Document version: 1.1 (2015-11-15)

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Tan	Student ID:	18249833
Other name(s):	Xhien Yi		
Unit name:	Operating System	Unit ID:	COMP2006
Lecturer / unit coordinator:	Soh	Tutor:	Arlen
Date of submission:	7 May, 2018	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature:  Date of signature: 7 May, 2018

(By submitting this form, you indicate that you agree with all the above text.)

Codes:

Thread:

```
1.  /**
2.   * FILE:    sds.c
3.   * AUTHOR:  Xhien Yi Tan
4.   * STUDENTID:   18249833
5.   * UNIT:    COMP2006
6.   * PURPOSE: This file is responsible for running the program using threads, especially Reader-
    Writer Problem
7.   */
8.
9.  #include <stdio.h>
10. #include <pthread.h>
11. #include <stdlib.h>
12. #include <stdint.h>
13. #include <string.h>
14. #include<semaphore.h>
15. #include<ctype.h>
16. #include "file.h"
17.
18. #define READER 0
19. #define WRITER 1
20. #define B 20
21. #define D 100
22.
23. pthread_mutex_t read, wrt, consume, output;
24.
25. int *dataFromFile;
26. char *filename = "shared_data.txt", *outputFileName = "sim_out.txt";
27. int t1, t2, readCount = 0, writeCount = 0, dataCount = 0, dataReadFilePosition = 0, bufferInde
    x = 0, b = B, d = D;
28. int data_buffer[B];
29. pthread_cond_t condW, condR;
30. int dataReadFromReader = 0, readersStopped = 0, iniNumWriter = 0, initNumReader = 0, finish =
    0, totalDataHasBeenRead = 0;
31.
32.
33. /* Struct for thread:
34.  * Counter - each thread has this counter
35.  * id - id for each thread
36.  * thread - thread object itself
37.  */
38.
39. typedef struct{
40.     int counter, id;
41.     pthread_t thread;
42. } Thread;
43.
44. /**
45. * PURPOSE : Consume data and increases each reader's counter
46. * IMPORTS : Reader thread
47. * EXPORTS : None
48. */
49. void *consumeData(void *thread) {
50.
51.     int i;
52.
53.     /* From i to where the writers stop reading */
54.     for (i = 0; i < totalDataHasBeenRead; i++) {
55.
56.         dataReadFromReader++;
57.
58.
```

```
59.         if ( ( ( ( Thread* )thread )->counter ) < d ) {
60.             ( ( ( Thread* )thread )->counter ) = ( ( ( Thread* )thread )->counter ) + 1;
61.         }
62.
63.         /* If a reader has finished consume data until the end of data_buffer */
64.         if ((i+1) == b)
65.         {
66.             /* Keep track of how many readers are waiting */
67.             readersStopped++;
68.
69.             /* If the number of readers waiting is the same as the initial readers, notify wri
    ters */
70.             if ( readersStopped == initNumReader )
71.             {
72.
73.                 dataReadFilePosition = 0;
74.                 readersStopped = 0;
75.
76.                 pthread_cond_broadcast(&condW);
77.                 pthread_mutex_unlock(&wrt);
78.
79.             }
80.
81.             /* If no writers left and this reader has finished reading all the data, then fini
    sh */
82.             if ( writeCount == 0 && ( ( ( Thread* )thread )->counter ) == d )
83.             {
84.
85.                 finish = 1;
86.             } else {
87.
88.                 /* Readers wait here */
89.                 pthread_mutex_unlock(&wrt);
90.                 pthread_cond_wait(&condR, &consume);
91.             }
92.         }
93.     }
94.
95.     return NULL;
96. }
97.
98.
99. /**
100.* PURPOSE : Read data from buffer and print read data
101.* IMPORTS : None
102.* EXPORTS : None
103.*/
104.void *readFromBuffer() {
105.
106.    int i;
107.
108.    /* To enable printing data, uncomment this */
109.
110.    /*
111.    *for( i = 0; i < b; i++ ) {
112.    *   printf("data_buffer data - %d\n", data_buffer[i]);
113.    *}
114.    */
115.
116.    return NULL;
117.}
118.
119.
120./**
121.* PURPOSE : Method for readers:
122.*           - To start reading from the buffer
```

```
*           - Sleeps after incrementing counter and reading
* IMPORTS : Reader thread
* EXPORTS : None
*/
void *reader(void *thread) {

    int done = 0;


    do {


        pthread_mutex_lock(&read);
        readCount = readCount + 1;

        if (readCount == 1)
        {
            pthread_mutex_lock(&wrt);
        }

        pthread_mutex_unlock(&read);

        /* Read data together */
        readFromBuffer();
        sleep(t1);

        /* Consume data */
        pthread_mutex_lock(&consume);
        consumeData(thread);
        pthread_mutex_unlock(&consume);


        /* When the finish flag is set to 1 by writer and this reader has finished consuming,
         * set done flag = 1 means done.
         */
        if (finish == 1 && ( ( ( Thread* )thread )->counter ) == d )
        {
            done = 1;
        }


        pthread_mutex_lock(&read);

        readCount = readCount - 1;
        if ( readCount == 0 )
        {

            pthread_cond_broadcast(&condW);
            pthread_mutex_unlock(&wrt);

        }

        pthread_mutex_unlock(&read);


    } while( done != 1 );

    printf("Reader ID %d has finished reading %d pieces of data from the data_buffer \n", ( (
    ( Thread* )thread )->id ),( ( ( Thread* )thread )->counter ) );


    pthread_mutex_lock(&output);
    writeToFile(outputFileName, thread, READER);
    pthread_mutex_unlock(&output);

