```
;/* ExRecLock1.c - Execute me to compile me with Lattice 5.10b
LC -b1 -cfistq -v -y -j73 ExRecLock1.c
Blink FROM LIB:c.o,ExRecLock1.o TO ExRecLock1 LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit ;*/
/* This is a simple example of using record locking to create an exclusive record    */
/* lock on a file, and writing to that record.  The example ExRecLock2 is almost     */
/* exactly the same as this example, except ExRecLock2 uses the record lock directly */
/* after ExRecLock1's record.  If you try to run ExRecLock1 (or ExRecLock2) while    */
/* another instance of ExRecLock1 (or ExRecLock2) is running, the second record lock */
/* attempt will fail.                                                                */

#include <clib/dos_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }                    /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }
#endif

#define RECORDSIZE   12
#define RECORDOFFSET 0

extern struct Library *DOSBase;

UBYTE *vers = "\0$VER: ExRecLock1 37.2";
UBYTE *string = "ExRecLock1\n";                           /* This string will be the */
                                                          /* contents of the record. */
void main(void)
{
    BPTR fh;

    if (DOSBase->lib_Version >= 37)   /* Record locking was introduced in Release 2, */
    {                                 /* but the standard startup code will open any */
                                      /* version of dos.library, so we have to ex-   */
                                      /* plicitly check the version number of DOS.   */

        if (fh = Open("t:testRLock", MODE_READWRITE))     /* Open the file, creating */
        {                                                 /*        it if necessary. */
            if (DOSTRUE == LockRecord(fh,           /* Lock the record as exclusive, */
                                      RECORDOFFSET, /* and do not wait if it is not  */
                                      RECORDSIZE,   /* available immediately.        */
                                      REC_EXCLUSIVE_IMMED, 0))
            {
                LONG error = RECORDOFFSET;

                                     /* If the record is beyond the end of the file, */
                if (Seek(fh, 0, OFFSET_END) < RECORDOFFSET)    /* lengthen the file. */
                    error = SetFileSize(fh, RECORDOFFSET, OFFSET_BEGINNING);

                if (error == RECORDOFFSET)    /* If there was no error with the file */
                {                             /*                 file size, continue. */
                    if (Seek(fh, RECORDOFFSET, OFFSET_BEGINNING) < 0)
                        PrintFault(IoErr(), "Seek() error");

                    if (Write(fh, string, RECORDSIZE) < 0)
                        PrintFault(IoErr(), "Write() error");
                    else
                        PutStr("Write successful, ");
                }
                PutStr("Waiting 10 seconds...\n");
                TimeDelay(UNIT_VBLANK, 10, 0);       /* Amiga.lib function that puts */
                          /* a task to sleep for a given amount of time.  This waits */
                          /* 10 seconds before unlocking the record to give the user */
                          /* a chance to start a second copy of this example.        */
                UnLockRecord(fh, RECORDOFFSET, RECORDSIZE);
            }
            else PrintFault(IoErr(), "Record Lock Failed");
            Close(fh);
        }
        else PrintFault(IoErr(), "Open Failed");
    }
    else PutStr("Need dos.library V37 or greater.\n");
}
```

```
;/* ExRecLock2.c - Execute me to compile me with Lattice 5.10b
LC -b1 -cfistq -v -y -j73 ExRecLock2.c
Blink FROM LIB:c.o,ExRecLock2.o TO ExRecLock2 LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit ;*/
/* This is a simple example of using record locking to create an exclusive record    */
/* lock on a file, and writing to that record.  The example ExRecLock1 is almost     */
/* exactly the same as this example, except ExRecLock1 uses the record lock directly */
/* before ExRecLock2's record.  If you try to run ExRecLock2 (or ExRecLock1) while   */
/* another instance of ExRecLock2 (or ExRecLock1) is running, the second record lock */
/* attempt will fail.                                                                */

#include <clib/dos_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }                    /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }
#endif

#define RECORDSIZE   12
#define RECORDOFFSET 12

extern struct Library *DOSBase;

