wait_for (a duration) ← used
wait until (s.t happen)

ENSC 351: Real-time and Embedded Systems

Craig Scratchley, Fall 2021

Multipart Project Part 3

(Figures to be updated from xmodem to ymodem.)

Please continue working with a partner. You will have next week and the week after, including 3 weekends, to work on this part of the project.

In part 2 of the project you wrote code to both send and receive YMODEM blocks and hooked in a simulator for a communication medium provided by a pair of telephone modems and the telephone system between them. The medium corrupted the data in some ways, but did not as configured inject spurious characters from simulated noise on the simulated telephone line. In particular, we had not enabled the ability for the medium to send an extra ACK character to the YMODEM sender while a block was being transmitted. We will enable that ability in Part 3.

As I will discuss again in class, just before the last byte in a block is transmitted to the YMODEM receiver, the YMODEM sender should dump any "glitch" characters that may have arrived at the sender due to noise on the telephone line. The sender dumps characters by calling myReadcond() specifying that the read should timeout immediately. If any characters are available, they are read in and discarded. If no characters are available, the myReadcond() call times out immediately (it returns the value 0 immediately).

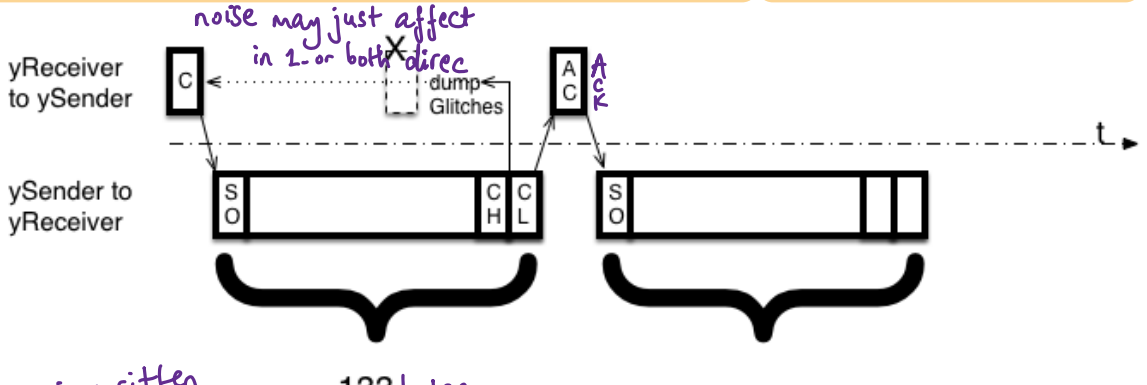


Figure 1: Glitches are dumped just before the last byte in a block is transmitted

waiting for what is written being read

How does the YMODEM sender know that all previously written characters have actually been transmitted to the modem and that now is the time for it to dump glitch characters? Well, in the code, we are calling myTcdrain() for the sender to know. For a terminal (i.e. console) device like a serial port, tcdrain() blocks the calling thread until all previously written characters have actually been sent. However, tcdrain() is not supported out-of-the-box for sockets, which we are using in Part 2 to simulate serial ports. In fact, our myTcdrain() currently does nothing but return 0.

You can enable the sending of extra ACKs in the medium by removing near the top of Medium.cpp the comment characters at the beginning of a line to yield this line:

```
#define SEND_EXTRA_ACKS
```

If you try this once you have finished Part 2, you will see that troubles currently arise transferring a file with the YMODEM protocol.

So, your job is to make modifications in the myIO.cpp file to get myTcdrain() working, similar to how it works for terminal devices, but when using sockets in socketpairs for communication. I recommend that you use condition variables, as discussed in Chapter 4, in order to achieve this. For our immediate needs in Part 2, we have only one thread using each socket (two threads using each socketpair – one thread per socket). However, try to make your code relatively general, where multiple threads might be affecting a socket in a socketpair. For example, thread A might write data to a socket and then block on a call to myTcdrain(), and then a thread B might block on a call to myTcdrain() for the same socket, and then a thread C might come along and read from the paired socket all the data written by thread A. At that point both the threads blocked on myTcdrain() should be unblocked and allowed to return from their respective calls to myTcdrain(). If you want, you can assume that there will be only one (blocked) reading thread at a time, so if one thread is blocked reading from a socket, no other thread will come along and also try to read from that same socket. Let us know if you try to relax that assumption. I will provide a testing project to you all to help in testing your modifications to myIO.cpp.

As long as there is memory available, the myIO.cpp file should work with as many descriptors as needed. We will be reasonable with our testing, but don't put your own arbitrary limit on the number of descriptors.

Please ask on Piazza if you have any questions or need any clarifications.

