Bahir Dar University

Bahir Dar Institute of Technology

Faculty of Computing

Department of Software Engineering

Operating system and System Programming

Individual Assignment on

int fcntl(int fd, int cmd, ... /* arg */)

Made by: Elias Ayana

ID: 1308324

Submitted to: Lecturer Wendimu Baye

Submission date: 17/11/2014 E.c

Table of contents

What, why, and how fcntl() system call?

As the name indicates that fcntl is abbreviated as 'file' control. It means it is based on the file handling process. The fcntl is a system call. It allows the program to place a read or a write lock. This function can be used to amend the file properties that are either opened already or can be opened through any action applied to it. It is a versatile function and is used to modify files in many ways like open, read and write, etc. This article is about the control functions on files.

The fcntl system call is the access point for several advanced operations on file descriptors. The first argument to fcntl is an open file descriptor, and the second is a value that indicates which operation is to be performed. For some operations, fcntl takes an additional argument

In the Linux operating system, the fcntl call is done through the descriptors. For instance, a read lock is placed on a readable file descriptor, and a similar case is for the write lock. A file descriptor represents the file number that is opened. It is convenient for the program to remember which file it is working on. When we open a file, the number that is not assigned already and is free is given to the file in the descriptor table of the processes file. And in the case of closing a file, the assigned number is removed from the process's descriptor table.

The fcntl function has 5 functions:

1. Copy an existing descriptor (cmd=F_DUPFD).
2. Get/set the file descriptor mark (cmd=F_GETFD or F_SETFD).
3. Get/set the file status mark (cmd= F_GETFL or F_SETFL).
4. Get/set asynchronous I/O ownership (cmd=F_GETOWN or F_SETOWN).
5. Get/set record lock (cmd=F_GETLK, F_SETLK or F_SETLKW).

1. F_DUPFD of cmd value: F_DUPFD
returns a (file) descriptor as described below:
· The smallest available descriptor greater than or equal to arg
· A reference to an
object that is the same as the original operator· If the object is a file (file), a new descriptor is returned. This descriptor and arg share the same offset (offset)
. The same access mode (read, write or read/write)
. The same file status flags (such as two File descriptors share the same status flag)
The close-on-exec flag combined with the new file descriptor is set to cross-access execve(2) system call

Actually call dup(oldfd); equivalent to fcntl(oldfd, F_DUPFD, 0);

And call dup2(oldfd, newfd); equivalent to

```
    close(oldfd);
    fcntl(oldfd, F_DUPFD, newfd);
```

2. F_GETFD and F_SETFD of cmd value:

F_GETFD

obtains the close-on-exec flag associated with the file descriptor fd, similar to FD_CLOEXEC. If the return value and the result of the FD_CLOEXEC AND operation are 0, the file remains cross-accessed exec(), otherwise, if it is run through exec, the file will be closed (arg is ignored)

F_SETFD sets the close-on-exec flag, the flag is The FD_CLOEXEC bit of the parameter arg is determined. It should be understood that many existing programs involving file descriptor flags do not use the constant FD_CLOEXEC, but set this flag to 0 (system default, not closed during exec) or 1 (closed during exec))

You must be careful when modifying the file descriptor flag or file status flag. 1. obtain the current flag value, then modify it as desired, and finally set the new flag value. You can't just execute F_SETFD or F_SETFL commands, this will turn off the previously set flag.

3. F_GETFL and F_SETFL of cmd value:

F_GETFL

obtains the file status flag of fd, as described below (arg is ignored). When the open function is described,
the file status flag has been explained. Unfortunately, the three access mode flags (O_RDONLY, O_WRONLY, and O_RDWR) do not occupy one bit each. (The values of these three flags are 0, 1 and 2. For historical reasons, these three values are mutually exclusive-a file can only have one of these three values.) Therefore, the mask word O_ACCMODE must first be used to associate with the storage. Take the mode bit and compare the result with these three values.

F_SETFL is

set to the arg descriptor status flag, several flags that can be changed are: O_APPEND, O_NONBLOCK, O_SYNC and O_ASYNC. The fcntl file status flag has a total of 7: O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_NONBLOCK, O_SYNC and O_ASYNC

Several flags that can be changed are described below: O_NONBLOCK Non-blocking I/O, if the read(2) call has no data to read, or if the write(2) operation will block, the read or write call will return -1 And EAGAIN error

O_APPEND

forces every write operation to be added at the end of the file size, which is equivalent to the O_APPEND flag

O_DIRECT of open(2) to

minimize or remove the cache impact of reading and writing. The system will try to avoid caching your read or write data. If caching cannot be avoided, it will minimize the impact of data that has already been cached. If this flag is not used well enough, performance will be greatly reduced.