```

```
    return NULL;
}

/**
* PURPOSE : Read data from shared_data and put it into buffer
* IMPORTS : Writer thread
* EXPORTS : None
*/
void *readFromSharedData(void* thread) {

    int done, i;


    do {
        dataReadFilePosition++;

        /* If all data from shared_data has been read, set flag to finish */
        if ( totalDataHasBeenRead == d ) {


            done = 1;
            finish = 1;

        /* If data_buffer is full, wait for readers to read and consume */
        } else if ( dataReadFilePosition > b ) {


            while( dataReadFilePosition > b ) {

                    /* Signal all readers to read */
                for (i = 0; i < initNumReader; i++) {
                    pthread_cond_signal(&condR);
                }

                if ( finish == 1 ) {
                    done = 1;
                    break;
                } else {

                    /* If only one reader left. wake it up */
                    if (readCount == 0) {
                        pthread_mutex_unlock(&read);
                    } else {
                        /* Writers wait here */
                        pthread_cond_wait(&condW, &wrt);
                    }

                }
            }

        } else {

            totalDataHasBeenRead++;
            ( ( ( Thread* )thread )->counter ) = ( ( ( Thread* )thread )->counter ) + 1;
            data_buffer[dataReadFilePosition - 1] = dataFromFile[dataReadFilePosition - 1];
        }

    } while( done != 1);


  return NULL;
}

/**
* PURPOSE : Method for writers:
```

```
254.*           - To start reading from the shared_data and put to buffer
255.*           - Sleeps after incrementing counter and writing
256.* IMPORTS : Writer thread
257.* EXPORTS : None
258.*/
259.void * writer(void *thread)
260.{
261.    int i,j;
262.
263.    pthread_mutex_lock(&wrt);
264.
265.    writeCount = writeCount + 1;
266.
267.
268.    readFromSharedData(thread);
269.    sleep(t2);
270.
271.
272.    pthread_mutex_unlock(&wrt);
273.
274.    /* When one of the writers come out from its CS.
275.     * means that writers has finish writing to buffer.
276.     * So that change the finish flag to 1 - means finished.
277.     */
278.    finish = 1;
279.
280.    for (i = 0; i < initNumReader; i++) {
281.        pthread_cond_signal(&condR);
282.    }
283.
284.
285.    writeCount = writeCount - 1;
286.
287.    printf("writer-%d has finished writing %d pieces of data to the data_buffer \n", ( ( ( Thr
    ead* )thread )->id ), ( ( ( Thread* )thread )->counter ));
288.
289.    pthread_mutex_lock(&output);
290.    writeToFile(outputFileName, thread, WRITER);
291.    pthread_mutex_unlock(&output);
292.
293.
294.    return NULL;
295.}
296.
297.
298./**
299.* PURPOSE : Method that kick starts the program
300.* IMPORTS : Number of readers, number of writers, t1, t2
301.* EXPORTS : None
302.*/
303.int main(int argc, char *argv[])
304.{
305.    int i;
306.
307.    initNumReader = atoi( argv[1] );
308.    iniNumWriter = atoi( argv[2] );
309.
310.    t1 = atoi( argv[3] );
311.    t2 = atoi( argv[4] );
312.
313.    if( t1 < 0 || t2 < 0 || initNumReader <= 0 || iniNumWriter <= 0 ) {
314.        printf("Invalid Inputs - Inputs can't be negative and Readers - Writers can't be zero\
    n");
315.    } else {
316.
317.        /* Initialise mutexes and conditional variables */
```

```
318.        pthread_t Readers_thr[initNumReader-1],Writer_thr[iniNumWriter-1];
319.        pthread_mutex_init(&read, NULL);
320.        pthread_mutex_init(&wrt, NULL);
321.        pthread_mutex_init(&output, NULL);
322.        pthread_mutex_init(&consume, NULL);
323.        pthread_cond_init(&condW, NULL);
324.        pthread_cond_init(&condR, NULL);
325.        Thread *writerThread,*readerThread;
326.
327.        /* Clean output file before writing */
328.        clearContentOutputFile(outputFileName);
329.
330.        /* Read file */
331.        dataFromFile = readFile(filename,d);
