/\*\*

Name: OS Assignment (process)

Date: 07/05/18

Author: Rares Popa

Student ID: 19159700

Unit: Operating Systems (COMP2006)

This code contains my finished Operating Systems Assignment,

Lisence for use is MIT

\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <sys/shm.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/mman.h>

#include <semaphore.h>

#include <pthread.h>

#include "colour.h" //Adds Colour to the output

//Define important Values with defaults

#define BUFFER\_SIZE 20 //Buffer Size

#define FILENAME "shared\_data.txt" //Filename to read from

#define READERS 5 //Amount of readers

#define WRITERS 2 //Amount of writers

#define SLEEP\_TIME 1 //Sleep amount

//Mutexes

sem\_t mutex; //General Mutex

sem\_t rf\_mutex; //Mutex used in read\_file

sem\_t w\_mutex; //Mutex used in writer

sem\_t \*semaphoresPtr;

//Counters

int reader\_count = 0; //Define the amount of readers

int writer\_count = 0; //Define the amount of writers

//define file/writer/reader position

int file\_pos, writer\_pos, reader\_pos;

//define num from file read and if allRead

int num, reader\_amount = -1, writer\_amount = -1;

/\*\*

--3 types of arrays--

data\_buffer - which holds values from the shared\_data file (MAX 20 Values) (Shared memory)

hasRead - Checks to see if the reader has read the data\_buffer (MAX (READER) Values)

readerPieces - Increments each time a reader has read the data\_buffer

\*\*/

//Header funtions

void signal\_next(int \*reader\_count, sem\_t \*\*semaphores);

void writer(int pid, int (\*data\_buffer)[BUFFER\_SIZE], int \*allRead, int (\*hasRead)[READERS],int \*reader\_count,int \*writer\_count, int \*reader\_pos, int \*endOfFile, sem\_t \*semaphores, int \*file\_pos);

void reader(int pid, int (\*data\_buffer)[BUFFER\_SIZE], int \*allRead, int (\*hasRead)[READERS],int \*reader\_count,int \*writer\_count, int \*writer\_pos, int \*endOfRead, int (\*readerPieces)[READERS], sem\_t \*semaphores);

void read\_file(int pid, int (\*\*data\_buffer)[BUFFER\_SIZE], int \*\*allRead, int (\*\*hasRead)[READERS], int \*\*reader\_pos, int \*\*endOfRead, int (\*\*readerPieces)[READERS], sem\_t \*\*semaphores);

void write\_file(int pid, int (\*\*data\_buffer)[BUFFER\_SIZE], int \*\*allRead, int (\*\*hasRead)[READERS], int \*\*writer\_pos, int \*\*endOfFile, int \*\*file\_pos);

void reset\_array(int (\*\*\*hasRead)[READERS]);

int check\_read(int (\*hasRead)[READERS]);

void validateArg(int argc, char\* argv[]);

void create\_shared\_memory(int \*data\_bufferFD,int \*hasReadFD,int \*allReadFD,int \*reader\_countFD,int \*writer\_countFD,int \*reader\_posFD,int \*writer\_posFD, int \*endOfReadFD, int \*endOfFileFD, int \*readerPiecesFD, int \*semaphoresFD, int \*file\_posFD, int (\*\*data\_bufferPtr)[BUFFER\_SIZE],int \*\*allReadPtr,int (\*\*hasReadPtr)[READERS],int \*\*reader\_countPtr, int \*\*writer\_countPtr, int \*\*reader\_posPtr,int \*\*writer\_posPtr, int \*\*endOfReadPtr, int \*\*endOfFilePtr, int (\*\*readerPiecesPtr)[READERS], sem\_t \*\*semaphoresPtr, int \*\*file\_posPtr);

void delete\_shared\_memory(int data\_bufferFD,int hasReadFD,int allReadFD,int reader\_countFD,int writer\_countFD,int reader\_posFD,int writer\_posFD, int endOfReadFD, int endOfFileFD, int readerPiecesFD, int semaphoresFD, int file\_posFD, int (\*\*data\_bufferPtr)[BUFFER\_SIZE],int \*\*allReadPtr,int (\*\*hasReadPtr)[READERS],int \*\*reader\_countPtr, int \*\*writer\_countPtr, int \*\*reader\_posPtr,int \*\*writer\_posPtr, int \*\*endOfReadPtr, int \*\*endOfFilePtr, int (\*\*readerPiecesPtr)[READERS], sem\_t \*\*semaphoresPtr, int \*\*file\_posPtr);

int main(int argc, char\* argv[])

