贵州大学 计算机科学与技术学院 实验报告

院（系）名称	示范性软件学院	班级	软工 206	课程名称	Linux 系统
实验名称	实验三	日期	11 月 6 日	指导教师	王老师
学号、姓名	2000770081 秦小龙	成绩(10 分制)：		批改日期：	

1. 实验名称

   文件系统-锁

2. 实验目的

   悉 Linux 系统多进程的文件共享（读、写）的锁方法。

2. 内容（原理、方法）

   用 fcntl 和 lockf 两种方式实现文件的记录锁操作，实现父子进程之间文件的共享。

   1）查阅联机手册，掌握文件操作的基本函数/系统调用：open, read, write, close, lseek。

   2）查阅联机手册，熟悉 lockf，fcntl 和 flock 函数的各种参数的含义。

   3）熟悉 Linux 下 C 语言程序的编写、编译、运行和测试技术。

文件锁是用于解决资源的共享使用的一种机制：当多个用户需要共享一个文件时，Linux 通常采用的方法是给文件上锁，来避免共享的资源产生竞争的状态。

文件锁包括建议性锁和强制性锁：

建议性锁：要求每个使用上锁文件的进程都要检查是否有锁存在，并且尊重已有的锁。在一般情况下，内核和系统都不使用建议性锁，它们依靠程序员遵守这个规定。

强制性锁：是由内核执行的锁，当一个文件被上锁进行写入操作的时候，内核将阻止其他任何文件对其进行读写操作。采用强制性锁对性能的影响很大，每次读写操作都必须检查是否有锁存在。使用 mount 命令带“mand”参数来重新挂载根文件系统，才能在文件系统级别使能强制锁功能。

lockf，fcntl 和 flock 的比较：

- flock 和 fcntl 是系统调用，而 lockf 是库函数。lockf 实际上是 fcntl 的封装
- flock 函数只能对整个文件上锁，而不能对文件的某一部分上锁，fcntl/lockf 可以对文件的某个区域上锁。
- flock 只能产生建议性锁；可以有共享锁和排它锁，lockf 只支持排它锁，但是 fcntl 里面参数 flock 可以有 RDLCK 读锁。
- flock 不能在 NFS 文件系统上使用，如果要在 NFS 使用文件锁，请使用 fcntl。
- flock 来自 BSD 而 lockf 来自 POSIX，所以 lockf 或 fcntl 实现的锁在类型上又叫做 POSIX 锁

3. 结果、分析与建议

（1） 查看 flock 函数手册

```
[root@hadoop100 experiment_003]# man 2 flock
```

```
FLOCK(2)                    Linux Programmer's Manual                    FLOCK(2)

NAME
       flock - apply or remove an advisory lock on an open file

SYNOPSIS
       #include <sys/file.h>

       int flock(int fd, int operation);

DESCRIPTION
       Apply or remove an advisory lock on the open file specified by fd.  The argu-
       ment operation is one of the following:

           LOCK_SH  Place a shared lock.  More than one process may  hold  a  shared
                    lock for a given file at a given time.

           LOCK_EX  Place an exclusive lock.  Only one process may hold an exclusive
                    lock for a given file at a given time.

           LOCK_UN  Remove an existing lock held by this process.

       A call to flock() may block if  an  incompatible  lock  is  held  by  another
       process.   To make a nonblocking request, include LOCK_NB (by ORing) with any
       of the above operations.

       A single file may not simultaneously have both shared and exclusive locks.

       Locks created by flock() are associated with an open file table entry.   This
       means  that  duplicate  file descriptors (created by, for example, fork(2) or
       dup(2)) refer to the same lock, and this lock may  be  modified  or  released
       using  any of these descriptors.  Furthermore, the lock is released either by
       an explicit LOCK_UN operation on any of these duplicate descriptors, or  when
       all such descriptors have been closed.

       If a process uses open(2) (or similar) to obtain more than one descriptor for
       the same file, these descriptors are treated independently  by  flock().   An
       attempt to lock the file using one of these file descriptors may be denied by
       a lock that the calling process has already placed via another descriptor.

       A process may hold only one type of lock (shared or  exclusive)  on  a  file.
       Subsequent  flock()  calls on an already locked file will convert an existing
       lock to the new lock mode.

       Locks created by flock() are preserved across an execve(2).

       A shared or exclusive lock can be placed on a file regardless of the mode  in
       which the file was opened.

RETURN VALUE
       On  success,  zero  is  returned.  On error, -1 is returned, and errno is set
       appropriately.

ERRORS
       EBADF  fd is not an open file descriptor.

 Manual page flock(2) line 1 (press h for help or q to quit)
```

