Semaphore `empty` was used by the reader and writer processes to implement the bounded buffer problem. The writer process calls `sem_wait(&locks->empty)` before obtaining the `read` lock, to ensure that there is space in the `data_buffer` array to write in. `empty` is decremented until 0, meaning that there is no space in the buffer to write in. The slowest reader for each index in the `data_buffer` array will call `sem_post(&locks->empty)` to increment it. This indicates to the writer that there is more space in the `data_buffer` to write in.

Semaphore `full` was used by the reader to wait for the writer to write to the buffer. Reader processes call `sem_wait(&locks->full)` if they are the fastest reader for the index, and there is no new data written to that index yet. Writer processes call `sem_post(&locks->full)` after writing, to let readers read the next index of the buffer. This increments the value of `full`.

Semaphores `full` and `empty` were used to avoid starvation of writers, since priority was given to the readers.

2. Variables Used

- `numDWritten` is an integer pointer that keeps track of the total number of elements that have been written from `shared_data` to the `data_buffer`. It is stored in shared memory
- `readcount` is an integer pointer that keeps track of the number of readers that are currently reading. The `count` semaphore variable is used to ensure mutual exclusion on this.
- `data_buffer` is an array of integers that is read by readers and written to by writers. The `read` semaphore variable is used to ensure mutual exclusion.
- `locks` is a pointer to a struct of `Locks`. It hold all the semaphores.
- `countarray` is an array of integers that keeps track of the number of readers that have read at each index. Each reader decrements the value at `countarray[i]` after reading the i^{th} element of the `data_buffer`. When `countarray[i]` = 0, `empty` is incremented (`sem_post(&locks->empty)` is called), and `countarray[i]` is reset to `r`.
- `fastestreader` is an integer pointer that keeps track of the highest index of the element that has been read. It is used by reader processes to allow the fastest reader to call `sem_wait(&locks->full)`. This indicates to the writer that the reader has caught up and is waiting for the writer to write more data into the buffer.

- `sharedarray` is an array of integers that stores all number in the `shared_data` file. The parent process opens the file and writes all contents to the `sharedarray`, which is then used by the writer processes to write into the `data_buffer`.

3. Testing

There are no known errors or memory leaks in the program. Multiple test cases were conducted on the lab machines in building 314, with different values of r, w, t1 and t2. Valgrind was used to check for memory leaks.

3.1 Different number of readers and writers

- **More readers than writers**

```
[19157364@lab232-b04 process]$ ./sds 5 2 1 1
```

sim_out file contents:

```
1 writer-16387 has finished writing 50 pieces of data to the data_buffer
2 writer-16388 has finished writing 50 pieces of data to the data_buffer
3 reader-16385 has finished reading 100 pieces of data from the data_buffer
4 reader-16382 has finished reading 100 pieces of data from the data_buffer
5 reader-16384 has finished reading 100 pieces of data from the data_buffer
6 reader-16383 has finished reading 100 pieces of data from the data_buffer
7 reader-16386 has finished reading 100 pieces of data from the data_buffer
```

- **More writers than readers**

```
[19157364@lab232-b04 process]$ ./sds 2 5 1 1
```

sim_out file contents:

```
1 writer-16582 has finished writing 20 pieces of data to the data_buffer
2 writer-16584 has finished writing 20 pieces of data to the data_buffer
3 writer-16581 has finished writing 20 pieces of data to the data_buffer
4 writer-16583 has finished writing 20 pieces of data to the data_buffer
5 writer-16580 has finished writing 20 pieces of data to the data_buffer
6 reader-16579 has finished reading 100 pieces of data from the data_buffer
7 reader-16578 has finished reading 100 pieces of data from the data_buffer
```

- **Equal readers and writers**

```
[19157364@lab219-c03 process]$ ./sds 2 2 1 1
```

sim_out file contents:

```
1 reader-7544 has finished reading 100 pieces of data from the data_buffer
2 writer-7547 has finished writing 50 pieces of data to the data_buffer
3 writer-7546 has finished writing 50 pieces of data to the data_buffer
4 reader-7545 has finished reading 100 pieces of data from the data_buffer
```

3.2 Different sleep times

- **`t1` and `t2` are 0**

```
[19157364@lab232-b04 process]$ ./sds 5 2 0 0
```

sim_out file contents:

```
1 writer-16669 has finished writing 42 pieces of data to the data_buffer
2 writer-16668 has finished writing 58 pieces of data to the data_buffer
3 reader-16664 has finished reading 100 pieces of data from the data_buffer
4 reader-16663 has finished reading 100 pieces of data from the data_buffer
5 reader-16667 has finished reading 100 pieces of data from the data_buffer
6 reader-16665 has finished reading 100 pieces of data from the data_buffer
7 reader-16666 has finished reading 100 pieces of data from the data_buffer
```