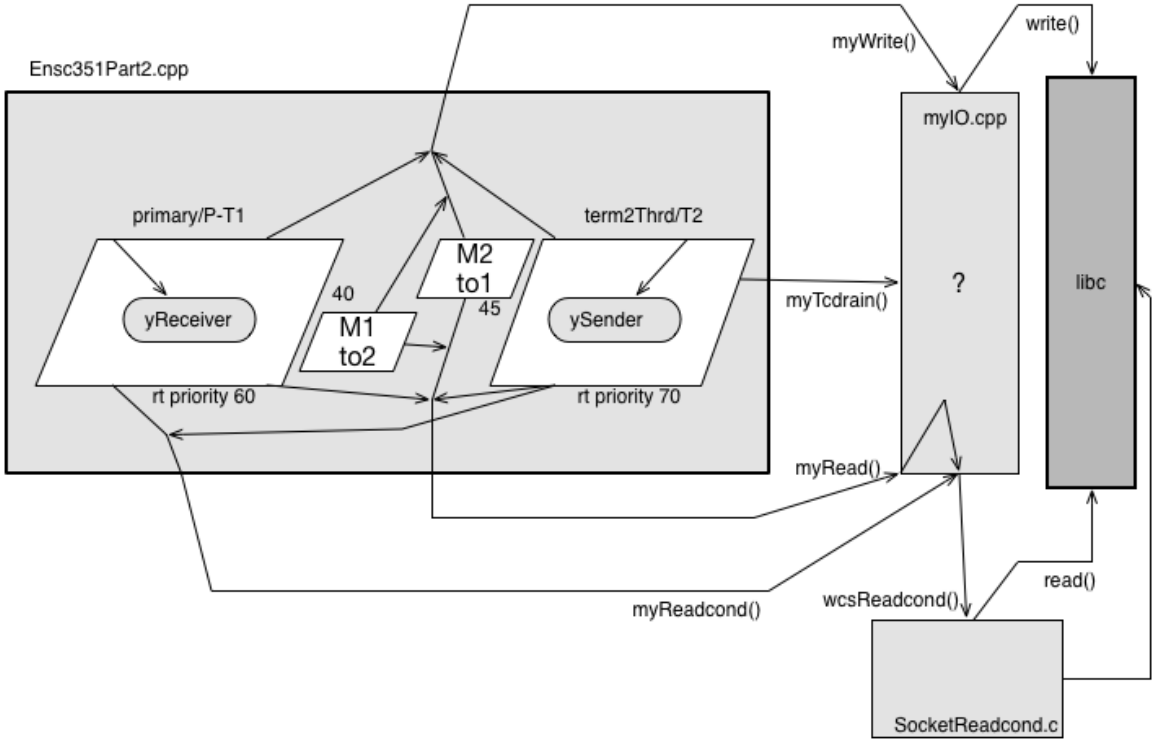
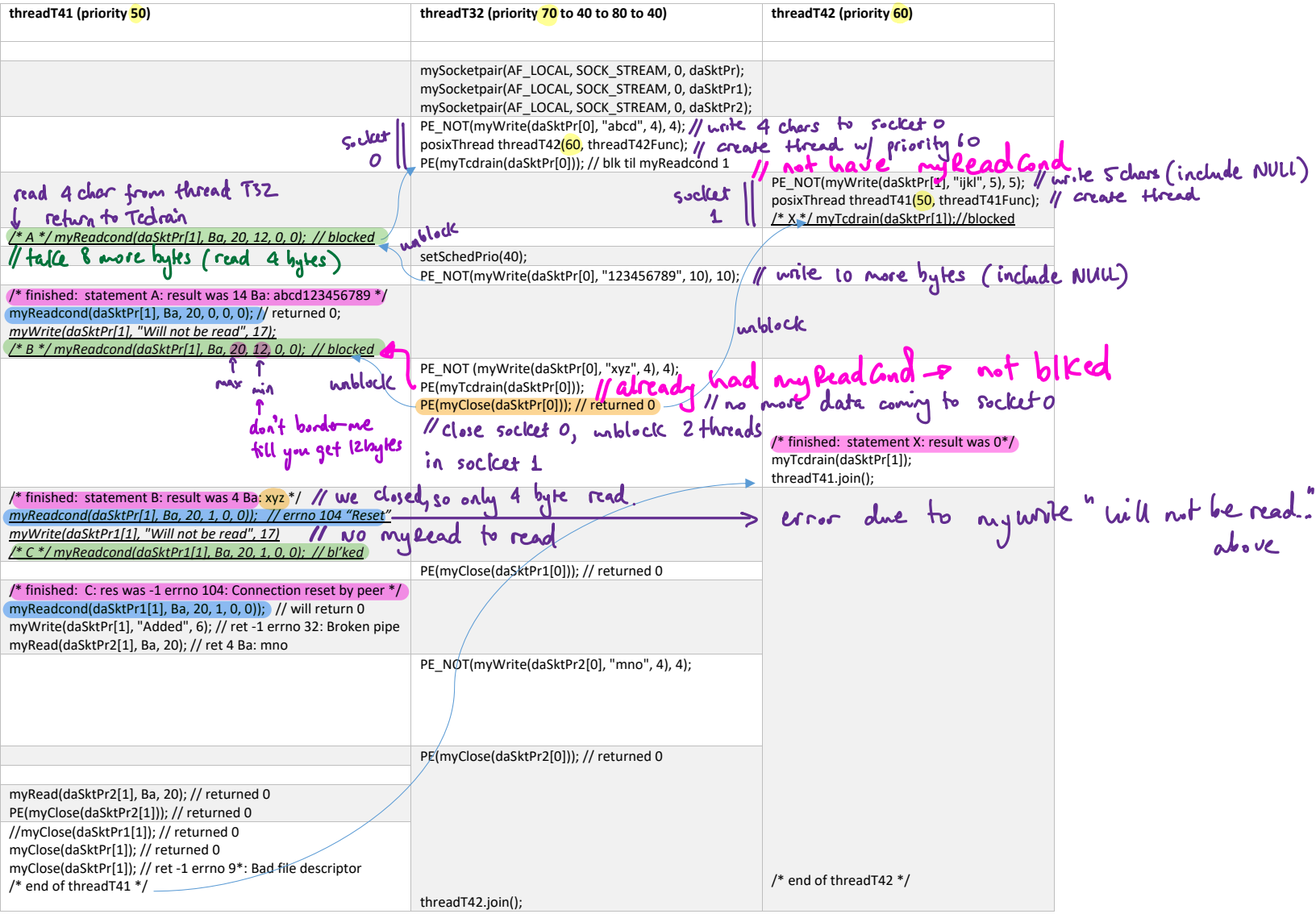


Figure 2: Collaboration Graph showing function calls for socketpair communication

call Tcdrain() after preparing next block.