（2） 查看 lockf 函数手册

```
[root@hadoop100 experiment_003]# man 3 lockf
```

```
LOCKF(3)                    Linux Programmer's Manual                    LOCKF(3)

NAME
       lockf - apply, test or remove a POSIX lock on an open file

SYNOPSIS
       #include <unistd.h>

       int lockf(int fd, int cmd, off_t len);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       lockf():
           _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 500 ||
           _XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED

DESCRIPTION
       Apply, test or remove a POSIX lock on a section of an open file.  The file is
       specified  by  fd, a file descriptor open for writing, the action by cmd, and
       the section consists of byte positions pos..pos+len-1 if len is positive, and
       pos-len..pos-1  if  len  is negative, where pos is the current file position,
       and if len is zero, the section extends from the  current  file  position  to
       infinity,  encompassing the present and future end-of-file positions.  In all
       cases, the section may extend past current end-of-file.

       On Linux, lockf() is just an interface on  top  of  fcntl(2)  locking.   Many
       other  systems  implement  lockf()  in  this  way, but note that POSIX.1-2001
       leaves the relationship between lockf() and fcntl(2)  locks  unspecified.   A
       portable application should probably avoid mixing calls to these interfaces.

       Valid operations are given below:

       F_LOCK Set  an exclusive lock on the specified section of the file.  If (part
              of) this section is already locked, the call blocks until the previous
              lock is released.  If this section overlaps an earlier locked section,
              both are merged.  File locks are released as soon as the process hold-
              ing  the  locks  closes  some  file  descriptor for the file.  A child
              process does not inherit these locks.

       F_TLOCK
              Same as F_LOCK but the call never blocks and returns an error  instead
              if the file is already locked.

       F_ULOCK
              Unlock  the  indicated  section  of the file.  This may cause a locked
              section to be split into two locked sections.

       F_TEST Test the lock: return 0 if the specified section is unlocked or locked
              by  this process; return -1, set errno to EAGAIN (EACCES on some other
              systems), if another process holds a lock.

RETURN VALUE
       On success, zero is returned.  On error, -1 is returned,  and  errno  is  set
       appropriately.

ERRORS
 Manual page lockf(3) line 1 (press h for help or q to quit)
```

（3） 查看 fcntl 手册

```
[root@hadoop100 experiment_003]# man fcntl
```

```
FCNTL(2)                    Linux Programmer's Manual                    FCNTL(2)

NAME
       fcntl - manipulate file descriptor

SYNOPSIS
       #include <unistd.h>
       #include <fcntl.h>

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

DESCRIPTION
       fcntl() performs one of the operations described below on the open file
       descriptor fd.  The operation is determined by cmd.

       fcntl() can take an optional third argument.  Whether or not this argument is
       required is determined by cmd.  The required argument type is indicated in
       parentheses after each cmd name (in most cases, the required type is int, and
       we identify the argument using the name arg), or void is specified if the
       argument is not required.

   Duplicating a file descriptor
       F_DUPFD (int)
              Find the lowest numbered available file descriptor greater than or
              equal to arg and make it be a copy of fd.  This is different from
              dup2(2), which uses exactly the descriptor specified.

              On success, the new descriptor is returned.

              See dup(2) for further details.

       F_DUPFD_CLOEXEC (int; since Linux 2.6.24)
              As for F_DUPFD, but additionally set the close-on-exec flag for the
              duplicate descriptor.  Specifying this flag permits a program to avoid
              an additional fcntl() F_SETFD operation to set the FD_CLOEXEC flag.
              For an explanation of why this flag is useful, see the description of
              O_CLOEXEC in open(2).

   File descriptor flags
       The following commands manipulate the flags associated with a file descrip-
       tor.  Currently, only one such flag is defined: FD_CLOEXEC, the close-on-exec
       flag.  If the FD_CLOEXEC bit is 0, the file descriptor will remain open
       across an execve(2), otherwise it will be closed.

       F_GETFD (void)
              Read the file descriptor flags; arg is ignored.

       F_SETFD (int)
              Set the file descriptor flags to the value specified by arg.

   File status flags
       Each open file description has certain associated status flags, initialized
       by open(2) and possibly modified by fcntl().  Duplicated file descriptors
       (made with dup(2), fcntl(F_DUPFD), fork(2), etc.) refer to the same open file
       description, and thus share the same file status flags.

Manual page fcntl(2) line 1 (press h for help or q to quit)
```