O_ASYNC

allows SIGIO signals to be sent to the process group when I/O is available, for example: when there is data to read

4. F_GETOWN and F_SETOWN of cmd value:

F_GETOWN

obtains the process id or process group id that is currently receiving SIGIO or SIGURG signals, and the process group id returns a negative value (arg is ignored)

F_SETOWN

sets the process id that will receive SIGIO and SIGURG signals Or process group id, the process group id is specified by providing a negative arg (a process group ID of the absolute value of arg), otherwise arg will be considered as the process id

5. F_GETLK, F_SETLK or F_SETLKW of cmd value : Get/set record lock function, return 0 if successful, return -1 if there is an error, and the cause of the error is stored in errno. F_GETLK obtains the lock pointed to by the first blocking lock description through the third parameter arg (a structure pointing to flock). The obtained information will overwrite the information of the flock structure passed to fcntl(). If no lock is found that can prevent the generation of this lock (flock), this structure will not be changed unless the lock type is set to F_UNLCK F_SETLK according to the lock information described in the third parameter arg of the pointer to the structure flock Set or clear the segment lock of a file. F_SETLK is used to implement shared (or read) lock (F_RDLCK) or exclusive (write) lock (F_WRLCK), and these two locks (F_UNLCK) can also be removed. If the shared lock or exclusive lock cannot be set, fcntl() will immediately return EAGAIN F_SETLKW . This command is the same as F_SETLK except that the shared lock or exclusive lock is blocked by other locks. If the shared lock or exclusive lock is blocked by other locks, the process will wait until the request can be completed. When fcntl() is waiting for a certain area of the file, a signal is captured. If this signal is not specified SA_RESTART, fcntl will be interrupted

When a shared lock is set to a certain section of a file, other processes can set the shared lock to this section or part of this section. Shared locks prevent any other process from setting exclusive locks to any part of this protected area. If the file descriptor is not opened for read access, the request to set the shared lock will fail.

The exclusive lock prevents any other process from setting a shared lock or exclusive lock anywhere in this protected area. If the file descriptor is not opened for write access, the exclusive lock request will fail.

Pointer to structure flock:

struct flcok
{
short int l_type;/* locked state*/

//The following three parameters are used to lock the file in sections. If the entire file is locked, then: l_whence=SEEK_SET, l_start=0, l_len=0 short int l_whence;/*Determine the position of l_start*/
off_t l_start;/*The beginning of the locked area*/
off_t l_len;/*The size of the locked area*/

pid_t l_pid;/*The process of locking the action*/ };

l_type has three states: F_RDLCK establishes a lock for reading
F_WRLCK establishes a lock for writing
F_UNLCK deletes the previously established lock

There are also three ways for l_whence : SEEK_SET takes the beginning of the file as the starting position of the lock SEEK_CUR takes the current reading and writing position of the file as the starting position of the lock SEEK_END takes the end of the file as the starting position of the lock

There are two types of fcntl file locks: advisory locks and mandatory locks. The advisory locks are stipulated in this way : each process that uses the locked file must check whether there is a lock, and of course the existing locks must be respected. The kernel and the system as a whole insist on not using advisory locks, and they rely on programmers to comply with this rule. Mandatory lock is executed by the kernel : when a file is locked for writing, the kernel will block any read or write access to the file before the process that locks the file releases the lock, every time it is read or written Access has to check whether the lock exists.

The system default fcntl is advisory lock, mandatory lock is non-POSIX standard. If you want to use a mandatory lock, you need to remount the file system to make the entire system use mandatory lock, mount use parameter -0 mand to open the mandatory lock, or close the group execution permission of the locked file and open The set-GID permission bit of the

file. Suggested locks are only useful between cooperating processes . The understanding of the cooperating process is the most important. It refers to the process that affects other processes or is affected by other processes. Two examples are:
(1) We can run the same command in two windows at the same time , Operate on the same file, then these two processes are cooperating processes
(2) cat file | sort, then the processes generated by cat and sort are the cooperating processes using pipe

Use fcntl file lock for I/O operations must be careful: how the process handles the lock before starting any I/O operation, and how to complete all operations before unlocking the file must be considered. If the file is opened before the lock is set, or the file is closed after reading the lock, another process may access the file within a fraction of a second between the lock/unlock operation and the open/close operation. When a process locks a file, regardless of whether it releases the lock, as long as the file is closed, the kernel will automatically release the recommended lock on the file (this is also the biggest difference between a recommended lock and a mandatory lock), so Don't want to set up advisory locks to achieve the purpose of permanently preventing other processes from accessing files (compulsory locks are only possible); mandatory locks work for all processes.

