Notice

This newsletter may contain information covered by one or more licenses, copyrights, and non-disclosure agreements. Permission to copy without fee all or part of this material is granted to Institutional Members of the Usenix Association provided that copies are made for internal use at the member campus or plant site.

UNIX is a trademark of Bell Laboratories

CONTENTS

Guidelines for Submission of Newsletter Material .......... 1
Letters...................................................... 2
PWB RJE Connection to Honeywell 66 under GCOS Desired .... 8
News about UNIX.............................................. 8
Statistics Programs.......................................... 8
News about C........................................... 9
Alcyon Corporation Announces C68 Compiler................... 9
UNIX Reading List............................................ 10
Implementation Description for File Locking.................. 11
Guidelines for Submission of Newsletter Material

I would like to use the modern text preparation and communications facilities of UNIX to as great an extent as feasible in the preparation of the Newsletter. I have established an account on our PWB/UNIX system so that those who can provide us with machine manageable material can do so. The telephone number is (512) 474-5511. The login name is login and the password is usenix. (The system is also host utexas on the ARPANET.)

For those submitting paper copy of material, please produce your copy on a daisy wheel printer or similar high quality printing device. Line printer produced copy is typically not adequate for reproduction. Copy should be on 8 1/2" by 11" paper with a 1" margin on left, right, and bottom and 1 1/2" margin on top.

U. S. Mail submissions should be addressed to:

Login Newsletter
Computation Center
The University of Texas
Austin, TX 78712

Attn: Wally Wedel

Editorial

This issue of ;login: is the first of a series of overdue issues which you will be receiving over the next few weeks. I have sufficient material to produce all issues promised for the first half of this year and for the first one for the second half. Now that the Austin meeting is past I have been concentrating on getting all back issues out. They are all in various stages of production now.

This is the first issue to be published containing material which will be distributed to licensees only. If you are a Public member of UNIX you will not receive the article entitled "Implementation Description of File Locking". Future issues of the newsletter will also be published two sections.
Letters

Jason Gait
Associate Professor
Computer Science
Carrier Hall
University, Miss.
38677
February 23, 1981

Wally Wedel
Login Newsletter
Computation Center
Univ of Texas
Austin, Texas
78712

Dear Wally,

We have so taken to heart the recent complaints in \LOGIN: with regard to sparseness of technical contributions, that we felt obligated to produce the enclosed, if you think it fit to print.

We too hope that the newsletter will specialize in technical matters.

Very truly yours,

John Gait
We encountered a problem with the version of the c compiler distributed with the stock Bell Labs UNIX when we attempted to optimize self-refering code, e.g.

L4: mov $L4,-10(r5)

optimizes to

mov $L4,-10(r5)

which of course does not pass successfully through the assembly phase. We bypassed the difficulty by installing the following code in the main for loop of refcount() in c20.c:

```c
if(p->op==MOV)
    q=p->code;
    if(*q++=='$'&&(*q++=='L')){
        label=getn(q);
        labhash[label % LABHS]->refc++;
    }
```

The new code, where getn() is a slight modification of getnum(), looks for an instruction that moves a label somewhere, looks up the label in the hash table and increments its reference count. This prevents decref(), called later in refcount(), from zapping the label node.
Login Newsletter
Computation Center
The University of Texas
Austin, Texas 78712
Attn: Wally Wedel

Dear Mr. Wedel,

I have discovered an anomaly in the code generation of some pdp-11 C compilers. You may want to report this in "login:", since there may be people who are unaware of it.

In certain contexts the wrong type of conditional branch instruction is generated. Instructions such as "jge" may occur rather than instructions such as "jpl". I have observed the problem with both of the compilers distributed with UNIX version 7, Dennis M. Ritchie's compiler and Stephen C. Johnson's compiler. The expression,

\[(j-i < 0) == (j-i < \text{zero})\],

evaluates to zero (i.e. false) in the accompanying program as compiled with these compilers.

I have what seems to be a working fixed version of Dennis M. Ritchie's compiler. I will probably make these changes available in a future USENIX Association software distribution.

Although this anomaly seems serious, remember that many C programs have seemingly worked properly over a number of years.

Sincerely,

George Rosenberg

copies:
Dennis M. Ritchie
Stephen C. Johnson
/*
 * this seems to generate
 * a coding error
 * on some pdp-11 C compilers
 */

int zero = 0;
int selfinv;
int i;
int j;

main ()
{
    selfinv = ~((unsigned)~0) >> 1);
    i = selfinv + 0101;  /* negative */
    j = selfinv - 0100;  /* positive */

    printf("selfinv = 0%o, selfinv\n");
    printf("zero = %d, zero\n");
    printf("i - 0%o = %d, i, i\n");
    printf("j - 0%o = %d, j, j\n");

    printf("(j-i < 0) == (j-i < zero)%s0,
        ((j-i < 0) == (j-i < zero)) ? "true" : "false"
    ");
}

;login:
February 16, 1981.

