桂林电子科技大学

**实验六 文件系统编程**  实验报告

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | **实验6 文件系统编程** | | | | | | | |  | 辅导员意见：  成绩 辅导员  签 名 |
| 院 系 | 计算机与信息安全学院 | | | 专业 | | 计算机科学与技术 | | |
| 学 号 | 1600300128 | | | 姓名 | | 王涛 | | |
| 实验日期 | 2018 | 年 | 12 | | 月 | | 22 | 日 |
|  |  | | | | | | | |

## 实验目的

1、 理解Linux对文件系统的操作方法；

2、 理解库函数和系统调用的区别；

3、 掌握常用的与文件操作有关的库函数和系统调用。

## 实验要求

1、掌握常用的系统调用，包括creat、open、close、read、write、lseek、fcntl、chmod等；

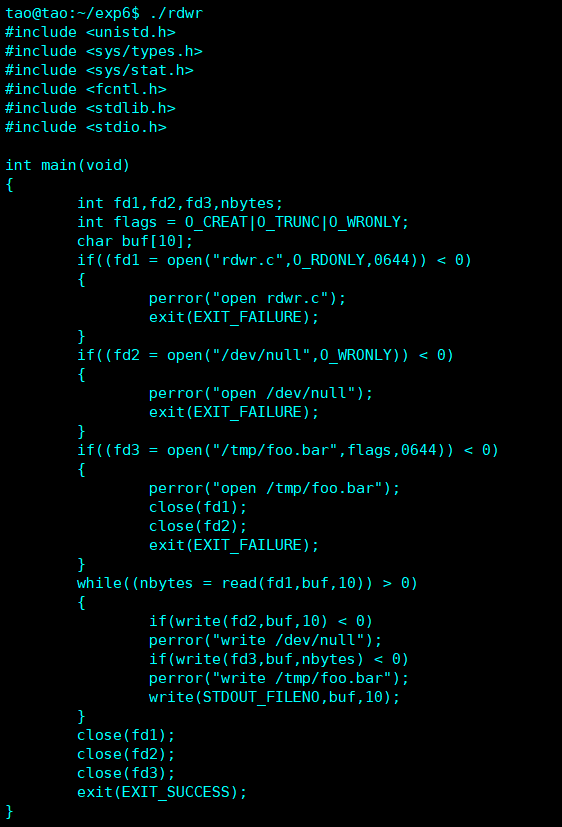
2、能够通过man命令理解系统调用中各个参数的含义，并应用在编程实践中。

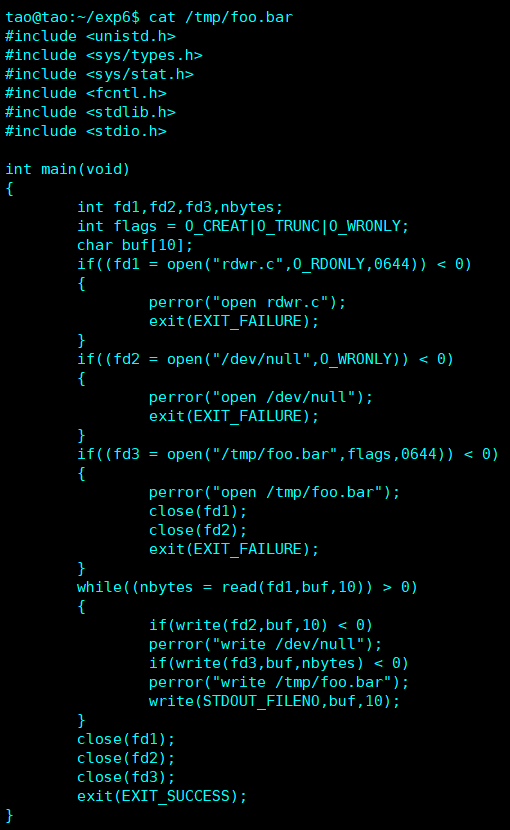
## 实验步骤

### 一、编辑、调试、运行以下程序

1. 编辑并运行程序rdwr.c，理解用到的系统调用，并解释说明该程序的功能。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | #include <unistd.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <stdlib.h>  #include <stdio.h>  int main(void)  {  int fd1,fd2,fd3,nbytes;  int flags = O\_CREAT|O\_TRUNC|O\_WRONLY;  char buf[10];  if((fd1 = open("rdwr.c",O\_RDONLY,0644)) < 0)  {  perror("open rdwr.c");  exit(EXIT\_FAILURE);  }  if((fd2 = open("/dev/null",O\_WRONLY)) < 0)  {  perror("open /dev/null");  exit(EXIT\_FAILURE);  }  if((fd3 = open("/tmp/foo.bar",flags,0644)) < 0)  {  perror("open /tmp/foo.bar");  close(fd1);  close(fd2);  exit(EXIT\_FAILURE);  }  while((nbytes = read(fd1,buf,10)) > 0)  {  if(write(fd2,buf,10) < 0)  perror("write /dev/null");  if(write(fd3,buf,nbytes) < 0)  perror("write /tmp/foo.bar");  write(STDOUT\_FILENO,buf,10);  }  close(fd1);  close(fd2);  close(fd3);  exit(EXIT\_SUCCESS);  } |

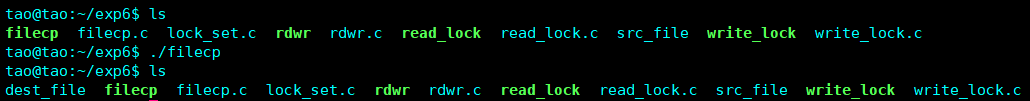




程序功能：打开rdwr.c文件，将其内容写入到/tmp/foo.bar中以及丢入/dev/null垃圾箱中。

2．编译、调试程序filecp.c使其能正常工作，运行之。解释这个程序的功能。注意观察BUFFER\_SIZE 和OFFSET对运行效率的影响

