main.c

#define \_CRT\_SECURE\_NO\_WARNINGS /\* to suppress Visual Studio 2010 compiler warning \*/

#include "Hw1TextFileReader.h"

#include "TestChecker.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

#include <ctype.h>

#include <math.h>

#include <windows.h>

#include <tchar.h>

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

{

TestLine \*test\_line\_array;

int num\_of\_threads = 0;

int \*num\_of\_threads\_ptr = &num\_of\_threads;

TextFileReader textFileReader;

if ( argc == 1 )

{

printf("ERROR - Not enough arguments\n");

exit(1);

}

textFileReader = CreateTextFileReader(argv[2]);

if (textFileReader.IsInitialized == FALSE)

{

printf("Failed to create the Text File Reader\n");

exit(1);

}

if (textFileReader.NumOfLines == 0)

exit(1);

test\_line\_array = (TestLine\*) malloc(textFileReader.NumOfLines \* sizeof(\*test\_line\_array));

if (test\_line\_array == NULL)

exit(1);

if (!TextFileReaderToTestLineArray(textFileReader, test\_line\_array,argv[1],num\_of\_threads\_ptr))

exit(1);

if (RunThreads(\*num\_of\_threads\_ptr, test\_line\_array))

exit(1);

if (PrintResultToFile(argv[3], argv[4], \*num\_of\_threads\_ptr, test\_line\_array, \*num\_of\_threads\_ptr))

exit(1);

free(test\_line\_array);

return 0;

}

TestCheker.h

#ifndef TEST\_CHECKER\_H

#define TEST\_CHECKER\_H

#include "Hw1TextFileReader.h"

#include <math.h>

#define MAX\_CHARS\_IN\_STRING 100

#define EXPECTED\_NAME 50

#define CMD\_LINE 100

typedef struct TestLine {

char\* CmdInput;

char\* OutputFileName;

char\* ExpectedFileName;

char\* ResultMsg;

char\* TimeMsg;

int ThreadInd;

} TestLine;

int TextFileReaderToTestLineArray(TextFileReader text\_file\_reader, TestLine\* test\_line\_array, const char \*exe\_file\_name,int \*num\_of\_threads);

int RunThreads(int num\_of\_threads, TestLine \*test\_line\_array);

int PrintResultToFile(char \*output\_file\_name1, char \*output\_file\_name2, int num\_of\_threads\_ptr, TestLine \*test\_line\_array, int num\_of\_threads);

#endif

TestCheker.c

#define \_CRT\_SECURE\_NO\_WARNINGS /\* to suppress Visual Studio 2010 compiler warning \*/

#include "TestChecker.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

#include <ctype.h>

#include <math.h>

#include <windows.h>

#include <tchar.h>

#define TIMEOUT\_IN\_MILLISECONDS 5000

#define BRUTAL\_TERMINATION\_CODE 0x55

FILETIME SubtractFiletimes(FILETIME Late, FILETIME Early);

char\* ConcatComandLine( char\* a, char\* b, ConstChar\* c);

int CreateTestLine(TestLine\* testLine)

{

testLine = (TestLine\*) malloc(sizeof(\*testLine));

if (testLine == NULL)

exit(1);

testLine->ThreadInd=0;

testLine->ResultMsg= "test #";

return 1;

}

int SaveTestLineProperty(TestLine\* test\_line, ConstLine file\_line, int words\_in\_line, const char \*exe\_file\_name)

{

int word\_ind = 0;

char\* command\_line = (char\*) malloc(strlen(exe\_file\_name) \* sizeof(char));

if (command\_line == NULL)

exit(1);

strcpy (command\_line,exe\_file\_name);

for (word\_ind = 0; word\_ind <words\_in\_line; word\_ind++)

{

command\_line = ConcatComandLine(command\_line," ",file\_line[word\_ind]);

if (word\_ind == words\_in\_line -1)