332.
333.        /* Create Writers and Readers */
334.        for(i=0;i<iniNumWriter;i++)
335.        {
336.            writerThread = ( Thread* )malloc( sizeof( Thread ) );
337.            writerThread->id = i;
338.            writerThread->counter = 0;
339.            pthread_create(&Writer_thr[i],NULL,writer,(void *)writerThread);
340.        }
341.
342.        for(i=0;i<initNumReader;i++)
343.        {
344.            readerThread = ( Thread* )malloc( sizeof( Thread ) );
345.            readerThread->id = i;
346.            readerThread->counter = 0;
347.            pthread_create(&Readers_thr[i],NULL,reader,(void *)readerThread);
348.        }
349.
350.        for(i=0;i<iniNumWriter;i++)
351.        {
352.            pthread_join(Writer_thr[i],NULL);
353.        }
354.
355.        for(i=0;i<initNumReader;i++)
356.        {
357.            pthread_join(Readers_thr[i],NULL);
358.        }
359.
360.        /* Clean up resources */
361.        pthread_mutex_destroy(&read);
362.        pthread_mutex_destroy(&wrt);
363.        pthread_mutex_destroy(&output);
364.        pthread_mutex_destroy(&consume);
365.        free(readerThread);
366.        free(writerThread);
367.    }
368.
369.
370.
371.    return 0;
372.}
```

```c
/**
 * FILE:    file.c
 * AUTHOR:  Xhien Yi Tan
 * STUDENTID:   18249833
 * UNIT:    COMP2006
 * PURPOSE: This file incharge of writing and reading data from file
 */


#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include<semaphore.h>
#include<ctype.h>
#include "file.h"

#define READER 0
#define WRITER 1

typedef struct{
    int counter, id;
    pthread_t thread;
} Thread;


/**
* PURPOSE : Clear sim_out file everytime we run this program,
*           so that the previous data is cleared before putting in new data.
* IMPORTS : Output file name
* EXPORTS : None
*/
void *clearContentOutputFile( char *outputfilename ) {

    FILE *f;
    f = fopen( outputfilename, "w" );

    if( f == NULL )
    {
        perror( outputfilename );
    }
    fclose(f);

    return NULL;

}

/**
* PURPOSE : Read file and store the data into array
* IMPORTS : Input file name, the total size for shared_data
* EXPORTS : Data from file
*/
int* readFile( char *filename, int d )
{
    FILE *f;
    int check, done = 0,i = 0, num;
    int *dataArray;


    dataArray = ( int* )malloc( d * sizeof( int ) );
    f = fopen( filename, "r" );


    if( f == NULL )
    {
```

```
67.         perror( filename );
68.     } else {
69.
70.         do {
71.
72.             check = fscanf( f, "%d ", &num );
73.
74.
75.             if ( check == EOF ) {
76.                 done = 1;
77.             } else if ( i == d ) {
78.                 done = 1;
79.             } else {
80.                 dataArray[i] = num;
81.                 i++;
82.             }
83.
84.         } while( done != 1);
85.     }
86.
87.     fclose(f);
88.
89.     return dataArray;
90. }
91.
92.
93. /**
94. * PURPOSE : Write data to file
95. * IMPORTS : Output file name, thread that wants tp input data, type - READER or WRI
    TER
96. * EXPORTS : None
97. */
98. void *writeToFile(char *filename, void* thread, int type) {
99.
100.                FILE *f;
101.                f = fopen( filename, "a" );
102.
103.                if( f == NULL ) {
104.                    perror( filename );
105.                } else {
106.
107.                    if ( type == READER ) {
108.                        fprintf(f, "reader-%d has finished reading %d pieces of data
     from the data_buffer. \n", ( ( ( Thread* )thread )->id ),( ( ( Thread* )thread )->
    counter ) );
109.                    } else {
110.                        fprintf(f, "writer-%d has finished writing %d pieces of data
     to the data_buffer. \n", ( ( ( Thread* )thread )->id ),( ( ( Thread* )thread )->co
    unter ) );
111.                    }
112.
113.                }
114.
115.                fclose(f);
116.                return NULL;
117.
118.        }
```