The Editor, LOGIN
University of Texas
Computation Center
Austin, Texas
USA 78712

Dear Group:

As I noted at the USENIX meeting in San Francisco, I am very unhappy that Western Electric has backed off in their policy regarding the C compiler. As I understand it, their policy is now that institutions with multiple different UNIX licenses may now standardize on any of the C compilers that they have a license for. While this helps solve some of the problems within institutions with regards to multiple licenses it does not do anything towards the problem of a standard C compiler that could be used for distributing software.

As in the case with other Universities, we have developed considerable software, which we provide, free of charge (except for a small distribution fee) to anyone who requests it. Unfortunately, unless the receiver of the distribution has exactly the same C compiler (or compilers) as we have, and also the same "standard" I/O libraries, the odds are very good that most of the programs cannot be regenerated on their system without considerable work.

I propose that USENIX officially request that Western allow USENIX to distribute versions of the Version 7 C compiler and a standard I/O library that (at a minimum) conform to "The C Programming Language" (Prentice-Hall) by Kernighan and Ritchie. The versions should run on UNIX V6, PWB, MINI-UNIX and V7, with the only differences being those required by the operating systems themselves.

It would be nice, to have the compilers available in some way to UNIX-similar installations running Coherent, or Isis or any of the other such systems.

I am unhappy with the way this topic was handled at the S.F. meeting. The Western Electric representative made his statement, but we were not allowed to ask questions or express our
dissatisfaction with the policy. When, at the end of the session, there was finally a chance for us to make a comment, there was no representative from Western there to hear us.

Sincerely,

W. E. Webb
Systems Analyst

WEW:fmt
PWB RJE Connection to Honeywell 66 under GCOS Desired.

The Banque Nationale de Paris is running a PDP 11/70 under PWB/UNIX with an RJE connection to an IBM 370/158. We would also like to connect to a Honeywell 66 system running GCOS. Does anyone know of software and hardware to do this?

R. Barzel, Manager
Technical Assistance and Methods Group
Banque Nationale de Paris
Division Technique
75450 Paris Cedex 09

Telephone 244-50-50

Statistics Programs

A library of 33 statistical programs is available for UNIX users. These programs were written using the BASIC-PLUS compiler which is usually available for PDP 11 systems.


Instructions for ordering the programs is listed in the Preface of the book.

Dennie Van Tassel
Computer Center
University of California
Santa Cruz, California 95064
UNIX™ Reading List


An issue devoted to UNIX; describes UNIX itself and several applications thereof.


Six papers that describe various aspects of a version of UNIX that is used primarily for program development and text processing.


Highlights the UNIX command language (which is a full programming language) and some of the ways in which it has been used.


Except for the references, this paper is in Japanese.


A readable description of a very neat UNIX facility.


Explains what's good about UNIX.


A textbook about building good software tools similar to those available in UNIX.


A book that describes in detail the principal language available in UNIX. Essentially all of UNIX is written in C, which has also been “ported” to a number of different computers.


A discussion of some UNIX features.


An enjoyable article that tells why they like UNIX in New South Wales.


Tells how UNIX was “ported” to an Interdata 7/32. This issue of the Operating Systems Review contains two other UNIX-related papers.


Ten years later, one of the creators of UNIX looks back.


The original, prize-winning UNIX paper, written by its two creators. A revised and updated version of this paper appears in the first reference above.


Describes a UNIX facility for keeping the history of, and for controlling updates to source code and text files.


Describes the original UNIX command language.
Alcyon Corporation announces a C compiler system which generates object code for the Motorola 68000. The C68 compiler system executes on any PDP-11 processor running the UNIX operating system. Consisting of a preprocessor, compiler, relocatable assembler, linking loader, support library, and utilities, this compiler system will generate programs which execute stand-alone or under operating system control on the 68000. The compiler produces assembly language which is assembled into relocatable object by the assembler. The linking loader combines relocatable object files along with referenced library functions and produces an executable object file which includes a symbol table. A utility program transmits the executable object over a serial communication channel in a format acceptable to the MACSBUD loader. Price for a single CPU binary license is $950. Distribution media is nine track magnetic tape.

Alcyon Corporation
8487 Commerce Avenue
San Diego, California 92121
(714) 578-0860
The Usenix Association

PURPOSE: The Usenix Association is an organization of Western Electric licensees and sub-licensees formed for the purpose of exchanging information and ideas about the UNIX operating system and the C Programming Language.

MEMBERSHIP: Four classes of membership in Usenix are offered:

1. Institutional Membership. Institutional Members are the voting members of the Usenix Association. This class of membership is open only to licensees or sub-licensees of Western Electric Co.

2. Non-voting Institutional Membership. This class of membership is open to corporate affiliates of AT&T.

3. Individual Membership. Open to employees of class 1 and 2 members and others who are bound by the software agreements with Western Electric and its licensees.