{

test\_line->OutputFileName = (char\*) malloc (sizeof(char) \* strlen(file\_line[word\_ind]));

if (test\_line->OutputFileName == NULL)

exit(1);

strcpy(test\_line->OutputFileName,file\_line[word\_ind]);

test\_line->CmdInput = (char\*) malloc (sizeof(char) \* strlen(command\_line));

if (test\_line->CmdInput == NULL)

exit(1);

strcpy(test\_line->CmdInput,command\_line);

}

}

//free (command\_line);

return 1;

}

int TextFileReaderToTestLineArray(TextFileReader text\_file\_reader, TestLine\* test\_line\_array, const char \*exe\_file\_name ,int \*num\_of\_threads)

{

int word\_size = 0;

int file\_line\_ind;

int test\_line\_ind = 0;

for (file\_line\_ind = 0; file\_line\_ind < text\_file\_reader.NumOfLines; file\_line\_ind++)

{

if (text\_file\_reader.WordsInLine[file\_line\_ind] > 0)

{

if (text\_file\_reader.WordsInLine[file\_line\_ind] > 1)

{

if (!CreateTestLine(&test\_line\_array[test\_line\_ind]))

return 0;

if (!SaveTestLineProperty(&test\_line\_array[test\_line\_ind], text\_file\_reader.WordsArr[file\_line\_ind], text\_file\_reader.WordsInLine[file\_line\_ind],exe\_file\_name))

{

return 0;

}

}

else if (strlen(text\_file\_reader.WordsArr[file\_line\_ind][0]) > 2)

{

test\_line\_array[test\_line\_ind].ExpectedFileName = (char\*) malloc (sizeof(char) \* strlen(text\_file\_reader.WordsArr[file\_line\_ind][0]));

if (test\_line\_array[test\_line\_ind].ExpectedFileName == NULL)

exit(1);

strcpy(test\_line\_array[test\_line\_ind].ExpectedFileName,text\_file\_reader.WordsArr[file\_line\_ind][0]);

test\_line\_array[test\_line\_ind].ThreadInd = test\_line\_ind;

test\_line\_ind++;

}

}

}

\*num\_of\_threads = test\_line\_ind;

return 1;

}

HANDLE CreateThreadSimple(LPTHREAD\_START\_ROUTINE StartAddress,

LPVOID ParameterPtr,

LPDWORD ThreadIdPtr)

{

return CreateThread(

NULL, /\* default security attributes \*/

0, /\* use default stack size \*/

StartAddress, /\* thread function \*/

ParameterPtr, /\* argument to thread function \*/

0, /\* use default creation flags \*/

ThreadIdPtr); /\* returns the thread identifier \*/

}

BOOL CreateNewProcess(LPTSTR CommandLine, PROCESS\_INFORMATION \*ProcessInfoPtr)

{

STARTUPINFO startinfo = { sizeof(STARTUPINFO), NULL, 0 };

return CreateProcess(

NULL, /\* module name (use command line). \*/

CommandLine, /\* Command line. \*/

NULL, /\* Process handle not inheritable. \*/

NULL, /\* Thread handle not inheritable. \*/

FALSE, /\* Set handle inheritance to FALSE. \*/

NORMAL\_PRIORITY\_CLASS, /\* creation/priority flags. \*/

NULL, /\* Use parent's environment block. \*/

NULL, /\* Use parent's starting directory. \*/

&startinfo, /\* Pointer to STARTUPINFO structure. \*/

ProcessInfoPtr /\* Pointer to PROCESS\_INFORMATION structure. \*/

);

}

int CompareCharsInFile(TestLine\* test\_line)

{

char ch1;

char ch2;

int since\_mismatch\_counter = 0;

FILE\* f\_expected = fopen(test\_line->ExpectedFileName, "r");

FILE\* f\_output = fopen(test\_line->OutputFileName, "r");

if (f\_output == NULL) {

printf("Cannot open %s for reading ", test\_line->OutputFileName);

exit(1);

} else if (f\_expected == NULL) {

printf("Cannot open %s for reading ", test\_line->ExpectedFileName);

exit(1);

} else {

ch1 = getc(f\_output);

ch2 = getc(f\_expected);

while ((ch1 != EOF) && (ch2 != EOF)) {

if (ch1 != ch2)