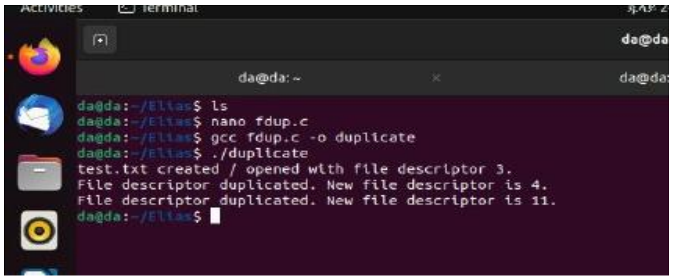
fcntl uses three parameters F_SETLK/F_SETLKW, F_UNLCK and F_GETLK to request, release, and test record locks respectively . Record locks are locks on a part of a file instead of the entire file. This kind of meticulous control allows processes to cooperate better to share file resources. Fcntl can be used for read locks and write locks. Read locks are also called shared locks, because multiple cooperating processes can establish read locks on the same part of the file; write locks are called exclusive locks. , Because only one cooperating process can establish a write lock on a certain part of the file at any time. If the cooperating processes operate on the file, they can add read lock to the file at the same time. Before adding write lock to a cooperating process, they must release the read lock and wrtie lock added to the file by other cooperating processes. Only one write lock can exist, and read lock and wrtie lock cannot coexist.

Examples

The following example uses F_DUPFD to duplicate a file

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
```

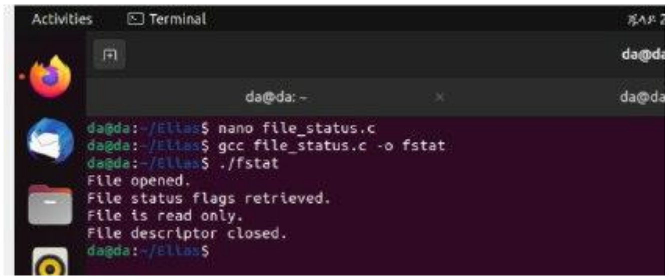
```
int main(void){

    int fdOne, fdTwo, fdThree;

    fdOne = open("misc.txt",  O_CREAT | O_TRUNC, S_IRWXU);

    if(fdOne < 0){
        printf("Error  opening / creating test.txt. ");
        if(errno==EACCES){
            printf("Error #%d (EACCES): Permission denied.\n", EACCES);
            exit(EXIT_FAILURE);
        }
    } else {
        printf("test.txt  created / opened ");
        printf("with  file descriptor %d.\n", fdOne);
    }

    //lets use the fcntl() function to copy the
    //file descriptor
   if((fdTwo = fcntl(fdOne, F_DUPFD, 0))<0){
       printf("Error  duplicating  file descriptor.\n");
       exit(EXIT_FAILURE);
   } else {
       printf("File  descriptor  duplicated.  ");
       printf("New  file descriptor is %d.\n", fdTwo);
   }

   //set the file descriptor  to be a higher  number;

    if((fdThree = fcntl(fdOne,  F_DUPFD, 11))<0){
        printf("Error  duplicating  file descriptor.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("File  descriptor  duplicated.  ");
        printf("New  file descriptor  is %d.\n", fdThree);
    }

    close(fdOne);
    close(fdTwo);
    close(fdThree);

    return 0;

}
```



```
da@da:~/Elias$ ls
da@da:~/Elias$ nano fdup.c
da@da:~/Elias$ gcc fdup.c -o duplicate
da@da:~/Elias$ ./duplicate
test.txt created / opened with file descriptor 3.
File descriptor duplicated. New file descriptor is 4.
File descriptor duplicated. New file descriptor is 11.
da@da:~/Elias$ 
```

The following example uses F_GETFL to get the file status flag of fd .

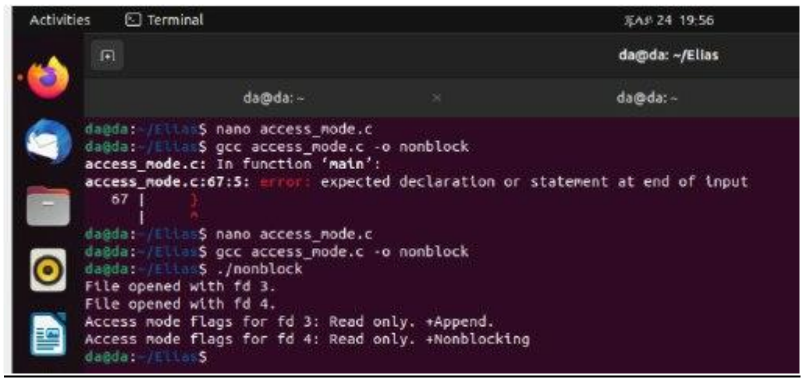
```
 #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>