4. Public Membership. Open to anyone with a bona fide interest in the purpose of the Usenix Association.

For further information write:

Usenix Association
Rockefeller University
Box 8
1230 York Avenue
New York, New York 10021
(212) 360-1182

Facts about UNIX and the Programming Language C

The UNIX operating system was developed by Ken Thompson and Dennis Ritchie of Bell Laboratories in Murray Hill, N.H., during the early 1970's. The C Programming Language was developed originally by Thompson and Ritchie as the implementation language for UNIX. UNIX is made available to the public by Western Electric Co. through its patent licensing office in Greensboro, North Carolina.

Implementation Description for File Locking

John L. Bass
ONYX Systems, Incorporated
73 East Trimble Road
San Jose, California 95131

ABSTRACT

This paper describes the file locking enhancement to UNIX as implemented under ONIX, a UNIX V7 system for the Zilog Z8000 processor.

1. Introduction to file locking issues.

UNIX allows any number of processes to read/write a shared file. What has been lacking in UNIX design, are primitives to allow effective use of this feature without the loss of data occurring when two or more processes try to update the same file area.

For many applications where updates are infrequent, creating a "lock" file as a semaphore has been effective. The next logical extension is to create proper semaphores using either a system call or device driver. The keynote here is that an application process cannot create a "critical region" in which it is guaranteed to complete before scheduling another process. This requires that semaphores must be implemented at the kernel level.

Several semaphore designs have been proposed and implemented at various UNIX sites in the last few years. The general problem in using such designs with multiple databases is that the lock is a separate item which must be mapped to the file by local convention. Furthermore, the region of the file controlled by the lock is often the entire file.

A difficult problem in using external semaphores, is handling multiple database servers in an online application. In this mode several processes are typically updating different records in a single large database. The semaphore scheme generally requires locking the entire database during updates. This results in response time delays due to serialization around the single semaphore. File locking should be enforced at record boundaries to reduce unnecessary serialization. When collisions occur while attempting to lock the same record, the application should be able to determine that the item is already locked.
When semaphores or other locking schemes are used, there is also the chance of system deadlocks. The potential of deadlock occurs when multiple processes need to access multiple lockable records or other resources.

2. Design goals for ONIX environment.

Locking under ONIX was designed to produce a convenient and powerful means of locking file segments. The goal was not to produce a generalized semaphore system, even though the end result seems quite usable as such. Since the locking applies to file segments, it seemed natural that files should be locked and unlocked in a manner similar to the way they are read and written.

The ONIX implementation incorporates the following features:

a) Locks are attributes of an open file. Locks are removed for a process when the process closes the file or terminates.

b) A lockable segment is any one or contiguous bytes. This allows the application to use whatever granularity is appropriate.

c) Multiple processes may lock multiple segments in a file.

d) There is no arbitrary limit to the number of segments which may be locked in a file. There is however a finite number of locks for a particular installation. This number is generally around 200 for ONIX systems.

e) Multiple lock requests for the same process which overlap or are adjacent are internally combined into a single locked segment. Unlock requests may release all or part of a locked segment. Unlocking the center of a segment will result in two locked segments remaining.

f) Read or write requests to a locked segment of a file will leave the process blocked until the locked region is released.

g) Potential deadlock conditions between processes using locked areas are detected and cause an error return. A deadlock condition may occur during a read, write, or locking system call due to some other process having a lock on the file segment. Prior to blocking the process on a locked segment, the lock list and process table are analyzed for a deadlock.

3. Introduction to use of locking system call.
The following sample C program is an example of using the file locking system call:

```c
#include <errno.h>
#include <stdio.h>

/*
 * simple database example to test file locking
 * John Bass, ONYX Systems Inc.
 * Fall 1980
 */

#define UNLOCK 0
#define LOCK 1
#define TLOCK 2

#define ERROR -1
extern int errno;

int rec_no; /* current record to process */
char data[20]; /* buffer for data */
long rec_addr; /* offset into file for record */

main()
{
    register i;
    int df;
    int retval;

    df = open("data_base", 2);
    if(df == ERROR) exit(1);

    /* main loop - get record number from operator, process record */
    for(rec_no = get_recno(); rec_no >= 0; rec_no = get_recno())
    {
        /* position file pointer
         * attempt to lock record
         * if lock fails determine why */
        rec_addr = rec_no * sizeof data;
        lseek(df, rec_addr, 0);
        retval = locking(df, TLOCK, (long)(sizeof data));
        if(retval == ERROR) {
            switch(errno) {
                case EACCES:
                    printf("Record Busy\n");
                    break;
                case EDEADLOCK:
                    printf("No free locks\n");
                    break;
                default:
                    printf("other error\n");
                    break;
            }
            continue;
        }
        /* process record */
        /* unlock record */
    }
}
```
```c
/*
  * zero buffer for handling records past EOF,
  * read record and update
  */
for(i=0;i<sizeof data;i++)data[i]=0;
read(df, &data, sizeof data);
rec_update();
/*
  * move file pointer back, and over write the record
  */
1seek(df, rec addr, 0);
write(df, &data, sizeof data);
/*
  * reposition file pointer and unlock area
  */
1seek(df, rec addr, 0);
locking(df, UNLOCK, (long)(sizeof data));
}
/*
  * prompt operator for next record number
  * return integer value, or ERROR for
  * no number given
  */
get_recno(){
  int retval;
  char in_line[100];

  printf("Record number? ");
  gets(in_line);
  if(in_line[0]) retval = atoi(in_line);
  else retval = ERROR;
  return(retval);
}
/*
  * prompt operator with current data
  * if operator provides input, use as new data
  */
rec_update(){
  register i;
  char in_line[100];

  printf(" old data:%s\0, data:");
  printf(" new data:");
  for(i=0; i < sizeof in_line; i++) in_line[i] = 0;

  gets(in_line);

  if(in_line[0]) {
    for(i=0; i<sizeof in_line; i++)
      data[i] = in_line[i];
  }
```
else printf("not changed\0");
}