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | #include <unistd.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <stdlib.h>  #include <stdio.h>  #define BUFFER\_SIZE 1024  #define SRC\_FILE\_NAME "src\_file"  #define DEST\_FILE\_NAME "dest\_file"  #define OFFSET 10240  int main()  {  int src\_file,dest\_file;  unsigned char buff[BUFFER\_SIZE];  int real\_read\_len;    src\_file = open(SRC\_FILE\_NAME,O\_RDONLY);  dest\_file = open(DEST\_FILE\_NAME,O\_WRONLY|O\_CREAT,0644);  if(src\_file < 0 || dest\_file < 0)  {  printf("Open file error!\n");  exit(1);  }  lseek(src\_file,-OFFSET,SEEK\_END);  while((real\_read\_len = read(src\_file,buff,sizeof(buff))) > 0)  {  write(dest\_file,buff,real\_read\_len);  }  close(dest\_file);  close(src\_file);  return 0;  } |



程序功能：将src\_file文件的最后10240字节复制存为dest\_file文件

3、

（1）编写一个文件lock\_set.c，该程序利用fcntl函数实现了一个lock\_set函数，该函数实现文件锁功能，F\_RDLCK为读锁，F\_WRLCK为写锁，F\_UNLCK为解锁。

lock\_set.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | int lock\_set(int fd,int type)  {  struct flock old\_lock,lock;  lock.l\_whence = SEEK\_SET;  lock.l\_start = 0;  lock.l\_len = 0;  lock.l\_type = type;  lock.l\_pid = -1;  fcntl(fd,F\_GETLK,&lock);  if(lock.l\_type != F\_UNLCK)  {  if(lock.l\_type == F\_RDLCK)  printf("Read lock already set by %d\n",lock.l\_pid);  else if(lock.l\_type == F\_WRLCK)  printf("Write lock already set by %d\n",lock.l\_pid);  }  lock.l\_type = type;  if((fcntl(fd,F\_SETLKW,&lock)) < 0)  {  printf("Lock failed:type = %d\n",lock.l\_type);  return 1;  }  switch(lock.l\_type)  {  case F\_RDLCK:  printf("Read lock set by %d\n",getpid());  break;  case F\_WRLCK:  printf("Write lock set by %d\n",getpid());  break;  case F\_UNLCK:  printf("Release lock set by %d\n",getpid());  return 1;  break;  default:  break;  }  return 0;  } |