int main(void){

   int fflags, rVal, fd;

   if((fd = open("afile", O_CREAT | O_APPEND, S_IRWXU))<0){
      printf("error opening file.\n");
      exit(EXIT_FAILURE);
   } else {
      printf("File opened.\n");
   }

   if((rVal = fcntl(fd, F_GETFL))<0){
      printf("Error getting file status flags.\n");
      exit(EXIT_FAILURE);
   } else {
      printf("File status flags retrieved.\n");
   }

   //O_ACCMODE has the value of three
   fflags = rVal & O_ACCMODE;
   if(fflags == O_RDONLY){
      printf("File is read only.\n");
   } else if (fflags == O_WRONLY){
      printf("File is write only.\n");
   } else if (fflags == O_RDWR){
      printf("File is read / write.\n");
   } else {
      printf("No idea.\n");
   }


   if(close(fd)<0){
      printf("Error closing file.\n");
      exit(EXIT_FAILURE);
   } else {
      printf("File descriptor closed.\n");
   }


   return 0;
}
```



```
da@da:~/Elias$ nano file_status.c
da@da:~/Elias$ gcc file_status.c -o fstat
da@da:~/Elias$ ./fstat
File opened.
File status flags retrieved.
File is read only.
File descriptor closed.
da@da:~/Elias$
```

The following example shows O_APPEND and O_NONBLOCK mode usage

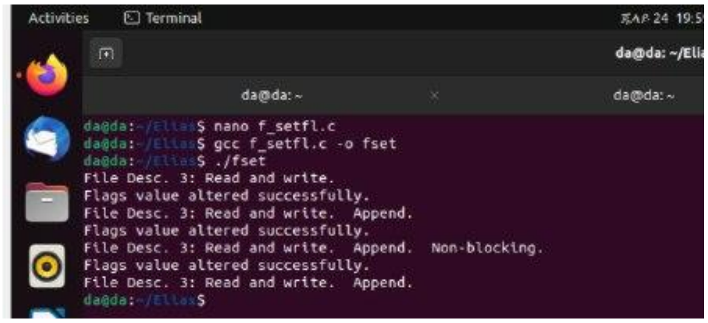
```
  #include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

void readFlags(int fd){
   int returnValue;
   if((returnValue  = fcntl(fd, F_GETFL,  0))<0){
       printf("Error opening file descriptor %d.\n", fd);
       return;
   }
   printf("Access mode flags for fd %d: ", fd);
   switch(returnValue  & O_ACCMODE){
       case O_RDONLY:
           printf("Read only. ");
           break;
       case O_WRONLY:
           printf("Write only. ");
           break;
       case O_RDWR:
           printf("Read and write. ");
           break;
       default:
           printf("Unknown access mode. ");
           break;
   } // end switch
   if(returnValue  & O_APPEND){
       printf("+Append.");
   }
   if(returnValue  & O_NONBLOCK){
       printf("+Nonblocking");
   }
   printf("\n");
} // end function readFlags ---------------

int main(void){

   int fdOne, fdTwo;

   if((fdOne  = open("append.txt", O_CREAT  | O_APPEND,  S_IRWXU))<0){
       printf("Error opening append.txt.\n");
       exit(EXIT_FAILURE);
   } else {
       printf("File opened with fd %d.\n", fdOne);
   }


   if((fdTwo  = open("nonblocking.txt", O_CREAT  | O_NONBLOCK,  S_IRWXU))<0){
      printf("Error opening nonblocking.txt");
      exit(EXIT_FAILURE);
   } else {
       printf("File opened with fd %d.\n", fdTwo);
   }

   readFlags(fdOne);
   readFlags(fdTwo);

   if((close(fdOne))<0){
       printf("Error closing fdOne.\n");
       exit(EXIT_FAILURE);
   }

   if((close(fdTwo))<0){
       printf("Error closing fdTwo.\n");
       exit(EXIT_FAILURE);
   }

   return 0;