Order of execution of code in Ensc351Part3-test.cpp and some output -- *Draft*

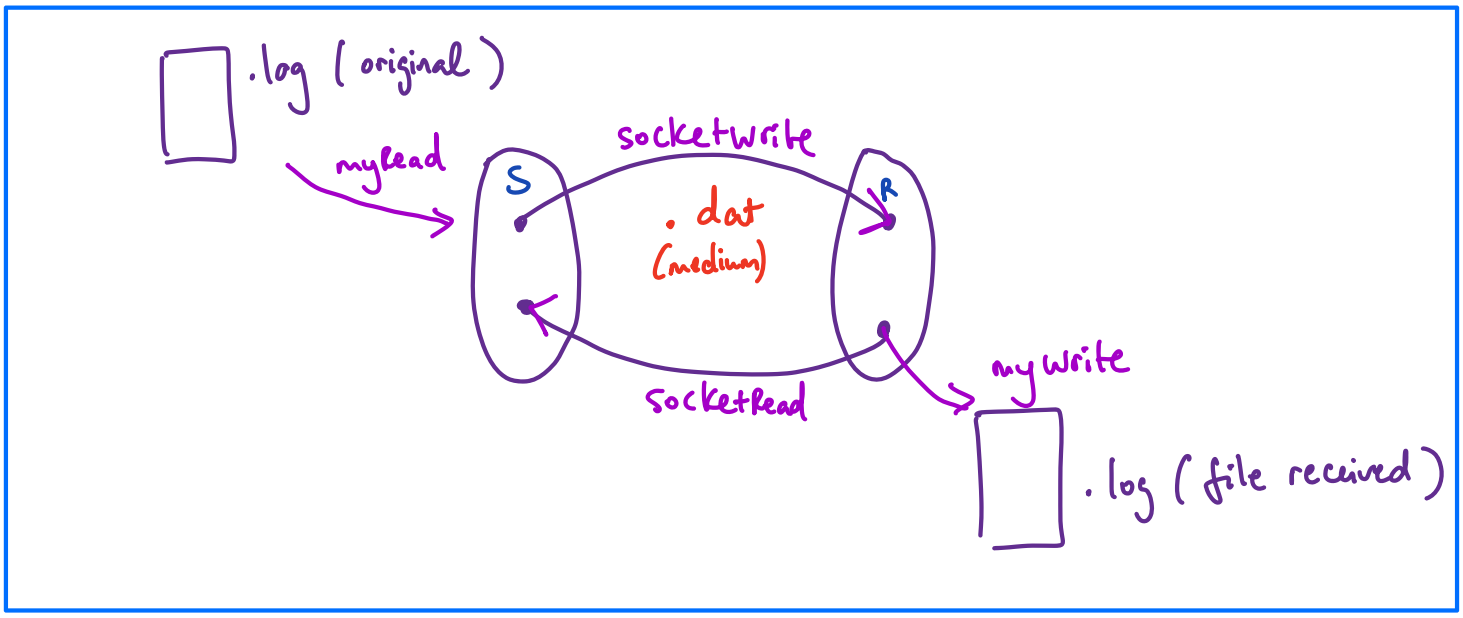


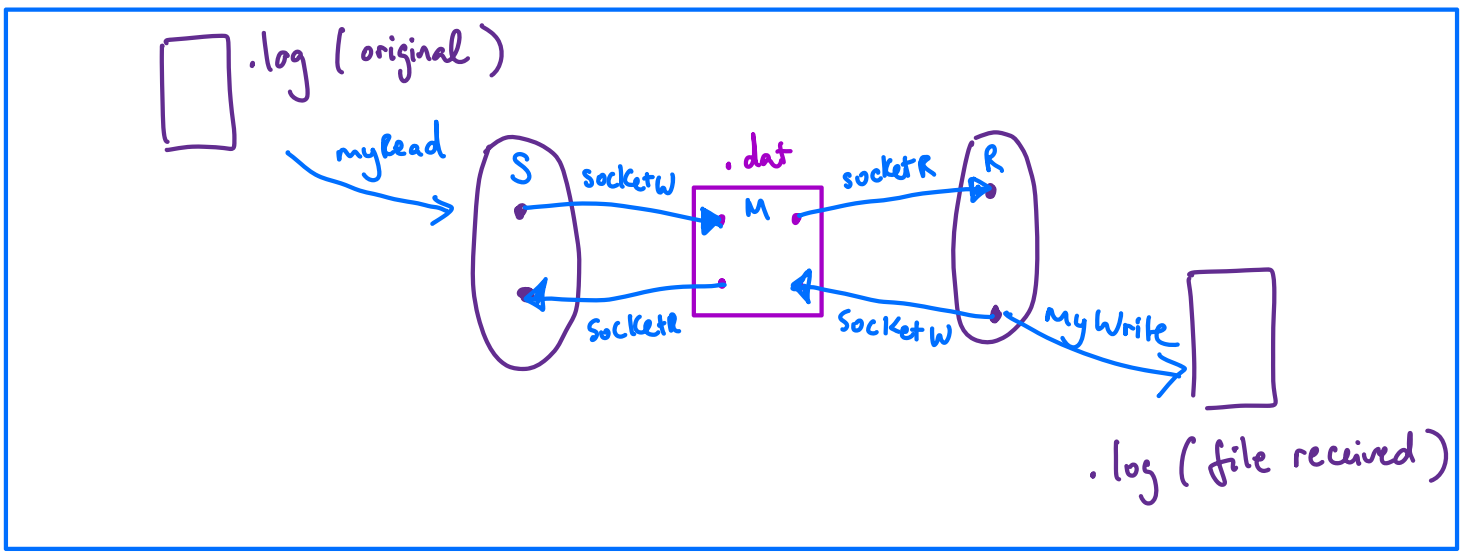
* The OS supplies the error string "Bad file descriptor" like this even though there is no "file" associated with the descriptor. I would have named it simply "Bad descriptor", but files came first in the OS design.

Valgrind conflicts w/ ASAN:

RClick Folder > properties > C/C++ Build > Environment

- Copy SAN_SET_WITH_THREAD to SAN_OPTIONS
- RClick setting > GCC C++ linker > Miscellaneous > Remove Linker Flags





threadTable (soln)

tw: total written
mTCR: maximum total can be read

Order of execution of code in Ensc351Part3-test.cpp and some output and details of the solution

which thread is blocked on cvDrain

threadT41 (priority 50)	threadT32 (priority 70 to 40 to 80 to 40)	threadT42 (priority 60)	daSktPr[0] - 3			daSktPr[1] - 4		
			tW/mTCR	pair	cvDrn	tW/mTCR	pair	cvDrn/cvR
			-	-	-	-	-	-

	mySocketpair(AF_LOCAL, SOCK_STREAM, 0, daSktPr);
	mySocketpair(AF_LOCAL, SOCK_STREAM, 0, daSktPr1);
	mySocketpair(AF_LOCAL, SOCK_STREAM, 0, daSktPr2);
			0/0	4		0/0	3	/

	PE_NOT(myWrite(daSktPr[0], "abcd", 4), 4);
	posixThread threadT42(60, threadT42Func);
	PE(myTcdrain(daSktPr[0])); // blocked
			0/0
			
			0/0
				4
				
				4
						4/0
						
						4/0
							3
							
							3
								/
								
								T32/

		PE_NOT(myWrite(daSktPr[1], "ijkl", 5), 5);
		posixThread threadT41(50, threadT41Func);
		/* X */ myTcdrain(daSktPr[1]);//blocked
			5/0
			
			5/0
				4
				
				4
					
					T42
						4/0
						
						4/0
							3
							
							3
								T32/
								
								T32/

/* A */ myReadcond(daSktPr[1], Ba, 20, 12, 0, 0); // blocked			5	4	T42	4/20	3	/T41
	setSchedPrio(40); PE_NOT(myWrite(daSktPr[0], "123456789", 10), 10);		5/0	4	T42	14/20	3	/

/* finished: statement A: result was 14 Ba: abcd123456789 */
myReadcond(daSktPr[1], Ba, 20, 0, 0, 0); // returned 0;
myWrite(daSktPr[1], "Will not be read", 17);
/* B */ myReadcond(daSktPr[1], Ba, 20, 12, 0, 0); // blocked
			5/0
			5/0
			22/0
			22/0
				4
				4
				4
				4
					T42
					T42
					T42
					T42
						0/0
						0/0
						0/0
						0/20
							3
							3
							3
							3
								/
								/
								/
								/T41

	setSchedPrio(80);
	PE_NOT (myWrite(daSktPr[0], "xyz", 4), 4);
	PE(myTcdrain(daSktPr[0]));
	PE(myClose(daSktPr[0])); // returned 0
	setSchedPrio(40);
			22/0
			
			22/0
				4
				
				-1
					T42	4/20
						
						4/20
							3
							
							-2
								/T41
								
								/

/* finished: statement B: result was 4 Ba: xyz */
myReadcond(daSktPr[1], Ba, 20, 1, 0, 0)); // errno 104 "Reset"
myWrite(daSktPr1[1], "Will not be read", 17)
/* C */ myReadcond(daSktPr1[1], Ba, 20, 1, 0, 0); // blocked
		/* finished: statement X: result was 0*/
		myTcdrain(daSktPr[1]);
		threadT41.join();
			22/0	-1		4/20	-2	/

	PE(myClose(daSktPr1[0]));							

/* finished: C: res was -1 errno 104: Connection reset by peer */
myReadcond(daSktPr1[1], Ba, 20, 1, 0, 0)); // will return 0
myWrite(daSktPr[1], "Added", 6); // ret -1 errno 32: Broken pipe
/* D */ myRead(daSktPr2[1], Ba, 20); // blocked								

	PE_NOT(myWrite(daSktPr2[0], "mno", 4), 4);							
/* finished: statement D: result was 4 Ba: mno */ /* E */ myRead(daSktPr2[1], Ba, 20);								
	PE(myClose(daSktPr2[0]));							

