数据管理

张海宁

贵州大学

hnzhang1@gzu.edu.cn

May 9, 2018

Overview

内存管理

内存管理概述

在所有的计算机系统中，内存都是一种稀缺资源。而且无论有多少内存，似乎总是不够用，这就对内存的管理提出了挑战。
联想一下电脑和手机的内存增长史，应用程序所需要占用的内存也在随着内存容量的增长而增加。

内存管理 I

malloc、free

```
#include<stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

The malloc() function allocates size bytes of memory and returns a pointer to the allocated memory.
The free() function deallocates the memory allocation pointed to by ptr. If ptr is a NULL pointer, no operation is performed.
A call to free should be made only with a pointer to memory allocated by a call to malloc, calloc, or realloc.

内存管理 II

分配、释放内存以及内存的合理使用

```
int main(){
 char * ch,*tmp;
 ch = (char *) malloc(1);
 tmp = (char *) malloc(1);
 printf("the addr of ch is:%p, tmp is %p\n",ch,tmp);
 sprintf(tmp,"java.\n");
 sprintf(ch,"hello mem, it may be breaks other address.\n");
 printf("the ch is: %s",ch);
 printf("the tmp is: %s",tmp);
 free(ch);
 free(tmp);
 exit(0);
}
```

Question: 这段代码的输出会是什么样的？

内存管理 III

内存的合理使用

```
int main(){
 char * ch2="h";
 char * tmp2="java.";
 printf("the addr of ch2 is:%p, tmp2 is %p\n",ch2,tmp2);
 ch2="hello mem, it may be breaks other address.";
 printf("the ch2 is: %s\n",ch2);
 printf("the tmp2 is: %s\n",tmp2);
 printf("the ch2 addr %p %p %p\n",&ch2[0],&ch2[1],&ch[3]);
 printf("the tmp2 addr %p %p\n",&tmp2[0],&tmp2[1]);
 exit(0);
}
```

Question: 这段代码的输出又会是什么样的？为什么与第6页不同？

文件锁定

文件锁定

文件锁定是多用户多任务操作系统中一个非常重要的组成部分。如果两个程序同时访问一个文件，一个读一个写，或者都在写，那么非常有可能出现文件内容不一致。Linux 使用文件锁定来解决这个问题。实现文件锁定有两种形式：

1. 创建锁文件
2. 锁定区域

创建锁文件 I

许多应用程序只要能够针对某个资源创建一个锁文件即可，然后其他的程序就可以通过检查这个文件的状态来判断它们自己是否被允许访问这个资源。
为了创建一个用途锁指示器的文件，使用 fcntl.h 头文件中定义的带 O_CREAT 和 O_EXECL 标志的 open 系统调用。这样能保证以一个原子操作同时完成两项工作：确定文件不存在，然后创建它。

O_EXCL error if O_CREAT and the file exists

创建锁文件 II

创建锁文件部分代码

```
int main(){
 int file_desc, save_errno;
  file_desc=open("LCK.test",O_RDWR|O_CREAT|O_EXCL,
     S_IRUSR);
  if(file_desc==-1){
   save_errno=errno;
   printf("Open failed with error code %d.\n",
     save_errno);
  }else{
   printf("Open succeeded.\n");
  }
 exit(EXIT_SUCCESS);
}
```

创建锁文件 III

执行第11页代码

```
$ ./lock1
Open succeeded.
$ ./lock1
Open failed with error code 17.
```

可以看到，第一次运行程序，会成功打开文件，第二次运行就会打开失败，其实不管再运行几次，都会失败。Why?
关于 errno 的更多信息可以查看第60页。

创建锁文件 IV

临界区：如果某个程序在执行的时候需要独占某个资源进行某操作，这部分操作被称为临界区，程序在进入临界区之前需要创建锁文件，在退出临界区之前需要删除（unlink）锁文件。

临界区代码