}
```



```
da@da:~/Elias$ nano access_mode.c
da@da:~/Elias$ gcc access_mode.c -o nonblock
access_mode.c: In function 'main':
access_mode.c:67:5: error: expected declaration or statement at end of input
   67 |     }
      |     ^
da@da:~/Elias$ nano access_mode.c
da@da:~/Elias$ gcc access_mode.c -o nonblock
da@da:~/Elias$ ./nonblock
File opened with fd 3.
File opened with fd 4.
Access mode flags for fd 3: Read only. +Append.
Access mode flags for fd 4: Read only. +Nonblocking
da@da:~/Elias$
```

The following example uses F_SETFL to set the file status flag of fd .

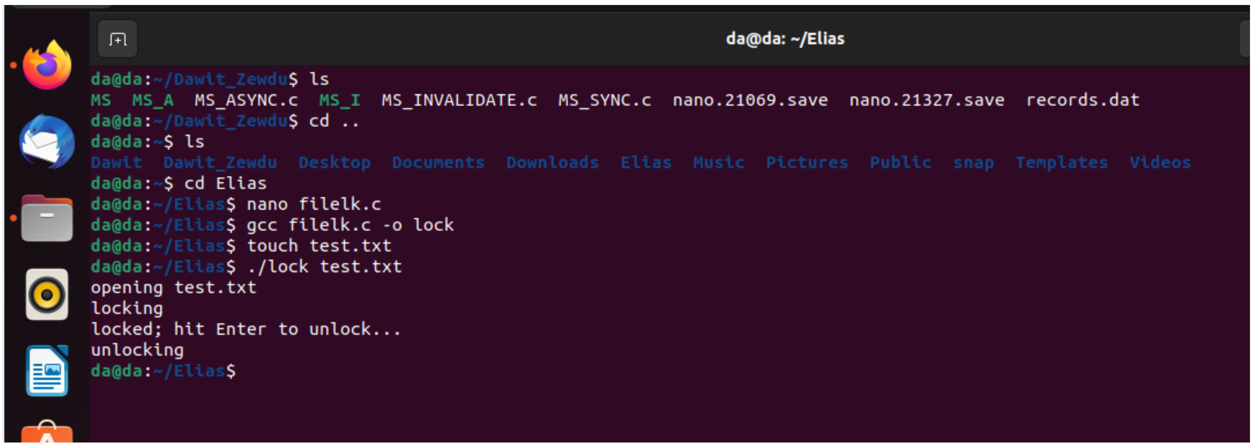
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int setFlags(int fd, int flags);
int removeFlags(int fd, int flags);
void strFlagReturn(int returnValue);
void printFlags(int fd);


int main(void){

    int fdOne = open("another.txt", O_CREAT | O_TRUNC | O_RDWR, S_IRWXU);

    if(fdOne < 0){
        printf("Error creating file descriptor.\n");
        exit(EXIT_FAILURE);
    }

    printFlags(fdOne);
    strFlagReturn(setFlags(fdOne, O_APPEND));
    printFlags(fdOne);
    strFlagReturn(setFlags(fdOne, O_NONBLOCK));
    printFlags(fdOne);
    strFlagReturn(removeFlags(fdOne, O_NONBLOCK));
    printFlags(fdOne);

    return 0;

}


void strFlagReturn(int returnVal){
    switch(returnVal){
        case 0:
            printf("Flags value altered successfully.\n");
            break;
        case -1:
            printf("Could not get flags to alter them.\n");
            break;
        case -2:
            printf("Could not alter flags after retrieving them.\n");
            break;
    }

}
void printFlags(int fd){
    int flags;

    if((flags=fcntl(fd, F_GETFL, 0))<0){
        printf("Couldn't access flags for fd %d.\n", fd);
        return;
    } else {
        printf("File Desc. %d: ", fd);
    }
    switch(flags & O_ACCMODE){
        case O_RDONLY:
            printf("Read only. ");
            break;
        case O_WRONLY:
            printf("Write only. ");
            break;
        case O_RDWR:
            printf("Read and write. ");
            break;
    } // end switch-----

    if(flags & O_APPEND){
        printf(" Append. ");
    }

    if(flags & O_NONBLOCK){
        printf(" Non-blocking.");
    }

    printf("\n");
}

//this uses logical and to remove flags
int removeFlags(int fd, int flags){
    int oldFlags;

    if((oldFlags = fcntl(fd, F_GETFL, 0))<0){
        return -1;
    }

    //AND the flag values
    //and inverse the flags parameter
    flags = ~flags & oldFlags;

    if(fcntl(fd, F_SETFL, flags) < 0){
        return -2;
    }
    return 0;

}
```



```
da@da:~/Elias$ nano f_setfl.c
da@da:~/Elias$ gcc f_setfl.c -o fset
da@da:~/Elias$ ./fset
File Desc. 3: Read and write.
Flags value altered successfully.
File Desc. 3: Read and write.  Append.
Flags value altered successfully.
File Desc. 3: Read and write.  Append.  Non-blocking.
Flags value altered successfully.
File Desc. 3: Read and write.  Append.
da@da:~/Elias$
```