In the example underlined calls to lseek, locking, read, and write represent a sample use sequence. The balance of the program is a mock application to test locking.

4. Error recovery issues.

EDEADLOCK may be returned as an error by any read, write, or locking call. The following sequence of events generally apply:

a) process A locks file segment 1.
b) process B locks file segment 2.
c) process A is put to sleep attempting to read, write, or lock file segment 2.
d) an EDEADLOCK error is returned for any read, write, or lock request by process B on segment 1.

The check for a deadlock condition works for any number of processes. It is suggested that the response to receiving EDEADLOCK on any read, write, or lock system call should be to release all locks the process may hold. Then, the process may retry locking all the required areas after waiting a reasonable period of time.

The EDEADLOCK test in the sample program will only be true when the system runs out of free locks. This error is generally appropriate since waiting for a free lock can cause a deadlock if any other locks are owned by the process. The recovery procedure should be the same as a normal EDEADLOCK error.

Subroutines which read records from a file being updated should buffer only on record boundaries. If only part of a record is buffered, the remaining part may get updated before the remainder is read. This would result in using a record which may contain inconsistent data.

Subroutines which put records to the file should unlock the file segment only after the entire record has been written.

When using the subroutines in "stdio" care must be taken to insure that the automatic buffering provided conforms to these rules. The setbuf routine may be used to set buffering to logical record boundaries on input files. The fflush routine may be used to force output data to be
written prior to unlocking.

Normal UNIX utilities use stdio, and may have logical data problems if used while the file is being updated.

5. Changes to V7 UNIX.

To install locking requires minor changes to these files:

a) /usr/include/sys/inode.h
b) /usr/sys/h/inode.h
c) /usr/include/sys/user.h
d) /usr/sys/h/user.h
e) /usr/include/errno.h
f) /usr/sys/conf/c.c
g) /usr/sys/sys/fio.c
h) /usr/sys/sys/sys2.c
i) /usr/sys/sys/sysent.c

A new file which contains the locking code is added as /usr/sys/sys/lockf.c.

The change for inode.h defines the locklist structure, and adds a pointer for the locklist in the incore inode structure. This pointer is used as the head of a locklist chain describing any locked segments.

The change for user.h and errno.h is to define EDEADLOCK as a valid error code. Sys_errlist may be changed to reflect this error number.

The change for c.c is to define the storage allocation for the locklist array.

The change for fio.c adds a hook for closef to remove any locks for this process.

The change for sys2.c adds the hooks for creat, read, and write. The change for creat was to abort the creat with EACCES error if any segment was locked to the process. The common code for read and write has been changed to sleep when locked segments are accessed.
The sysent.c file has changes to add the entry point for the locking system call. An assembly language routine should be written for your hardware and installed as /usr/src/libc/sys/locking.s. For the ONIX system the arguments were passed in registers, as reflected in the sysent.c entry.

Among the UNIX V7 files about 30 lines are added/changed, for the most part calling routines in lockf.c to do the work.

6. **Description of routines in lockf.c.**

6.1 **Organization of locking module:**

Locking is called through sysent. The arguments are assembled and verified. If in error, the request is aborted.

If the request is unlock, then deletion rules are used to modify or remove the lock on the requested regions.

If the request is to lock, then locked is called with the type of the lock request. If locked returns an error the lock request is aborted. Otherwise insertion rules are used to lock the requested region, and compacts requests if necessary.

6.2 **Organization of locked module:**

Locked is called by locking and the hooks in sys2.c. Locked takes two arguments, a pointer to the incore inode structure, and a flag which signifies either error return or sleeping when a locked area is found.

Locked scans the lock list until the entire requested region is available without blocking.

If an error return is requested, locked sets the error code and returns if the region is not available.