Process:

```c
/**
 * FILE:    sds.c
 * AUTHOR:  Xhien Yi Tan
 * STUDENTID:   18249833
 * UNIT:    COMP2006
 * PURPOSE: This file is responsible for running the program using process, especia
lly Reader-Writer Problem
 */

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

#include "file.h"

#define READER 0
#define WRITER 1
#define B 20
#define D 100
#define MAX_PROCESS 7

//Shared variables
int *dataFromFile, *dataBuffer;
char *filename = "shared_data.txt", *outputFileName = "sim_out.txt";
int t1, t2, dataReadFilePosition = 0,  bufferIndex = 0, b = B, d = D;

sem_t *writer_sem, *reader_sem, *consume_sem, *condR_sem, *condW_sem, *output_sem;

int data_buffer[B];
int dataReadFromReader = 0, readersStopped = 0, initNumWriter = 0, initNumReader =
0, finish = 0, totalDataHasBeenRead = 0;

int i, pid, shm_DataFromFile,shm_DataBuffer;
int *ptr_readCount, *ptr_writeCount, *ptr_initNumReader, *ptr_initNumWriter, *ptr_t
1, *ptr_t2;
int *ptr_done, *ptr_finish, *ptr_dataReadFilePosition, *ptr_totalDataHasBeenRead, *
ptr_dataReadFromReader, *ptr_readerStopped;
void *ptr_DataFromFile, *ptr_DataBuffer;


/* Struct for process:
 * Counter - each process has this counter
 * id - id for each process
 */
typedef struct{
    int counter, id;
} Process;


/**
* PURPOSE : Consume data and increases each reader's counter
* IMPORTS : Reader process
* EXPORTS : None
*/
void *consumeData(void* readerProcess) {
int i;
    //If no data has been read by writers or data buffer is empty, waits
```

```
62.     if ( *ptr_totalDataHasBeenRead == 0 )
63.     {
64.         /* While this reader process waits, unlocks other reader processes.
65.          * At the same time wake up writers to read and write.
66.          */
67.         sem_post(reader_sem);
68.         sem_post(writer_sem);
69.         sem_wait(condR_sem);
70.
71.     } else {
72.
73.         //From 0 to b - meaning from 0 to buffer size
74.         for (i = 0; i < *ptr_totalDataHasBeenRead; i++) {
75.
76.             *ptr_dataReadFromReader = *ptr_dataReadFromReader + 1;
77.
78.             //Increases the reader counter when it reads data
79.             if ( ( ( ( Process* )readerProcess )->counter ) < d ) {
80.                 ( ( ( Process* )readerProcess )->counter ) = ( ( ( Process* )reader
    Process )->counter ) + 1;
81.             }
82.
83.
84.             //When it reads until the end of data buffer
85.             if ( (i+1) == b )
86.             {
87.                 //Keep track number of waiting readers
88.                 *ptr_readerStopped = *ptr_readerStopped + 1;
89.
90.                 //When all readers finished reading - wake up writers
91.                 if ( *ptr_readerStopped == initNumReader )
92.                 {
93.                     //Reset the index of the data buffer to 0
94.                     *ptr_dataReadFilePosition = 0;
95.                     *ptr_readerStopped = 0;
96.                     sem_post(condW_sem);
97.                 }
98.
99.                 // When there is one last writer and this reader has finished readi
    ng all the data in the shared_data
100.                     if ( *ptr_writeCount == 0 && ( ( ( Process* )readerProcess )
    ->counter ) == d )
101.                     {
102.                         //Set finish
103.                         *ptr_finish = 1;
104.                     } else {
105.
106.                         //Else wake up writers and wait
107.                         sem_post(writer_sem);
108.                         sem_post(consume_sem);
109.                         sem_post(condW_sem);
110.                         sem_wait(condR_sem);
111.
112.                     }
113.                 }
114.
115.             }
116.         }
117.
118.         return NULL;
119.     }
120.
121.
122.     /**
123.     * PURPOSE : Read data from buffer and print read data
124.     * IMPORTS : Reader Process
```

```
125.        * EXPORTS : None
126.        */
127.       void *readFromBuffer(void *readerProcess) {
128.
129.           //Reading data together - uncomment this if you want to print
130.           /*
131.            *for (int i = 0; i < b; i++) {
132.            *  printf("Reader ID %d priting data_buffer data - %d\n", ( ( ( Process
    * )readerProcess )->id ), ( ( int* )ptr_DataBuffer)[i]);
133.            } */
134.           return NULL;
135.       }
136.
137.       /**
138.       * PURPOSE : Method for readers:
139.       *           - To start reading from the buffer
140.       *           - Sleeps after incrementing counter and reading
141.       * IMPORTS : Reader process
142.       * EXPORTS : None
143.       */
144.       void* reader(void *readerProcess) {
145.
146.           int done = 0, i;
147.
148.           do {
149.
150.               //Lock CS for updating read count
151.               sem_wait( reader_sem );
152.               *ptr_readCount = *ptr_readCount + 1;
153.
154.               if ( *ptr_readCount == 1) {
155.                   sem_wait( writer_sem );
156.               }
157.
158.               sem_post( reader_sem );
159.
160.
161.               //Read data from buffer together
162.               readFromBuffer(readerProcess);
163.               sleep( *ptr_t1 );
164.
165.               //Lock CS for consuming data - consume one after one
166.               sem_wait( consume_sem );
167.               consumeData(readerProcess);
168.               sem_post( consume_sem );
169.
170.
171.               /* When the finish flag is set to 1 by writer and this reader has fi
    nished consuming,
172.                * set done flag = 1 means done.
173.                */
174.               if (*ptr_finish == 1 && ( ( ( Process* )readerProcess )->counter ) =
    = d ) {
175.                   done = 1;
176.               }
177.
178.               //Lock CS for decrement read count