{

//File descriptor

int data\_bufferFD, hasReadFD, allReadFD, reader\_countFD, writer\_countFD, reader\_posFD, writer\_posFD, endOfReadFD, endOfFileFD, readerPiecesFD, semaphoresFD, file\_posFD;

//Shared memory pointers

int (\*data\_bufferPtr)[BUFFER\_SIZE], \*allReadPtr,(\*hasReadPtr)[READERS] = {0}, \*reader\_countPtr, \*writer\_countPtr, \*reader\_posPtr, \*writer\_posPtr, \*endOfReadPtr, \*endOfFilePtr, (\*readerPiecesPtr)[READERS] = {0}, \*file\_posPtr;

//Validates the command line arguements

// validateArg(argc, argv); - Working on it

//Creates the shared memory

create\_shared\_memory(&data\_bufferFD, &hasReadFD, &allReadFD, &reader\_countFD, &writer\_countFD, &reader\_posFD, &writer\_posFD, &endOfReadFD, &endOfFileFD, &readerPiecesFD, &semaphoresFD, &file\_posFD, &data\_bufferPtr, &allReadPtr, &hasReadPtr, &reader\_countPtr, &writer\_countPtr, &reader\_posPtr, &writer\_posPtr, &endOfReadPtr, &endOfFilePtr, &readerPiecesPtr, &semaphoresPtr, &file\_posPtr);

//Initiates all the semaphores

sem\_init(&mutex, 0, 1);

sem\_init(&rf\_mutex, 0, 1);

sem\_init(&w\_mutex, 0, 1);

//Initiates values

int ii, jj;

\*allReadPtr = 0;

\*reader\_countPtr = 0;

\*writer\_countPtr = 0;

\*endOfReadPtr = 0;

\*endOfFilePtr = 0;

\*file\_posPtr = 0;

semaphoresPtr[0] = mutex;

semaphoresPtr[1] = rf\_mutex;

semaphoresPtr[2] = w\_mutex;

//Initiates Writers

pid\_t writer\_pid = getpid();

for (ii = 0; ii < WRITERS; ii++)

{

if (writer\_pid > 0)

{

writer\_amount++;

printf("%sCreating Writer %d%s\n", GRN, ii, RESET);

writer\_pid = fork();

}

}

//Initiates Readers

pid\_t reader\_pid = getpid();

for (jj = 0; jj < READERS; jj++)

{

if (reader\_pid > 0)

{

reader\_amount++;

printf("%sCreating Reader %d%s\n", GRN, jj, RESET);

reader\_pid = fork();

}

}

//Checks to make sure its a child

if (writer\_pid == 0 && reader\_pid != 0)

{

writer(writer\_amount,data\_bufferPtr,allReadPtr,hasReadPtr,reader\_countPtr,writer\_countPtr,writer\_posPtr,endOfFilePtr,semaphoresPtr, file\_posPtr);

}

else if (reader\_pid == 0 && writer\_pid != 0)

{

reader(reader\_amount,data\_bufferPtr,allReadPtr,hasReadPtr,reader\_countPtr,writer\_countPtr,reader\_posPtr,endOfReadPtr,readerPiecesPtr,semaphoresPtr);

}

else

{

//Else it is terminated

for (ii = 0; ii < READERS+WRITERS; ii++)

{

wait(NULL);

}

}

//destroy semaphores

sem\_destroy(&mutex);

sem\_destroy(&rf\_mutex);

sem\_destroy(&w\_mutex);