（4） 编写程序练习文件锁

- 编写 lock-1.c 文件

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <wait.h>

#define PATH "/root/course/experiments/experiment_003/lock"
int main()
{
    int fd;
    pid_t pid;

    fd = open(PATH, O_RDWR|O_CREAT|O_TRUNC, 0644);
    if (fd < 0) {
        perror("open()");
        exit(1);
    }

    if (flock(fd, LOCK_EX) < 0) {
        perror("flock()");
        exit(1);
    }
    printf("%d: locked!\n", getpid());

    pid = fork();
    if (pid < 0) {
        perror("fork()");
        exit(1);
    }

    if (pid == 0) {
/*
 *          fd = open(PATH, O_RDWR|O_CREAT|O_TRUNC, 0644);
 *                  if (fd < 0) {
 *                                  perror("open()");
 *                                                  exit(1);
 *                                                          }
 *                                                          */
        if (flock(fd, LOCK_EX) < 0) {
            perror("flock()");
            exit(1);
        }
        printf("%d: locked!\n", getpid());
        exit(0);
    }
    wait(NULL);
    unlink(PATH);
    sleep(10);
    exit(0);
}

~
~
"lock-1.c" 54L, 1176C
```

- 编译运行

```
[root@hadoop100 experiment_003]# gcc lock-1.c -o lock-1
[root@hadoop100 experiment_003]# # ./lock-1&
```

```
[root@hadoop100 experiment_003]# lslocks | grep lock
codeblocks        3494 POSIX   5B WRITE 0          0          0 /tmp/Code::Blocks-root
master            1393 FLOCK  33B WRITE 0          0          0 /var/lib/postfix/master
lock
```

- 去掉注释重新编译运行

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <wait.h>

#define PATH "/root/course/experiments/experiment_003/lock"
int main()
{
    int fd;
    pid_t pid;

    fd = open(PATH, O_RDWR|O_CREAT|O_TRUNC, 0644);
    if (fd < 0) {
        perror("open()");
        exit(1);
    }

    if (flock(fd, LOCK_EX) < 0) {
        perror("flock()");
        exit(1);
    }
    printf("%d: locked!\n", getpid());

    pid = fork();
    if (pid < 0) {
        perror("fork()");
        exit(1);
    }

    if (pid == 0) {
          fd = open(PATH, O_RDWR|O_CREAT|O_TRUNC, 0644);
                  if (fd < 0) {
                                  perror("open()");
                                                  exit(1);
                                                          }

        if (flock(fd, LOCK_EX) < 0) {
            perror("flock()");
            exit(1);
        }
        printf("%d: locked!\n", getpid());
        exit(0);
    }
    wait(NULL);
    unlink(PATH);
    sleep(10);
    exit(0);
}
~
~
~
~
```

```
[root@hadoop100 experiment_003]# gcc lock-2.c -o lock-2
[root@hadoop100 experiment_003]# ./lock-2&
[1] 4032
[root@hadoop100 experiment_003]# 4032: locked!
```

```
[root@hadoop100 experiment_003]# lslocks | grep lock
codeblocks       3494 POSIX   5B WRITE  0          0          0 /tmp/Code::Blocks-root
lock-2           4033 FLOCK   0B WRITE* 0          0          0 /root/course/experiment
s/experiment_003/lock
lock-2           4032 FLOCK   0B WRITE  0          0          0 /root/course/experiment
s/experiment_003/lock
master           1393 FLOCK  33B WRITE  0          0          0 /var/lib/postfix/master
.lock
```

```
[root@hadoop100 experiment_003]# ps
    PID TTY          TIME CMD
   3619 pts/1    00:00:00 bash
   4032 pts/1    00:00:00 lock-2
   4033 pts/1    00:00:00 lock-2
   4078 pts/1    00:00:00 ps
```

```
[root@hadoop100 experiment_003]# kill -9 4032
[root@hadoop100 experiment_003]# ps
   PID TTY          TIME CMD
  3619 pts/1    00:00:00 bash
  4033 pts/1    00:00:00 lock-2
  4093 pts/1    00:00:00 ps
[1]+  已杀死               ./lock-2
[root@hadoop100 experiment_003]# lslocks | grep lock
codeblocks       3494 POSIX   5B WRITE  0          0          0 /tmp/Code::Blocks-root
lock-2           4033 FLOCK   0B WRITE* 0          0          0 /root/course/experiment
s/experiment_003/lock
master           1393 FLOCK  33B WRITE  0          0          0 /var/lib/postfix/master
.lock
[root@hadoop100 experiment_003]#
```