{

since\_mismatch\_counter++;

ch1 = getc(f\_output);

ch2 = getc(f\_expected);

}

else

{

ch1 = getc(f\_output);

ch2 = getc(f\_expected);

if (since\_mismatch\_counter != 0)

{

since\_mismatch\_counter++;

}

}

}

}

fclose(f\_output);

fclose(f\_expected);

if (since\_mismatch\_counter == 0 || since\_mismatch\_counter < 10)

{

// printf("Files are identical ");

return 0;

}

else

{

// printf("Files are Not identical ");

return 1;

}

}

void CompareResultMessage(DWORD exitcode,TestLine\* test\_line)

{

test\_line->ResultMsg = (char\*) malloc (sizeof(char) \* 32);

if (test\_line->ResultMsg == NULL)

exit(1);

if (exitcode != 0)

{

sprintf(test\_line->ResultMsg, "test #%d : Crashed #%d\n", test\_line->ThreadInd+1,exitcode);

}

else

{

if (!CompareCharsInFile(test\_line))

sprintf(test\_line->ResultMsg, "test #%d : Secceeded\n", test\_line->ThreadInd+1);

else

sprintf(test\_line->ResultMsg, "test #%d : Failed\n", test\_line->ThreadInd+1);

}

}

int RunSnoopy (TestLine\* test\_line)

{

PROCESS\_INFORMATION procinfo;

DWORD waitcode;

DWORD exitcode;

BOOL retVal;

SYSTEMTIME UTC\_time, UTC\_total\_time,local\_start\_time, local\_user\_time, local\_kernel\_time;

FILETIME start\_time, exit\_time, total\_time, user\_time, kernel\_time;

test\_line->TimeMsg = (char\*) malloc (sizeof(char) \* 500);

if (test\_line->TimeMsg == NULL)

exit(1);

/\* Start the child process. \*/

retVal = CreateNewProcess(test\_line->CmdInput, &procinfo);

if (retVal == 0)

{

printf("Process Creation Failed!\n");

return 1;

}

waitcode = WaitForSingleObject(

procinfo.hProcess,

INFINITE);

if (waitcode != WAIT\_OBJECT\_0)

{

printf("Unexpected output value of 0x%x from WaitForSingleObject(). "

"Ending function.\n", waitcode);

CloseHandle(procinfo.hProcess);

return 1;

}

GetExitCodeProcess(procinfo.hProcess, &exitcode);

CompareResultMessage(exitcode,test\_line);

GetProcessTimes(

procinfo.hProcess,

&start\_time,

&exit\_time,

&kernel\_time,

&user\_time

);

FileTimeToSystemTime(&start\_time,&UTC\_time);

SystemTimeToTzSpecificLocalTime(NULL,&UTC\_time, &local\_start\_time);

total\_time = SubtractFiletimes(exit\_time, start\_time);

FileTimeToSystemTime(&total\_time,&UTC\_total\_time);

FileTimeToSystemTime(&user\_time,&UTC\_time);

SystemTimeToTzSpecificLocalTime(NULL,&UTC\_time, &local\_user\_time);

FileTimeToSystemTime(&kernel\_time,&UTC\_time);

SystemTimeToTzSpecificLocalTime(NULL,&UTC\_time, &local\_kernel\_time);

sprintf(test\_line->TimeMsg, "test #%d : start\_time=%02d:%02d:%02d:%d total\_time=%d user\_time=%d kernel\_time=%d\n",

test\_line->ThreadInd,local\_start\_time.wHour, local\_start\_time.wMinute, local\_start\_time.wSecond, local\_start\_time.wMilliseconds,

((UTC\_total\_time.wMinute\*60000)+ (UTC\_total\_time.wSecond\*1000) + UTC\_total\_time.wMilliseconds),

((local\_user\_time.wMinute\*60000)+ (local\_user\_time.wSecond\*1000) + local\_user\_time.wMilliseconds),

((local\_kernel\_time.wMinute\*60000)+ (local\_kernel\_time.wSecond\*1000) + local\_kernel\_time.wMilliseconds));