If sleeping is necessary, the deadlock module is called to see if sleeping will cause a deadlock — if so, locked returns an error. If sleeping is ok, locked then sleeps on the list item until it is released. When the list item is released, locked rescans the list looking for other locked regions.

6.3 **Organization of deadlock module:**
Deadlock is called from locked to trace back through the locklist and proc table for a deadlock. If a deadlock is detected, then it error returns; otherwise, it returns NULL.

Deadlock is called with a pointer to the list item on which the sleep will occur. The id entry in that list item is used to see if that process is currently sleeping on another list item. If the "other" process is not sleeping on a list item, no deadlock is pending, and a normal return is made. If the "other" process is sleeping on a list item, and that item was created by this process, a deadlock is pending, and error return is made. Otherwise the id in this new list item is used to check other processes in the chain until one is found which is either not sleeping on a list item, or sleeping on a lock by this process.

6.4 Organization of the unlock module:

Unlock is called by closef to release any locks to the argument inode which are held by the process. The routine scans that inode's list, removing all items by that process.

6.5 Organization of the lockalloc module:

On the first call lockalloc initializes the free lock list.

Lockalloc returns either the next free item on the locklist, or NULL if none are available.

6.6 Organization of the lockfree module:

Lockfree is called by locking to add a locklist entry back into the free list. If any process is sleeping on the lock, the process is awakened.

6.7 Organization of the lockadd module:

Lockadd is called from locking to insert a lockitem into the lock list for an inode. The arguments and data from the current process are used to fill in the entries for the lock.

6.7 User mode test routines:

The remaining code in lockf.c may be conditionally enabled for debugging. If compiled with a debugging flag defined, lockf.c may be executed as a user process to aid testing.
Included are a simple main program to accept requests from standard input, and display the internal state on standard output. Simple stubs are provided to dummy the kernel interface.

7. Code for changes and lockf.c

For each of the following changed files a few lines from UNIX V7 are included as landmarks. Large segments of missing code are represented by line containing "***".

7.1 Listing of changes for /usr/include/sys/inode.h and /usr/sys/h/inode.h

```c
struct
...
inode...
••• (new code follows)
struct locklist *i_locklist; /* locked region list */
```

`#define NFLOCKS 200`

```c
/* file locking hooks -- Sept 1980, John Bass */
struct
locklist
{
    /* NOTE link must be first in struct */
    struct
        locklist *ll_link; /* link to next lock region */
    int ll_flags;    /* misc flags */
    struct
        proc *ll_proc; /* process which owns region */
    off_t ll_start; /* starting offset */
    off_t ll_end;   /* ending offset, zero is eof */
};
```

7.2 Listing of changes for /usr/sys/conf/c.c:

```c
struct
locklist
locklist[NFLOCKS]; /* alloc locklist table */
```

7.3 Listing of changes for /usr/sys/sys/fio.c:

```c
close(fp)
register struct file *fp;
{
    struct chan *cp;
    if(fp == NULL)
        return;
```
### 7.4 Listing of changes for /usr/sys/sys/sys2.c:

```c
/* file locking hook -- J. Bass fall 1980 */
unlock(fp->f_inode);
/* file locking hook -- J. Bass fall 1980 */
```

```c
7.4 Listing of changes for /usr/sys/sys/sys2.c:

```c
rdwr(mode)
register mode;

```c
***
```c
if((fp->flag&FPIPE) != 0) {
***
```c
} else {
***
```c
***
```c
(following lines are changed for locking of read/write)
```c
if((ip->i_mode&(IFCHR&IFBLK)) == 0) {
/* file locking hook -- Sept 1980, John Bass */
```c
if(ip->i_locklist &&
locked(ip,ip->u.u_offset,ip->u.u_offset+ip->u.u_count))
```c
return;
```c
plock(ip);
```c
}
***
```c
(end of locking change)
```c
if(mode == FREAD)
```c
readi(ip);
***
```c
creat()

```c
}
***
```c
if(ip == NULL) {
***
```c
} else {
/* file locking hook -- Jan 1981, John Bass */
```c
if(ip->i_locklist &&
locked(ip->i_mode,(long)(1L<<30))) {
```c
iput(ip);
```c
return;
}
```c
open1(ip, FWRITE, 1);
```c
}
***
```c
(end of locking changes)
```