//Removes shared memory

delete\_shared\_memory(data\_bufferFD, hasReadFD, allReadFD, reader\_countFD, writer\_countFD, reader\_posFD, writer\_posFD, endOfReadFD, endOfFileFD, readerPiecesFD, semaphoresFD, file\_posFD, &data\_bufferPtr, &allReadPtr, &hasReadPtr, &reader\_countPtr, &writer\_countPtr, &reader\_posPtr, &writer\_posPtr, &endOfReadPtr, &endOfFilePtr, &readerPiecesPtr, &semaphoresPtr, &file\_posPtr);

return 0;

}

/\*\*

\* Depending on the read count determines on what happens

\* > If there are > 0 readers then pthread will broadcast (Waking up all

\* sleeping readers)

\* > If there are 0 readers then the writers will be signalled and

\* woken up

\*

\* @void - No param inputs

\*\*/

void signal\_next(int \*reader\_count, sem\_t \*\*semaphores)

{

if (\*reader\_count > 0)

{

//Wake up next reader

signal(&(\*semaphores)[0]);

}

else

{

//Wake up next writer

signal(&(\*semaphores)[2]);

}

}

/\*\*

\* Handles the reading with mutexes, it outputs the info about the

\* reader and its current status. At the end it outputs how many

\* pieces the reader has read

\*

\* @pid - PID of the reader

\* @data\_buffer - Contains the shared memory location

\* @allRead - Checker to make sure all the readers have read (true/false)

\* @hasRead - Array to check if reader has read data

\* @reader\_count - amount of readers

\* @writer\_count - amount of writers

\* @reader\_pos - reader position in array

\* @endOfRead - check if its the end of reading

\* @readerPieces - amount the reader has read

\* @semaphores - array of the semaphores

\*\*/

void reader(int pid, int (\*data\_buffer)[BUFFER\_SIZE], int \*allRead, int (\*hasRead)[READERS], int \*reader\_count, int \*writer\_count, int \*reader\_pos, int \*endOfRead, int (\*readerPieces)[READERS], sem\_t \*semaphores)

{

//Until finished reading loop

while (\*endOfRead == 0)

{

wait(&semaphores[0]);

printf("Reader-%d is trying to enter into database\n", pid);

\*reader\_count++;

//if more than one reader and more than 0 writers make the reader wait

if (\*reader\_count > 1 || \*writer\_count > 0)

{

printf("Reader-%d is waiting until signalled\n", pid);

//Make reader wait

wait(&semaphores[0]);

}

signal(&semaphores[0]);

printf("Reader-%d is reading the database\n", pid);

//Read the data\_buffer

read\_file(pid,&data\_buffer,&allRead,&hasRead,&reader\_pos,&endOfRead, &readerPieces,&semaphores);

wait(&semaphores[0]);

reader\_count--;

printf("Reader-%d is leaving the database\n", pid);

signal\_next(reader\_count,&semaphores);

signal(&semaphores[0]);

//Sleep for amount of time

sleep(SLEEP\_TIME);

}

//Output reader result

printf("Reader-%d has finished reading %d pieces of data from the data\_buffer\n", pid, (\*readerPieces)[pid]);

}

/\*\*

\* Handles the reading of the file and writing to the shared\_data, it

\* outputs the info about the writer and its current status.

\*

\* @pid - PID of the reader

\* @data\_buffer - Contains the shared memory location

\* @allRead - Checker to make sure all the readers have read (true/false)

\* @hasRead - Array to check if reader has read data

\* @reader\_count - amount of readers

\* @writer\_count - amount of writers

\* @writer\_pos - writer position in array

\* @endOfFile - check if its the end of file

\* @file\_pos - where the writer is up to reading

\* @semaphores - array of the semaphores

\*\*/

void writer(int pid, int (\*data\_buffer)[BUFFER\_SIZE], int \*allRead, int (\*hasRead)[READERS], int \*reader\_count, int \*writer\_count, int \*writer\_pos, int \*endOfFile, sem\_t \*semaphores, int \*file\_pos)

{

//Until end of file loop

while (\*endOfFile == 0)

{

printf("Writer-%d is trying to enter into database\n", pid);

wait(&semaphores[2]);

//if more then 0 readers and 0 writers make the writer wait

while (\*reader\_count > 0 || \*writer\_count > 0)

{

//if more than 0 reader and more than 0 writers make the writer wait

printf("Writer-%d is waiting until signalled\n", pid);

wait(&semaphores[2]);