UBYTE *vers = "\0$VER: ExRecLock2 37.2";
UBYTE *string = "ExRecLock2\n";                           /* This string will be the */
                                                          /* contents of the record. */
void main(void)
{
    BPTR fh;

    if (DOSBase->lib_Version >= 37)   /* Record locking was introduced in Release 2, */
    {                                 /* but the standard startup code will open any */
                                      /* version of dos.library, so we have to ex-   */
                                      /* plicitly check the version number of DOS.   */

        if (fh = Open("t:testRLock", MODE_READWRITE))     /* Open the file, creating */
        {                                                 /*        it if necessary. */
            if (DOSTRUE == LockRecord(fh,           /* Lock the record as exclusive, */
                                      RECORDOFFSET, /* and do not wait if it is not  */
                                      RECORDSIZE,   /* available immediately.        */
                                      REC_EXCLUSIVE_IMMED, 0))
            {
                LONG error = RECORDOFFSET;

                                     /* If the record is beyond the end of the file, */
                if (Seek(fh, 0, OFFSET_END) < RECORDOFFSET)    /* lengthen the file. */
                    error = SetFileSize(fh, RECORDOFFSET, OFFSET_BEGINNING);

                if (error == RECORDOFFSET)    /* If there was no error with the file */
                {                             /*                 file size, continue. */
                    if (Seek(fh, RECORDOFFSET, OFFSET_BEGINNING) < 0)
                        PrintFault(IoErr(), "Seek() error");

                    if (Write(fh, string, RECORDSIZE) < 0)
                        PrintFault(IoErr(), "Write() error");
                    else
                        PutStr("Write successful, ");
                }
                PutStr("Waiting 10 seconds...\n");
                TimeDelay(UNIT_VBLANK, 10, 0);       /* Amiga.lib function that puts */
                          /* a task to sleep for a given amount of time.  This waits */
                          /* 10 seconds before unlocking the record to give the user */
                          /* a chance to start a second copy of this example.        */
                UnLockRecord(fh, RECORDOFFSET, RECORDSIZE);
            }
            else PrintFault(IoErr(), "Record Lock Failed");
            Close(fh);
        }
        else PrintFault(IoErr(), "Open Failed");
    }
    else PutStr("Need dos.library V37 or greater.\n");
}
```

```
; /* LockRecord.c - Execute me to compile me with SAS C 5.10b
lc -cfis -v -d0 -b0 -j73 LockRecord.c
blink FROM LockRecord.o to LockRecord LIB lib:amiga.lib
quit
*
* AmigaMail LockRecord()/UnLockRecord() example.
*/
#include <exec/memory.h>
#include <exec/lists.h>
#include <dos/dosextens.h>
#include <dos/rdargs.h>
#include <dos/record.h>
#include <utility/tagitem.h>

#include <clib/alib_protos.h>
#include <clib/dos_protos.h>
#include <clib/exec_protos.h>
#include <clib/utility_protos.h>

void GetCommandLine(BPTR, struct RDArgs *rdargs, UBYTE *cmdbuffer);
void DoLockRecord(BPTR, struct RDArgs *rdargs);
void DoUnLockRecord(BPTR fh, struct RDArgs *rdargs);
void ListRecordLocks(void);
struct LockNode *FindRecordLock(ULONG offset, ULONG length);

/* List and node structures to keep track of record locks */
struct LockNode {
    struct LockNode *ln_Succ;
    struct LockNode *ln_Pred;
    ULONG ln_Counter;
    ULONG ln_Offset;
    ULONG ln_Length;
    ULONG ln_Mode;
};

struct LockList {
    struct LockNode *lh_Head;
    struct LockNode *lh_Tail;
    struct LockNode *lh_TailPred;
    ULONG lh_Counter;
};

/* Pseudo data file */
#define TESTFILE "t:locktest"

#define LOCK_TEMPLATE "OFFSET/K/N,LENGTH/K/N,EXCLUSIVE/S,IMMEDIATE/S,TIMEOUT/K/N"
#define UNLOCK_TEMPLATE "OFFSET/K/N,LENGTH/K/N"

#define OFFSET_POS      0
#define LENGTH_POS      1
#define EXCLUSIVE_POS   2
#define IMMEDIATE_POS   3
#define TIMEOUT_POS     4

struct Library *SysBase;
struct DosLibrary *DOSBase;
struct Library *UtilityBase;

struct LockList *locklist;

LONG main(void)
{
    BPTR fh;
    struct RDArgs *rdargs;
    struct CSource *csource;
    UBYTE *cmdbuffer;
    struct LockNode *lnode, *nnode;
    LONG error = RETURN_OK;

    SysBase = (*((struct Library **) 4));

    /* Fails silently if < 37 */
    if (DOSBase = (struct DosLibrary *)OpenLibrary("dos.library", 37))
    {
        UtilityBase = DOSBase->dl_UtilityBase;

        if (locklist = AllocMem(sizeof(struct LockList), MEMF_CLEAR))
