- open book
- statechart parts.
- no definition.
- look at code & answer questions
- 2nd half of chap 4 (timeouts, notion of time)
- condition variable

- part 3_ circBuf _ myIO. cpp
- sequential order
- required / released order

the instructors' answer, *where instructors collectively construct a single answer*

Hi.

So I haven't written the final exam yet, but I do expect Part 4 and StateCharts to be part of the final exam.

Timeouts is a big part of Part 5, and I expect to have some timeout-related stuff on the exam. Make sure that you have read Section 4.3 of the textbook. I'll be talking about that in class.

I expect to bring the CircularBuffer code and Atomic Variables into the final exam. I spoke about the CircularBuffer code in class already, but I haven't covered Atomic Variables yet. I plan to do much of that in tomorrow's lecture, and so if you haven't done so already you should definitely consider reading much of Chapter 5 of the textbook. You can skip subsection 5.2.7, the discussion of memory_order_consume, and the entries in section 5.3.7 that we haven't been working with this term. Don't worry too much about all the nitty-gritty details in Chapter 5, but please plan to understand the higher-level issues that are discussed. I will talk about the memory orderings in class and the code that you will need to understand for the final exam.

Also, expect mutexes and condition variables to be present in the final exam as well.

I will discuss more about the final exam in the remaining classes.

Craig

// condition variable

```
353
354 static struct uart_data *uart_data_init(void)
355 {
356         struct uart_data *uart;
357
358         uart = kzalloc(sizeof(*uart), GFP_KERNEL);
359         if (!uart)
360                 return NULL;
361         init_waitqueue_head(&uart->readable);
362         init_waitqueue_head(&uart->writeable);
363         INIT_KFIFO(uart->in_fifo);
364         INIT_KFIFO(uart->out_fifo);
365         mutex_init(&uart->read_lock);
366         mutex_init(&uart->write_lock);
367         tasklet_init(&uart->uart_tasklet,
368                      uart_tasklet_func, (unsigned long)uart);
369
370         uart->mem_start = insmodaddr;
```

↖ kernel version of thread

Table 4.1 Functions that accept timeouts

Class/Namespace	Functions	Return values
`std::this_thread` namespace	`sleep_for(duration)` `sleep_until(time_point)`	N/A
`std::condition_variable` or `std::condition_variable_any`	`wait_for(lock, duration)` `wait_until(lock, time_point)`	`std::cv_status::timeout` or `std::cv_status::no_timeout`
	`wait_for(lock, duration, predicate)` `wait_until(lock, time_point, predicate)`	`bool`—the return value of the *predicate* when awakened
`std::timed_mutex` or `std::recursive_timed_mutex`	`try_lock_for(duration)` `try_lock_until(time_point)`	`bool`—`true` if the lock was acquired, `false` otherwise
`std::unique_lock<TimedLockable>`	`unique_lock(lockable, duration)` `unique_lock(lockable, time_point)`	N/A—`owns_lock()` on the newly constructed object; returns `true` if the lock was acquired, `false` otherwise
	`try_lock_for(duration)` `try_lock_until(time_point)`	`bool`—`true` if the lock was acquired, `false` otherwise
`std::future<ValueType>` or `std::shared_future<ValueType>`	`wait_for(duration)` `wait_until(time_point)`	`std::future_status::timeout` if the wait timed out, `std::future_status::ready` if the future is ready, or `std::future_status::deferred` if the future holds a deferred function that hasn't yet started

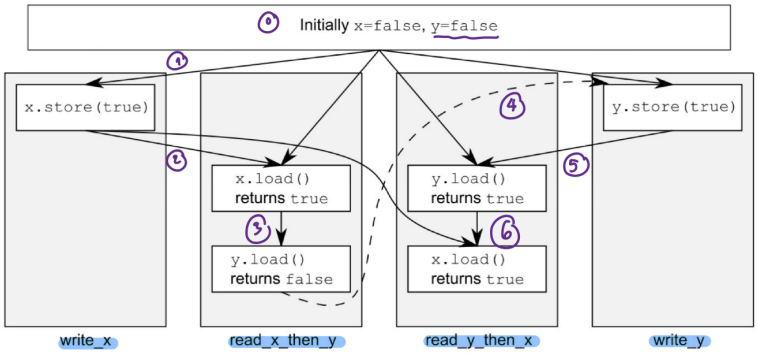


Figure 5.3 Sequential consistency and happens-before

↳ slows your program down.

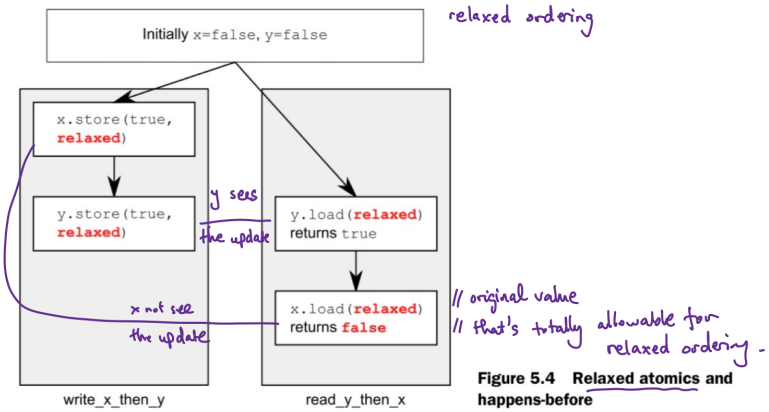
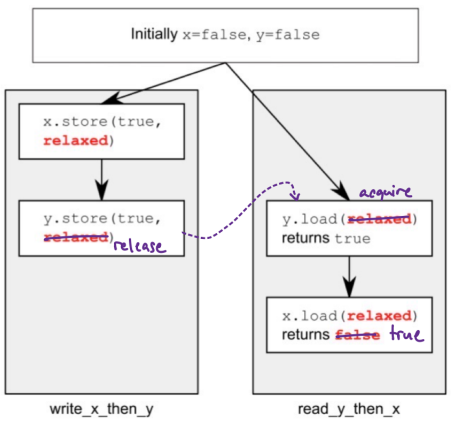


Figure 5.4 Relaxed atomics and happens-before

// runs much more quickly but the order isn't cared



b/c synchronization of y.store/load w/ release/acquire.
x.store happens before x.load, so "relaxed" sees the update.

release/acquire and

Figure 5.5 Relaxed atomics and happens-before

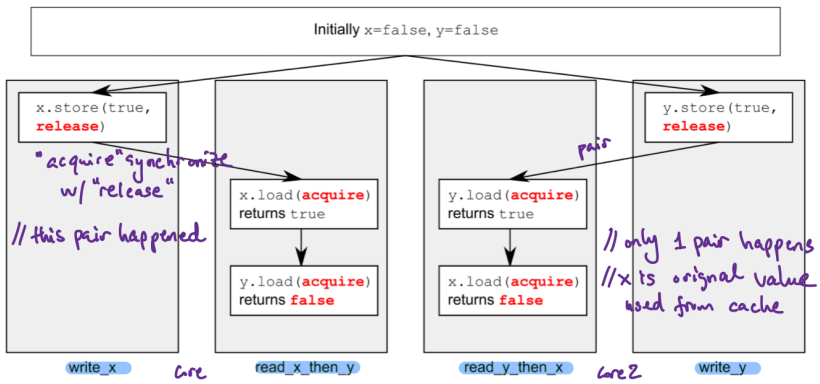


Figure 5.6 Acquire-release and happens-before

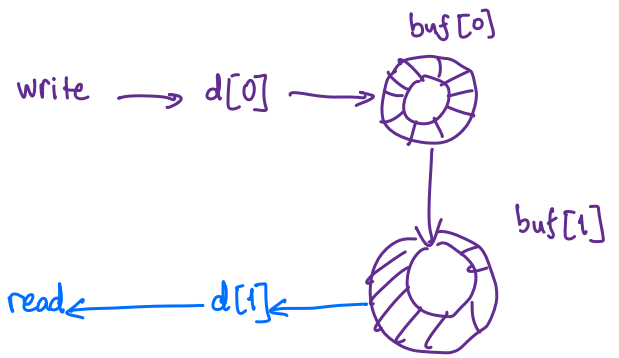
POSIX func (select()) check to see if data comes from either des.