}

\*writer\_count++;

signal(&semaphores[2]);

printf("Writer-%d is writing into the database\n", pid);

write\_file(pid,&data\_buffer,&allRead,&hasRead,&writer\_pos,&endOfFile, &file\_pos);

wait(&(semaphores[2]));

printf("Writer-%d is leaving the database\n", pid);

\*writer\_count--;

signal\_next(reader\_count,&semaphores);

signal(&semaphores[2]);

//Sleep for amount of time

sleep(SLEEP\_TIME);

}

}

/\*\*

\* Reads the data\_buffer and depending on the 3 types of outputs

\* it will produce an outcome

\* > -2147483648 - means it is the end of the array

\* > data == 0 - means that the data\_buffer is empty

\* else we check to make sure all the readers have read the data\_buffer

\* if all have read we increment the reader\_pos and set the allRead

\* condition to 1 (true). It outputs the read status and the data the

\* reader has read.

\*

\* @pid - PID of the reader

\* @data\_buffer - Contains the shared memory location

\* @allRead - Checker to make sure all the readers have read (true/false)

\* @hasRead - Array to check if reader has read data

\* @reader\_pos - reader position in array

\* @endOfRead - check if its the end of reading

\* @readerPieces - amount the reader has read

\* @semaphores - array of the semaphores

\*\*/

void read\_file(int pid, int (\*\*data\_buffer)[BUFFER\_SIZE], int \*\*allRead, int (\*\*hasRead)[READERS], int \*\*reader\_pos, int \*\*endOfRead, int (\*\*readerPieces)[READERS], sem\_t \*\*semaphores)

{

wait(&(\*semaphores)[1]);

int data = (\*\*data\_buffer)[\*\*reader\_pos];

//End of the array

if (data == -2147483648)

{

\*\*endOfRead = 1;

printf("> No more data left to read!\n");

}

//Array is empty

else if (data == 0)

{

printf("%s> Waiting for more data to be input%s\n", BLU, RESET);

}

else //Valid data

{

//Check to make sure all the readers have read the data\_buffer

if (check\_read(\*hasRead) == 0)

{

//Check if given reader has not read the data\_buffer

if ((\*\*hasRead)[pid] != 1)

{

//Increment the amount of pieces which the reader has read

(\*\*readerPieces)[pid]++;

//Set the hasRead flag to 1 (True)

(\*\*hasRead)[pid] = 1;

//Output status

printf("- %sReader-%d%s | > %d\n", MAG, pid, RESET, data);

}

//If all the readers have read the data\_buffer then

if (check\_read(\*hasRead) != 0)

{

//Set all read true (0)

\*\*allRead = \*\*allRead - 1;

printf("= All data read!\n");

//If reader\_pos is not 20 increment else reset the reader\_pos

if (\*\*reader\_pos != 20)

{

\*\*reader\_pos++;

}

else

{

\*\*reader\_pos = 0;

}

//Reset the values in the array

reset\_array(&hasRead);

}

}

}

signal(&(\*semaphores)[1]);

}

/\*\*

\* Reads the file line by line and writes it to the data\_buffer,

\* once the data\_buffer has been filled to 20 it is reset back to 0.

\* Once all the values in the file have been read it inputs the value

\* -2147483648 at the end of the array to show it is done. Outputs

\* the status of the writer.

\*

\* @pid - PID of the reader

\* @data\_buffer - Contains the shared memory location

\* @allRead - Checker to make sure all the readers have read (true/false)

\* @hasRead - Array to check if reader has read data

\* @reader\_pos - reader position in array

\* @endOfRead - check if its the end of reading

\* @file\_pos - where the writer is up to reading

\* @semaphores - array of the semaphores

\*\*/

void write\_file(int pid, int (\*\*data\_buffer)[BUFFER\_SIZE], int \*\*allRead, int (\*\*hasRead)[READERS], int \*\*writer\_pos, int \*\*endOfFile, int \*\*file\_pos)

{

FILE \*f;

int i;

//Open file for reading only

f = fopen(FILENAME,"r");

if (f == NULL){

printf("Error! opening file");