/* finished: statement E: result was 0*/
myClose(daSktPr2[1]); // returned 0
myClose(daSktPr1[1]); // returned 0
myClose(daSktPr[1]); // returned 0
myClose(daSktPr[1]); // ret -1 errno 9 "Bad file descriptor"
myRead(daSktPr[1], Ba, 20); // ret -1 errno 9
// ...
/* end of threadT41 */								

	threadT42.join();	/* end of threadT42 */						

[i] For this course, we will accept a return value of 0 in addition to errno 104 (Connection Reset by Peer)

[ii] The OS supplies the error string “Bad file descriptor” like this even though there is no “file” associated with the descriptor. I would have named it simply “Bad descriptor”, but files came first in the OS design.

myIO (soln)

```
//
============================================================
==========
// Name        : myIO.cpp
// Author(s)   : Craig Scratchley
//             :
// Version     : November, 2021 -- tcdrain for socketpairs.
// Copyright   : Copyright 2021, W. Craig Scratchley, SFU
// Description : An implementation of tcdrain-like behaviour for
socketpairs.
//
============================================================
==========
```

// there are 3 solution options

```
// dangerous in the general case, but uncomment the below to use
// (2) wcsReadcond to block instead of cvRead condition (1) variable
//#define WCSREADCOND
#ifndef WCSREADCOND
// if not using WCSREADCOND, then (3) CIRCBUF is an option, to use
//     a circular buffer instead of read() and write() functions.
//#define CIRCBUF
#endif

// Uncomment the line below to turn on debugging output
//#define REPORT_INFO

#include <sys/socket.h>
#include <unistd.h>          // for posix i/o functions
#include <stdlib.h>
#include <termios.h>         // for tcdrain()
#include <fcntl.h>           // for open/creat
#include <errno.h>
#include <stdarg.h>
#include <mutex>
#include <shared_mutex>
#include <condition_variable>
#include <map>
#include <memory>
#include "AtomicCOUT.h"
#include "SocketReadcond.h"
#include "VNPE.h"
#ifdef CIRCBUF
    #include "RageUtil_CircularBuffer.h"
```

```
#endif

using namespace std;

//Unnamed namespace
namespace{

    class socketInfoClass;

    typedef shared_ptr<socketInfoClass> socketInfoClassSp;
    map<int, socketInfoClassSp> desInfoMap {
        {-2, make_shared<socketInfoClass>(-1)}, // marker for a
descriptor whose pair is closed
        {0, nullptr}, // init for stdin,
        {1, nullptr}, //          stdout,
        {2, nullptr}  //          stderr
    };

    //  A shared mutex used to protect desInfoMap so only a
single thread can modify the map at a time.
    //  This also means that only one call to functions like
mySocketpair() or myClose() can make progress at a time.
    //  This mutex is also used to prevent a paired socket from
being closed at the beginning of a myWrite or myTcdrain function.
    shared_mutex mapMutex;

    class socketInfoClass {
        unsigned totalWritten = 0;// byte written to a buffer
        unsigned maxTotalCanRead = 0;// max total bytes can be read
        condition_variable cvDrain;    by a thread in a socket
#ifndef WCSREADCOND
        condition_variable cvRead;
#endif
#ifdef CIRCBUF
        CircBuf<char> circBuffer;
        bool connectionReset = false;
#endif
        mutex socketInfoMutex;
    public:
        int pair;   // Cannot be private because myWrite and
myTcdrain using it.
                    // -1 when descriptor closed, -2 when paired
descriptor is closed
```

avoid risk when combine code w/ partner whose also have same name class. So only our class can access this unnamed namespace.

// funcs belong to unnamed namespace can only be used in this file but not others. Keep it private!!

this map is not thread safe → need to lock mutex

(1) used to blk reading activity

(2)

(3)

```
        socketInfoClass(unsigned pairInit)
        :pair(pairInit) {
    #ifdef CIRCBUF
                circBuffer.reserve(1100); // note constant of
1100
    #endif
        }

   /*
    * Function:  if necessary, make the calling thread wait for a
reading thread to drain the data
    */
   int draining()
   { // operating on object for paired descriptor of original des
      unique_lock<mutex> socketLk(socketInfoMutex);

      //  paired descriptor could have been closed before we
constructed socketLk
      //  once the reader decides the drainer should wakeup, it
should wakeup
      if (pair >= 0 && totalWritten > maxTotalCanRead)
         cvDrain.wait(socketLk); // what about spurious wakeup?
//       cvDrain.wait(socketLk, [this]{return pair < 0 ||
totalWritten <= maxTotalCanRead;});
                    these 2 condition code can replace w/ comment line
      if (pair == -2) {
         errno = EBADF; // check errno
         return -1;
      }
      return 0;
   }

   int writing(int des, const void* buf, size_t nbyte)
   {
      // operating on object for paired descriptor
      lock_guard<mutex> socketLk(socketInfoMutex);
      // consider unlocking mapMutex
#ifdef CIRCBUF
      int written = circBuffer.write((const char*) buf, nbyte);
#else
      int written = write(des, buf, nbyte);
#endif
```

```
        if (written > 0) {
            totalWritten += written;
#ifndef WCSREADCOND
            cvRead.notify_one();
#endif
        }
#ifdef REPORT_INFO
        COUT << " mw:" << des << ";" << totalWritten << flush;
#endif
        return written;
    }

    int reading(int des, void * buf, int n, int min, int time, int
timeout)
    {
        int bytesRead;
        unique_lock<mutex> socketLk(socketInfoMutex);

        // would not have got this far if pair == -1
        if ((!maxTotalCanRead && totalWritten >= (unsigned) min)
|| pair == -2) {
#ifdef CIRCBUF
            if (pair == -2)
                if (connectionReset) {
                    errno = ECONNRESET;
                    bytesRead = -1;
                    connectionReset = false;
                }
                else
                    bytesRead = 0;
            else {
                bytesRead = circBuffer.read((char *) buf, n);
#else
              // wcsReadcond should not wait in this situation.
              // bytesRead = wcsReadcond(des, buf, n, min, time,
timeout);
            if (min == 0 && totalWritten == 0 && pair >= 0)
                bytesRead = 0;
            else {
                bytesRead = read(des, buf, n); // at least min
will be waiting or pair == -2
#endif
                 if (bytesRead > 0) {
```