C : C char
A : ACK
N : NAK or 'N'
E : EOT
! : CAN

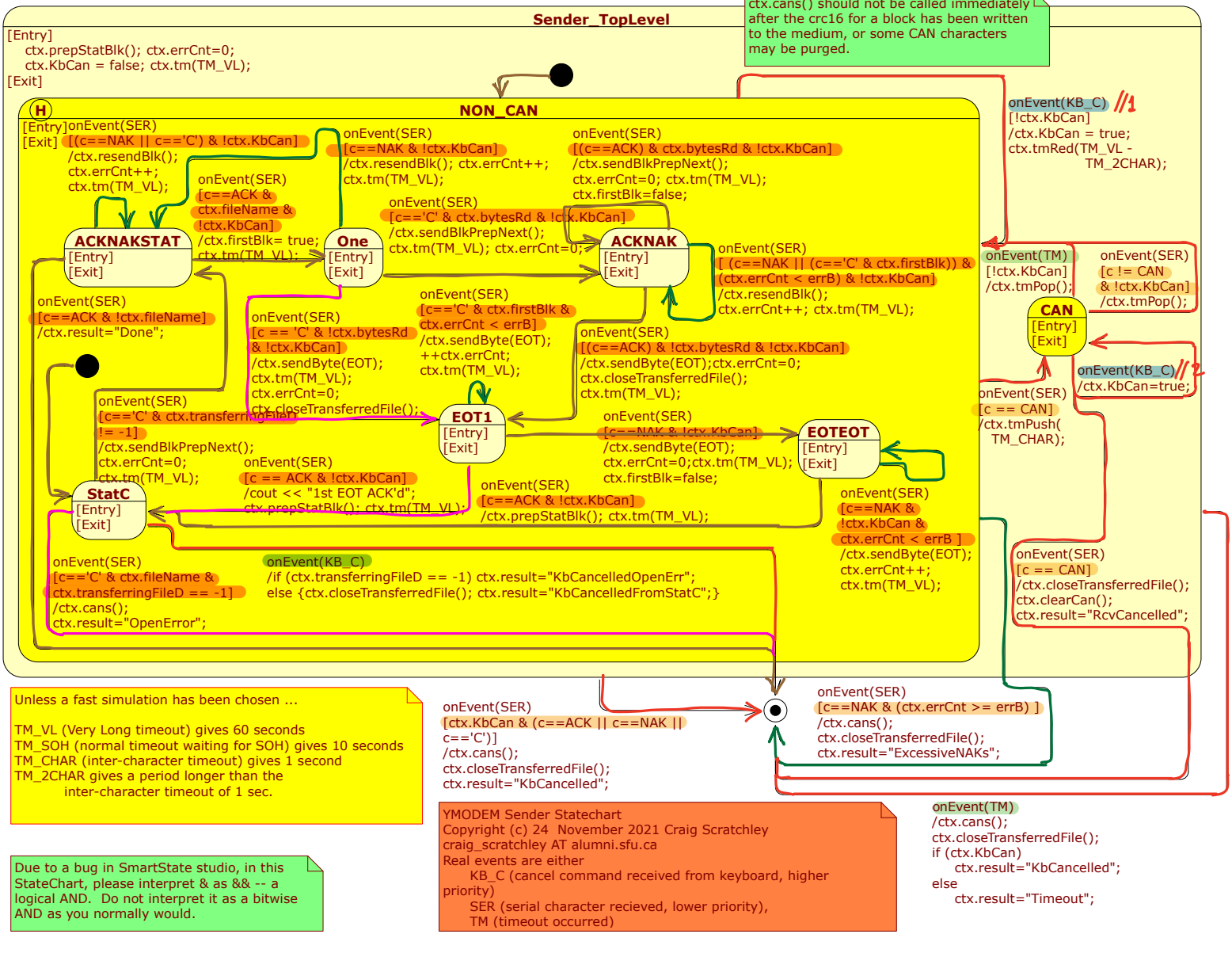
(c) ← ( ) do w/ receiver
[ ] ← do w/ sender
{ } ← medium
< > ← medium in 1-direction

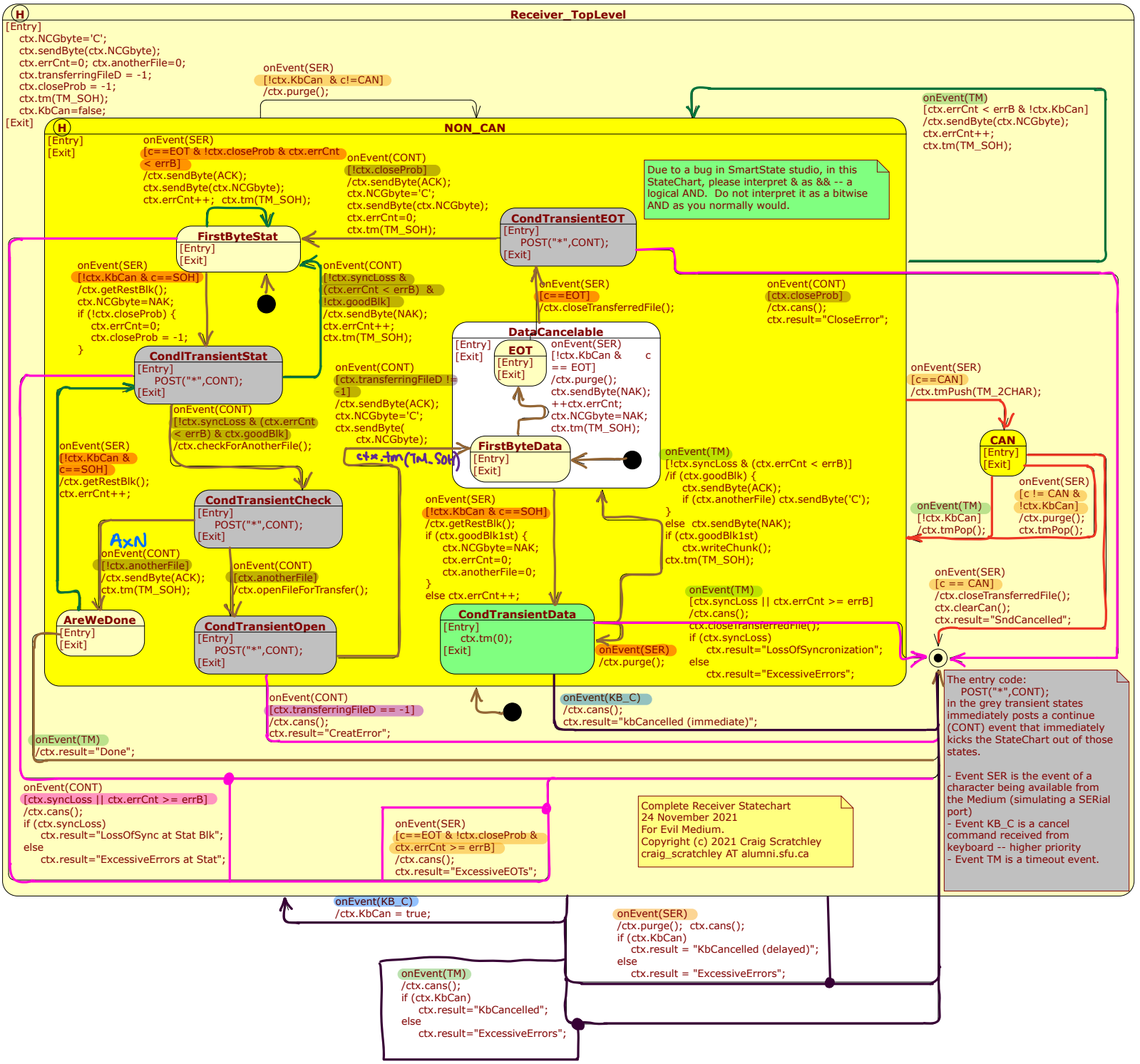
x : corrupted
+ : added / glitched / extra
_ : dropped char