179.               sem_wait( reader_sem );
180.
181.               *ptr_readCount = *ptr_readCount - 1;
182.
183.               if ( *ptr_readCount == 0) {
184.                   //Wake up remaining writers
185.                   sem_post( condW_sem );
186.                       sem_post( writer_sem );
187.               }
```

```
188.
189.                 sem_post( reader_sem );
190.             } while( done != 1 );
191.
192.             printf("Reader ID %d has finished reading %d pieces of data from the dat
    a_buffer \n", ( ( ( Process* )readerProcess )->id ),( ( ( Process* )readerProcess )
    ->counter ) );
193.
194.
195.             //Lock CS for writing data to shared output file
196.             sem_wait( output_sem );
197.             writeToFile(outputFileName, readerProcess, READER);
198.             sem_post( output_sem );
199.
200.
201.
202.
203.             return NULL;
204.         }
205.
206.         /**
207.         * PURPOSE : Read data from shared_data and put it into buffer
208.         * IMPORTS : Writer process
209.         * EXPORTS : None
210.         */
211.         void *readFromSharedData(void *writerProcess) {
212.
213.             int done = 0;
214.
215.             do {
216.
217.                 //Increments when data has been read by writers
218.                 *ptr_dataReadFilePosition = *ptr_dataReadFilePosition + 1;
219.
220.                 //This is where EOF happens
221.                 if ( *ptr_totalDataHasBeenRead == d ) {
222.
223.                     //Set flags done and fnish to 1
224.                     done = 1;
225.                     *ptr_finish = 1;
226.
227.                 //If the buffer is full
228.                 } else if ( *ptr_dataReadFilePosition > b ) {
229.
230.                     //While the buffer is full, writer waits
231.                     while( *ptr_dataReadFilePosition > b ) {
232.
233.                         //If fnished the stops this process
234.                         if ( *ptr_finish == 1 )
235.                         {
236.                             done = 1;
237.                             break;
238.                         }
239.
240.
241.                         /* Wake up other writers and readers that are waiting,
242.                          * then waits since buffer is full.
243.                          */
244.                         sem_post(writer_sem);
245.                         sem_post(condR_sem);
246.                         sem_wait(condW_sem);
247.
248.                     }
249.
250.                 } else {
251.
```

```
252.                    //Put data into buffer from shared_data and increases the counte
    r for this writer
253.                    *ptr_totalDataHasBeenRead = *ptr_totalDataHasBeenRead + 1;
254.                    ( ( ( Process* )writerProcess )->counter ) = ( ( ( Process* )wri
    terProcess )->counter ) + 1;
255.                    ( ( int* )ptr_DataBuffer)[*ptr_dataReadFilePosition - 1 ] = ( (
    int* )ptr_DataFromFile)[*ptr_dataReadFilePosition - 1];
256.                }
257.
258.            } while( done != 1);
259.
260.
261.
262.            return NULL;
263.        }
264.
265.        /**
266.        * PURPOSE : Method for writers:
267.        *           - To start reading from the shared_data and put to buffer
268.        *           - Sleeps after incrementing counter and writing
269.        * IMPORTS : Writer process
270.        * EXPORTS : None
271.        */
272.        void* writer(void *writerProcess) {
273.
274.            int i;
275.            //Lock critical section for writer
276.            sem_wait( writer_sem );
277.
278.            *ptr_writeCount = *ptr_writeCount + 1;
279.
280.            //Read from shared_data and put into buffer
281.            readFromSharedData(writerProcess);
282.            sleep(*ptr_t2);
283.
284.            //Unlock critical section for writer
285.            sem_post( writer_sem );
286.
287.            /* When one of the writers finish reading,
288.             * means that data_buffer has been finished reading.
289.             * Set finish flag = 1, let all others processes knows.
290.             */
291.            *ptr_finish = 1;
292.
293.            *ptr_writeCount = *ptr_writeCount - 1;
294.
295.            printf("writer-%d has finished writing %d pieces of data to the data_buf
    fer \n", ( ( ( Process* )writerProcess )->id ), ( ( ( Process* )writerProcess )->co
    unter ));
296.
297.            //Another CS for writing the result to a shared output file
298.            sem_wait( output_sem );
299.            writeToFile(outputFileName, writerProcess, WRITER);
300.            sem_post( output_sem );
301.
302.            //If there is any writers left and all data has been written, wake up wr
    iters
303.            if ( *ptr_writeCount != 0 && *ptr_finish == 1 ) {
304.                for( i = 0; i < *ptr_writeCount; i++ ) {
305.                    sem_post(condW_sem);
306.                }
307.
308.            }
309.            return NULL;
310.        }
311.
```

```
312.
313.
314.        /**
315.        * PURPOSE : Create readers and writers
316.        * IMPORTS : The sum of readers and writers
317.        * EXPORTS : None
318.        */
319.        void createProcessors(int nprocesses)
320.        {
321.            pid_t pid;
322.            int i;
323.            Process *writerProcess,*readerProcess;
324.
325.            int parentId = getpid();
326.
327.            //Create Maximum number of processes
328.            for (i = 0; i < MAX_PROCESS; ++i)
329.            {
330.                fork();
331.            }
332.
333.            int childPid = getpid() - parentId;
334.
335.            //Create writers
336.            if(childPid > 0 && childPid <= *ptr_initNumWriter) {