notify for read

??

```
                    totalWritten -= bytesRead;
                    if (totalWritten <= maxTotalCanRead /* &&
pair >= 0 */)
                        cvDrain.notify_all();
                }
            }
        }
        else {
            maxTotalCanRead +=n;
            cvDrain.notify_all(); // totalWritten must be less than
min
#ifdef WCSREADCOND
            socketLk.unlock(); // this might be dangerous if there
are multiple readers
            // (a) consider myWrite() is called here, with
sufficient (extra) bytes.
            // (b1) myWrite() could be called again here, which
means that more bytes
            //      could be read, and totalWritten could be 0
            // tcdrain() might have been called here, and might
have to block
             bytesRead = wcsReadcond(des, buf, n, min, time,
timeout);
            // (b2) or myWrite() could be called again here, which
would definitely
             //      mean that totalWritten will end up > 0
            // what if tcdrain() is called here, we don't know if
it would have to block or not.
            socketLk.lock();
#else // better way to use w/ cvRead.wait()
            if (time != 0 || timeout != 0) {
                COUT << "Currently only supporting no timeouts or
immediate timeout" << endl;
                exit(EXIT_FAILURE);
            }

            cvRead.wait(socketLk, [this, min] {
                // after notify, myTcdrain() might have been
called, and block
                return totalWritten >= (unsigned) min || pair <
0;});
            if (pair == -1) { // shouldn't normally happen
                errno = EBADF; // check errno value
```

```
                return -1;
            }
#ifdef CIRCBUF
            bytesRead = circBuffer.read((char *) buf, n);
            if (connectionReset && bytesRead == 0) {
                errno = ECONNRESET;
                bytesRead = -1;
                connectionReset = false;
            }
#else
            // choice below seems to affect "Connection reset by
peer"
            bytesRead = read(des, buf, n);
            //bytesRead = wcsReadcond(des, buf, n, min, time,
timeout);
#endif // #ifdef CIRCBUF
#endif // #ifdef WCSREADCOND

          if (bytesRead != -1)
             totalWritten -= bytesRead;
            maxTotalCanRead -= n;
#ifndef WCSREADCOND
          if (totalWritten > 0 || pair < 0) // || pair == -2
             cvRead.notify_one(); // can this affect errno?
#endif
       }
#ifdef REPORT_INFO
       COUT << " mr:" << des << ";" << bytesRead << ";" <<
totalWritten << flush;
#endif
       return bytesRead;
    } // .reading()

    /*
     * Function:  Closing des. Should be done only after all other
operations on des have returned.
     */
    void closing(int des)
    {
       // mapMutex already locked at this point, so no
mySocketpair or other myClose
       if(pair != -2) { // pair has not already been closed
          socketInfoClassSp des_pair(desInfoMap[pair]);
```

i.e.: If one lock in thread 2, one in thread 3 — if you can't lock one, the other also can't
you can't lock each other, but need another thread 4 to lock them

```
            unique_lock<mutex> socketLk(socketInfoMutex,
 defer_lock);
            unique_lock<mutex> condPairlk(des_pair-
 >socketInfoMutex, defer_lock);
            lock(condPairlk, socketLk); // lock both mutex at the same time,
            pair = -1; // this is first socket in the pair to be
 closed
            des_pair->pair = -2; // paired socket will be the
 second of the two to close.
              if (totalWritten > maxTotalCanRead) {
                  // by closing the socket we are throwing away any
 buffered data.
                  // notification will be sent immediately below to
 any myTcdrain waiters on paired descriptor.
#ifdef CIRCBUF
                  des_pair->connectionReset = true;
#endif
                  cvDrain.notify_all();
              }
#ifndef WCSREADCOND
              if (maxTotalCanRead > 0) {
                  // there shouldn't be any threads waiting in
 myRead() or myReadcond() on des, but just in case.
                  cvRead.notify_all();
              }

            if (des_pair->maxTotalCanRead > 0) {
                // no more data will be written from des
                // notify a thread waiting on reading on paired
 descriptor
                des_pair->cvRead.notify_one();
            }
#endif
            if (des_pair->totalWritten > des_pair->maxTotalCanRead)
 {
                // there shouldn't be any threads waiting in
 myTcdrain on des, but just in case.
                des_pair->cvDrain.notify_all();
            }
        }
    } // .closing()
    }; // socketInfoClass
```

```
    // get shared pointer for des info
    socketInfoClassSp getDesInfoP(int des) {
        auto iter = desInfoMap.find(des);
        if (iter == desInfoMap.end())
            return nullptr; // des not in use
        else
            return iter->second; // return the shared pointer
    }

    // get shared pointer for des info (locked version)
    socketInfoClassSp lockedGetDesInfoP(int des) {
        shared_lock<shared_mutex> desInfoLk(mapMutex);
        return getDesInfoP(des);
    }

    // get shared pointer for paired des info (locked version)
    socketInfoClassSp lockedGetDesPairInfoP(int des) {
        shared_lock<shared_mutex> desInfoLk(mapMutex);
        socketInfoClassSp desInfoP = getDesInfoP(des);
        if (!desInfoP)
            return nullptr; // des not in use
        else // since *desInfoP is good, pair != -1
             // locking mapMutex above makes sure that desinfoP-
>pair is not closed here
            return desInfoMap[desInfoP->pair];
    } // lockedGetDesPairInfoP()
} // unnamed namespace

/*
 * Function: Calling the reading member function to read
 * Return:      An integer with number of bytes read, or -1 for
an error.
 * see https://developer.blackberry.com/native/reference/core/
com.qnx.doc.neutrino.lib_ref/topic/r/readcond.html
 *
 */
int myReadcond(int des, void * buf, int n, int min, int time, int
timeout) {
   socketInfoClassSp desInfoP = lockedGetDesInfoP(des);
   if (!desInfoP)
      return wcsReadcond(des, buf, n, min, time, timeout);
   return desInfoP->reading(des, buf, n, min, time, timeout);
}
```

```
/*
 * Function: Reading directly from a file or from a socketpair
descriptor)
 * Return:      the number of bytes read , or -1 for an error
 */
ssize_t myRead(int des, void* buf, size_t nbyte) {
   socketInfoClassSp desInfoP = lockedGetDesInfoP(des);
   if (!desInfoP)
      return read(des, buf, nbyte); // des is closed or not from
a socketpair
   // myRead (for sockets) usually reads a minimum of 1 byte
   return desInfoP->reading(des, buf, nbyte, 1, 0, 0);
}

/*
 * Return:      the number of bytes written, or -1 for an error
 */
ssize_t myWrite(int des, const void* buf, size_t nbyte) {
    auto desPairInfoSp = lockedGetDesPairInfoP(des); // make a
local shared pointer
    if (desPairInfoSp && desPairInfoSp->pair != -1)
        return desPairInfoSp->writing(des, buf, nbyte);
    return write(des, buf, nbyte); // des is not from a pair of
sockets or socket or pair closed
}

/*
 * Function:  make the calling thread wait for a reading thread
to drain the data
 */
int myTcdrain(int des) {
    auto desPairInfoSp = lockedGetDesPairInfoP(des); // make a
local shared pointer
    if (desPairInfoSp) {
        if (desPairInfoSp->pair == -1)
            return 0; // paired descriptor is closed.
        else
            return desPairInfoSp->draining();
    }
    return tcdrain(des); // des is not from a pair of sockets or
socket closed
}
```

```
/*
 * Function: Open a file and get its file descriptor.
 * Return:      return value of open
 */
int myOpen(const char *pathname, int flags, ...) //, mode_t mode)
{
    mode_t mode = 0;
    // in theory we should check here whether mode is needed.
    va_list arg;
    va_start (arg, flags);
    mode = va_arg (arg, mode_t);
    va_end (arg);
   int des = open(pathname, flags, mode);
   if (des != -1){
       lock_guard<shared_mutex> desInfoLk(mapMutex);
      desInfoMap[des] = nullptr;}
   return des;
}

/*
 * Function: Create a new file and get its file descriptor.
 * Return:      return value of creat
 */
int myCreat(const char *pathname, mode_t mode)
{
   lock_guard<shared_mutex> desInfoLk(mapMutex);
   int des = creat(pathname, mode);
   if (des != -1)
      desInfoMap[des] = nullptr;
   return des;
}

/*
 * Function: Create pair of sockets and put them in desInfoMap
 * Return:      return an integer that indicate if it is
successful (0) or not (-1)
 */
int mySocketpair(int domain, int type, int protocol, int des[2])
{
   lock_guard<shared_mutex> desInfoLk(mapMutex);
   int returnVal = socketpair(domain, type, protocol, des);
   if(returnVal != -1) {
```

move inside if()

what happen if we switch the order of these 2 lines?