```
void writeAfile(){
 FILE *out = fopen("lcktest2","a");
 char st='a', end='z', c;
 for(c=st;c<end;c++){
  fputc(c,out);  fflush(out);
 }
 fputc('\n',out); fclose(out);
 printf("write done.\n");
}
```

创建锁文件 V

普通模式访问代码

```
int main(){
  int file_desc, tries=2;
  while(tries--){
    writeAfile();
  }
  exit(EXIT_SUCCESS);
}
```

创建锁文件 VI

使用锁文件方式

```
int main(){
 int file_desc, tries=2;
 while(tries--){
  file_desc=open(lock_file,O_RDWR|O_CREAT|O_EXCL,
    S_IRUSR);
  if(file_desc==-1){
   printf("%d : Lock already present.\n",getpid());
   sleep(3);   tries++;
  }else{
   printf("%d : Got access.\n",getpid());
   writeAfile(); close(file_desc);
   unlink(lock_file); sleep(1);
  } } exit(EXIT_SUCCESS);}
```

创建锁文件 VII

查看结果

普通方式执行程序

```
$./lock2n & ./lock2n
aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyy

abcdeafbcdhefgiijklmnopqjrksltmnopqrstuvwxy
uvwxy
```

创建锁文件方式执行程序

```
$./lock2 & ./lock2
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxy
```

创建锁文件 VIII

通过第16页，可以看出，通过创建锁文件的方式，达到了一个程序单独占用某个文件的目的，确保了同一时间段只能有一个程序来独占一个文件进行操作。

锁定区域

创建锁文件的方式会造成对文件的独占式访问，这不适合用来访问大型共享文件。比如一个日志文件，随时都可能会有数据写入，为了让其他程序也能读取这个不会停止写入的文件，需要有一种协调机制来提供对一个文件的同时访问。
文件中的锁定区域可以用来解决这个问题，即文件的一部分被锁定，但其他程序可以访问这个文件的其他部分。这种机制被称为文件段锁定或文件区锁定。

两个系统调用

1. lockf
2. fcntl
   最常用

fcntl

原型

```
#include<fcntl.h>

int fcntl(int fildes, int cmd, ...);
```

fcntl 对一个打开的文件描述符进行操作，并根据 cmd 参数的设置完成不同的任务。其有三个用于文件锁的命令：

1. F_GETLK 获取锁信息。若失败，返回-1。
2. F_SETLK 设置锁。若失败，返回-1。
3. F_SETLKW 设置锁，直到成功。

F_GETLK

fcntl 这个系统调用可以使用 F_GETLK 这个参数来获取文件某一区域当前的锁定状态。其返回值有两种：

1. -1
   调用失败，表明 F_GETLK 无法获取信息。
2. 非-1
   调用成功，表明 F_GETLK 获取到了锁信息。
   - 如果文件的该区域已经存在一个锁的话，F_GETLK 会将本次调用中的 flock 结构中的 l_pid 修改为锁定进程的进程号。
   - 若文件的该区域没有锁，则不做修改。

flock

当使用第19页这些参数的时候，第三个参数必须是一个指向 flock 结构的指针，fcntl 完整的原型应该是：