### 7.5 Listing of changes for /usr/sys/sysent.c:

```c
*** (add following line in a reasonable place)
```c
int locking(); /* file locking hook -- Sept 1980, John Bass */
***
```c
struct sysent sysent[64] =
```c
{
*** (add following line where reasonable -- note syscall number)
```c
45, 4, locking, /* 45 = file locking call */
```c
```
7.6 Listing of new code added as /usr/sys/sys/lockf.c:

/*
 * file lock routines
 * John Bass, PO Box 1223, San Luis Obispo, CA 93401
 * original design spring 1976, CalPoly, San Luis Obispo
 * Deadlock idea from Ed Grudzien at Basys April 1980
 * extensions fall 1980, Onyx Systems Inc., San Jose
 */
#include <sys/param.h>
#include <sys/inode.h>
#include <sys/file.h>
#include <sys/proc.h>
#include <sys/dir.h>
#include <sys/user.h>

#define MAXSIZE (long)(1L<<30)    /* number larger than any request */
/*
 * locking -- handles syscall requests
 */

locking()
{
    struct file *fp;
    struct inode *ip;
    int foo;
    /*
    * define order and type of syscall args
    */
    register struct a {
        int fdes;
        int flag;
        off_t size;
    } *uap = (struct a *)u.u ap;
    register struct locklist *cl, *nl;
    off_t LB, UB;

    /*
    * check for valid open file
    */
    fp = getf(uap->fdes);
    if(fp == NULL) return;
    ip = fp->f_inode;
    if (((ip->i_flag&IFMT) == IFDIR) {
        u.u_error = EACCES;
        return;
    }

    /*
    * validate ranges
    * kludge for zero length
    */
    LB = fp->f_un.f_offset;
    if( uap->size ){
UB = LB + uap->size;
if(UB <= 0) UB = MAXSIZE;
}
else UB = MAXSIZE;

/*
 * test for unlock request
 */
if(uap->flag == 0) {
    /*
     * starting at list head scan
     * for locks in the range by
     * this process
     */
    cl = &ip->i_locklist; /* note addr is pointer in inode */
    while(nl = cl->ll_link) {
        /*
         * if not by this process skip to next lock
         */
        if(nl->ll_proc != u.u_procp) {
            cl = nl;
            continue;
        }
        /*
         * check for locks in proper range
         */
        if( UB <= nl->ll_start )
            break;
        if( nl->ll_end <= LB ) {
            cl = nl;
            continue;
        }
        /*
         * for locks fully contained within
         * requested range, just delete the item
         */
        if( LB <= nl->ll_start && nl->ll_end <= UB ) {
            cl->ll_link = nl->ll_link;
            lockfree(nl);
            continue;
        }
        /*
         * if someone is sleeping on this lock
         * do a wakeup, we may free the region
         * being slept on
         */
        if(nl->ll_flags & IWANT) {
            nl->ll_flags &= ~IWANT;
            wakeup(nl);
        }
        /*
         * middle section is being removed
         * add new lock for last section
         */
/* modify existing lock for first section.  
   * if no locks, return in error  
   */
if( nl->ll_start < LB && UB < nl->ll_end) {
   if( lockadd(nl,UB,nl->ll_end) ) return;
   nl->ll_end = LB;
   break;
}
/*
   * first section is being deleted  
   * just move starting point up  
   */
if( LB <= nl->ll_start && UB < nl->ll_end) {
   nl->ll_start = UB;
   break;
}
/*
   * must be deleting last part of this section  
   * move ending point down  
   * continue looking for locks covered by upper  
   * limit of unlock range  
   */
nl->ll_end = LB;
cl = nl;
}
/*  
   * end of scan for unlock request  
   */
return;

/*
   * request must be a lock of some kind  
   * check to see if the region is lockable by this  
   * process  
   */
if(locked(uap->flag, ip, LB, UB)) return;
cl = &ip->i_locklist;  /* note addr is pointer in inode */
/*
   * simple case, no existing locks, simply add new lock  
   */
if( (nl=cl->ll_link) == NULL ) {
   lockadd(cl, LB, UB);
   return;
}
/*
   * simple case, lock is before existing locks,  
   * simply insert at head of list  
   */
if( UB < nl->ll_start ) {
   lockadd(cl,LB,UB);
   return;
}
/*
• ending range of lock is same as start of lock by
• another process, simply insert at head of list
*/
if( UB <= nl->ll_start && u.u_procp != nl->ll_proc ) {
  lockadd(cl, LB, UB);
  return;
}
/
• request range overlaps with beginning of first request