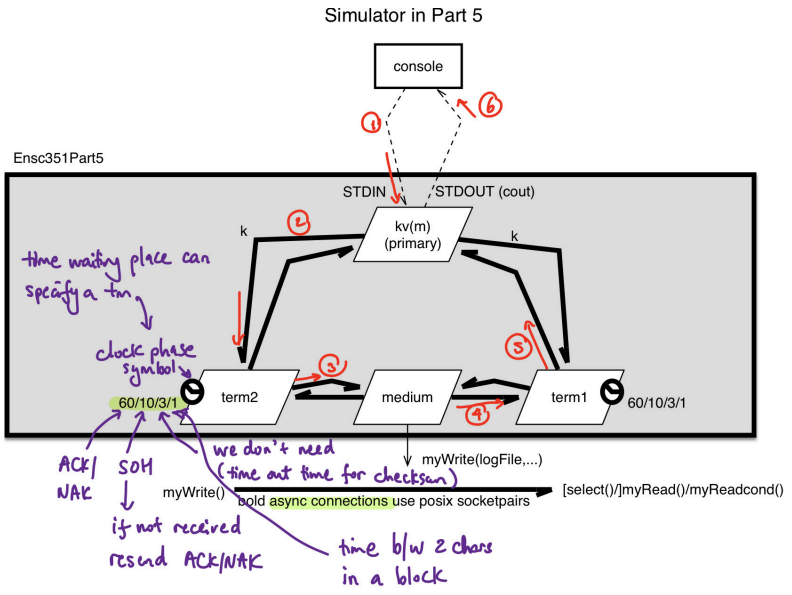
[w2] : sender written blk2
(bd2) : bad blk2
(f0) : good Blk1st blk0
[d1] : dump 1 glitch
AxN : ACK corrupted to NAK
r16 : resend blk16
{+A} : extra ACK — getNextBlk() calls purge()
<+35> : purge() to dump glitches
dumpglitches() used before CRC
{A+0} : glitches in the other direction.



- write to dataSkp[0], data will appear in buf[1] through buf[0]
- read from dataSkp[1], will read data from the same buf, here buf[1]


Sender_TopLevel
[Entry]
ctx.prepStatBlk(); ctx.errCnt=0;
ctx.KbCan = false; ctx.tm(TM_VL);
[Exit]
ctx.cans() should not be called immediately after the crc16 for a block has been written to the medium, or some CAN characters may be purged.
H
NON_CAN
[Entry]onEvent(SER)
[Exit] [((c==NAK || c=='C') & !ctx.KbCan]
/ctx.resendBlk();
ctx.errCnt++;
ctx.tm(TM_VL);
onEvent(SER)
[c==ACK & ctx.fileName & !ctx.KbCan]
/ctx.firstBlk= true;
ctx.tm(TM_VL);
ACKNAKSTAT
[Entry]
[Exit]
onEvent(SER)
[c==NAK & !ctx.KbCan]
/ctx.resendBlk(); ctx.errCnt++;
ctx.tm(TM_VL);
One
[Entry]
[Exit]
onEvent(SER)
[((c==ACK) & ctx.bytesRd & !ctx.KbCan]
/ctx.sendBlkPrepNext();
ctx.errCnt=0; ctx.tm(TM_VL);
ctx.firstBlk=false;
onEvent(SER)
[c=='C' & ctx.bytesRd & !ctx.KbCan]
/ctx.sendBlkPrepNext();
ctx.tm(TM_VL); ctx.errCnt=0;
ACKNAK
[Entry]
[Exit]
onEvent(SER)
[(c==NAK || (c=='C' & ctx.firstBlk)) & (ctx.errCnt < errB) & !ctx.KbCan]
/ctx.resendBlk();
ctx.errCnt++; ctx.tm(TM_VL);
onEvent(SER)
[c=='C' & ctx.firstBlk & ctx.errCnt < errB]
/ctx.sendByte(EOT);
++ctx.errCnt;
ctx.tm(TM_VL);
onEvent(SER)
[c == 'C' & !ctx.bytesRd & !ctx.KbCan]
/ctx.sendByte(EOT);
ctx.tm(TM_VL);
ctx.errCnt=0;
ctx.closeTransferredFile();
onEvent(SER)
[((c==ACK) & !ctx.bytesRd & !ctx.KbCan]
/ctx.sendByte(EOT);ctx.errCnt=0;
ctx.closeTransferredFile();
ctx.tm(TM_VL);
onEvent(SER)
[c=='C' & ctx.transferringFileD != -1]
/ctx.sendBlkPrepNext();
ctx.errCnt=0;
ctx.tm(TM_VL);
EOT1
[Entry]
[Exit]
onEvent(SER)
[c==NAK & !ctx.KbCan]
/ctx.sendByte(EOT);
ctx.errCnt=0;ctx.tm(TM_VL);
ctx.firstBlk=false;
EOTEOT
[Entry]
[Exit]
onEvent(SER)
[c == ACK & !ctx.KbCan]
/cout << "1st EOT ACK'd";
ctx.prepStatBlk(); ctx.tm(TM_VL);
onEvent(SER)
[c==ACK & !ctx.KbCan]
/ctx.prepStatBlk(); ctx.tm(TM_VL);
onEvent(SER)
[c==NAK & !ctx.KbCan & ctx.errCnt < errB]
/ctx.sendByte(EOT);
ctx.errCnt++;
ctx.tm(TM_VL);
StatC
[Entry]
[Exit]
onEvent(SER)
[c=='C' & ctx.fileName & ctx.transferringFileD == -1]
/ctx.cans();
ctx.result="OpenError";
onEvent(KB_C)
/if (ctx.transferringFileD == -1) ctx.result="KbCancelledOpenErr";
else {ctx.closeTransferredFile(); ctx.result="KbCancelledFromStatC";}
onEvent(KB_C) //1
[!ctx.KbCan]
/ctx.KbCan = true;
ctx.tmRed(TM_VL - TM_2CHAR);
onEvent(TM)
[!ctx.KbCan]
/ctx.tmPop();
onEvent(SER)
[c != CAN & !ctx.KbCan]
/ctx.tmPop();
CAN
[Entry]
[Exit]
onEvent(KB_C) //2
/ctx.KbCan=true;
onEvent(SER)
[c == CAN]
/ctx.tmPush(TM_CHAR);
onEvent(SER)
[c == CAN]
/ctx.closeTransferredFile();
ctx.clearCan();
ctx.result="RcvCancelled";
onEvent(SER)
[ctx.KbCan & (c==ACK || c==NAK || c=='C')]
/ctx.cans();
ctx.closeTransferredFile();
ctx.result="KbCancelled";
onEvent(SER)
[c==NAK & (ctx.errCnt >= errB)]
/ctx.cans();
ctx.closeTransferredFile();
ctx.result="ExcessiveNAKs";
onEvent(TM)
/ctx.cans();
ctx.closeTransferredFileD();
if (ctx.KbCan)
ctx.result="KbCancelled";
else
ctx.result="Timeout";
Unless a fast simulation has been chosen ...
TM_VL (Very Long timeout) gives 60 seconds
TM_SOH (normal timeout waiting for SOH) gives 10 seconds
TM_CHAR (inter-character timeout) gives 1 second
TM_2CHAR gives a period longer than the inter-character timeout of 1 sec.
Due to a bug in SmartState studio, in this StateChart, please interpret & as && -- a logical AND. Do not interpret it as a bitwise AND as you normally would.
YMODEM Sender Statechart
Copyright (c) 24 November 2021 Craig Scratchley
craig_scratchley AT alumni.sfu.ca
Real events are either
KB_C (cancel command received from keyboard, higher priority)
SER (serial character recieved, lower priority),
TM (timeout occurred)