```
int fcntl(int fildes, int cmd, struct flock *flk)
```

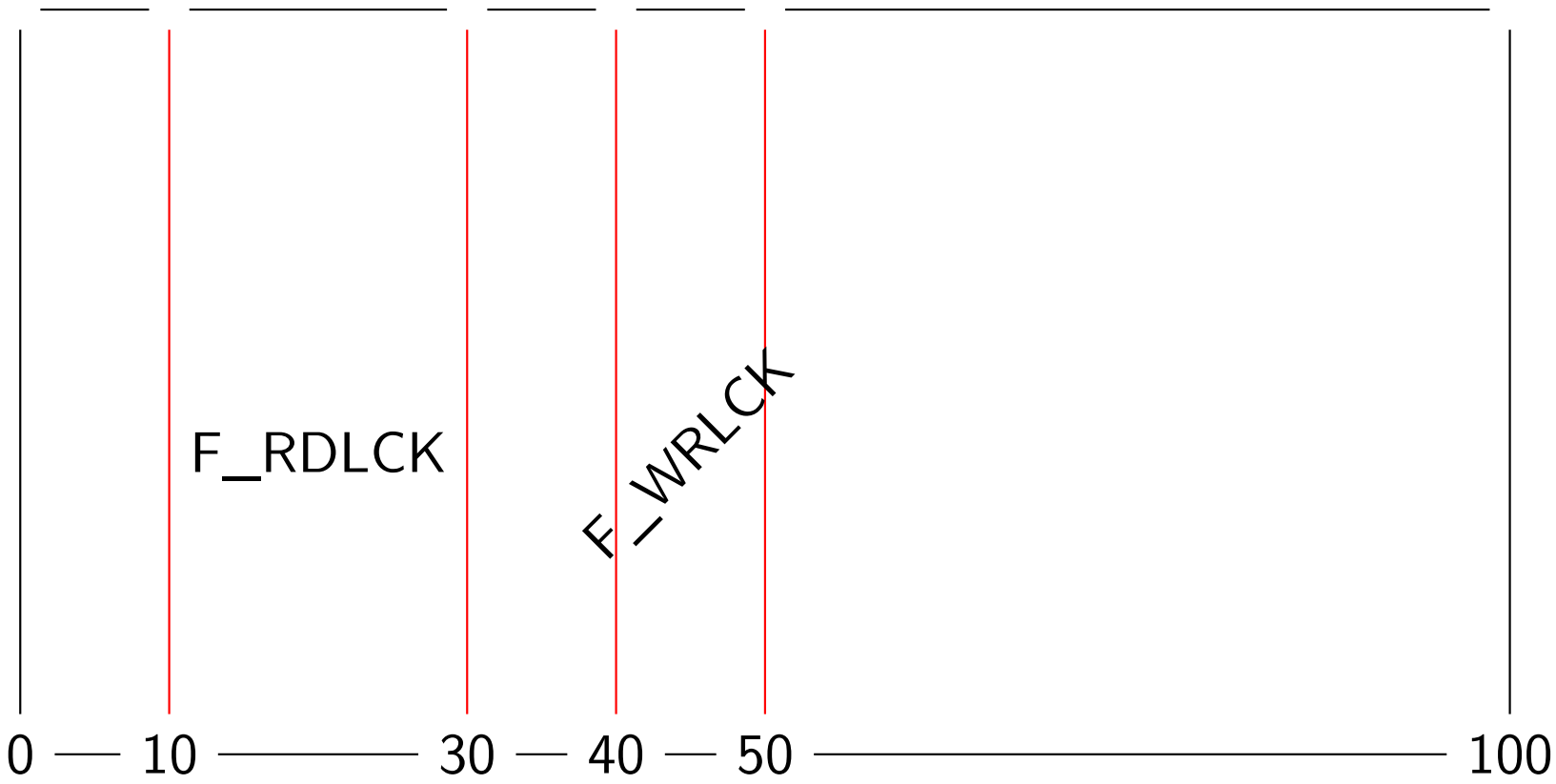
flock(file lock) 结构依赖于具体的实现，但至少包含以下成员：

1. short l_type;
   - F_RDLCK 共享（读）锁。
   - F_UNLCK 解锁。
   - F_WRLCK 独占（写）锁。
2. short l_whence SEEK_SET SEEK_CUR SEEK_END。
3. off_t l_start 相对于 whence 的偏移量（字节）。
4. off_t l_len 字节个数。
5. pid_t l_pid 记录持有锁的进程号。

锁定状态下的读写操作

当对文件的区域加锁之后，访问文件应当使用 read 和 write 调用，不要使用 fread 和 fwrite，因为后者会缓存比要求更多的数据。

使用 fcntl 锁定文件 I



使用 fcntl 锁定文件 II

头文件及声明