        {
            NewList((struct List *)locklist);

            /* Allocate RDArgs structure to parse command lines */
            if (rdargs = AllocDosObject(DOS_RDARGS, TAG_END))
            {
                csource = &rdargs->RDA_Source;

                /* Get buffer to read command lines in */
                if (csource->CS_Buffer = AllocMem(512, MEMF_CLEAR))
                {
                    csource->CS_Length = 512;
                    csource->CS_CurChr = 0;

                    /* Buffer to isolate command keyword */
                    if (cmdbuffer = AllocMem(80, MEMF_CLEAR))
                    {

                        /* Open a testfile, create it if necessary */
                        if (fh = Open(TESTFILE, MODE_READWRITE))
                        {

                            /* Process command lines */
                            GetCommandLine(fh, rdargs, cmdbuffer);

                            /* Try to get rid of outstanding record locks */
                            lnode = locklist->lh_Head;
                            while (nnode = lnode->ln_Succ)
                            {

                                /* Try to unlock pending locks */
                                if ((UnLockRecord(fh,
                                                  lnode->ln_Offset,
                                                  lnode->ln_Length)) == DOSFALSE)
                                {
                   Printf("Error unlocking record %ld with offset %ld length %ld\n",
                                         lnode->ln_Counter,
                                         lnode->ln_Offset,
                                         lnode->ln_Length);
                                    if (IoErr())
                                        PrintFault(IoErr(), NULL);
                                }
                                /* Remove node no matter what */
                                FreeMem(lnode, sizeof(struct LockNode));
                                lnode = nnode;
                            };

                            Close(fh);
                        }
                        FreeMem(cmdbuffer, 80);
                    } else
                        SetIoErr(ERROR_NO_FREE_STORE);

                    FreeMem(csource->CS_Buffer, 512);
                } else
                    SetIoErr(ERROR_NO_FREE_STORE);

                FreeDosObject(DOS_RDARGS, rdargs);
            } else
                SetIoErr(ERROR_NO_FREE_STORE);

            FreeMem(locklist, sizeof(struct LockList));
        } else
            SetIoErr(ERROR_NO_FREE_STORE);


        error = IoErr();
        if (error)
        {
            PrintFault(IoErr(), NULL);
            error = RETURN_FAIL;
        }

        CloseLibrary((struct Library *)DOSBase);
    }
    return(error);
}
```

```
void GetCommandLine(BPTR fh, struct RDArgs *rdargs, UBYTE *cmdbuffer)
{
    struct CSource *csource = &rdargs->RDA_Source;
    UBYTE *cmdlinebuffer = csource->CS_Buffer;
    LONG error;

    /* Prompt for command line */
    Write(Output(), "Cmd> ", 5);

    /* Loop forever, waiting for commands */
    for (;;)
    {
        /* Get command line */
        if ((FGets(Input(), cmdlinebuffer, 512)) != NULL)
        {

            /* Use ReadItem() to isolate actual command */
            error = ReadItem(cmdbuffer, 80, csource);

            /* Break on error */
            if (error == ITEM_ERROR)
                break;

            /* Make sure I've got something */
            else if (error != ITEM_NOTHING)
            {
                /* cmdbuffer now contains the command:
                 *
                 * KNOWN COMMANDS:
                 * QUIT
                 * LIST
                 * LOCKRECORD
                 * UNLOCKRECORD
                 */

                if ((Stricmp("QUIT", cmdbuffer)) == 0)
                    break;
                else if ((Stricmp("HELP", cmdbuffer)) == 0)
                {
                    Printf("Available commands:\n");
                    Printf("LOCKRECORD %s\n", LOCK_TEMPLATE);
                    Printf("UNLOCKRECORD %s\n", UNLOCK_TEMPLATE);
                    Printf("LIST\n");
                    Printf("QUIT\n");
                }
                else if ((Stricmp("LIST", cmdbuffer)) == 0)
                    ListRecordLocks();  /* Show all current locks */
                else
                {

                    /* Note that I've already isolated the command
                     * keyword, so I'm using Source->CS_CurChr to point
                     * after it.