Receiver_TopLevel
[Entry]
ctx.NCGbyte='C';
ctx.sendByte(ctx.NCGbyte);
ctx.errCnt=0; ctx.anotherFile=0;
ctx.transferringFileD = -1;
ctx.closeProb = -1;
ctx.tm(TM_SOH);
ctx.KbCan=false;
[Exit]
onEvent(SER)
[!ctx.KbCan & c!=CAN]
/ctx.purge();
onEvent(TM)
[ctx.errCnt < errB & !ctx.KbCan]
/ctx.sendByte(ctx.NCGbyte);
ctx.errCnt++;
ctx.tm(TM_SOH);
NON_CAN
[Entry]
[Exit]
onEvent(SER)
[c==EOT & !ctx.closeProb & ctx.errCnt < errB]
/ctx.sendByte(ACK);
ctx.sendByte(ctx.NCGbyte);
ctx.errCnt++; ctx.tm(TM_SOH);
onEvent(CONT)
[!ctx.closeProb]
/ctx.sendByte(ACK);
ctx.NCGbyte='C';
ctx.sendByte(ctx.NCGbyte);
ctx.errCnt=0;
ctx.tm(TM_SOH);
Due to a bug in SmartState studio, in this StateChart, please interpret & as && -- a logical AND. Do not interpret it as a bitwise AND as you normally would.
CondTransientEOT
[Entry]
POST("*",CONT);
[Exit]
FirstByteStat
[Entry]
[Exit]
onEvent(SER)
[!ctx.KbCan & c==SOH]
/ctx.getRestBlk();
ctx.NCGbyte=NAK;
if (!ctx.closeProb) {
ctx.errCnt=0;
ctx.closeProb = -1;
}
onEvent(CONT)
[!ctx.syncLoss & (ctx.errCnt < errB) & !ctx.goodBlk]
/ctx.sendByte(NAK);
ctx.errCnt++;
ctx.tm(TM_SOH);
onEvent(SER)
[c==EOT]
/ctx.closeTransferredFile();
onEvent(CONT)
[ctx.closeProb]
/ctx.cans();
ctx.result="CloseError";
DataCancelable
[Entry]
[Exit]
EOT
[Entry]
[Exit]
onEvent(SER)
[!ctx.KbCan & c == EOT]
/ctx.purge();
ctx.sendByte(NAK);
++ctx.errCnt;
ctx.NCGbyte=NAK;
ctx.tm(TM_SOH);
CondInTransientStat
[Entry]
POST("*",CONT);
[Exit]
onEvent(CONT)
[ctx.transferringFileD != -1]
/ctx.sendByte(ACK);
ctx.NCGbyte='C';
ctx.sendByte(
ctx.NCGbyte);
ctx.tm(TM_SOH);
onEvent(CONT)
[!ctx.syncLoss & (ctx.errCnt < errB) & ctx.goodBlk]
/ctx.checkForAnotherFile();
onEvent(SER)
[!ctx.KbCan & c==SOH]
/ctx.getRestBlk();
ctx.errCnt++;
FirstByteData
[Entry]
[Exit]
onEvent(TM)
[!ctx.syncLoss & (ctx.errCnt < errB)]
/if (ctx.goodBlk) {
ctx.sendByte(ACK);
if (ctx.anotherFile) ctx.sendByte('C');
}
else ctx.sendByte(NAK);
if (ctx.goodBlk1st)
ctx.writeChunk();
ctx.tm(TM_SOH);
onEvent(SER)
[c==CAN]
/ctx.tmPush(TM_2CHAR);
CAN
[Entry]
[Exit]
onEvent(SER)
[c != CAN & !ctx.KbCan]
/ctx.purge();
ctx.tmPop();
onEvent(TM)
[!ctx.KbCan]
/ctx.tmPop();
CondTransientCheck
[Entry]
POST("*",CONT);
[Exit]
AxN
onEvent(CONT)
[!ctx.anotherFile]
/ctx.sendByte(ACK);
ctx.tm(TM_SOH);
onEvent(CONT)
[ctx.anotherFile]
/ctx.openFileForTransfer();
onEvent(SER)
[!ctx.KbCan & c==SOH]
/ctx.getRestBlk();
if (ctx.goodBlk1st) {
ctx.NCGbyte=NAK;
ctx.errCnt=0;
ctx.anotherFile=0;
}
else ctx.errCnt++;
CondTransientData
[Entry]
ctx.tm(0);
[Exit]
onEvent(TM)
[ctx.syncLoss || ctx.errCnt >= errB]
/ctx.cans();
ctx.closeTransferredFile();
if (ctx.syncLoss)
ctx.result="LossOfSyncronization";
else
ctx.result="ExcessiveErrors";
onEvent(SER)
/ctx.purge();
onEvent(SER)
[c == CAN]
/ctx.closeTransferredFile();
ctx.clearCan();
ctx.result="SndCancelled";
AreWeDone
[Entry]
[Exit]
CondTransientOpen
[Entry]
POST("*",CONT);
[Exit]
onEvent(CONT)
[ctx.transferringFileD == -1]
/ctx.cans();
ctx.result="CreatError";
onEvent(KB_C)
/ctx.cans();
ctx.result="kbCancelled (immediate)";
onEvent(TM)
/ctx.result="Done";
The entry code:
POST("*",CONT);
in the grey transient states immediately posts a continue (CONT) event that immediately kicks the StateChart out of those states.
- Event SER is the event of a character being available from the Medium (simulating a SERial port)
- Event KB_C is a cancel command received from keyboard -- higher priority
- Event TM is a timeout event.
onEvent(CONT)
[ctx.syncLoss || ctx.errCnt >= errB]
/ctx.cans();
if (ctx.syncLoss)
ctx.result="LossOfSync at Stat Blk";
else
ctx.result="ExcessiveErrors at Stat";
onEvent(SER)
[c==EOT & !ctx.closeProb & ctx.errCnt >= errB]
/ctx.cans();
ctx.result="ExcessiveEOTs";
Complete Receiver Statechart
24 November 2021
For Evil Medium.
Copyright (c) 2021 Craig Scratchley
craig_scratchley AT alumni.sfu.ca
onEvent(KB_C)
/ctx.KbCan = true;
onEvent(SER)
/ctx.purge(); ctx.cans();
if (ctx.KbCan)
ctx.result = "KbCancelled (delayed)";
else
ctx.result = "ExcessiveErrors";
onEvent(TM)
/ctx.cans();
if (ctx.KbCan)
ctx.result="KbCancelled";
else
ctx.result="ExcessiveErrors";