```
//lock3.c
#include<unistd.h>
#include<fcntl.h>
#include<stdlib.h>
#include<stdio.h>
const char *test_file="lck3.lock";
int main(){
 int file_desc;
 int byte_count;
 char * byte_to_write="A";
 struct flock region_1;
 struct flock region_2;
 int res;
```

使用 fcntl 锁定文件 III

打开一个文件

```
//open a file discription
 file_desc=open(test_file, O_RDWR | O_CREAT, 0666);
 if(!file_desc){
  fprintf(stderr,"Unable to open %s.\n",test_file);
  exit(EXIT_FAILURE);
 }
//  write some data
 for(byte_count=0;byte_count<100;byte_count++){
  write(file_desc, byte_to_write, 1);
 }
```

使用 fcntl 锁定文件 IV

设置锁信息

```
//set region 1 as read lock
 region_1.l_type = F_RDLCK;
 region_1.l_whence = SEEK_SET;
 region_1.l_start = 10;
 region_1.l_len=20;

//set region 2 as write lock
 region_2.l_type = F_WRLCK;
 region_2.l_whence = SEEK_SET;
 region_2.l_start = 40;
 region_2.l_len=10;
```

使用 fcntl 锁定文件 V

锁定文件

```
//lock file
 printf("Process %d locking file.\n",getpid());
 res = fcntl(file_desc, F_SETLK, &region_1);
 if(res==-1){
  fprintf(stderr,"proc %d failed to lock region 1.\n"
     , getpid());
 }
 res = fcntl(file_desc, F_SETLK, &region_2);
 if(res==-1){
  fprintf(stderr,"proc %d failed to lock region 2.\n"
     , getpid());
 }
```

使用 fcntl 锁定文件 VI

关闭文件

```
//sleep
  sleep(20);
  printf("process %d prepare to close the file.\n"
     , getpid());
  close(file_desc);
  exit(EXIT_SUCCESS);
}
```

程序对某个文件拥有的所有锁都将在相应的文件描述符关闭时自动清除。

测试文件上的锁

编写一个 lock4.c 文件，在 lock3.c 的程序在文件 lck3.lock 上设置锁的同时，使用 lock4.c 的程序再加锁，对比结果。

测试文件上的锁 I

头文件及声明

```
//lock4.c
#include<unistd.h>
#include<fcntl.h>
#include<stdlib.h>
#include<stdio.h>
const char *test_file="lck3.lock";
#define SIZE_TO_TRY 5
void show_lock_info(struct flock *to_show);
int main(){
 int file_desc;
 struct flock region_to_test;
 int res;
 int start_byte;
```

测试文件上的锁 II

打开一个文件，并开始一个循环

```
//open a file discription
 file_desc=open(test_file, O_RDWR | O_CREAT, 0666);
 if(!file_desc){
  fprintf(stderr,"Unable to open %s.\n",test_file);
  exit(EXIT_FAILURE);
 }
 for(start_byte=0;start_byte<99;
   start_byte+=SIZE_TO_TRY){
```

测试文件上的锁 III

设置写锁

```
//set a write lock
 region_to_test.l_type = F_WRLCK;
 region_to_test.l_whence = SEEK_SET;
 region_to_test.l_start = start_byte;
 region_to_test.l_len=SIZE_TO_TRY;
 region_to_test.l_pid=-1;
```

测试文件上的锁 IV

测试写锁

```
//test the write lock can be set or not
 printf("Testing the F_WRLCK on region from %d to %d
  \n",start_byte,start_byte+SIZE_TO_TRY);
 res=fcntl(file_desc,F_GETLK,&region_to_test);
 if(res==-1){
  fprintf(stderr,"F_GETLK(F_WRLCK) failed!\n");
  exit(EXIT_FAILURE);
 }
 if(region_to_test.l_pid!=-1){
  printf("F_WRLCK would fail.F_GETLK returned:\n");
  show_lock_info(&region_to_test);
 }else{
  printf("F_WRLCK - lock would succeed.\n");
 }
```

测试文件上的锁 V

设置读锁