- **`t1` and `t2` are more than 0**

```
[19157364@lab232-b04 process]$ ./sds 5 2 3 4
```

sim_out file contents:

```
1 writer-16709 has finished writing 50 pieces of data to the data_buffer
2 writer-16708 has finished writing 50 pieces of data to the data_buffer
3 reader-16706 has finished reading 100 pieces of data from the data_buffer
4 reader-16707 has finished reading 100 pieces of data from the data_buffer
5 reader-16704 has finished reading 100 pieces of data from the data_buffer
6 reader-16703 has finished reading 100 pieces of data from the data_buffer
7 reader-16705 has finished reading 100 pieces of data from the data_buffer
```

3.3 Memory Leaks

```
==28408== HEAP SUMMARY:
==28408==     in use at exit: 0 bytes in 0 blocks
==28408==   total heap usage: 2 allocs, 2 frees, 1,136 bytes allocated
==28408==
==28408== All heap blocks were freed -- no leaks are possible
==28407== HEAP SUMMARY:
==28407==     in use at exit: 0 bytes in 0 blocks
==28407==   total heap usage: 2 allocs, 2 frees, 1,136 bytes allocated
==28407==
==28407== All heap blocks were freed -- no leaks are possible
```

3.4 Test Input Files

- **contents of the shared_data file that was used in testing**

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
```

4. Readme File

4.1 Purpose

The purpose of this program is to simulate the operation of the readers-writers and bounded buffer problem using processes.

4.2 Known bugs

There are no known bugs in the program. Testing was conducted on various lab machines in building 314.

4.3 Files

- sds.c
- sds.h
- makefile
- shared_data

4.4 To compile and run the program:

1. Navigate to the process directory

```
cd Documents/OS/assignment/process
```

2. Run make

```
make
```

3. Execute the program with the arguments [r] for the number of readers, [w] for the number of writers, [t1] for the reader sleep time, [t2] for the writer sleep time

```
./sds r w t1 t2
(example: ./sds 5 2 1 1)
```

4.5 To remove all object files:

1. Run make clean

```
make clean
```

5. Code

5.1 sds.c

```
/*********************************************************
 * FILE: sds.c
 * DATE: 4/5/2018
 * UNIT: OS, 2018 S1
 * AUTHOR: Kai Li Shi 19157364
 * PURPOSE: Implements the Readers Writers problem for
 *          Simple Data Sharing using processes
 *********************************************************/
```

```c
#include "sds.h"

int main (int argc, char * argv[])
{
    // check that all arguments are entered
    if (argc != 5)
    {
        printf("Usage: ./sds r w, t1 t2\n");
        return -1;
    }

    // rename argument names
    r = atoi(argv[1]);
    w = atoi(argv[2]);
    t1 = atoi(argv[3]);
    t2 = atoi(argv[4]);

    // validate arguments
    if (r < 1 || w < 1 || t1 < 0 || t2 < 0)
    {
       printf("Error. Command line arguments must be positive\n");
       return -1;
    }

    int parentID = getpid();

    // create shared memory
    int databuffer_fd = shm_open("databuffer", O_CREAT | O_RDWR, 0666);
    int readcount_fd = shm_open("readcount", O_CREAT | O_RDWR, 0666);
    int numDWritten_fd = shm_open("numDWritten", O_CREAT |O_RDWR, 0666);
    int locks_fd = shm_open("locks", O_CREAT | O_RDWR, 0666);
    int countarray_fd = shm_open("countarray", O_CREAT | O_RDWR, 0666);
    int fastestreader_fd = shm_open("fastestreader", O_CREAT | O_RDWR, 0666);
    int sharedarray_fd = shm_open("sharedarray", O_CREAT | O_RDWR, 0666);

    // error checking
    if (databuffer_fd == -1 || readcount_fd == -1 || locks_fd == -1 ||
            numDWritten_fd == -1 || fastestreader_fd == -1 || sharedarray_fd == -1)
    {
        printf("error creating shared data\n");
        return -1;
    }

    // configure sizes
    error += ftruncate(databuffer_fd, sizeof(int) * b);
    error += ftruncate(readcount_fd, sizeof(int));
    error += ftruncate(numDWritten_fd, sizeof(int));
    error += ftruncate(locks_fd, sizeof(Locks));
    error += ftruncate(countarray_fd, sizeof(int)*b);
    error += ftruncate(fastestreader_fd, sizeof(int));
    error += ftruncate(sharedarray_fd, sizeof(int)*d);

    if (error != 0)
    {
        printf("error ftruncating\n");
        return -1;
    }