}

//Get the file position from last time

fseek(f, \*\*file\_pos, 0);

if (fscanf(f,"%d", &num) != EOF)

{

//Check if all the readers have read the data\_buffer

if (\*\*allRead == 0)

{

printf("+ %sWriter%s | > %d\n", CYN, RESET, num);

(\*\*data\_buffer)[\*\*writer\_pos] = num;

//If writer\_pos is not 20 increment else reset the writer\_pos

if (\*\*writer\_pos != 20)

{

\*\*writer\_pos++;

}

else

{

\*\*writer\_pos = 0;

}

//Set the file\_pos for next time

\*\*file\_pos = ftell(f);

//Set the allRead to false (1) as no readers would have read it

\*\*allRead = \*\*allRead + 1;

}

else //Not all readers have read the buffer

{

printf("= Must wait for all readers to read the data\n");

}

}

else

{

//End of file reached

\*\*endOfFile = 1;

//Min possible int (Because low chance of ever being used)

(\*\*data\_buffer)[\*\*writer\_pos] = -2147483648;

}

fclose(f);

}

/\*\*

\* Resets the array to where all the values are 0

\*

\* @hasRead - Array of what readers have read (true/false)

\*\*/

void reset\_array(int (\*\*\*hasRead)[READERS])

{

for (int ii = 0; ii < READERS; ii++)

{

(\*\*\*hasRead)[ii] = 0;

}

}

/\*\*

\* Checks to make sure all the readers have a value in the array

\* location, if it is not == 1 then it returns 0 (false)

\*

\* @hasRead - Array of what readers have read (true/false)

\*\*/

int check\_read(int (\*hasRead)[READERS])

{

int jj;

for (jj = 0; jj < READERS; jj++)

{

if ((\*hasRead)[jj] != 1)

{

return 0;

}

}

}

/\*\*

\* Validate the command line parameters to make sure the user

\* gives the correct input

\*

\* @param - argc number of parameters

\* @param - argv command line parameters

\*/

void validateArg(int argc, char\* argv[])

{

//Make sure correct number of command line parameters

if (argc != 4)

{

printf("Incorrect use: /process READERS WRITERS SLEEP\_TIME\n");

exit(1);

}

//Make sure READERS is valid

if (atoi(argv[1]) < 1)

{

printf("All data needs to be read! More than 0 readers is needed\n");

exit(1);

}

//Make sure WRITERS is valid

if (atoi(argv[2]) < 1)

{

printf("Data doesn't make it in itself! More than 0 writers needed!\n");

exit(1);

}

//Make sure SLEEP\_TIME is valid

if (atoi(argv[3]) < 1)

{

printf("We all need rest! Sleep time must be above 0\n");

exit(1);

}

}

/\*\*

\* Initiates and creates the shared memory location

\*

\* @pidFD - PID of the reader (File descriptor)

\* @data\_bufferFD - Contains the shared memory location (File descriptor)

\* @allReadFD - Checker to make sure all the readers have read (true/false) (File descriptor)

\* @hasReadFD - Array to check if reader has read data (File descriptor)

\* @reader\_countFD - amount of readers (File descriptor)

\* @writer\_countFD - amount of writers (File descriptor)

\* @writer\_posFD - writer position in array (File descriptor)

\* @endOfFileFD - check if its the end of file (File descriptor)

\* @file\_posFD - where the writer is up to reading (File descriptor)

\* @semaphoresFD - array of the semaphores (File descriptor)

\* @pidPtr - A pointer to the PID of the reader (Pointer to memory)

\* @data\_bufferPtr - Contains the shared memory location (Pointer to memory)

\* @allReadPtr - Checker to make sure all the readers have read (true/false) (Pointer to memory)

\* @hasReadPtr - Array to check if reader has read data (Pointer to memory)

\* @reader\_countPtr - amount of readers (Pointer to memory)

\* @writer\_countPtr - amount of writers (Pointer to memory)

\* @writer\_posPtr - writer position in array (Pointer to memory)

\* @endOfFilePtr - check if its the end of file (Pointer to memory)

\* @file\_posPtr - where the writer is up to reading (Pointer to memory)