337.                writerProcess = ( Process* )malloc( sizeof( Process ) );
338.                writerProcess->id = childPid;
339.                writerProcess->counter = 0;
340.                writer(writerProcess);
341.            }
342.
343.            //Create ID for readers
344.            int readerLeft = 0;
345.            if ( *ptr_initNumWriter == *ptr_initNumReader ) {
346.                readerLeft = *ptr_initNumWriter + *ptr_initNumReader;
347.            } else  {
348.                readerLeft = ( *ptr_initNumWriter ) + *ptr_initNumReader;
349.            }
350.
351.            //Create readers
352.            if(childPid > *ptr_initNumWriter && childPid <= readerLeft) {
353.                readerProcess = ( Process* )malloc( sizeof( Process ) );
354.                readerProcess->id = childPid - *ptr_initNumWriter;
355.                readerProcess->counter = 0;
356.                reader(readerProcess);
357.            }
358.
359.
360.        }
361.
362.
363.        /**
364.        * PURPOSE : Method that kick starts the program
365.        * IMPORTS : Number of readers, number of writers, t1, t2
366.        * EXPORTS : None
367.        */
368.        int main ( int argc, char *argv[] )
369.        {
370.            int i, j;
371.
372.            //Set initial numbers for both readers and writers
373.            initNumReader = atoi( argv[1] );
374.            initNumWriter = atoi( argv[2] );
375.
376.            //Convert arguments to both t1 and t2
377.            t1 = atoi( argv[3] );
```

```
378.           t2 = atoi( argv[4] );
379.
380.
381.           if( t1 < 0 || t2 < 0 || initNumReader <= 0 || initNumWriter <= 0 ) {
382.               printf("Invalid Inputs - Inputs can't be negative and Readers - Writ
    ers can't be zero\n");
383.           } else {
384.
385.               //Clear output file before doing anything else
386.               clearContentOutputFile(outputFileName);
387.
388.               //Read data from the file
389.               dataFromFile = readFile(filename,d);
390.
391.
392.               //Create shared memory data
393.               shm_DataFromFile = shm_open( "dataFromFile", O_CREAT | O_RDWR, 0666
    );
394.               ftruncate( shm_DataFromFile, sizeof( dataFromFile ) );
395.               ptr_DataFromFile = mmap( NULL, sizeof( dataFromFile ), PROT_READ | P
    ROT_WRITE, MAP_SHARED, shm_DataFromFile, 0 );
396.
397.               shm_DataBuffer = shm_open( "data_Buffer", O_CREAT | O_RDWR, 0666 );

398.               ftruncate( shm_DataBuffer, sizeof( dataBuffer ) );
399.               ptr_DataBuffer = mmap( NULL, sizeof( dataBuffer ), PROT_READ | PROT_
    WRITE, MAP_SHARED, shm_DataBuffer, 0 );
400.
401.
402.               //Create shared memory data
403.               ptr_t1 = mmap(NULL, sizeof *ptr_t1, PROT_READ | PROT_WRITE, MAP_SHAR
    ED | MAP_ANONYMOUS, 0, 0);
404.               ptr_t2 = mmap(NULL, sizeof *ptr_t2, PROT_READ | PROT_WRITE, MAP_SHAR
    ED | MAP_ANONYMOUS, 0, 0);
405.               ptr_initNumReader = mmap(NULL, sizeof *ptr_initNumReader, PROT_READ
    | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
406.               ptr_initNumWriter = mmap(NULL, sizeof *ptr_initNumWriter, PROT_READ
    | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
407.               ptr_readCount = mmap(NULL, sizeof *ptr_readCount, PROT_READ | PROT_W
    RITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
408.               ptr_writeCount = mmap(NULL, sizeof *ptr_writeCount, PROT_READ | PROT
    _WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
409.               ptr_done = mmap(NULL, sizeof *ptr_done, PROT_READ | PROT_WRITE, MAP_
    SHARED | MAP_ANONYMOUS, 0, 0);
410.               ptr_finish = mmap(NULL, sizeof *ptr_finish, PROT_READ | PROT_WRITE,
    MAP_SHARED | MAP_ANONYMOUS, 0, 0);
411.               ptr_dataReadFilePosition = mmap(NULL, sizeof *ptr_dataReadFilePositi
    on, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
412.               ptr_dataReadFromReader = mmap(NULL, sizeof *ptr_dataReadFromReader,
    PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
413.               ptr_totalDataHasBeenRead = mmap(NULL, sizeof *ptr_totalDataHasBeenRe
    ad, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
414.               ptr_readerStopped = mmap(NULL, sizeof *ptr_readerStopped, PROT_READ
    | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
415.
416.               //Initialise shared memory data
417.               *ptr_t1 = t1;
418.               *ptr_t2 = t2;
419.               *ptr_initNumReader = initNumReader;
420.               *ptr_initNumWriter = initNumWriter;
421.               *ptr_readCount = 0;
422.               *ptr_writeCount = 0;
423.               *ptr_done = 0;
424.               *ptr_finish = 0;
425.               *ptr_dataReadFilePosition = 0;
426.               *ptr_dataReadFromReader = 0;
```

```
427.                *ptr_totalDataHasBeenRead = 0;
428.                *ptr_readerStopped = 0;
429.
430.                sem_unlink("/writer");
431.                sem_unlink("/reader");
432.                sem_unlink("/consume");
433.                sem_unlink("/condR");
434.                sem_unlink("/condW");
435.                sem_unlink("/output");
436.
437.                //Check for errors
438.                if ( ( condW_sem = sem_open("/condW", O_CREAT|O_EXCL, 0, 1) ) == SEM
    _FAILED )
439.                {
440.                    perror("sem open");
441.                    exit(EXIT_FAILURE);
442.                }
443.
444.                if ( ( condR_sem = sem_open("/condR", O_CREAT|O_EXCL, 0, 1) ) == SEM
    _FAILED )
445.                {
446.                    perror("sem open");
447.                    exit(EXIT_FAILURE);