（2）编写一个文件read\_lock.c，该文件调用lock\_set函数对文件hello上读锁。

read\_lock.c

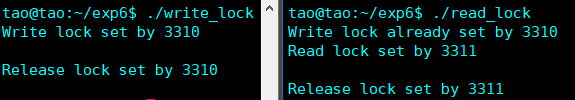
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | #include <unistd.h>  #include <sys/file.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <stdio.h>  #include <stdlib.h>  #include "lock\_set.c"  int main(void)  {  int fd;  fd = open("hello",O\_RDWR|O\_CREAT,0644);  if(fd<0)  {  printf("Open file error.\n");  exit(1);  }  lock\_set(fd,F\_RDLCK);  getchar();  lock\_set(fd,F\_UNLCK);  getchar();  close(fd);  exit(0);  } |

（3）编写一个文件write\_lock.c，该文件调用lock\_set函数对文件hello上写锁。

write\_lock.c

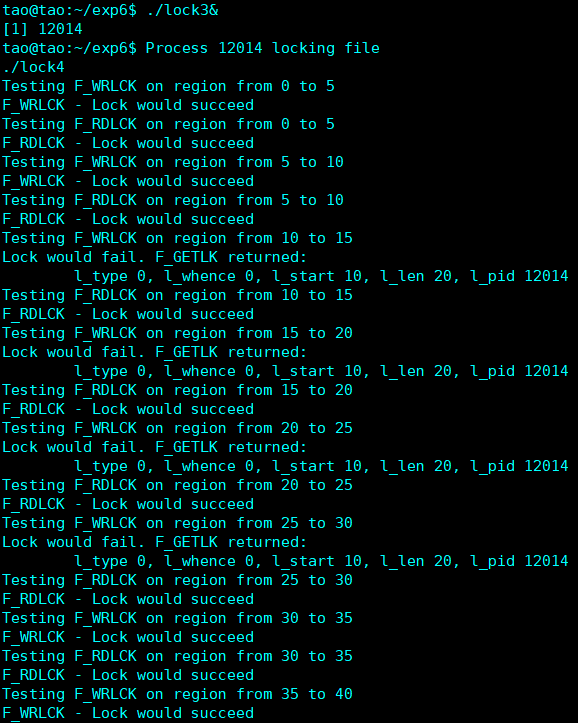
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | #include <unistd.h>  #include <sys/file.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <stdio.h>  #include <stdlib.h>  #include "lock\_set.c"  int main(void)  {  int fd;  fd = open("hello",O\_RDWR|O\_CREAT,0644);  if(fd<0)  {  printf("Open file error.\n");  exit(1);  }  lock\_set(fd,F\_WRLCK);  getchar();  lock\_set(fd,F\_UNLCK);  getchar();  close(fd);  exit(0);  } |

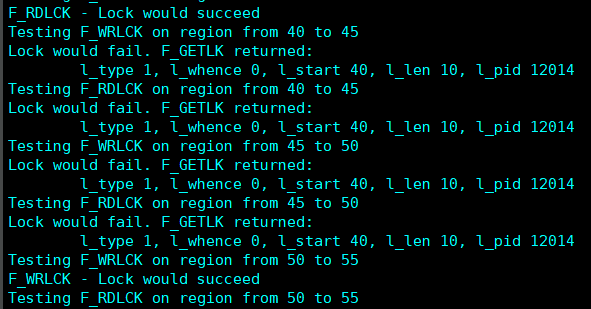
（4）在两个终端分别运行read\_lock.c和write\_lock.c，观察实验结果，并对实验结果进行解析。