    // pointer to the shared data
    int * data_buffer = (int*) mmap(0, sizeof(int) * b, PROT_WRITE | PROT_READ, MAP_SHARED,
                                    databuffer_fd, 0);
    int * readcount = (int*)mmap(0, sizeof(int), PROT_WRITE | PROT_READ, MAP_SHARED,
                                 readcount_fd, 0);
    int * numDWritten = (int*)mmap(0, sizeof(int), PROT_WRITE | PROT_READ, MAP_SHARED,
                                   numDWritten_fd, 0);
    Locks * locks = (Locks*)mmap(0, sizeof(Locks), PROT_WRITE | PROT_READ, MAP_SHARED,
                                 locks_fd, 0);
    int * countarray = (int*)mmap(0, sizeof(int)*b, PROT_WRITE | PROT_READ, MAP_SHARED,
                                  countarray_fd, 0);
    int * fastestreader = (int*)mmap(0, sizeof(int), PROT_WRITE | PROT_READ, MAP_SHARED,
                                     fastestreader_fd, 0);
    int * sharedarray = (int*)mmap(0, sizeof(int)*d, PROT_WRITE | PROT_READ, MAP_SHARED,
                                   sharedarray_fd, 0);
```

```
// initialise semaphores
error += sem_init(&locks->read, -1, 1);
error += sem_init(&locks->count, -1, 1);
error += sem_init(&locks->empty, -1, b);
error += sem_init(&locks->full, -1, 0);

if (error != 0)
{
    printf("error initialising semaphores\n");
    return -1;
}

// initialise the countarray
for (int i = 0; i < b; i++)
{
    countarray[i] = r;
}

// initialise shared memory
*readcount = 0;
*numDWritten = 0;
*fastestreader = -1;

// open the shared_data file and add all elements to
// the sharedarray for writers to access
FILE * fp = fopen("shared_data", "r");
if (fp == NULL)
{
    printf("Error opening shared_data file\n");
    return -1;
}
for (int i = 0; i < d; i++)
{
    int number;
    fscanf(fp, "%d,", &number);
    sharedarray[i] = number;
}
fclose(fp);
fclose(fopen("sim_out", "w"));


pid_t pid;

// create r children processes
for (int i = 0; i < r; i++)
{
    if (parentID == getpid())
    {
        pid = fork();
        if (pid == 0)
        {
            reader(locks, data_buffer, readcount, countarray,
                   numDWritten, fastestreader);
            exit(0);
        }
    }
}

// create w children processes
for (int i = 0; i < w; i++)
{
    if (parentID == getpid())
    {
        pid = fork();
        if (pid == 0)
        {
            writer(locks, data_buffer, sharedarray, numDWritten);
            exit(0);
        }
    }
}

// parent waits for children processes
```

```
    for (int i = 0; i < r+w; i++)
    {
        wait(NULL);
    }

    // clean up memory
    error += close(databuffer_fd);
    error += close(readcount_fd);
    error += close(numDWritten_fd);
    error += close(locks_fd);
    error += close(countarray_fd);
    error += close(fastestreader_fd);
    error += close(sharedarray_fd);
    error += sem_destroy(&locks->read);
    error += sem_destroy(&locks->count);
    error += sem_destroy(&locks->empty);
    error += sem_destroy(&locks->full);
    error += munmap(data_buffer, sizeof(int));
    error += munmap(readcount, sizeof(int));
    error += munmap(numDWritten, sizeof(int));
    error += munmap(locks, sizeof(Locks));
    error += munmap(countarray, sizeof(int) * b);
    error += munmap(fastestreader, sizeof(int));
    error += munmap(sharedarray, sizeof(int) *d);

    if (error != 0)
    {
        printf("error closing shared memory\n");
        return -1;
    }

    return 0;
}

/* METHOD writer
 * IMPORTS counter (int), locks (Locks*), data_buffer (int*), f (int*), numDWritten (int*)
 * EXPORTS none
 * PURPOSE allows writers to writer to the data_buffer one at a time*/
void writer(Locks * locks, int * data_buffer, int * f, int * numDWritten)
{
    int counter = 0;
    int exit = FALSE;
    while (!exit) //(*numDWritten < d)
    {
        // writer decrements empty
        sem_wait(&locks->empty);
        // writer decrements read
       sem_wait(&locks->read);