• modify starting point in lock to include requested region
*/
if( UB >= nl->ll_start && LB < nl->ll_start ) {
  nl->ll_start = LB;
}
/
• scan thru remaining locklist
*/
c1 = nl;
do {
  /* actions for requests at end of list */
  if( ( nl = cl->ll_link ) == NULL ) {
  /*
  • request overlaps tail of last entry
  • extend end point
  */
    if( LB <= cl->ll_end && u.u_procp == cl->ll_proc ) {
      if( UB > cl->ll_end ) cl->ll_end = UB;
      return;
    }
  } /*
  • otherwise add new entry
  */
  lockadd(cl, LB, UB);
  return;
} /*
• if more locks in range skip to next
• otherwise stop scan
*/
if( nl->ll_start < LB ) {
  cl = nl;
} else {
  break;
}
} while(foo==1); /* compiler bug for do()while(constant) */
/
• if upper bound is fully resolved were done
• otherwise fix end of last entry or add new entry
*/
if(UB <= cl->ll_end) return;
if(LB <= cl->ll_end && u.u_procp == cl->ll_proc) cl->ll_end = UB;
else {
    if( lockadd(cl, LB, UB) ) return;
    cl = cl->ll_link;
}

#ifndef end point set above may overlap later entries
if so delete or modify them to perform the compaction
#end
while( (nl=cl->ll_link) != NULL ) {
    /*
    * if we found lock by another process we must
    * be done since we validated the range above
    *
    if(u.u_procp != nl->ll_proc) return;
    /*
    * if the new endpoint no longer overlaps were done
    */
    if(cl->ll_end < nl->ll_start) return;
    /*
    * if the new range overlaps the first part of the
    * next lock, take its end point
    * and delete the next lock
    * we should be done
    */
    if(cl->ll_end <= nl->ll_end) {
        cl->ll_end = nl->ll_end;
        cl->ll_link = nl->ll_link;
        lockfree(nl);
        return;
    }
    /*
    * the next lock is fully included in the new range
    * so it may be deleted
    */
    cl->ll_link = nl->ll_link;
    lockfree(nl);
}

/*
* locked -- routine to scan locks and check for a locked condition
*/
locked(flag, ip, LB, UB)
register struct inode *ip;
off_t LB, UB;
{
    register struct locklist *nl = ip->i_locklist;

    /*
    * scan list while locks are in requested range
    */
    while(nl != NULL && nl->ll_start < UB) {
    /*

# skip locks for this process
# and those out of range
*/

if (nl->ll_proc == u.u_procp || nl->ll_end <= LB) {
    nl = nl->ll_link;
    if (nl == NULL) return(NULL);
    continue;
}

/*
 * must have found lock by another process
 * if request is to test only, then exit with 
 * error code
*/

if (flag > 1) {
    u.u_error = EACCES;
    return(1);
}

/*
 * will need to sleep on lock, check for deadlock first
 * abort on error
*/

if (deadlock(nl) != NULL) return(1);

/*
 * post want flag to get awoken
 * then sleep till lock is released
*/

nl->llFlags = IWANT;

sleep(~(caddr_t)nl, PSLEP);

/*
 * set scan back to begining to catch
 * any new areas locked
 * or a partial delete
*/

nl = ip->i_locklist;

/*
 * abort if any errors
*/

if (u.u_error) return(1);

}

return(NULL);

/*
 * deadlock -- routine to follow chain of locks and proc table entries
 * to find deadlocks on file locks.
*/

deadlock(lp)
register struct locklist *lp;
{
    register struct locklist *nl;

    /*
* scan while the process owning the lock is sleeping *

while(lp->ll_proc->p_stat == SSLEEP) {

  * if the object the process is sleeping on is
  * NOT in the locktable everything is ok
  * fall out of loop and return NULL
  */
  nl = lp->ll_proc->p_wchan;
    break;

  * the object was a locklist entry
  * if the owner of that entity is this
  * process then a deadlock would occur
  * set error exit and return
  */
  if(nl->ll_proc == u.u_procp) {
    u.u_error = EDEADLOCK;
    return(nl);
  }

  * the object was a locklist entry
  * owned by some other process
  * continue the scan with that process
  */
  lp = nl;
}
return(NULL);

unlock -- called by close to release all locks for this process

unlock(ip)
register struct inode *ip;
{
  register struct locklist *nl;
  register struct locklist *cl;

  if(cl = &ip->i_locklist)
    while( (nl = cl->ll_link) != NULL ) {
      if(nl->ll_proc == u.u_procp) {
        cl->ll_link = nl->ll_link;
        lockfree(nl);
      } else cl = nl;
    }
}

lockalloc -- allocates free list, returns free lock items
# lockalloc