先运行写锁程序，在3310进程设置了写锁，再运行读锁程序，此时写锁未释放需等待，等写锁释放之后读锁才被设置。

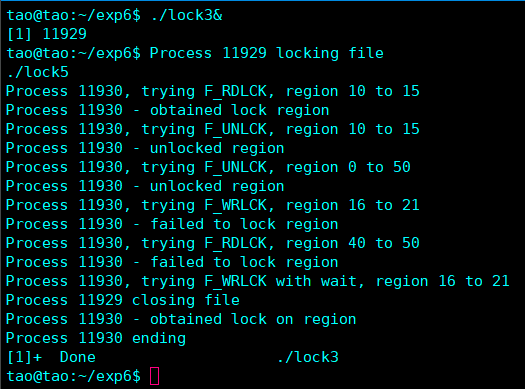
（5）运行lock3.c和lock4.c，对实验结果进行说明





说明：lock3在10-30阶段设置了共享读锁，40-50阶段设置了独占写锁。lock4在10-30阶段设置读锁时能成功，设置写锁不能成功，在40-50阶段读锁和写锁都不能设置。所以共享锁允许其他进程设置相同的锁，独占锁其他进程无法设置任何锁。

（6）运行lock3.c和lock5.c，对实验结果进行说明



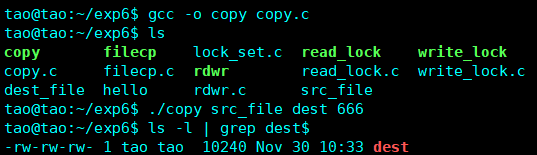
说明：lock3在10-30设置了共享读锁，40-50设置了独占写锁。lock5在10-15阶段设置了读锁，又释放了0-50阶段的所有的锁，但是在16-21阶段设置写锁时失败，在40-50阶段设置读锁时失败，可见此时lock3设置的锁并没有因为lock5释放所有锁而释放。等待lock3运行结束释放了锁，lock5才成功设置锁。所以，一个进程只能释放自己本进程设置的锁，不能释放其他进程设置的锁。

### 二、根据要求设计编写程序

（1）编写一个程序，把一个文件的内容复制到另一个文件，即实现简单的copy功能，并且修改该文件的访问权限。要求：只使用open、read、write、close、chmod等系统调用，程序的第一个参数是源文件，第二个参数是目的文件。

copy.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <unistd.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <stdlib.h>  #include <stdio.h>  #define BUFFER\_SIZE 1024  int main(int argc, char\*\* argv)  {  int src\_file,dest\_file;  unsigned char buff[BUFFER\_SIZE];  int real\_read\_len;  src\_file = open(argv[1],O\_RDONLY);  dest\_file = open(argv[2],O\_WRONLY|O\_CREAT);  if(src\_file < 0 || dest\_file < 0)  {  printf("Open file error!\n");  exit(1);  }  while((real\_read\_len = read(src\_file,buff,sizeof(buff))) > 0)  {  write(dest\_file,buff,real\_read\_len);  }  chmod(argv[2], strtoul(argv[3], NULL,8));  close(dest\_file);  close(src\_file);  return 0;  } |



（2）从文件file.in中读取最后10KB数据并复制到另一个文件，要求采用以下两种方式：

库函数方式

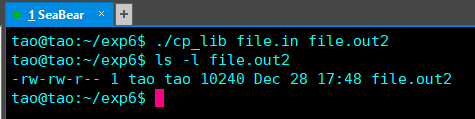
系统调用方式

设置相同的缓冲区，对上述两种方式的结果进行比较。

库函数方式：cp\_lib.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #include <stdio.h>  #define BUFFER\_SIZE 2048  #define OFFSET 10240  int main(int argc, char \*\*argv)  {  FILE \*src\_file=NULL,\*dest\_file=NULL;  unsigned char buff[BUFFER\_SIZE];  int real\_read\_len;    src\_file = fopen(argv[1],"r");  dest\_file = fopen(argv[2],"w");  if(src\_file == NULL || dest\_file == NULL)  {  printf("Open file error!\n");  return 1;  }  fseek(src\_file,-OFFSET,SEEK\_END);  while((real\_read\_len = fread(buff,BUFFER\_SIZE,1,src\_file)) > 0)  {  fwrite(buff,1,BUFFER\_SIZE,dest\_file);  }  fclose(dest\_file);  fclose(src\_file);  return 0;  } |