```
//set a read lock
 region_to_test.l_type = F_RDLCK;
 region_to_test.l_whence = SEEK_SET;
 region_to_test.l_start = start_byte;
 region_to_test.l_len=SIZE_TO_TRY;
 region_to_test.l_pid=-1;
```

测试文件上的锁 VI

测试读锁

```
//test the read lock can be set or not
 printf("Testing the F_RDLCK on region from %d to %d
  \n",start_byte,start_byte+SIZE_TO_TRY);
 res=fcntl(file_desc,F_GETLK,&region_to_test);
 if(res==-1){
  fprintf(stderr,"F_GETLK(F_RDLCK) failed!\n");
  exit(EXIT_FAILURE);
 }
 if(region_to_test.l_pid!=-1){
  printf("F_RDLCK would fail.F_GETLK returned:\n");
  show_lock_info(&region_to_test);
 }else{
  printf("F_RDLCK - lock would succeed.\n");
 }
```

测试文件上的锁 VII

关闭文件及函数实现

```
  }
  close(file_desc);
  exit(EXIT_SUCCESS);
}
void show_lock_info(struct flock *to_show){
  printf("\tl_type %d, ",to_show->l_type);
  printf("l_whence %d, ",to_show->l_whence);
  printf("l_start %d, ",(int)to_show->l_start);
  printf("l_len %d, ",(int)to_show->l_len);
  printf("l_pid %d\n ",to_show->l_pid);
}
```

测试文件上的锁 VIII

先使 lock3 在后台运行，保持对文件的锁定，随即运行 lock4 程序，对同一文件的各部分依次进行加锁测试。

运行 lock3 lock4

```
$ ./lock3 &
$ ./lock4
```

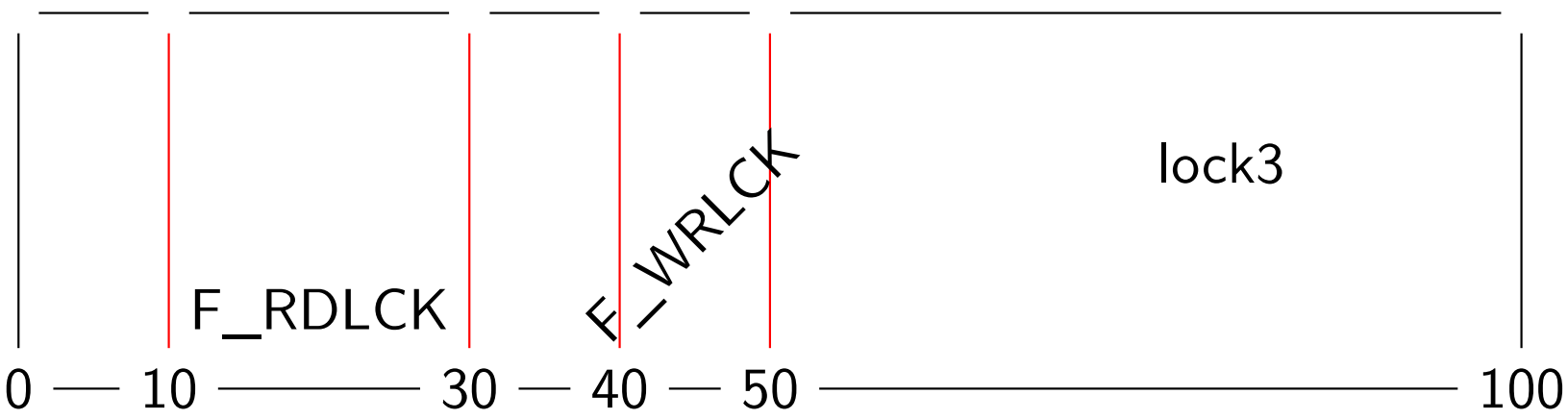
测试文件上的锁 IX

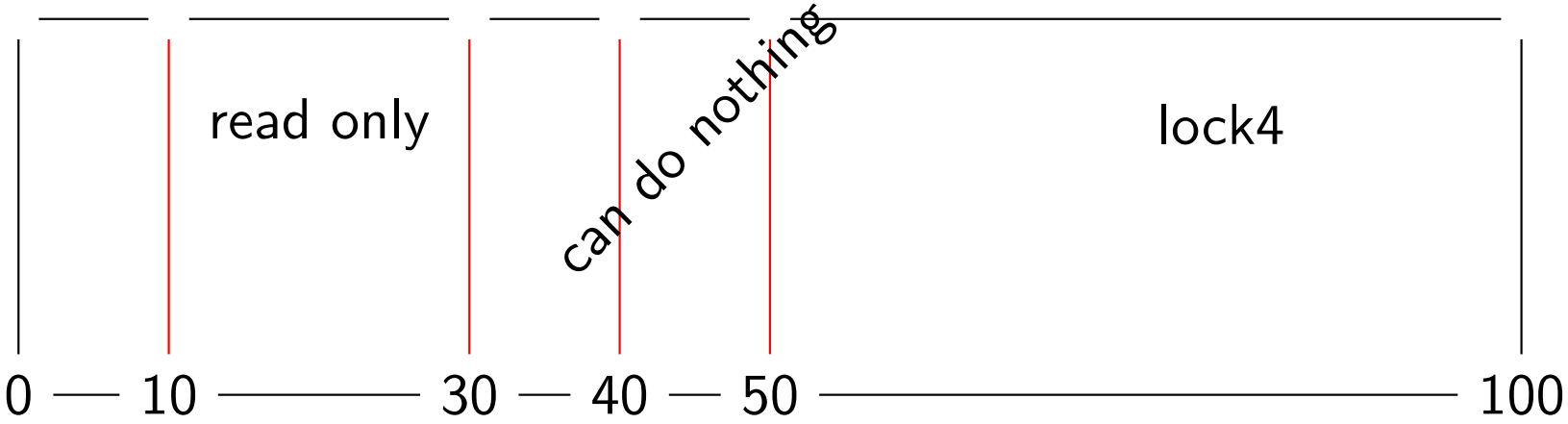
部分程序输出