socketpair is thread safe (which not modify

our Map, so it can be put above mutex → holds mutex for shorter time → more efficient so other thread donnot wait too long

```
        desInfoMap.emplace(des[0], make_shared<socketInfoClass>(des[1]));
        desInfoMap.emplace(des[1], make_shared<socketInfoClass>(des[0]));
    }
    return returnVal;
}

/*
 * Function: Closing des
 *     myClose() should not be called until all other calls using the descriptor have finished.
 */
int myClose(int des) {
    {
        lock_guard<shared_mutex> desInfoLk(mapMutex);
        auto iter = desInfoMap.find(des);
        if (iter != desInfoMap.end()) { // if in the map
            if (iter->second) // if shared pointer exists
                iter->second->closing(des); // -1 or 0
            desInfoMap.erase(des);
        }
    }
    return close(des);
}
```

// check if put mutex at the beginning gives any error?

Ensc351Part3_test.cpp

```
/* Ensc351Part3-test.cpp -- October 21 -- Copyright 2021 Craig
Scratchley */

/* This program can be used to test your changes to myIO.cpp
 *
 * Put this project in the same workspace as your Ensc351 library
project,
 * and build it.
 *
 * With 3 created threads for a total of 4 threads, the output
that I get with my solution
 * is in the file "output-fromSolution".
 *
 */

#include <sys/socket.h>
#include <stdlib.h>          // for exit()
#include <sched.h>
#include "posixThread.hpp"
#include "VNPE.h"
#include "myIO.h"

// Uncomment the below to send output to the file "output".
//#define OUTPUT_TO_FILE

#ifdef OUTPUT_TO_FILE
    #include <fstream>
    std::ofstream COUT("output");
    // When RUNning the Release configuration, output will appear
in the Eclipse project.
    // When using the "viaConnection" launcher, output will
appear at path /home/osboxes/output
#else
    //#define COUT cout
    #include "AtomicCOUT.h"
#endif


#define REPORT0(S) COUT << threadName << ": " << #S << ";
statement will now be started\n";  \
    S; \
    COUT << threadName << ": " << #S << "; statement has now
finished\n";
```

```
#define REPORT1(FC) {COUT << threadName << ": " << #FC << " will
now be called\n";  \
    int RV = FC; \
    COUT << threadName << ": " << #FC << " result was " << RV; \
    if (RV == -1) COUT << " errno " << errno << ": " <<
strerror(errno); \
    COUT << "\n"; }
#define REPORT2(FC) {COUT << threadName << ": " << #FC << " will
now be called\n";  \
    int RV = FC; \
    COUT << threadName << ": " << #FC << " result was " << RV; \
    if (RV == -1) COUT << " errno " << errno << ": " <<
strerror(errno); \
    else if (RV > 0) COUT << " Ba: " << Ba; \
    COUT << "\n";}

using namespace std;
using namespace pthreadSupport;

static int daSktPr[2];    // Descriptor Array for Socket Pair
static int daSktPr2[2];    // Descriptor Array for 2nd Socket
Pair
static int daSktPr1[2];    // Descriptor Array for another Socket
Pair
//static int daSktPr0[2];    // Descriptor Array for yet another
Socket Pair
cpu_set_t cpu_set;
int myCpu=0;

void threadT41Func(void) // starts at priority 50
{
    char    Ba[20];
    const char* threadName = "T41";
    PE_0(pthread_setname_np(pthread_self(), threadName));
    //
    // Blank lines below indicate that the statement above the
blank line
    // will finish after one or more other threads in the process
have made progress.
    REPORT2(myReadcond(daSktPr[1], Ba, 20, 12, 0, 0));   // will
block until myWrite of 10 characters

    REPORT2(myReadcond(daSktPr[1], Ba, 20, 0, 0, 0));
```

```
    REPORT1(myWrite(daSktPr[1], "Will not be read", 17));
    REPORT2(myReadcond(daSktPr[1], Ba, 20, 12, 0, 0));   // will
block until myClose(daSktPr[0])

    REPORT2(myReadcond(daSktPr[1], Ba, 20, 1, 0, 0));   // will
return -1 with error 104, Connection reset by peer
    REPORT1(myWrite(daSktPr1[1], "Will not be read", 17));
    REPORT2(myReadcond(daSktPr1[1], Ba, 20, 1, 0, 0));   // will
block until myClose(daSktPr1[0])

    REPORT2(myReadcond(daSktPr1[1], Ba, 20, 1, 0, 0));   // will
return 0
   REPORT1(myWrite(daSktPr[1], "Added", 6));
    REPORT2(myRead(daSktPr2[1], Ba, 20));

    REPORT2(myRead(daSktPr2[1], Ba, 20));

    REPORT1(myClose(daSktPr2[1]));
    REPORT1(myClose(daSktPr1[1]));
   REPORT1(myClose(daSktPr[1]));
   REPORT1(myClose(daSktPr[1]));
    REPORT2(myRead(daSktPr[1], Ba, 20));
    REPORT2(myReadcond(daSktPr[1], Ba, 20, 0, 0, 0));
    REPORT1(myWrite(daSktPr[1], Ba, 20));
    REPORT1(myTcdrain(daSktPr[1]));
}

void threadT42Func(void) // starts at priority 60
{
    const char* threadName = "T42";
    PE_0(pthread_setname_np(pthread_self(), threadName));
    //
//    REPORT1(myClose(daSktPr0[1]));
//
    REPORT1(PE_NOT(myWrite(daSktPr[1], "ijkl", 5), 5));
    REPORT0(posixThread threadT41(50, threadT41Func));
    REPORT1(myTcdrain(daSktPr[1])); // will block until
myClose(daSktPr[0])

    REPORT1(myTcdrain(daSktPr[1]));
    REPORT0(threadT41.join());

    // output happens at this time from the above REPORT0
```

```
}

void threadT32Func(void) // priority 70 -> priority 40 ->
priority 80 -> priority 40
{
//    char    Ba[20];
    const char* threadName = "T32";
    PE_0(pthread_setname_np(pthread_self(), threadName));
    //
//    REPORT1(mySocketpair(AF_LOCAL, SOCK_STREAM, 0,
daSktPr0)); // daSktPrA
    REPORT1(mySocketpair(AF_LOCAL, SOCK_STREAM, 0, daSktPr));
    REPORT1(mySocketpair(AF_LOCAL, SOCK_STREAM, 0, daSktPr1));
    REPORT1(mySocketpair(AF_LOCAL, SOCK_STREAM, 0, daSktPr2));
    //
//    REPORT1(PE_NOT(myWrite(daSktPr0[0], "abcd", 4), 4));
//    REPORT0(posixThread threadT42(60, threadT42Func));
//    REPORT2(myReadcond(daSktPr0[0], Ba, 20, 1, 0, 0));   //
will return -1 with error 104, Connection reset by peer
//
//    REPORT2(myReadcond(daSktPr0[0], Ba, 20, 1, 0, 0));   //
will return 0
    REPORT1(PE_NOT(myWrite(daSktPr[0], "abcd", 4), 4));
    REPORT0(posixThread threadT42(60, threadT42Func));
    REPORT1(myTcdrain(daSktPr[0])); // will block until 1st
myReadcond(..., 12, ...);

    REPORT1(setSchedPrio(40));
    REPORT1(PE_NOT(myWrite(daSktPr[0], "123456789", 10), 10)); //
don't forget nul termination character

    REPORT1(setSchedPrio(80));
    REPORT1(PE_NOT(myWrite(daSktPr[0], "xyz", 4), 4));
    REPORT1(PE(myTcdrain(daSktPr[0])));
    REPORT1(myClose(daSktPr[0]));
    REPORT1(setSchedPrio(40));

    sched_yield(); // I noticed once the above priority change
was not immediately respected
    REPORT1(myClose(daSktPr1[0]));

    REPORT1(PE_NOT(myWrite(daSktPr2[0], "mno", 4), 4));
```

```
    REPORT1(myClose(daSktPr2[0]));

    REPORT0(threadT42.join());
}

int main() {
    CPU_SET(myCpu, &cpu_set);
    const char* threadName = "Pri";
    PE_0(pthread_setname_np(pthread_self(), threadName));
    PE(sched_setaffinity(0, sizeof(cpu_set), &cpu_set)); // set
processor affinity for current thread

    // Pre-allocate some memory for the process.
    // Seems to make priorities work better, at least
    // when using gdb.
    // For some programs, the amount of memory allocated
    // here should perhaps be greater.
    void* ptr = malloc(20000);
    free(ptr);

   try{
       sched_param sch;
        int policy = -1;
       int primaryPriority = 90;

       getSchedParam(&policy, &sch);
      if (sch.__sched_priority < 98)
          cout << "**** If you are debugging, debugger is not
running at a high priority. ****\n" <<
                          " **** This could cause problems with
debugging.  Consider debugging\n" <<
                          " **** with the proper debug launch
configuration ****" << std::endl;
       cout << "Primary Thread was executing at policy " <<
policy << " and priority " <<  sch.sched_priority << endl;
      sch.__sched_priority = primaryPriority;
      setSchedParam(SCHED_FIFO, sch); //SCHED_FIFO == 1,
SCHED_RR == 2
      getSchedParam(&policy, &sch);
       cout << "Primary Thread now executing at policy (should
be 1) " << policy << " and priority (should be " <<
primaryPriority << ") " <<  sch.sched_priority << endl;
```

```
        REPORT0(posixThread T32(SCHED_FIFO, 70, threadT32Func));
        REPORT0(T32.join());

        return 0;
    }
    catch (std::system_error& error){
       cout << "Error: " << error.code() << " - " << error.what()
<< endl;
        return error.code().value();
    }
     catch (...) { throw; }
}
```