File Locking Example

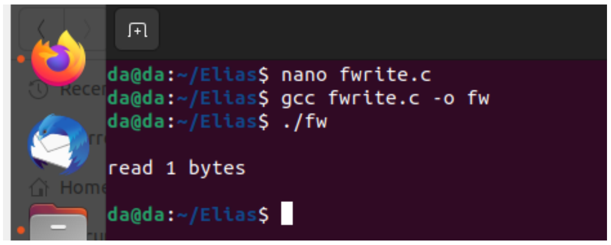
```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
 char* file = argv[1];
 int fd;
 struct flock lock;

 printf ("opening %s\n", file);
 /* Open a file descriptor to the file. */
 fd = open (file, O_WRONLY);
 printf ("locking\n");
 /* Initialize the flock structure. */
 memset (&lock, 0, sizeof(lock));
 lock.l_type = F_WRLCK;
 /* Place a write lock on the file. */
 fcntl (fd, F_SETLKW, &lock);

 printf ("locked; hit Enter to unlock... ");
 /* Wait for the user to hit Enter. */
 getchar ();

 printf ("unlocking\n");
 /* Release the lock. */
 lock.l_type = F_UNLCK;
 fcntl (fd, F_SETLKW, &lock);

 close (fd);
 return 0;
}
```



```
da@da:~/Dawit_Zewdu$ ls
MS  MS_A  MS_ASYNC.c  MS_I  MS_INVALIDATE.c  MS_SYNC.c  nano.21069.save  nano.21327.save  records.dat
da@da:~/Dawit_Zewdu$ cd ..
da@da:~$ ls
Dawit  Dawit_Zewdu  Desktop  Documents  Downloads  Elias  Music  Pictures  Public  snap  Templates  Videos
da@da:~$ cd Elias
da@da:~/Elias$ nano filelk.c
da@da:~/Elias$ gcc filelk.c -o lock
da@da:~/Elias$ touch test.txt
da@da:~/Elias$ ./lock test.txt
opening test.txt
locking
locked; hit Enter to unlock...
unlocking
da@da:~/Elias$
```

File read example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <error.h>

char buf[500000];

int main(int  argc,char *argv[])
{
        int ntowrite,nwrite;
        const char *ptr ;
        int flags;

        ntowrite = read(STDIN_FILENO,buf,sizeof(buf));
        if(ntowrite  <0)
        {
               perror("read STDIN_FILENO   fail:");
               exit(1);
        }
        fprintf(stderr,"read %d bytes\n",ntowrite);

        if((flags  = fcntl(STDOUT_FILENO,F_GETFL,0))==-1)
        {
               perror("fcntl F_GETFL  fail:");
               exit(1);
        }
        flags |= O_NONBLOCK;
        if(fcntl(STDOUT_FILENO,F_SETFL,flags)==-1)
        {
               perror("fcntl F_SETFL  fail:");
               exit(1);
        }

        ptr = buf;
        while(ntowrite  > 0)
        {
               nwrite  = write(STDOUT_FILENO,ptr,ntowrite);
               if(nwrite  == -1)
               {

                      perror("write  file  fail:");
               }
               if(nwrite  > 0)
               {
                ptr += nwrite;
                ntowrite -= nwrite;
            }
        }

        flags &= ~O_NONBLOCK;
        if(fcntl(STDOUT_FILENO,F_SETFL,flags)==-1)
        {
            perror("fcntl F_SETFL fail2:");
        }
        return 0;
}
```



```
da@da:~/Elias$ nano fwrite.c
da@da:~/Elias$ gcc fwrite.c -o fw
da@da:~/Elias$ ./fw

read 1 bytes

da@da:~/Elias$ 
```