\* @semaphoresPtr - array of the semaphores (Pointer to memory)

\*/

void create\_shared\_memory(int \*data\_bufferFD,int \*hasReadFD,int \*allReadFD,int \*reader\_countFD,int \*writer\_countFD,int \*reader\_posFD,int \*writer\_posFD, int \*endOfReadFD, int \*endOfFileFD, int \*readerPiecesFD, int \*semaphoresFD, int \*file\_posFD, int (\*\*data\_bufferPtr)[BUFFER\_SIZE],int \*\*allReadPtr,int (\*\*hasReadPtr)[READERS],int \*\*reader\_countPtr, int \*\*writer\_countPtr, int \*\*reader\_posPtr,int \*\*writer\_posPtr, int \*\*endOfReadPtr, int \*\*endOfFilePtr, int (\*\*readerPiecesPtr)[READERS], sem\_t \*\*semaphoresPtr, int \*\*file\_posPtr)

{

//Create shared memory

\*data\_bufferFD = shm\_open("data\_buffer", O\_CREAT | O\_RDWR, 0666);

\*hasReadFD = shm\_open("hasRead", O\_CREAT | O\_RDWR, 0666);

\*allReadFD = shm\_open("allRead", O\_CREAT | O\_RDWR, 0666);

\*reader\_countFD = shm\_open("reader\_count", O\_CREAT | O\_RDWR, 0666);

\*writer\_countFD = shm\_open("writer\_count", O\_CREAT | O\_RDWR, 0666);

\*reader\_posFD = shm\_open("reader\_pos", O\_CREAT | O\_RDWR, 0666);

\*writer\_posFD = shm\_open("writer\_pos", O\_CREAT | O\_RDWR, 0666);

\*endOfReadFD = shm\_open("endOfRead", O\_CREAT | O\_RDWR, 0666);

\*endOfFileFD = shm\_open("endOfFile", O\_CREAT | O\_RDWR, 0666);

\*readerPiecesFD = shm\_open("readerPieces", O\_CREAT | O\_RDWR, 0666);

\*semaphoresFD = shm\_open("semaphores", O\_CREAT | O\_RDWR, 0666);

\*file\_posFD = shm\_open("file\_pos", O\_CREAT | O\_RDWR, 0666);

//Check to make sure there are no errors

if (\*data\_bufferFD == -1)

{

fprintf(stderr, "Error creating shared memory blocks\n");

exit(1);

}

//Set the size of the shared memory

if (ftruncate(\*data\_bufferFD, sizeof(int)\*BUFFER\_SIZE))

{

fprintf(stderr, "Error setting size of data\_buffer\n");

exit(1);

}

ftruncate(\*hasReadFD, sizeof(int)\*READERS);

ftruncate(\*allReadFD, sizeof(int));

ftruncate(\*reader\_countFD, sizeof(int));

ftruncate(\*writer\_countFD, sizeof(int));

ftruncate(\*reader\_posFD, sizeof(int));

ftruncate(\*writer\_posFD, sizeof(int));

ftruncate(\*endOfReadFD, sizeof(int));

ftruncate(\*endOfFileFD, sizeof(int));

ftruncate(\*readerPiecesFD, sizeof(int)\*READERS);

ftruncate(\*semaphoresFD, sizeof(sem\_t)\*3);

ftruncate(\*file\_posFD, sizeof(int));

//Map shared memory to addresses

\*data\_bufferPtr = mmap(NULL, sizeof(int)\*BUFFER\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*data\_bufferFD, 0);

\*hasReadPtr = mmap(0, sizeof(int)\*READERS, PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*hasReadFD, 0);

\*allReadPtr = mmap(0, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*allReadFD, 0);

\*reader\_countPtr = mmap(0, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*reader\_countFD, 0);

\*writer\_countPtr = mmap(0, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*writer\_countFD, 0);

\*reader\_posPtr = mmap(0, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*reader\_posFD, 0);

\*writer\_posPtr = mmap(0, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*writer\_posFD, 0);

\*endOfReadPtr = mmap(0, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*endOfReadFD, 0);

\*endOfFilePtr = mmap(0, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*endOfFileFD, 0);