RageUtil_CircularBuffer.h

not truely safe.

```
/* CircBuf - A fast, [limited] thread-safe, lockless circular
buffer. */
/* read()/write() interface adjusted by Craig Scratchley to be
similar
 * to the posix read() and write() functions in order to increase
efficiency.
 * Craig Scratchley -- 2011 - 2020 */
```

ATI has RAGE graphic card

```
#ifndef RAGE_UTIL_CIRCULAR_BUFFER
#define RAGE_UTIL_CIRCULAR_BUFFER

#include <cstring> // for memcpy
#include <atomic>
#include <algorithm> // for min

/* Lock-free circular buffer.  This should be threadsafe if one
thread is reading
 * and another is writing. */
template<class T>
class CircBuf
{
    T *buf;
    /* read_pos is the position data is read from; write_pos is
the position
     * data is written to.  If read_pos == write_pos, the buffer
is empty.
     *
     * There will always be at least one position empty, as a
completely full
     * buffer (read_pos == write_pos) is indistinguishable from an
empty buffer.
     *
     * Invariants: read_pos < size, write_pos < size. */
    unsigned size;
    unsigned m_iBlockSize;

    // Craig says:  making the variables volatile won't make this
code thread safe.
    /* These are volatile to prevent reads and writes to them from
being optimized. */ // do exactly what the program does (ex: you close twice,
    // volatile std::atomic<unsigned> read_pos, write_pos; the program
    std::atomic<unsigned> read_pos, write_pos; will ignore once b/c why
    // volatile unsigned read_pos, write_pos;      you need to close twice?,
                                                   volatile prevent that)
```

① let's the processor can see that 64 bits are read or written
② make sure all processors agree to the operation orders (usually we break in half 32 bits to send/read (will slowing down your processors to have agreements)

```
public:
    CircBuf()
    {
        buf = nullptr;
        clear();
    }

    ~CircBuf()
    {
        delete[] buf;
    }

// atomics cause a problem for swap() and copy assignment
// void swap( CircBuf &rhs )
// {
//     std::swap( size, rhs.size );
//     std::swap( m_iBlockSize, rhs.m_iBlockSize );
//       // a correct compiler will not swap atomics
//     std::swap( read_pos, rhs.read_pos );
//     std::swap( write_pos, rhs.write_pos );
//     std::swap( buf, rhs.buf );
// }
//
// CircBuf &operator=( const CircBuf &rhs )
// {
//     CircBuf c( rhs );
//     this->swap( c );
//     return *this;
// }

    CircBuf( const CircBuf &cpy )
    {
        size = cpy.size;
        read_pos = cpy.read_pos;
        write_pos = cpy.write_pos;
        m_iBlockSize = cpy.m_iBlockSize;
        if( size )
        {
            buf = new T[size];
            std::memcpy( buf, cpy.buf, size*sizeof(T) );
        }
        else
        {
```

```
            buf = nullptr;
        }
    }

    /* Return the number of elements available to read. */
    unsigned num_readable() const
    {
        const int rpos = read_pos;
        const int wpos = write_pos;
        if( rpos < wpos )
            /* The buffer looks like "eeeeDDDDeeee" (e = empty, D =
data). */
            return wpos - rpos;
        else if( rpos > wpos )
            /* The buffer looks like "DDeeeeeeeeDD" (e = empty, D =
data). */
            return size - (rpos - wpos);
        else // if( rpos == wpos )
            /* The buffer looks like "eeeeeeeeeeee" (e = empty, D =
data). */
            return 0;
    }

    /* Return the number of writable elements. */
    unsigned num_writable() const
    {
        const int rpos = read_pos;
        const int wpos = write_pos;

        int ret;
        if( rpos < wpos )
            /* The buffer looks like "eeeeDDDDeeee" (e = empty, D =
data). */
            ret = size - (wpos - rpos);
        else if( rpos > wpos )
            /* The buffer looks like "DDeeeeeeeeDD" (e = empty, D =
data). */
            ret = rpos - wpos;
        else // if( rpos == wpos )
            /* The buffer looks like "eeeeeeeeeeee" (e = empty, D =
data). */
            ret = size;
```

```
        /* Subtract the blocksize, to account for the element that
we never fill
         * while keeping the entries aligned to m_iBlockSize. */
        return ret - m_iBlockSize;
    }

    unsigned capacity() const { return size; }

    void reserve( unsigned n, int iBlockSize = 1 )
    {
        m_iBlockSize = iBlockSize;

        clear();
        delete[] buf;
        buf = nullptr;

        /* Reserve an extra byte.  We'll never fill more than n
bytes; the extra
         * byte is to guarantee that read_pos != write_pos when
the buffer is full,
         * since that would be ambiguous with an empty buffer. */
        if( n != 0 )
        {
            size = n+1;
            size = ((size + iBlockSize - 1) / iBlockSize) *
iBlockSize; // round up

            buf = new T[size];
        }
        else
            size = 0;
    }

    void clear()
    {
        read_pos = write_pos = 0;
    }

    /* Indicate that n elements have been written. */
    void advance_write_pointer( int n )
    {
        write_pos = (write_pos + n) % size;
    }
```