       // exit if 100 numbers already written
       if (*numDWritten == d)
        {
            exit = TRUE;
        }
        else // write 1 number to the buffer
        {
            data_buffer[*numDWritten % b] = f[*numDWritten];
           (*numDWritten)++;
           counter++;
        }

        // writer increments read
         sem_post(&locks->read);
        //writer increments full
         sem_post(&locks->full);
         sleep(t2);
    }
    sem_post(&locks->full);

    // writes out message to sim_out
    char message[300];
    sprintf(message, "writer-%d has finished writing %d pieces of data to the data_buffer\n",
getpid(), counter);
    writeMessage(message);
```

```
}

/* METHOD: reader
 * IMPORTS: locks (Locks *), data_buffer (int*), readcount (int*),
 *          countarray (int*), numDWritten (int*), fastestreader (int*)
 * EXPORTS: none
 * PURPOSE:  reads elements from the data_buffer while there are elements to read.
 *           the last reader of each index will call sem_post().
 *           the first reader of each index will call sem_wait().*/
void reader(Locks * locks, int * data_buffer, int * readcount, int* countarray, int *
numDWritten, int * fastestreader)
{
    int counter = 0;
    while (counter < d)
    {
       // the fastest reader will call sem_wait(&locks->full)
       if ((countarray[counter%b] == r) && *fastestreader < counter)
       {
           *fastestreader = counter;
           sem_wait(&locks->full);
       }

       // lock the count to update readcount
        sem_wait(&locks->count);
        (*readcount)++;
        if (*readcount == 1)
        {
           // lock the read semaphore if it is the first reader
            sem_wait(&locks->read);
        }

        sem_post(&locks->count);

       // keep reading while there are values in the data_buffer
        while (counter < *numDWritten)
        {
            // read the value at data_buffer[counter%b]
            countarray[counter%b]--;
            if (countarray[counter%b] == 0)
            {
                 // if it is the last reader for the index
                 // then reset the counter and let the
                 // writer know that it can overwrite the value
                 countarray[counter%b] = r;
                 sem_post(&locks->empty);
            }
            counter++;
        }

        // update readcount
       sem_wait(&locks->count);
       (*readcount)--;
       if (*readcount == 0)
       {
           // unlock read semaphore if it is the last reader
           sem_post(&locks->read);
       }
       sem_post(&locks->count);
       sleep(t1);
    }

      // writes message to sim_out
      char message[300];
      sprintf(message, "reader-%d has finished reading %d pieces of data from the
data_buffer\n", getpid(), counter);
      writeMessage(message);
}


/* METHOD: writeMessage
 * IMPORTS: message (string)
 * EXPORTS: none
 * PURPOSE: writes message string to the "sim_out" file */
void writeMessage(char * message)
```

```
{
    FILE* fp = fopen("sim_out", "a");
    if (fp == NULL)
    {
        printf("error opening sim_out: %s\n", message);
        fclose(fp);
    }
    else
    {
        int error = fprintf(fp, message);
        if (error == EOF)
        {
            printf("error opening sim_out: %s\n", message);
        }
        fclose(fp);
    }
}
```

5.2 sds.h

```
/*********************************************************
 * FILE: sds.h
 * DATE: 5/5/2018
 * UNIT: OS, 2018 S1
 * AUTHOR: Kai Li Shi 19157364
 * PURPOSE: header file for sds.c
 *          contains method declarations and global variables
 * *******************************************************/

#ifndef PROCESSES_SDS_H
#define PROCESSES_SDS_H

#define FALSE 0
#define TRUE 1
#define d 100
#define b 20

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

typedef struct {
    sem_t read;
    sem_t count;
    sem_t full;
    sem_t empty;
} Locks;

int r, w, t1, t2, error;

void writeMessage(char * message);
void writer(Locks * locks, int * data_buffer, int * f, int * numDWritten);
void reader(Locks * locks, int * data_buffer, int * readcount, int * countarray,
            int * numDWritten, int * fastestreader);

#endif //PROCESSES_SDS_H
```

5.3 makefile

```
EXEC = sds
OBJ = sds.o
```

```
CFLAGS = -Wall -g -std=c99 -pthread -D_XOPEN_SOURCE=500 -lrt -Wno-deprecated -Wu>>nused-
variable
CC = gcc

$(EXEC) : $(OBJ)
    $(CC) $(OBJ) -o $(EXEC) $(CFLAGS)

sds.o : sds.c sds.h
    $(CC) -c sds.c $(CFLAGS)

clean:
    rm -f $(EXEC) $(OBJ)
```