Simulator in Part 5
console
Ensc351Part5
STDIN
STDOUT (cout)
kv(m)
(primary)
k
k
time waiting place can specify a tm
clock phase symbol
term2
medium
term1
60/10/3/1
60/10/3/1
ACK/ NAK
SOH
if not received resend ACK/NAK
we don't need (time out time for checksum)
time b/w 2 chars in a block
myWrite()
myWrite(logFile,...)
bold async connections use posix socketpairs
[select()/]myRead()/myReadcond()

ret -1 errno 32: broken pipe : myWrite (1)
ret -1 errno 104: "Reset" : myRead (1)
ret 0 myRead (1)
} called after myClose (0) (peer side)
ret -1 errno 9: bad file descriptor : myPOSIX (0) called after myClose (0)

myWrite (0) called before
myWrite (0) is not called before

myioFinal.cpp

```
/* myioFinal-prep.cpp -- Dec. 12 -- Copyright 2021 Craig Scratchley */
#define AF_LOCAL 1
#define SOCK_STREAM 1   } // open local socket
#include <condition_variable>
#include <mutex>
#include <shared_mutex>
#include <map>
#include <cstring>          // for memcpy ?
#include <atomic>
#include <iostream>
#include "posixThread.hpp" // you are not examined on details of this class

using namespace std;
using namespace std::chrono; // time based → used for time condition variable

/* Lock-free circular buffer.  This should be threadsafe if one thread is
reading
 * and another is writing. */
template<class T>
class CircBuf {
    /* 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. */
    static const unsigned size = 8 + 1; // capacity of 8 elements
    T buf[size];
    std::atomic<unsigned> read_pos, write_pos;
public:
    CircBuf() {
        read_pos = write_pos = 0;
    }

//  void get_write_pointers( T *pPointers[2], unsigned pSizes[2] ) {
//      const int wpos = write_pos.load( memory_order_relaxed );
//      const int rpos = read_pos.load( memory_order_acquire );
//        pPointers[0] = buf + wpos;
//        /* Subtract 1 below, to account for the element that we never fill.
*/
//      if( rpos <= wpos ) {
//          // The buffer looks like "eeeeDDDDeeee" or "eeeeeeeeeeee" (e =
empty, D = data)
//          pPointers[1] = buf;
//          if (rpos) {
//                pSizes[0] = size - wpos;
//                pSizes[1] = rpos - 1;
//          }
//          else {
//                pSizes[0] = size - wpos - 1;
//                pSizes[1] = rpos; // 0
//          }
//      } else /* if( rpos > wpos ) */ {
//          /* The buffer looks like "DDeeeeeeeeDD" (e = empty, D = data). */
//          pPointers[1] = nullptr; // could comment out
//          pSizes[0] = rpos - wpos - 1;
```

capacity ← differentiate when empty or full

has 1 unused element

write_pos & read_pos point at same spot

```
//          pSizes[1] = 0;
//      }
//  }
//
//  void get_read_pointers( T *pPointers[2], unsigned pSizes[2] ) {
//      const int rpos = read_pos.load( memory_order_relaxed );
//      const int wpos = write_pos.load( memory_order_acquire );
//        pPointers[0] = buf + rpos;
//      if( rpos <= wpos ) {
//          // The buffer looks like "eeeeDDDDeeee" or "eeeeeeeeeeee" (e =
empty, D = data)
//          pPointers[1] = nullptr; // could comment out
//          pSizes[0] = wpos - rpos;
//          pSizes[1] = 0;
//      } else if( rpos > wpos ) {
//          /* The buffer looks like "DDeeeeeeeeDD" (e = empty, D = data). */
//          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 ) {// need to know where to the
        T *p[2];                                                 so it can write
        unsigned sizes[2];

        p[0] = buf + write_pos.load( memory_order_relaxed );
        const unsigned rpos = read_pos.load( memory_order_acquire );
        /* Subtract 1 below, to account for the element that we never fill.
*/
        if( rpos <= write_pos.load( memory_order_relaxed ) ) {
            // The buffer looks like "eeeeDDDDeeee" or "eeeeeeeeeeee" (e =
empty, D = data)
            p[1] = buf;
            if (rpos) {
                sizes[0] = size - write_pos.load( memory_order_relaxed );
                sizes[1] = rpos - 1;
            }
            else {
                sizes[0] = size - write_pos.load( memory_order_relaxed ) - 1;
                sizes[1] = rpos; // 0
            }
        } else /* if( rpos > wpos ) */ {
            /* The buffer looks like "DDeeeeeeeeDD" (e = empty, D = data). */
            p[1] = nullptr; // could comment out
            sizes[0] = rpos - write_pos.load( memory_order_relaxed ) - 1;
            sizes[1] = 0;
        }
```

SFENCE - store fence.

myioFinal.cpp

```
        buffer_size = min( buffer_size,
sizes[0]+sizes[1]);//max_write_sz=sizes[0]+sizes[1]
        const int from_first = min( buffer_size, sizes[0] );
        memcpy( p[0], buffer, from_first*sizeof(T) );
        if( buffer_size > sizes[0] )
            memcpy(p[1], buffer+from_first, max(buffer_size-sizes[0],
0u)*sizeof(T) ); // if necessary
update position ← write_pos.store( (write_pos.load( memory_order_relaxed ) +
buffer_size) % size,
                        memory_order_release ); // copy first then update position
        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 ) { // need to know where is the write_pos
        T *p[2];                                          so it can read
        unsigned sizes[2];

        p[0] = buf + read_pos.load( memory_order_relaxed );
        const unsigned wpos = write_pos.load( memory_order_acquire );
        if( read_pos.load( memory_order_relaxed ) <= wpos ) {
            // The buffer looks like "eeeeDDDDeeee" or "eeeeeeeeeeee" (e =
empty, D = data)
            p[1] = nullptr; // could comment out
            sizes[0] = wpos - read_pos.load( memory_order_relaxed );
            sizes[1] = 0;
        } else /* if( rpos > wpos ) */ {
            /* The buffer looks like "DDeeeeeeeeDD" (e = empty, D = data). */
            p[1] = buf;
            sizes[0] = size - read_pos.load( memory_order_relaxed );
            sizes[1] = wpos;
        }

        buffer_size = min( buffer_size,
sizes[0]+sizes[1]);//max_read_sz=sizes[0]+sizes[1];
        const int from_first = min( buffer_size, sizes[0] );
        memcpy( buffer, p[0], from_first*sizeof(T) ); // copy from buf to where the read()
        if( buffer_size > sizes[0] )                     wants it to go
            memcpy( buffer+from_first, p[1], max(buffer_size-sizes[0],
0u)*sizeof(T) );
        read_pos.store( (read_pos.load( memory_order_relaxed ) + buffer_size)
% size,
                        memory_order_release ); // update the read_pos. Interact w/
        return buffer_size;                        write_pos.load in the writing()
    } /* Original source Copyright (c) 2004 Glenn Maynard. */   release - acquire
};  /* See Craig for more details and code before improvements. */

namespace{ //Unnamed (anonymous) namespace

    class socketInfoClass; // similar to Part3_sol
    typedef shared_ptr<socketInfoClass> socketInfoClassSp;
    map<int, socketInfoClassSp> desInfoMap {
//        {-2, make_shared<socketInfoClass>(-1)}, // marker for a descriptor
whose pair is closed
```