CloseHandle(procinfo.hProcess); /\* Closing the handle to the process \*/

CloseHandle(procinfo.hThread); /\* Closing the handle to the main thread of the process \*/

return 0;

}

int RunThreads(int num\_of\_threads, TestLine \*test\_line\_array)

{

DWORD\* ThreadIDs; /\* An array of threadIDs \*/

HANDLE\* ThreadHandles; /\* An array of thread handles \*/

DWORD exitcode;

int i;

ThreadIDs =(DWORD\*) malloc(num\_of\_threads\*sizeof(\*ThreadIDs));

if (ThreadIDs == NULL){

exit(1);

}

ThreadHandles =(HANDLE\*) malloc(num\_of\_threads\*sizeof(\*ThreadHandles));

if (ThreadHandles == NULL){

exit(1);

}

for (i=0; i< num\_of\_threads; i++)

{

ThreadHandles[i] = CreateThreadSimple(

(LPTHREAD\_START\_ROUTINE)RunSnoopy, /\* thread function \*/

&test\_line\_array[i], /\* argument to thread function \*/

&ThreadIDs[i]); /\* returns the thread identifier \*/

}

/\* Wait for all threads to end: \*/

WaitForMultipleObjects(

num\_of\_threads,

ThreadHandles,

TRUE, /\* wait until all threads finish \*/

INFINITE);

Sleep(10);

for (i = 0; i < num\_of\_threads; i++)

{

GetExitCodeThread(ThreadHandles[i], &exitcode);

if (exitcode != 0)

{

exit(1);

}

CloseHandle(ThreadHandles[i]);

}

free (ThreadIDs);

free (ThreadHandles);

return 0;

}

char\* ConcatComandLine( char\* a, char\* b, ConstChar\* c)

{

int total\_size = strlen(a)+strlen(b)+strlen(c);

char\* string = (char\*) malloc(total\_size\* sizeof(char));

if (string == NULL)

{

exit(1);

}

strcpy(string,a);

strcat(string,b);

strcat(string,c);

return string;

}

int PrintResultToFile(char \*output\_file\_name1, char \*output\_file\_name2,int num\_of\_threads\_ptr, TestLine \*test\_line\_array, int num\_of\_threads)

{

int test\_line\_ind = 0;

FILE\* correctness\_file = fopen(output\_file\_name1,"w+");

FILE\* runtime\_file = fopen(output\_file\_name2,"w+");

if(NULL == correctness\_file || NULL == runtime\_file )

{

printf("\n fopen() Error!!!\n");

return 1;

}

for (test\_line\_ind = 0; test\_line\_ind < num\_of\_threads; test\_line\_ind++)

{

fprintf(correctness\_file,"%s",test\_line\_array[test\_line\_ind].ResultMsg);

fprintf(runtime\_file,"%s", test\_line\_array[test\_line\_ind].TimeMsg);

}

fclose(correctness\_file);

fclose(runtime\_file);

return 0;

}

FILETIME SubtractFiletimes(FILETIME Late, FILETIME Early)

{

typedef unsigned \_\_int64 Unsigned64BitType;

Unsigned64BitType Late64BitVal;

Unsigned64BitType Early64BitVal;

Unsigned64BitType Difference64BitVal;

FILETIME DifferenceAsFILETIME;

const Unsigned64BitType Low32BitsMask = 0x00000000FFFFFFFF;

Late64BitVal = ((Unsigned64BitType)(Late.dwHighDateTime) << 32) +

Late.dwLowDateTime;

Early64BitVal = ((Unsigned64BitType)(Early.dwHighDateTime) << 32) +

Early.dwLowDateTime;

Difference64BitVal = Late64BitVal - Early64BitVal;

DifferenceAsFILETIME.dwLowDateTime =

(DWORD)(Difference64BitVal & Low32BitsMask);

DifferenceAsFILETIME.dwHighDateTime =

(DWORD)(Difference64BitVal >> 32);

return DifferenceAsFILETIME;

}