```
Testing the F_WRLCK on region from 10 to 15
F_WRLCK would fail.F_GETLK returned:
l_type 1, l_whence 0, l_start 10, l_len 20, l_pid 4732
Testing the F_RDLCK on region from 10 to 15
F_RDLCK - lock would succeed.

Testing the F_WRLCK on region from 40 to 45
F_WRLCK would fail.F_GETLK returned:
l_type 3, l_whence 0, l_start 40, l_len 10, l_pid 4732
Testing the F_RDLCK on region from 40 to 45
F_RDLCK would fail.F_GETLK returned:
l_type 3, l_whence 0, l_start 40, l_len 10, l_pid 4732
```

测试文件上的锁 X


F_RDLCK
F_WRLCK
lock3
0
10
30
40
50
100


read only
can do nothing
lock4
0
10
30
40
50
100

文件锁的竞争

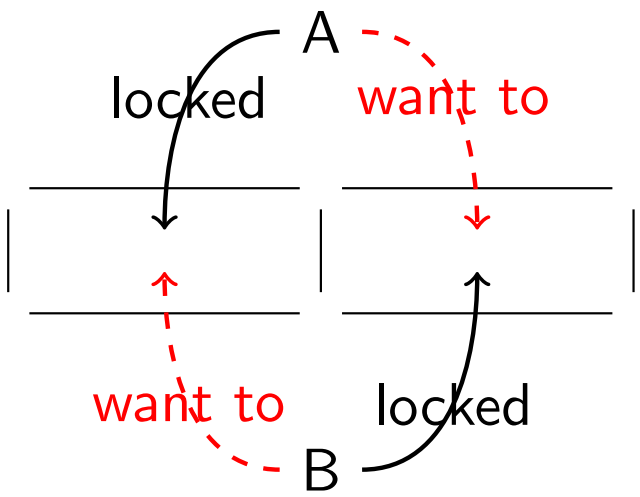
针对某一文件的特定区域，存在三种状态：

1. 无锁
   新程序可以随意加锁
2. have a read lock(shared lock)
   可以加 read lock
3. have a write lock(exclusive lock)
   新程序不可加锁

针对解锁有个特别要说明的是：即使一个程序对某一区域不拥有任何锁，同时不管该区域拥有什么锁，这个程序针对这一区域的解锁操作都会返回成功，因为对于本程序来说，只要解锁操作的结果是本身对文件的特定区域不拥有任何锁，那么这个解锁操作就是成功的。

死锁

假设 AB 两个程序需要同时更新字节 1 和字节 2，程序 A 先锁定了字节 1，程序 B 先锁定了字节 2 。然后程序 A 尝试锁定字节 2，而此时字节 2 正在被程序 B 锁定，A 只好继续锁定字节 1，等待机会锁定字节 2；而程序 B 相反，它锁定了字节 2，等待机会去锁定字节 1 。这就出现了 A 和 B 各处占有，又各自等待的情况，最终的结果就是两个程序都无法继续执行下去，这种现象就是死锁。



真的锁住了？

fcntl 的锁只是建议性的锁，并不会真正的阻止我们从文件中读写数据，对锁的检测是程序的责任。
规范性。

数据库

dbm

dbm 是 Linux 平台和 Unix-like 平台上的一个基础的、但是非常有效的与数据存储有关的函数库，通常称为 dbm 数据库。dbm 数据库很适合用来存储索引化的静态数据，
一些使用 RPM 包管理机制的 linux 发行版，比如 Red Hat 和 SUSE 使用 dbm 来进行包信息的存储管理。

dbm 的结构

与 MySQL 等数据库不同，dbm 不使用表来存储数据。其使用一种叫做datum的结构来存储数据。一个 dbm 数据库中包含若干个“索引-数据”对，每个索引或数据都是一个 datum 结构的数据。datum 的具体实现不尽相同，但至少会包含以下两个内容：

1. void *dptr;
   dptr 指向数据的开始。
2. size_t dsize
   指明数据的大小。

就像打开或关闭文件需要使用 FILE 结构一样，使用 dbm 数据库时，需要使用DBM这种结构。

dbm 常用函数

dbm_open 函数原型

```
#include <ndbm.h>
DBM *dbm_open(const char *filename,
    int file_open_flags, mode_t file_mode);
```

dbm_open 函数用来打开一个存在的数据库或都创建一个数据库。
其接收的第一个参数是一个数据库文件名; 第二个参数和第三个参数与打开文件的 open 系统调用是一致的。
其返回值是一个 DBM 类型的指针，以供接下来访问数据库使用。如果失败的话，会返回 0 。

dbm 常用函数

dbm_store 函数原型

```
#include <ndbm.h>
int dbm_store(DBM *database_descriptor,
   datum key, datum content, int store_mode);
```

dbm_store 是往数据库里存储数据的。为了存储数据，需要定义两个 datum 结构：一个用来放索引，一个用来放实际的数据。
store_mode 是用来声明如果一个索引已经存在的话，本次的存储数据操作会如何应对：如果是 DBM_INSERT，那么本次操作会失败，并返回 1; 如果是 DBM_REPLACE，新数据会覆盖旧数据，并返回 0 。如果有其他错误的话，会返回负值。

dbm 常用函数

dbm_fetch 函数原型

```
#include <ndbm.h>
datum dbm_fetch(DBM *database_descriptor, datum key);
```

dbm_fetch 用于从数据库中查询数据。如果与 key 相关的数据被查询到，返回的 datum 结构中的 dptr 和 dsize 会根据实际的数据设置，如果没有找到，那么 datum 中的 dptr 会被设置成 NULL。

dbm 常用函数

dbm_close 函数原型