// if we delete all relaxed, release, acquire, it will back to default which is sequential order, → it is correct but slow

```
    //      {0, nullptr}, // init for stdin, stdout, stderr
    //      {1, nullptr},
    //      {2, nullptr}
    };

    //  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 shared 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;
        unsigned maxTotalCanRead = 0;
        condition_variable cvDrain;
        condition_variable cvRead;
        CircBuf<char> circBuffer;
        bool connectionReset = false;
        mutex socketInfoMutex;
    public:
        int pair;   // Cannot be private because myWrite and myTcdrain using
it.
                    // -1 when descriptor closed, -2 when paired descriptor
is closed
    socketInfoClass(unsigned pairInit)
    :pair(pairInit) { }

     // If necessary, make the calling thread wait for a reading thread to
drain the data
    int draining(shared_lock<shared_mutex> &desInfoLk) { // operating on
object for paired descriptor of original des
        unique_lock<mutex> socketLk(socketInfoMutex);
        desInfoLk.unlock();

        //  once the reader decides the drainer should wakeup, it should
wakeup
        if (pair >= 0 && totalWritten > maxTotalCanRead)
            cvDrain.wait(socketLk); // what about spurious wakeup?
        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);
        int written = circBuffer.write((const char*) buf, nbyte);
        if (written > 0) {
            totalWritten += written;
            cvRead.notify_one();
        }
```

// block P thread
→ switch to "50" thread
→ go to line 387

```
        return written;
    }

    int reading2(int des, void * buf, int n, int min, int time)
    {                                                          // wait for that amount of time to get min bytes
        int bytesRead;
        unique_lock<mutex> socketLk(socketInfoMutex);

        // would not have got this far if pair == -1
        if ((!maxTotalCanRead && totalWritten >= (unsigned) min) || pair == -
2) {
            if (pair == -2)
                if (connectionReset) {
                    errno = ECONNRESET;
                    bytesRead = -1;
                    connectionReset = false;
                }
                else
                    bytesRead = 0;
            else {
                bytesRead = circBuffer.read((char *) buf, n);
                if (bytesRead > 0) {
                    totalWritten -= bytesRead;
                    if (totalWritten <= maxTotalCanRead /* && pair >= 0 */)
                        cvDrain.notify_all();
                }
            }
        }
        else {
            maxTotalCanRead +=n;
            cvDrain.notify_all(); // totalWritten must be less than min
            if (time)          // return a status -> so you know if it's timeout
                cvRead.wait_for(socketLk, duration<int, deci>{time}, [this,
min] {  // from "so" thread, switch back to "p" thread
                    return totalWritten >= (unsigned) min || pair < 0;});
            else
                cvRead.wait(socketLk, [this, min] {
                    return totalWritten >= (unsigned) min || pair < 0;});
            if (pair == -1) { // shouldn't normally happen
                errno = EBADF; // check errno value
                return -1;
            }
            bytesRead = circBuffer.read((char *) buf, n);
            if (connectionReset && bytesRead == 0) {
                errno = ECONNRESET;
                bytesRead = -1;
                connectionReset = false;
            }
            if (bytesRead != -1)
                totalWritten -= bytesRead;
            maxTotalCanRead -= n;
            if (totalWritten > 0 || pair < 0) // || pair == -2
                cvRead.notify_one(); // can this affect errno?
        }
        return bytesRead;
    } // .reading2()
```

```
    int 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]);
            unique_lock<mutex> socketLk(socketInfoMutex, defer_lock);
            unique_lock<mutex> condPairlk(des_pair->socketInfoMutex,
defer_lock);
            lock(condPairlk, socketLk);
            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();
            }
        }
        // unlock condPairlk here or even above a little bit.
        return 0;
    } // .closing()
    }; // socketInfoClass

    // get shared pointer for des
    socketInfoClassSp getDesInfoP(int des) { // same as Part3_sol
        auto iter = desInfoMap.find(des);
        if (iter == desInfoMap.end())
            return nullptr; // des not in use
        else
            return iter->second; // return the shared pointer
    }
} // unnamed namespace
```

```
// see
https://developer.blackberry.com/native/reference/core/com.qnx.doc.neutrino.l
ib_ref
int myReadcond2(int des, void * buf, int n, int min, int time) {
    shared_lock<shared_mutex> desInfoLk(mapMutex);
    socketInfoClassSp desInfoP = getDesInfoP(des);
    desInfoLk.unlock();
    if (!desInfoP) {
        // not an open "socket" [created with mySocketpair()]
        errno = EBADF; return -1;
    }
    return desInfoP->reading2(des, buf, n, min, time);
}

ssize_t myWrite(int des, const void* buf, size_t nbyte) {
    shared_lock<shared_mutex> desInfoLk(mapMutex);
    socketInfoClassSp desInfoP = getDesInfoP(des);
    if(desInfoP) {
        if (desInfoP->pair >= 0) {
            // locking mapMutex above makes sure that desinfoP->pair is not
closed here
            return desInfoMap[desInfoP->pair]->writing(des, buf, nbyte);
        }
        else {
            errno = EPIPE; return -1;
        }
    }
    errno = EBADF; return -1;
}

int myTcdrain(int des) {
    shared_lock<shared_mutex> desInfoLk(mapMutex);
    socketInfoClassSp desInfoP = getDesInfoP(des);
    if(desInfoP) {
        if (desInfoP->pair == -2)
            return 0; // paired descriptor is closed.
        else { // pair == -1 won't be in *desInfoP now
            // locking mapMutex above makes sure that desinfoP->pair is not
closed here
            return desInfoMap[desInfoP->pair]->draining(desInfoLk);
        }
    }
    errno = EBADF; return -1;
}

int mySocketpair(int domain, int type, int protocol, int des[2]) {
    lock_guard<shared_mutex> desInfoLk(mapMutex);
    des[0] = 3; // This is the only function that has materially changed ...
    des[1] = 4; //  ... from the code distributed before the exam.
    desInfoMap[des[0]] = make_shared<socketInfoClass>(des[1]);
    desInfoMap[des[1]] = make_shared<socketInfoClass>(des[0]);
    return 0;
}
int myClose(int des) {
    int retVal = 1;
    lock_guard<shared_mutex> desInfoLk(mapMutex);
```

// cross from one thread to paired thread.
i.e. write to des3 but will buffer to des4 where it is read from