what memory ordering we need? (release/acquire)

```
   /* Indicate that n elements have been read. */
   void advance_read_pointer( int n )
   {
      read_pos = (read_pos + n) % size;
   }

   void get_write_pointers( T *pPointers[2], unsigned pSizes[2] )
   {
      const int rpos = read_pos;
      const int wpos = write_pos;

      if( rpos <= wpos )
      {
         /* The buffer looks like "eeeeDDDDeeee" or
"eeeeeeeeeeee" (e = empty, D = data). */
         pPointers[0] = buf+wpos;
         pPointers[1] = buf;

         pSizes[0] = size - wpos;
         pSizes[1] = rpos;
      }
      else if( rpos > wpos )
      {
         /* The buffer looks like "DDeeeeeeeeDD" (e = empty, D =
data). */
         pPointers[0] = buf+wpos;
         pPointers[1] = nullptr;

         pSizes[0] = rpos - wpos;
         pSizes[1] = 0;
      }

      /* Subtract the blocksize, to account for the element that
we never fill
       * while keeping the entries aligned to m_iBlockSize. */
      if( pSizes[1] )
         pSizes[1] -= m_iBlockSize;
      else
         pSizes[0] -= m_iBlockSize;
   }

   /* Like get_write_pointers, but only return the first range
```

```
available. */
   T *get_write_pointer( unsigned *pSizes )
   {
      T *pBothPointers[2];
      unsigned iBothSizes[2];
      get_write_pointers( pBothPointers, iBothSizes );
      *pSizes = iBothSizes[0];
      return pBothPointers[0];
   }

   void get_read_pointers( T *pPointers[2], unsigned pSizes[2] )
   {
      const int rpos = read_pos;
      const int wpos = write_pos;

      if( rpos <= wpos )
      {
         /* The buffer looks like "eeeeDDDDeeee" (e = empty, D =
data). */
           /*    or   */
           /* The buffer looks like "eeeeeeeeeeee" (e = empty, D
= data). */
         pPointers[0] = buf+rpos;
         pPointers[1] = nullptr;

         pSizes[0] = wpos - rpos;
         pSizes[1] = 0;
      }
      else
      {
         /* The buffer looks like "DDeeeeeeeeDD" (e = empty, D =
data). */
         pPointers[0] = buf+rpos;
         pPointers[1] = buf;

         pSizes[0] = size - rpos;
         pSizes[1] = wpos;
      }
   }

   /* Write buffer_size elements from buffer into the circular
buffer object,
    * and advance the write pointer.  Return the number of
```

```
elements that were
    * able to be written.  If
    * the data will not fit entirely, as much data as possible
will be fit
    * in. */
   unsigned write( const T *buffer, unsigned buffer_size )
   {
      using std::min;
      using std::max;
      T *p[2];
      unsigned sizes[2];
      get_write_pointers( p, sizes );

      unsigned max_write_size = sizes[0] + sizes[1];
      if( buffer_size > max_write_size )
         buffer_size = max_write_size;

      const int from_first = min( buffer_size, sizes[0] );
      std::memcpy( p[0], buffer, from_first*sizeof(T) );
      if( buffer_size > sizes[0] )
         std::memcpy( p[1], buffer+from_first, max(buffer_size-
sizes[0], 0u)*sizeof(T) );

      advance_write_pointer( buffer_size );

      return buffer_size;
   }

   /* Read buffer_size elements into buffer from the circular
buffer object,
    * and advance the read pointer.  Return the number of
elements that were
    * read.  If buffer_size elements cannot be read, as many
elements as
    * possible will be read */
   unsigned read( T *buffer, unsigned buffer_size )
   {
      using std::max;
      using std::min;
      T *p[2];
      unsigned sizes[2];
      get_read_pointers( p, sizes );
```

```
       unsigned max_read_size = sizes[0] + sizes[1];
       if( buffer_size > max_read_size )
          buffer_size = max_read_size;

       const int from_first = min( buffer_size, sizes[0] );
       std::memcpy( buffer, p[0], from_first*sizeof(T) );
       if( buffer_size > sizes[0] )
          std::memcpy( buffer+from_first, p[1], max(buffer_size-
sizes[0], 0u)*sizeof(T) );

       /* Set the data that we just read to 0xFF.  Thisa8 way, if
we're passing pointers
        * through, we can tell if we accidentally get a stale
pointer. */
       std::memset( p[0], 0xFF, from_first*sizeof(T) );
       if( buffer_size > sizes[0] )
          std::memset( p[1], 0xFF, max(buffer_size-sizes[0],
0u)*sizeof(T) );

       advance_read_pointer( buffer_size );
       return buffer_size;
   }
};

#endif

/*
 * Copyright (c) 2004 Glenn Maynard
 * All rights reserved.
 *
 * Permission is hereby granted, free of charge, to any person
obtaining a
 * copy of this software and associated documentation files (the
 * "Software"), to deal in the Software without restriction,
including
 * without limitation the rights to use, copy, modify, merge,
publish,
 * distribute, and/or sell copies of the Software, and to permit
persons to
 * whom the Software is furnished to do so, provided that the
above
 * copyright notice(s) and this permission notice appear in all
copies of
```

```
 * the Software and that both the above copyright notice(s) and
this
 * permission notice appear in supporting documentation.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS
 * OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT OF
 * THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
HOLDERS
 * INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY
SPECIAL INDIRECT
 * OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING
FROM LOSS
 * OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR
 * OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH
THE USE OR
 * PERFORMANCE OF THIS SOFTWARE.
 */
```