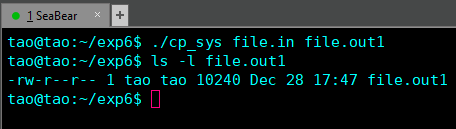
结果：



系统调用方式：cp\_sys.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #include <unistd.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <stdlib.h>  #include <stdio.h>  #define BUFFER\_SIZE 2048  #define OFFSET 10240  int main(int argc, char \*\*argv)  {  int src\_file,dest\_file;  unsigned char buff[BUFFER\_SIZE];  int real\_read\_len;  src\_file = open(argv[1],O\_RDONLY);  dest\_file = open(argv[2],O\_WRONLY|O\_CREAT,0644);  if(src\_file < 0 || dest\_file < 0)  {  printf("Open file error!\n");  exit(1);  }  lseek(src\_file,-OFFSET,SEEK\_END);  while((real\_read\_len = read(src\_file,buff,BUFF\_SIZE)) > 0)  {  write(dest\_file,buff,real\_read\_len);  }  close(dest\_file);  close(src\_file);  return 0;  } |

结果：



### 三、实验扩展：学习inotify

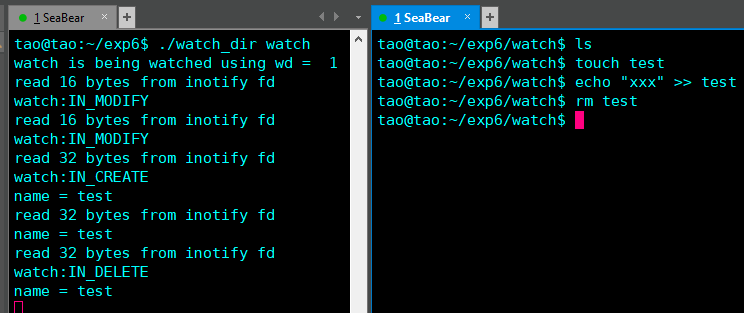
1、利用inotify API实现对目录监控

参考论文：基于Linux的文件实时备份系统设计与实现\_刘斌

watch\_dir.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | #include<unistd.h>  #include<stdio.h>  #include<stdlib.h>  #include<sys/types.h>  #include<sys/inotify.h>  #define BUF\_LEN 1000  void display(struct inotify\_event \*i, char \*\* argv)  {  if(i->wd == 1){  if(i ->mask & IN\_CREATE)  printf("%s:IN\_CREATE\n",argv[1]);  if(i ->mask & IN\_DELETE)  printf("%s:IN\_DELETE\n",argv[1]);  if(i ->mask & IN\_ATTRIB)  printf("%s:IN\_ACCESS\n",argv[1]);  if(i ->mask & IN\_ACCESS)  printf("%s:IN\_MODIFY\n",argv[1]);  }  if(i -> len > 0)  printf("name = %s\n",i->name);  }  int main(int argc,char \*\*argv)  {  int ifd, wd,read\_size,result;  char buf[BUF\_LEN];  char \*p;  struct inotify\_event \*event;  fd\_set inputs, msgfds;  ifd = inotify\_init(); //初始化实例  if(ifd == -1)  {  printf("init error\n");  return 1;  }  wd = inotify\_add\_watch(ifd,argv[1],IN\_MODIFY|IN\_ACCESS|IN\_CREATE|IN\_DELETE); //追加目录监控项  if(wd == -1)  {  printf("error wd\n");  return 1;  }  printf("%s is being watched using wd = %d\n", argv[1],wd);  FD\_ZERO(&inputs);  FD\_SET(ifd,&inputs); //将inotify的文件描述符加入到select中  while(1) //开启非阻塞select模式  {  int fd;  msgfds = inputs;  result = select(FD\_SETSIZE, &msgfds, (fd\_set \*)0, (fd\_set \*)0, (struct timeval \*) 0);  if (result < 1)  perror("select error");  for(fd = 0; fd < FD\_SETSIZE; fd++)  {  if(FD\_ISSET(fd, &msgfds)) //收到消息 处理消息  {  if(fd == ifd)  {  read\_size = read(ifd,buf,BUF\_LEN);  if(read\_size == -1)  printf("read error\n");  printf("read %ld bytes from inotify fd\n",(long)read\_size);  for(p = buf; p < buf+read\_size;)  {  event = (struct inotify\_event \*)p;  display(event,argv);  p+=sizeof(struct inotify\_event) + event->len;  }  }  }  }  }  close(ifd);  return 0;  } |