// hard-coded the socket des w/o lib func

} // only creating 1 socketpair (socket 3 & 4), we'll use circular buff

myioFinal.cpp

```
cout << "The next line will timeout in 5 or so seconds (50 deciseconds)" << endl;

    auto iter = desInfoMap.find(des);
    if (iter != desInfoMap.end()) { // if in the map
        if (iter->second) // if shared pointer exists
            retVal = iter->second->closing(des);
        desInfoMap.erase(des);
    }
    if (retVal == 1) { // if not-in-use
        errno = EBADF;
        retVal = -1;
    }
    return retVal;
}

const int BSize = 20;

char    B[BSize]; // initially zeroed (filled with NUL characters)
char    B2[BSize]; // initially zeroed (filled with NUL characters)
static int daSktPr[2]; // Descriptor Array for Socket Pair

void coutThreadFunc(void) {
    int RetVal;
    CircBuf<char> buffer;
    //buffer.reserve(5); // note constant of 5
    buffer.write("123456789", 10); // don't forget NUL termination character
    RetVal = buffer.read(B, BSize);
    cout << "Output Line 1 - RetVal: " << RetVal << " B: " << B << endl;
    RetVal = myReadcond2(daSktPr[1], B, BSize, 10, 50);
    cout << "Output Line 2 - RetVal: " << RetVal << " B: " << B << endl;
    // myWrite(daSktPr[1], "wxyz", 5); // don't forget NUL termination char
    // myWrite(daSktPr[0], "ab", 3); // don't forget NUL termination character
    RetVal = myReadcond2(daSktPr[1], B, BSize, 5, 0);
    cout << "Output Line 3 - RetVal: " << RetVal << endl;
    if (RetVal == -1)
        cout << " Error:  " << strerror(errno) << endl;
    else if (retVal > 0)
        cout << " B: " << B;
    // RetVal = myReadcond2(daSktPr[1], B, BSize, 5, 0);
    cout << "Output Line 4 - RetVal: " << RetVal;
    if (RetVal == -1)
        cout << " Error:  " << strerror(errno);
    else if (RetVal > 0)
        cout << " B: " << B;
    cout << endl;
    RetVal = myWrite(daSktPr[0], "123456789", 10); // don't forget NUL
    termination char // there is no myReadcond2(daSktPr[1])
    cout << "Output Line 5 - RetVal: " << RetVal;
    if (RetVal == -1)
        cout << " Error:  " << strerror(errno) << endl;
    else cout << endl;
}

int main() { // debug launch configuration ensures gdb runs at FIFO priority
98 ...
    cpu_set_t cpu_set;
```

```
RetVal = myReadcond2(daSktPr[1], B, BSize, 5, 0);
cout << "Output Line 4b- RetVal: " << RetVal;
if (RetVal == -1)
    cout << " Error:  " << strerror(errno);
else if (RetVal > 0)
    cout << "  B: " << B;
cout << endl;
```

```
Output Line 1 - RetVal: 8  B: 12345678
The next line will timeout in 5 or so seconds (50 deciseconds)
Output Line 2 - RetVal: 7  B: abc123
Output Line 3 - RetVal: 3  B: ab
Output Line 4 - RetVal: -1 Error:  Connection reset by peer
Output Line 4b- RetVal: 0
Output Line 5 - RetVal: -1 Error:  Bad file descriptor
```

min byte = 10, but buf-size=8, so it will cause tm, and printout whatever it has.

size of buf = 8

// 8, 12345678

// 7, abc 123 ~~4~~

//5s - timeout

// circular Buff

read [0]

read [1]

// release myReadcond2

// is not shown b/c no myReadcond2(d[2])

// 3, ab ~~123 8~~

cout << endl;

no write [0]

// 0, ab

B isn't shown

read [1]

// there is no myReadcond2(daSktPr[1])

// -1, bad file descriptor

```
    CPU_SET(0, &cpu_set);
    const char* threadName = "Pri";
    pthread_setname_np(pthread_self(), threadName);
    sched_setaffinity(0, sizeof(cpu_set), &cpu_set); // set processor
affinity    // tells primary thread to restrict to core0

    // 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); // allocate memory
    free(ptr); // free right away

    try {        // ... realtime policy.
        sched_param sch;
        sch.__sched_priority = 60; // primary thread priority
        pthreadSupport::setSchedParam(SCHED_FIFO, sch); //SCHED_FIFO == 1,
SCHED_RR == 2
        mySocketpair(AF_LOCAL, SOCK_STREAM, 0, daSktPr);
        myWrite(daSktPr[0], "abc", 3); // we don't want to write NULL in the socket
        pthreadSupport::posixThread coutThread(SCHED_FIFO, 50, // round-robin
coutThreadFunc); // lower    // keep running until there is a thread w/ higher priority
        myTcdrain(daSktPr[0]); // block "50" thread and back to "P" thread

        myWrite(daSktPr[0], "123", 4); // don't forget NUL termination
character
        myReadcond2(daSktPr[0], B2,  4  , 4 , 0); // never tm, wait when it does a job
                                     max   min
        pthreadSupport::setSchedPrio(40); // lower primary thread priority

        myClose(daSktPr[0]);

        coutThread.join();
        return 0;
    }
    catch (system_error& error) {
        cout << "Error: " << error.code() << " - " << error.what() << '\n';
    }
    catch (...) { throw; }
}
```