448.                }
449.                if ( ( consume_sem = sem_open("/consume", O_CREAT|O_EXCL, 0, 1) ) ==
     SEM_FAILED )
450.                {
451.                    perror("sem open");
452.                    exit(EXIT_FAILURE);
453.                }
454.                if ( ( writer_sem = sem_open("/writer", O_CREAT|O_EXCL, 0, 1) ) == S
    EM_FAILED )
455.                {
456.                    perror("sem open");
457.                    exit(EXIT_FAILURE);
458.                }
459.
460.                if ( ( reader_sem = sem_open("/reader", O_CREAT|O_EXCL, 0, 1) ) == S
    EM_FAILED )
461.                {
462.                    perror("sem open");
463.                    exit(EXIT_FAILURE);
464.                }
465.
466.                if ( ( output_sem = sem_open("/output", O_CREAT|O_EXCL, 0, 1) ) == S
    EM_FAILED )
467.                {
468.                    perror("sem open");
469.                    exit(EXIT_FAILURE);
470.                }
471.
472.                //Transfer file data to shared memory
473.                for (i = 0; i < d; i++) {
474.                    ( ( int* )ptr_DataFromFile)[i] = dataFromFile[i];
475.                    ( (int* )ptr_DataBuffer)[i] = 0;
476.                }
477.
478.                //Create readers and writers
479.                createProcessors(initNumReader + initNumWriter);
480.
481.                //Free processors
482.                for (j = 0; j < (initNumReader + initNumWriter) * 2; j++) {
483.                    wait(NULL);
484.                }
485.
486.                //Clean up resources
```

```
487.                munmap( ptr_DataFromFile, sizeof( dataFromFile ) );
488.                munmap( ptr_DataBuffer, sizeof( dataBuffer ) );
489.                munmap( ptr_t1, sizeof( *ptr_t1 ) );
490.                munmap( ptr_t2, sizeof( *ptr_t2 ) );
491.                munmap( ptr_initNumReader, sizeof( *ptr_initNumReader ) );
492.                munmap( ptr_initNumWriter, sizeof( *ptr_initNumWriter ) );
493.                munmap( ptr_readCount, sizeof( *ptr_readCount ) );
494.                munmap( ptr_writeCount, sizeof( *ptr_writeCount ) );
495.                munmap( ptr_done, sizeof( *ptr_done ) );
496.                munmap( ptr_finish, sizeof( *ptr_finish ) );
497.                munmap( ptr_dataReadFilePosition, sizeof( *ptr_dataReadFilePosition
    ) );
498.                munmap( ptr_dataReadFromReader, sizeof( *ptr_dataReadFromReader ) );

499.                munmap( ptr_totalDataHasBeenRead, sizeof( *ptr_totalDataHasBeenRead
    ) );
500.                munmap( ptr_readerStopped, sizeof( *ptr_readerStopped ) );
501.
502.                close( shm_DataFromFile );
503.                close( shm_DataBuffer );
504.            }
505.
506.
507.            return 0;
508.
509.        }
```

```
1.  /**
2.   * FILE:    file.c
3.   * AUTHOR:  Xhien Yi Tan
4.   * STUDENTID:   18249833
5.   * UNIT:    COMP2006
6.   * PURPOSE: This file incharge of writing and reading data from file
7.   */
8.
9.
10. #include <stdio.h>
11. #include <pthread.h>
12. #include <stdlib.h>
13. #include <stdint.h>
14. #include <string.h>
15. #include<semaphore.h>
16. #include<ctype.h>
17. #include "file.h"
18.
19. #define READER 0
20. #define WRITER 1
21.
22. typedef struct{
23.     int counter, id;
24. } Process;
25.
26.
27.
28. /**
29. * PURPOSE : Clear sim_out file everytime we run this program,
30. *           so that the previous data is cleared before putting in new data.
31. * IMPORTS : Output file name
32. * EXPORTS : None
33. */
34. void *clearContentOutputFile( char *outputfilename ) {
```

```
35.
36.     FILE *f;
37.     f = fopen( outputfilename, "w" );
38.
39.     if( f == NULL )
40.     {
41.         perror( outputfilename );
42.     }
43.     fclose(f);
44.
45.     return NULL;
46.
47. }
48.
49. /**
50. * PURPOSE : Read file and store the data into array
51. * IMPORTS : Input file name, the total size for shared_data
52. * EXPORTS : Data from file
53. */
54. int* readFile( char *filename, int d )
55. {
56.     FILE *f;
57.     int check, done = 0,i = 0, num;
58.     int *dataArray;
59.
60.
61.     dataArray = ( int* )malloc( d * sizeof( int ) );
62.     f = fopen( filename, "r" );
63.
64.
65.     if( f == NULL )
66.     {
67.         perror( filename );
68.     } else {
69.
70.         do {
71.
72.             check = fscanf( f, "%d ", &num );
73.
74.
75.             if ( check == EOF ) {
76.                 done = 1;
77.             } else if ( i == d ) {
78.                 done = 1;
79.             } else {
80.                 dataArray[i] = num;
81.                 i++;
82.             }
83.
84.         } while( done != 1);
85.     }
86.
87.     fclose(f);
88.
89.     return dataArray;
90. }
91.
92.
93. /**
94. * PURPOSE : Write data to file
95. * IMPORTS : Output file name, thread that wants tp input data, type - READER or WRI
    TER
96. * EXPORTS : None
97. */
98. void *writeToFile(char *filename, void* process, int type) {
99.
```

```
100.                FILE *f;
101.                f = fopen( filename, "a" );
102.
103.                if( f == NULL ) {
104.                    perror( filename );
105.                } else {
106.
107.                    if ( type == READER ) {
108.                        fprintf(f, "reader-%d has finished reading %d pieces of data from the data_buffer. \n", ( ( ( Process* )process )->id ),( ( ( Process* )process )->counter ) );
109.                    } else {
110.                        fprintf(f, "writer-%d has finished writing %d pieces of data to the data_buffer. \n", ( ( ( Process* )process )->id ),( ( ( Process* )process )->counter ) );
111.                    }
112.
113.                }
114.
115.                fclose(f);
116.                return NULL;
117.
118.        }
```

