/\*\*

Name: OS Assignment (pthreads)

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 <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

pthread\_mutex\_t mutex; //General Mutex

pthread\_mutex\_t r\_mutex; //Mutex used in read\_file

//Counters

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

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

//Define all the conditions

pthread\_cond\_t condw, condr;

//define file/writer/reader position

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

//define end of the file/read

int endOfFile, endOfRead;

//define num from file read and if allRead

int num, allRead = 1;

/\*\*

--3 types of arrays--

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

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

\*\*/

int data\_buffer[BUFFER\_SIZE], hasRead[READERS] = {0}, readerPieces[READERS] = {0};

//Header funtions

void signal\_next();

void \*writer(void \*ptr);

void \*reader(void \*ptr);

void read\_file(int reader);

void write\_file();

void reset\_array();

int check\_read();

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

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

{

validateArg(argc, argv);

int ii, jj;

pthread\_t read[READERS];

pthread\_t write[WRITERS];

pthread\_mutex\_init(&mutex, 0);

pthread\_mutex\_init(&r\_mutex, 0);

pthread\_cond\_init(&condw, 0); //init writer

pthread\_cond\_init(&condr,0); //init reader

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

//Create pthreads

pthread\_create(&write[ii], NULL, writer, (void \*) ii);

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

}

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

//Create pthreads

pthread\_create(&read[jj], NULL, reader, (void \*) jj);

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

}

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

//Join pthreads

pthread\_join(write[ii], NULL);

printf("%sWriter %d finished%s\n", RED, ii, RESET);

}

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

//Join pthreads

pthread\_join(read[jj], NULL);

printf("%sReader %d finished%s\n", RED, jj, RESET);

}

//destroy mutexes/conditions

pthread\_cond\_destroy(&condw);

pthread\_cond\_destroy(&condr);

pthread\_mutex\_destroy(&r\_mutex);

pthread\_mutex\_destroy(&mutex);

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()

{

if (reader\_count > 0)

{

//Wake up all the readers

pthread\_cond\_broadcast(&condr);

}

else

{

//Wake up next writer

pthread\_cond\_signal(&condw);

}

}

/\*\*

\* 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

\*

\* @ptr - A pointer to the PID of the reader

\*\*/

void \*reader(void \*ptr)

{

int pid = (int) ptr;

while (endOfRead == 0)

{

pthread\_mutex\_lock(&mutex);

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

pthread\_cond\_wait(&condr, &mutex);

}

pthread\_mutex\_unlock(&mutex);

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

//Read the data\_buffer

read\_file(pid);

pthread\_mutex\_lock(&mutex);

reader\_count--;

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

signal\_next();

pthread\_mutex\_unlock(&mutex);

//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.

\*

\* @ptr - A pointer to the PID of the reader

\*\*/

void \*writer(void \*ptr)

{

int pid = (int) ptr;

while (endOfFile == 0)

{

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

pthread\_mutex\_lock(&mutex);

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

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

{

//cond\_wait unlocks the mutex, waits to be signaled,

//then re-acquires the mutex

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

pthread\_cond\_wait(&condw, &mutex);

}

writer\_count++;

pthread\_mutex\_unlock(&mutex);

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

write\_file();

pthread\_mutex\_lock(&mutex);

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

writer\_count--;

signal\_next();

pthread\_mutex\_unlock(&mutex);

//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

\*\*/

void read\_file(int pid)

{

pthread\_mutex\_lock(&r\_mutex);

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() == 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 (check\_read() != 0)

{

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();

}

}

}

pthread\_mutex\_unlock(&r\_mutex);

}

/\*\*

\* 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.

\*

\* @void - No param inputs

\*\*/

void write\_file()

{

FILE \*f;

int i;

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 == 1)

{

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 0 as no readers would have read it

allRead = 0;

}

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

\*

\* @void - No param inputs

\*\*/

void reset\_array()

{

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)

\*

\* @void - No param inputs

\*\*/

int check\_read()

{

for (int 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: /pthread 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);

}

}