                     */
                    csource->CS_Buffer += csource->CS_CurChr;
                    csource->CS_CurChr = 0;

                    if ((Stricmp("LOCKRECORD", cmdbuffer)) == 0)
                        DoLockRecord(fh, rdargs);
                    else if ((Stricmp("UNLOCKRECORD", cmdbuffer)) == 0)
                        DoUnLockRecord(fh, rdargs);
                    else
                        PrintFault(ERROR_NOT_IMPLEMENTED, cmdbuffer);

                    /* Reset CSource */
                    csource->CS_Buffer = cmdlinebuffer;
                }

                /* Output new prompt. Make sure csource is OK. */
                Write(Output(), "Cmd> ", 5);
                csource->CS_CurChr = 0;
            }
        } else
            break;
    }
}
```

```
void DoLockRecord(BPTR fh, struct RDArgs *rdargs)
{
    struct RDArgs *readargs;
    LONG rargs[5];
    ULONG offset, length, timeout, mode;
    ULONG result;
    struct LockNode *lnode;

    offset = length = timeout = mode = 0;
    rargs[0] = rargs[1] = rargs[2] = rargs[3] = rargs[4] = 0;

    if (readargs = ReadArgs(LOCK_TEMPLATE, rargs, rdargs))
    {

        if (rargs[OFFSET_POS])
            offset = *((LONG *)rargs[OFFSET_POS]);
        if (rargs[LENGTH_POS])
            length = *((LONG *)rargs[LENGTH_POS]);
        if (rargs[TIMEOUT_POS])
            timeout = *((LONG *)rargs[TIMEOUT_POS]);

        /* Type of locking */
        if (rargs[EXCLUSIVE_POS])
        {
            if (rargs[IMMEDIATE_POS])
                mode = REC_EXCLUSIVE_IMMED;
            else
                mode = REC_EXCLUSIVE;
        }
        else
        {
            if (rargs[IMMEDIATE_POS])
                mode = REC_SHARED_IMMED;
            else
                mode = REC_SHARED;
        }

        rargs[0] = offset;
        rargs[1] = length;
        switch (mode)
        {
            case REC_EXCLUSIVE_IMMED:
                rargs[2] = (LONG)"REC_EXCLUSIVE_IMMED";
                break;
            case REC_EXCLUSIVE:
                rargs[2] = (LONG)"REC_EXCLUSIVE";
                break;
            case REC_SHARED_IMMED:
                rargs[2] = (LONG)"REC_SHARED_IMMED";
                break;
            case REC_SHARED:
                rargs[2] = (LONG)"REC_SHARED";
                break;
        }
        rargs[3] = timeout;


        /* Show what I'm going to do */
        VFPrintf(Output(),
            "LockRecord: Offset %ld, Length %ld, Mode %s, Timeout %ld...",
            rargs);
        Flush(Output());

        /* Lock the record. Parameters are not checked. It is f.e. possible to
         * specify an offset larger than the size of the file. Possible since
         * Record Locks are not related to the file itself, only the means for
         * you to do arbitration.
         *
         * Note that the timeout value is in ticks...