// call on same des

// new thread exists

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

- atomic variable – code faster if does not have sequential order
- via Connection → connected to gdb server

```
Output Line 1 - RetVal: 8   B: 12345678
The next line will timeout in 5 or so seconds (50 deciseconds)
Output Line 2 - RetVal: 8   B: abc12321
Output Line 2b - RetVal: 4   B2: wxyz
Output Line 3 - RetVal: 3   B: ab
Output Line 3b- RetVal: -1 Error:  Connection reset by peer
Output Line 3c- RetVal: 0   B: ab
Output Line 4 - RetVal: 0   B: ab
Output Line 4b- RetVal: 0   B: ab
Output Line 5 - RetVal: -1 Error:  Bad file descriptor
Output Line 5b - RetVal: 0   B: ab
```

```
//============================================================================
========
// Name        : posixThread.hpp
// Author(s)   :
//             : Craig Scratchley
//             : Eton Kan
// Version     : November 2019
// Copyright   : Copyright 2019, Craig Scratchley and Eton Kan
// Description : A derived class that allows creating threads with policy and
priority.
//============================================================================
========

#ifndef PTHREAD_POSIXTHREAD
#define PTHREAD_POSIXTHREAD

#include <iostream>
#include <cstring>
#include <thread>
#include <system_error>
#include <sched.h>
#include <pthread.h>
#include <semaphore.h>

namespace pthreadSupport
{
    int getSchedParam(int *policy, sched_param *sch)
    ;
    int getSchedParam(pthread_t th, int *policy, sched_param *sch)
    ;
    int setSchedParam(int policy, sched_param sch)
    ;
    int setSchedParam(pthread_t th, int policy, sched_param sch)
    ;
    int setSchedPrio(int priority)
    ;
    int setSchedPrio(pthread_t th, int priority)
    ;
    int get_priority_max(int policy)
    ;
    int get_priority_min(int policy)
    ;

    //Set new thread's priority when it is higher than creating thread's
priority
    template<typename _Callable, typename... _Args>
    void ctorHelperHigher(int policy, sched_param sch, sem_t &sem4,
_Callable&& __f, _Args&&... __args){
        if (pthread_setschedparam(pthread_self(), policy, &sch)){
            sem_post(&sem4);
            throw std::system_error(errno, std::system_category());
            return;
        }
        sem_post(&sem4);
        __f(std::forward<_Args>(__args)...);
    }
```

```
    //Set new thread's priority when it is lower than creating thread's
priority
    template<typename _Callable, typename... _Args>
    void ctorHelperLower(int policy, sched_param sch, sem_t &sem4,
_Callable&& __f, _Args&&... __args){
        sem_post(&sem4);
        if (!setSchedParam(policy, sch)) {
            sched_yield(); // is this needed?
            __f(std::forward<_Args>(__args)...);
        }
    }

    class posixThread : public std::thread
    {
    private:
        //Helper function for setting thread's policy and schedule parameter
(including's priority (lower or higher))
        template<typename _Callable, typename... _Args>
        thread beforeThread(int policy, sched_param sch, _Callable&& __f,
_Args&&... __args) {
            sem_t sem4;
            thread th;
            sched_param curSch;
            int curPolicy = -1;

            getSchedParam(&curPolicy, &curSch); //Getting creating thread
info

            if(policy == -1){
                policy = curPolicy;
            }

            // do we need try/catch block here?  *wcs*
            sem_init(&sem4,0,0);
            if(curSch.sched_priority <= sch.sched_priority){
                th = std::thread([&]{ctorHelperHigher(policy, sch, sem4,
std::forward<_Callable>(__f), std::forward<_Args>(__args)...);});
                sched_yield(); // this may not be needed.
            }
            else{
                // don't need sem4 stuff in this case probably.
                th = std::thread([&]{ctorHelperLower(policy, sch, sem4,
std::forward<_Callable>(__f), std::forward<_Args>(__args)...);});
            }
            sem_wait(&sem4);
            sem_destroy(&sem4);
            return th;
        }

//      //Flag variable is used as a place holder so the code can distinguish
if it is setting policy or priority
//      template<typename _Callable, typename... _Args>
//      thread beforeThread(int policy, ??? policy_flag, _Callable&& __f,
_Args&&... __args) {
//  [...]
//      }
```

```
        template<typename _Callable, typename... _Args>
        thread beforeThread(int policy, int prio, _Callable&& __f, _Args&&...
__args) {
            sched_param curSch;
            int curPolicy = -1;

            getSchedParam(&curPolicy, &curSch); //Getting creating thread
info
            curSch.sched_priority = prio;
            return beforeThread(policy, curSch, std::forward<_Callable>(__f),
std::forward<_Args>(__args)...);
        }

    public:
        using thread::thread;

        //Start thread with custom priority and creating thread's policy
        template<typename _Callable, typename... _Args>
        explicit
        posixThread(int prio, _Callable&& __f, _Args&&... __args) :
            std::thread(beforeThread(-1, prio, std::forward<_Callable>(__f),
std::forward<_Args>(__args)...)){}

        //Start thread with custom policy and creating thread's parameters
(including priority)
        // ??

        //Start thread with custom policy and custom priority
        template<typename _Callable, typename... _Args>
        explicit
        posixThread(int policy, int prio, _Callable&& __f, _Args&&... __args)
:
            std::thread(beforeThread(policy, prio,
std::forward<_Callable>(__f), std::forward<_Args>(__args)...)){}

        //Start thread with custom parameters and creating thread's policy
        template<typename _Callable, typename... _Args>
        explicit
        posixThread(sched_param sch, _Callable&& __f, _Args&&... __args) :
            std::thread(beforeThread(-1, sch, std::forward<_Callable>(__f),
std::forward<_Args>(__args)...)){}

        //Start thread with custom parameter (including priority) and custom
policy
        template<typename _Callable, typename... _Args>
        explicit
        posixThread(int policy, sched_param sch, _Callable&& __f, _Args&&...
__args) :
            std::thread(beforeThread(policy, sch,
std::forward<_Callable>(__f), std::forward<_Args>(__args)...)){}

        void getSchedandPolicy(int *policy, sched_param *sch){
            getSchedParam(this->native_handle(), policy, sch);
        }

        int getPolicy(){
            sched_param sch;
```

```
            int policy = -1;
            getSchedParam(this->native_handle(), &policy, &sch);
            return policy;
        }

        sched_param getParam(){
            sched_param sch;
            int policy = -1;
            getSchedParam(this->native_handle(), &policy, &sch);
            return sch;
        }

        int getPriority(){
            sched_param sch;
            int policy = -1;
            getSchedParam(this->native_handle(), &policy, &sch);
            return sch.sched_priority;
        }

        int setPriority(int prio){
            return setSchedPrio(this->native_handle(), prio);
        }

        //Set schedule parameters and use the existing policy of this thread
        int setParam(sched_param sch){
            sched_param curSch;
            int curPolicy = -1;
            getSchedParam(this->native_handle(), &curPolicy, &curSch);
            return setSchedParam(this->native_handle(), curPolicy, sch);
        }

        int setParamAndPolicy(int policy, sched_param sch){
            return setSchedParam(this->native_handle(), policy, sch);
        }
    };
}

#endif
```

```
//============================================================================
========
// Name        : posixThread.cpp
// Author(s)   :
//             : Craig Scratchley
//             : Eton Kan
// Version     : November 2019
// Copyright   : Copyright (C) 2019, Craig Scratchley and Eton Kan
// Description : Set schedule parameters, priority and policy for
current/given thread
//============================================================================
========

#include <sched.h>
#include <pthread.h>
#include <system_error>

namespace pthreadSupport
{
    //Overloaded function: get policy and sched_param (including priority)
for the current thread
    int getSchedParam(int *policy, sched_param *sch)
    {
        if(pthread_getschedparam(pthread_self(), policy, sch)) {
            throw std::system_error(errno, std::system_category());
            return -1;
        }
        return 0;
    }

    //Overloaded function: get policy and sched_param (including priority)
for the given thread
    int getSchedParam(pthread_t th, int *policy, sched_param *sch)
    {
        if(pthread_getschedparam(th, policy, sch)) {
            throw std::system_error(errno, std::system_category());
            return -1;
        }
        return 0;
    }

    //Overloaded function: set sched_param (including priority) and policy
for the current thread
    int setSchedParam(int policy, sched_param sch)
    {
        if(pthread_setschedparam(pthread_self(), policy, &sch)) {
            throw std::system_error(errno, std::system_category());
            return -1;
        }
        return 0;
    }

    //Overloaded function: set sched_param (including priority) and policy
for the given thread
    int setSchedParam(pthread_t th, int policy, sched_param sch)
    {
        if(pthread_setschedparam(th, policy, &sch)) {
```

```
            throw std::system_error(errno, std::system_category());
            return -1;
        }
        return 0;
    }

    //Overloaded function: set priority for the current thread
    int setSchedPrio(int priority)
    {
        if(pthread_setschedprio(pthread_self(), priority)) {
            throw std::system_error(errno, std::system_category());
            return -1;
        }
        return 0;
    }

    //Overloaded function: set priority for the given thread
    int setSchedPrio(pthread_t th, int priority)
    {
        if(pthread_setschedprio(th, priority)) {
            throw std::system_error(errno, std::system_category());
            return -1;
        }
        return 0;
    }

    int get_priority_max(int policy)
    {
        int max = sched_get_priority_max(policy);
        if(max == -1){
            throw std::system_error(errno, std::system_category());
            return -1;
        }
        return max;
    }

    int get_priority_min(int policy)
    {
        int min = sched_get_priority_min(policy);
        if(min == -1){
            throw std::system_error(errno, std::system_category());
            return -1;
        }
        return min;
    }
}
```