\*readerPiecesPtr = mmap(0, sizeof(int)\*READERS, PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*readerPiecesFD, 0);

\*semaphoresPtr = mmap(0, sizeof(sem\_t)\*3, PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*semaphoresFD, 0);

\*file\_posPtr = mmap(0, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, \*file\_posFD, 0);

}

/\*\*

\* Deletes and cleans up the shared memory location

\*

\* @pidFD - PID of the reader (File descriptor)

\* @data\_bufferFD - Contains the shared memory location (File descriptor)

\* @allReadFD - Checker to make sure all the readers have read (true/false) (File descriptor)

\* @hasReadFD - Array to check if reader has read data (File descriptor)

\* @reader\_countFD - amount of readers (File descriptor)

\* @writer\_countFD - amount of writers (File descriptor)

\* @writer\_posFD - writer position in array (File descriptor)

\* @endOfFileFD - check if its the end of file (File descriptor)

\* @file\_posFD - where the writer is up to reading (File descriptor)

\* @semaphoresFD - array of the semaphores (File descriptor)

\* @pidPtr - A pointer to the PID of the reader (Pointer to memory)

\* @data\_bufferPtr - Contains the shared memory location (Pointer to memory)

\* @allReadPtr - Checker to make sure all the readers have read (true/false) (Pointer to memory)

\* @hasReadPtr - Array to check if reader has read data (Pointer to memory)

\* @reader\_countPtr - amount of readers (Pointer to memory)

\* @writer\_countPtr - amount of writers (Pointer to memory)

\* @writer\_posPtr - writer position in array (Pointer to memory)

\* @endOfFilePtr - check if its the end of file (Pointer to memory)

\* @file\_posPtr - where the writer is up to reading (Pointer to memory)

\* @semaphoresPtr - array of the semaphores (Pointer to memory)

\*/

void delete\_shared\_memory(int data\_bufferFD,int hasReadFD,int allReadFD,int reader\_countFD,int writer\_countFD,int reader\_posFD,int writer\_posFD, int endOfReadFD, int endOfFileFD, int readerPiecesFD, int semaphoresFD, int file\_posFD, int (\*\*data\_bufferPtr)[BUFFER\_SIZE],int \*\*allReadPtr,int (\*\*hasReadPtr)[READERS],int \*\*reader\_countPtr, int \*\*writer\_countPtr, int \*\*reader\_posPtr,int \*\*writer\_posPtr, int \*\*endOfReadPtr, int \*\*endOfFilePtr, int (\*\*readerPiecesPtr)[READERS], sem\_t \*\*semaphoresPtr, int \*\*file\_posPtr);

{

//fclose

close(data\_bufferFD);

close(hasReadFD);

close(reader\_countFD);

close(writer\_countFD);

close(reader\_posFD);

close(writer\_posFD);

close(endOfReadFD);

close(endOfFileFD);

close(readerPiecesFD);

close(semaphoresFD);

close(file\_posFD);

//Unmap

munmap(data\_bufferPtr, sizeof(int)\*BUFFER\_SIZE);

munmap(hasReadPtr, sizeof(int)\*READERS);

munmap(allReadPtr, sizeof(int));

munmap(reader\_countPtr, sizeof(int));

munmap(writer\_countPtr, sizeof(int));

munmap(reader\_posPtr, sizeof(int));

munmap(writer\_posPtr, sizeof(int));

munmap(endOfReadPtr, sizeof(int));

munmap(endOfFilePtr, sizeof(int));

munmap(readerPiecesPtr, sizeof(int)\*READERS);

munmap(semaphoresPtr, sizeof(sem\_t)\*3);

munmap(file\_posPtr, sizeof(int));

//Unlink

shm\_unlink("data\_buffer");

shm\_unlink("hasRead");

shm\_unlink("allRead");

shm\_unlink("reader\_count");

shm\_unlink("writer\_count");

shm\_unlink("reader\_pos");

shm\_unlink("writer\_pos");

shm\_unlink("endOfRead");

shm\_unlink("endOfWrite");

shm\_unlink("readerPieces");

shm\_unlink("semaphores");

shm\_unlink("file\_pos");

}