         */
        result = LockRecord(fh, offset, length, mode, timeout);

        if (result == DOSTRUE)
        {
            Write(Output(), "OK\n", 3);

            /* Add a node to track this record lock */
```

```
            if (lnode = AllocMem(sizeof(struct LockNode), MEMF_CLEAR))
            {
                lnode->ln_Counter = locklist->lh_Counter++;
                lnode->ln_Offset = offset;
                lnode->ln_Length = length;
                lnode->ln_Mode = mode;

                AddTail((struct List *)locklist, (struct Node *)lnode);
            }
            else
            {
                /* Not enough memory for node. You’re on your own... */
                Write(Output(), "Not enough memory to track record lock.\n", 40);
            }
        }
        else
        {
            Write(Output(), "FAILED\n", 7);
            if (IoErr())
                PrintFault(IoErr(), NULL);
        }

        /* Release memory associated with readargs */
        FreeArgs(readargs);
    } else
        PrintFault(IoErr(), NULL);
}

void DoUnLockRecord(BPTR fh, struct RDArgs *rdargs)
{
    struct RDArgs *readargs;
    LONG rargs[2];
    ULONG offset, length;
    ULONG result;
    struct LockNode *lnode;

    offset = length = 0;
    rargs[0] = rargs[1] = 0;

    if (readargs = ReadArgs(LOCK_TEMPLATE, rargs, rdargs))
    {

        if (rargs[OFFSET_POS])
            offset = *((LONG *)rargs[OFFSET_POS]);
        if (rargs[LENGTH_POS])
            length = *((LONG *)rargs[LENGTH_POS]);

        rargs[0] = offset;
        rargs[1] = length;

        /* Show what I’m going to do */
        VFPrintf(Output(), "UnLockRecord: Offset %ld, Length %ld...", rargs);
        Flush(Output());

        /* Unlock indicated record with indicated offset and length.
         * If the same record (same offset/length) is locked multiple times,
         * only one, the first one in the list , will be unlocked.
         */
        result = UnLockRecord(fh, offset, length);

        if (result == DOSTRUE) {
            Write(Output(), "OK\n", 3);

            /* Remove node associated with this lock */
            if (lnode = FindRecordLock(offset, length))
            {
                Remove((struct Node *)lnode);
                FreeMem(lnode, sizeof(struct LockNode));
            }
        }
        else
        {
            Write(Output(), "FAILED\n", 7); /* Keep locknode */
            if (IoErr())
                PrintFault(IoErr(), NULL);
        }
        /* Release memory associated with readargs */
        FreeArgs(readargs);
    } else
        PrintFault(IoErr(), NULL);
}

void ListRecordLocks(void)
{
    struct LockNode *lnode;
    LONG rargs[4];

    for (lnode = locklist->lh_Head; lnode->ln_Succ; lnode = lnode->ln_Succ)
    {
        rargs[0] = lnode->ln_Counter;
        rargs[1] = lnode->ln_Offset;
        rargs[2] = lnode->ln_Length;

        switch (lnode->ln_Mode)
        {
            case REC_EXCLUSIVE_IMMED:
                rargs[3] = (LONG)"REC_EXCLUSIVE_IMMED";
                break;
            case REC_EXCLUSIVE:
                rargs[3] = (LONG)"REC_EXCLUSIVE";
                break;
            case REC_SHARED_IMMED:
                rargs[3] = (LONG)"REC_SHARED_IMMED";
                break;
            case REC_SHARED:
                rargs[3] = (LONG)"REC_SHARED";
                break;
        }

        VFPrintf(Output(), "RecordLock #%ld: Offset %ld Length %ld Mode %s\n", rargs);
    }
    Flush(Output());
}

struct LockNode *FindRecordLock(ULONG offset, ULONG length)
{
    struct LockNode *lnode;

    for (lnode = locklist->lh_Head; lnode->ln_Succ; lnode = lnode->ln_Succ)
    {
        if ((lnode->ln_Offset == offset) && lnode->ln_Length == length)
            return(lnode);
    }
    return(NULL);
}
```