```c
struct locklist *
lockalloc()
{
    register struct locklist *fl = &locklist[0];
    register struct locklist *nl;

    /*
    * if first entry has never been used
    * link the locklist table into the freelist
    */
    if(fl->ll_proc == NULL) {
        fl->ll_proc = &proc[0];
        for(nl= &locklist[1]; nl < &locklist[NFLOCKS]; nl++) {
            lockfree(nl);
        }
    }

    /*
    * if all the locks are used error exit
    */
    if( (nl=fl->ll_link) == NULL) {
        u.u_error = EDEADLOCK;
        return(NULL);
    }

    /*
    * return the next lock on the list
    */
    fl->ll_link = nl->ll_link;
    nl->ll_link = NULL;
    return(nl);
}
```

# lockfree

```c
lockfree(lp)
register struct locklist *lp;
{
    register struct locklist *fl = &locklist[0];

    /*
    * if some process is sleeping on this lock
    * wake them up
    */
    if(lp->ll_flags & IWANT) {
        lp->ll_flags &= ~IWANT;
        wakeup(lp);
    }

    /*
    * add the lock into the free list
    */
    lp->ll_link = fl->ll_link;
    fl->ll_link = lp;
```
lockadd -- routine to add item to list

lockadd(cl, LB, UB)
register struct locklist *cl;
off_t LB, UB;
{
    register struct locklist *nl;

    /*
     * get a lock, return if none available
     */
    nl = lockalloc();
    if(nl == NULL) {
        return(1);
    }

    /*
     * link the new entry into list at current spot
     * fill in the data from the args
     */
    nl->ll_link = cl->ll_link;
    cl->ll_link = nl;
    nl->ll_proc = u.u_proc;
    nl->ll_start = LB;
    nl->ll_end = UB;
    return(0);
}

#if debugging
/
******************************************************************************
** the following code is the user state test fixture for the locking code ****
******************************************************************************
#

#define TESTMAX 10
struct proc proc[TESTMAX];
struct user u;
struct locklist locklist[NFLOCKS];
struct file file[TESTMAX];
struct inode inode[TESTMAX];

main()
{
    int val;
    register struct a {
        int fdes;
        char flag;
        off_t size;
    } *uap = (struct a *) u.u_ap;
    while(1) {
        printf("proc fdes flag offset size:\n");
        scanf("%d", &val); u.u_proc = val;
scanf("%d", &val); uap->fdes = val;
scanf("%d", &val); uap->flag = val;
scanf("%d", &val); file[uap->fdes].f_un.f_offset = val;
scanf("%d", &val); uap->size = val;

if(uap->flag > 2) exit(0);

u.u_procp = &proc[(int)u.u_procp];
u.u_error = NULL;
locking();
if(u.u_error) printf("\n***error is %d, u.u_error);
prntlocks(file[uap->fdes].f_inode);
"
}
sleep(a, b) {
    printf("sleeping on (%x)\n", a);
    u.u_procp->p_wchan = a;
    u.u_procp->p_stat = SSLEEP;
    u.u_error = 99;
}
wakeup(a)
{
    register struct proc *p;
    for(p = proc; p < &proc[TESTMAX]; p++)
        if(p->p_wchan == a) {
            p->p_wchan = 0;
            p->p_stat = 0;
            printf("process(%d) awake\n", p-&proc[0]);
        }
}
getf(a){
    file[a].f_inode = &inode[a];
    return(&file[a]);
}
prntlocks(ip){
    register struct locklist *ll;
    register struct proc *p;
    for(ll = ip->i_locklist; ll; ll = ll->ll_link)
        printf("ll(%x) ll_link(%x) ll_procp(%d) ll_lb(%d) ll_ub(%d)\n", ll,
            ll->ll_link,
            ll->ll_procp - &proc[0],
            ll->ll_start,
            ll->ll_end);
    for(p = proc; p < maxproc; p++)
        if(p->p_stat == SSLEEP && p->p_wchan > = locklist &&
            p->p_wchan < &locklist[NLOCKS])
            printf("proc(%d) sleeping\n", p-&proc[0]);
}
#endif
NAME
locking - provide exclusive file regions for reading or writing

SYNOPSIS
locking(fildes, mode, size);
lsize;

DESCRIPTION
Locking will allow a specified number of bytes to be accessed only by the locking process. Other processes which attempt to lock, read, or write the locked area will sleep until the area becomes unlocked.

Fildes is the word returned from a successful open, creat, dup, or pipe system call.

Mode is zero to unlock the area. Mode is one or two for making the area locked. If the mode is one, and the area has some other lock on it, then the process will sleep until the entire area is available. If the mode is two, and the area is locked, an error will be returned.

Size is the number of contiguous bytes to be locked or unlocked. The area to be locked starts at the current offset in the file. If size is zero the area to end of file is locked.

The potential for a deadlock occurs when a process controlling a locked area is put to sleep by accessing another processes locked area. Thus calls to locking, read, or write scan for a deadlock prior to sleeping on a locked area. An error return is made if sleeping on the locked area would cause a deadlock.

Lock requests may, in whole or part, contain or be contained by a previously locked area for the same process. When this or adjacent areas occur, the areas are combined into a single area. If the request requires a new lock element with the lock table full, an error is returned, and the area is not locked.

Unlock requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining areas are still locked by the process. Release of the center section of a locked area requires an additional lock element to hold the cut off section. If the lock table is full, an error is returned, and the requested area is not released.
While locks may be applied to special files or pipes, read/write operations will not be blocked. Locks may not be applied to a directory.

**SEE ALSO**
open(2), creat(2), read(2), write(2), dup(2), close(2)

**DIAGNOSTICS**
The value -1 is returned if the file does not exist, or if a deadlock using file locks would occur. EACCES will be returned for lock requests in which the area is already locked by another process. EDEADLOCK will be returned by locking, read, or write if a deadlock would occur. EDEADLOCK will also be returned when the locktable overflows.

**ASSEMBLER**
\[(\text{locking} = 45)\]
\[(\text{file descriptor in r0})\]
\[(\text{mode in r1})\]
\[(\text{size in r2-r3})\]
\[\text{syslocking}\]