- 用 lockf(fd, F_LOCK, 0)替换上述文件中的 flock，并恢复原来的注释（子进程中打开文件），再重新测试。

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <wait.h>

#define PATH "/root/course/experiments/experiment_003/lock"
int main()
{
    int fd;
    pid_t pid;

    fd = open(PATH, O_RDWR|O_CREAT|O_TRUNC, 0644);
    if (fd < 0) {
        perror("open()");
        exit(1);
    }

    if (lockf(fd, F_LOCK, 0) < 0) {
        perror("flock()");
        exit(1);
    }
    printf("%d: locked!\n", getpid());

    pid = fork();
    if (pid < 0) {
        perror("fork()");
        exit(1);
    }

    if (pid == 0) {
/*
 *         fd = open(PATH, O_RDWR|O_CREAT|O_TRUNC, 0644);
 *                 if (fd < 0) {
 *                                 perror("open()");
 *                                                 exit(1);
 *                                                                 }
 *                                                                 */
        if (lockf(fd, F_LOCK, 0) < 0) {
            perror("flock()");
            exit(1);
        }
        printf("%d: locked!\n", getpid());
        exit(0);
    }
    wait(NULL);
    unlink(PATH);
    sleep(10);
    exit(0);
}

~
```

```
[root@hadoop100 experiment_003]# gcc lock-3.c -o lock-3
[root@hadoop100 experiment_003]# ./lock-3&
[1] 4135
[root@hadoop100 experiment_003]# 4135: locked!
```

```
[root@hadoop100 experiment_003]# lslocks |grep lock
codeblocks       3494 POSIX   5B WRITE  0          0          0 /tmp/Code::Blocks-root
lock-2           4033 FLOCK   0B WRITE* 0          0          0 /root/course/experiment
s/experiment_003/lock
lock-3           4136 POSIX   0B WRITE* 0          0          0 /root/course/experiment
s/experiment_003/lock
lock-3           4135 POSIX   0B WRITE  0          0          0 /root/course/experiment
s/experiment_003/lock
master           1393 FLOCK  33B WRITE  0          0          0 /var/lib/postfix/master
.lock
[root@hadoop100 experiment_003]#
```

- 用 fcntl 实现强制性锁

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char **argv) {
  if (argc > 1) {
    int fd = open(argv[1], O_WRONLY);
    if(fd == -1) {
      printf("Unable to open the file\n");
      exit(1);
    }
    static struct flock lock;

    lock.l_type = F_WRLCK;
    lock.l_start = 0;
    lock.l_whence = SEEK_SET;
    lock.l_len = 0;
    lock.l_pid = getpid();

    int ret = fcntl(fd, F_SETLKW, &lock);
    printf("Return value of fcntl:%d\n",ret);
    if(ret==0) {
      while (1) {
        scanf("%c", NULL);
      }
    }
  }
}
```

```
[root@hadoop100 experiment_003]# gcc mlock.c -o mlock
```

4. 附录(如源程序)
   源码如上截图