```
#include <ndbm.h>
void dbm_close(DBM *database_descriptor);
```

关闭数据库。

创建一个 dbm 数据库

创建数据库

```
#include<ndbm.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(){
//create a db
DBM *db = dbm_open("mydbm",O_CREAT|O_RDWR,0666);
if(!db){
  printf("the dbm database open failure.");
}else{
  printf("the dbm database open successed.\n");
}
dbm_close(db);
exit(0);
```

插入数据

插入数据

```
//insert a recode
datum key,value;
char * k="gzu";
char * v="guizhou university";
key.dptr=k;  key.dsize=strlen(k);
value.dptr=v;  value.dsize=strlen(v);
int rz = dbm_store(db,key,value,DBM_INSERT);
if(rz==0){
 printf("the data insert successed.\n");
 dbm_close(db);
}else{
 printf("the data insert failure.\n");
 dbm_close(db);
}
```

查询数据

查询数据

```
//retrive a recode
datum valueRetrive;
valueRetrive = dbm_fetch(db,key);
if(valueRetrive.dptr!=NULL){
 char * dt;
 strncpy(dt,valueRetrive.dptr,valueRetrive.dsize);
 //printf("the retrived data is: %s.\n",
   valueRetrive.dptr);
 printf("the retrived data is: %s.\n",dt);
 printf("the data longth is:%l.\n",valueRetrive.dsize)
}else{
 printf("no data was retrived.\n");
}
```

dbm 其他的一些函数

其他函数原型

```
#include <ndbm.h>
int dbm_delete(DBM *database_descriptor, datum key);
int dbm_error(DBM *database_descriptor);
int dbm_clearerr(DBM *database_descriptor);
datum dbm_firstkey(DBM *database_descriptor);
datum dbm_nextkey(DBM *database_descriptor);
```

delete,error,clearerr

- dbm_delete
  The dbm_delete function is used to delete entries from the database. It takes a key datum just like dbm_fetch but rather than retrieve the data, it deletes the data. It returns 0 on success.
- dbm_error
  The dbm_error function simply tests whether an error has occurred in the database, returning 0 if there is none.
- dbm_clearerr
  The dbm_clearerr clears any error condition flag that may be set in the database.

dbm_firstkey and dbm_nextkey

These routines are normally used as a pair to scan through all the keys of all the items in a database. The loop structure required is as follows:

dbm_firstkey and dbm_nextkey

```
DBM *db_ptr;
datum key;
for(key = dbm_firstkey(db_ptr); key.dptr;
   key = dbm_nextkey(db_ptr));
```

作业

1. 试回答第6、第7页的代码输出结果为什么不同[1]（下周三课前收作业）
2. 练习使用创建锁文件和锁定区域两种不同的形式对文件进行锁定
3. 练习 dbm 库的使用

[1]此网址的内容可能会有帮助：
https://www.cs.bu.edu/teaching/c/string/intro/

The End

Appendix

sprintf

Description

The C library function int sprintf(char *str, const char *format, ...) sends formatted output to a string pointed to, by str.

Declaration

Following is the declaration for sprintf() function.

```
int sprintf(char *str, const char *format, ...)
```

errno

关于文件操作相关的错误代码所代表的错误信息可以查看:

```
/usr/include/sys/errno.h

\#define EEXIST          17
/* File exists */
```

本课程相关资源下载

1. ppt
   https://github.com/gmsft/ppt/tree/master/linux
2. 实验指导书
   https://github.com/gmsft/ppt/tree/master/book/linux

about man page

The manual is generally split into eight numbered sections, organized as follows (on Research Unix, BSD, macOS and Linux):

section	description
1	General commands
2	System calls
3	Library function(C standard library)
4	Special files(devices) and drivers
5	File formats and conventions
6	Games and screensavers
7	Miscellanea
8	System administration commands and daemons

Table: man page

在终端中运行 man read 与 man 2 read ， 观察其输出的区别。