Introduction

My program requires 4 input arguments – number of readers, number of writers, sleeping time for readers and sleeping time for writers. Input and output file names and the value of b and d are hard coded, changes made to any of this requires recompiling the program.

The steps for both thread and process are similar. First, I will read all the data from the shared_data into an array we called dataFromFIle instead of reading one after one by moving the file pointer. After that, the writers only need to retrieve data from that array and write to data_buffer where readers will read and consume. After that, create the specified number of readers and writers. The OS will randomly select either writer or reader to run, it completely depends on the OS itself.

If writer is selected first, then it will retrieve data from dataFromFIle and write to data_buffer at a single time. Once the buffer is full, all writers will be blocked and wait for readers to consume it. After the readers read and consume the data from the data_buffer, it will notify the writers to writer and at the same time, set the index of the data_buffer to 0 and write from the front again until all the data from dataFromFIle has been finished writing into data_buffer, then writers stop. Readers stop when they finish reading all the data from the dataFromFIle. While they stop, they output the results to sim_out which is the output file.

If reader is selected first, it will wait since the data_buffer is empty at that moment, no data can be retrieved. After that writer will do the same steps as I described from the top.

In additional to what I described, only one **writer** at a time can access to dataFromFIle and write to data_buffer and **readers** can read the data together but only one **reader** at a time can consume the data at a time to ensure that all readers consume the right data. Other than that, I created a Thread and Process typedef struct for each of the problems.

How to compile and run the program

In order to run the program, first compile using “make” then run using “./sds r w t1 t2” where r is the number of readers, w is the number of writers, t1 is the sleeping time for readers, and t2 is the sleeping time for writers.

Discuss how any mutual exclusion is achieved and what processes/threads access the shared resources

Threads

Mutual Exclusion

Mutual exclusion is achieved by using mutex lock and unlock as well as conditional wait and signal. I have used a total of 4 mutexes and 2 conditional variables.

Mutex:
read – Ensure one reader at a time can increments read count.

write – Ensure one writer can access its critical section for reading and writing to the data buffer at a time.
consume – Ensure one reader at a time can consume the data in the data buffer.
output – Ensure one thread at a time can write to the output file to avoid any conflicts.

Conditional Variables:
condR – Used by reader thread to wait writers to put data into buffer when buffer is empty.
condW – Used by writer thread to wait readers to consume when buffer is full.

Shared resources

Both reader and writer threads share a number of resources.

readCount – number of readers reading.
writeCount – number of writers reading and writing.
dataBuffer – an array of integers shared between them to read and write.
All the mutexes and conditional variables.
Integers for tracking number of readers stopped, number of data has been read by readers, number of data has been read by writers.

Is my program working or not?

I believe my program is working because it outputs the correct results to the output file. However, in some rare cases, it will result is deadlock and the chance of getting deadlock is extremely low.

Tests Used

I have provided 3 testcases in the testcases folder. Tests I did include:

- Large data sets for d = 100, d = 1000, d = 5000
- Set d < b, r < w, r > w, r = w, as well as various number of sleeping time

Sample Input and output

d = 100 (total number of integers in the shared_data)
b = 20 (total number of integers in data_buffer)
r = 5
w = 5
t1 and t2 = 1

Input - ./sds 5 5 1 1
Output File:

```
writer-3 has finished writing 15 pieces of data to the data_buffer.
reader-2 has finished reading 100 pieces of data from the data_buffer.
writer-2 has finished writing 40 pieces of data to the data_buffer.
reader-4 has finished reading 100 pieces of data from the data_buffer.
reader-1 has finished reading 100 pieces of data from the data_buffer.
reader-3 has finished reading 100 pieces of data from the data_buffer.
reader-0 has finished reading 100 pieces of data from the data_buffer.
writer-4 has finished writing 0 pieces of data to the data_buffer.
writer-0 has finished writing 37 pieces of data to the data_buffer.
writer-1 has finished writing 8 pieces of data to the data_buffer.
```

As you can see total number of data wrote by **writers** is the same as the number of d.
At the same time, all **readers** finished reading 100 pieces of data, which is the same as the number d.

Processes

Mutual Exclusion

Mutual exclusion is achieved by using semaphore wait and post. I have used a total of 7 semaphores.

Semaphores:
reader_sem – Ensure one reader at a time can increments read count.
write_sem – Ensure one writer can access its critical section for reading and writing to the data buffer at a time.
consume_sem – Ensure one reader at a time can consume the data in the data buffer.
output_sem – Ensure one process at a time can write to the output file to avoid any conflicts.
condR_sem – Used by reader process to wait writers to put data into buffer when buffer is empty.
condW_sem – Used by writer process to wait readers to consume when buffer is full.

Shared resources

Both reader and writer processes share a number of resources. In process, I need to explicitly create shared memory between processes.

readCount – number of readers reading.
writeCount – number of writers reading and writing.
dataBuffer – an array of integers shared between them to read and write.
All the semaphores.
Integers for tracking number of readers stopped, number of data has been read by readers, number of data has been read by writers.

Is my program working or not?

I believe my program is partially working because it sometimes outputs the correct results to the output file, but sometimes don't. One limitation I have come across is creating

number of processes. In my program, I created a constant for maximum number of processes can be created and the current value is 7, meaning that the maximum number of processes can be created is 128 (for loop from 0 to 7, and each call fork). In order to increase that, you need to change the value in the constant. Another limitation is sometime the writers over read the shared_data.

Tests Used

I have provided 3 testcases in the testcases folder. Tests I did include:

- Large data sets for d = 100, d = 1000, d = 5000
- Set d < b, r < w, r > w, r = w, as well as various number of sleeping time

Sample Input and output

d = 100 (total number of integers in the shared_data)
b = 20 (total number of integers in data_buffer)
r = 5
w = 5
t1 and t2 = 1

```
1   reader-3 has finished reading 100 pieces of data from the data_buffer.
2   reader-1 has finished reading 100 pieces of data from the data_buffer.
3   reader-2 has finished reading 100 pieces of data from the data_buffer.
4   reader-5 has finished reading 100 pieces of data from the data_buffer.
5   writer-1 has finished writing 7 pieces of data to the data_buffer.
6   writer-2 has finished writing 24 pieces of data to the data_buffer.
7   reader-4 has finished reading 100 pieces of data from the data_buffer.
8   writer-4 has finished writing 9 pieces of data to the data_buffer.
9   writer-5 has finished writing 51 pieces of data to the data_buffer.
10  writer-3 has finished writing 9 pieces of data to the data_buffer.
11
```