结果：



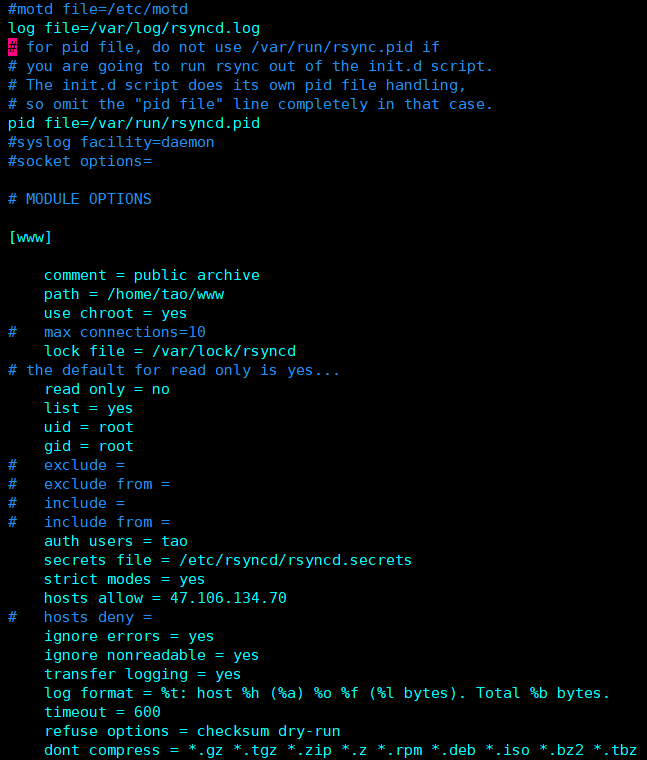
启动watch\_dir监听watch文件目录的创建、删除、修改等操作。

2、利用inotify tools实现服务器数据实时同步

参考网站：<https://www.cnblogs.com/qinhir/p/6589403.html>

https://blog.csdn.net/zpf336/article/details/51659666

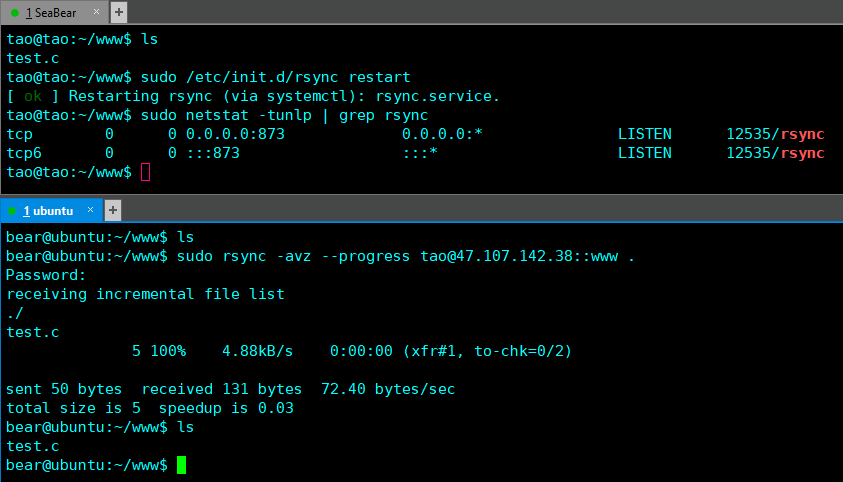
rsyncd.conf 配置：



rsyncd.secrets 保存用户及密码



测试同步目录：

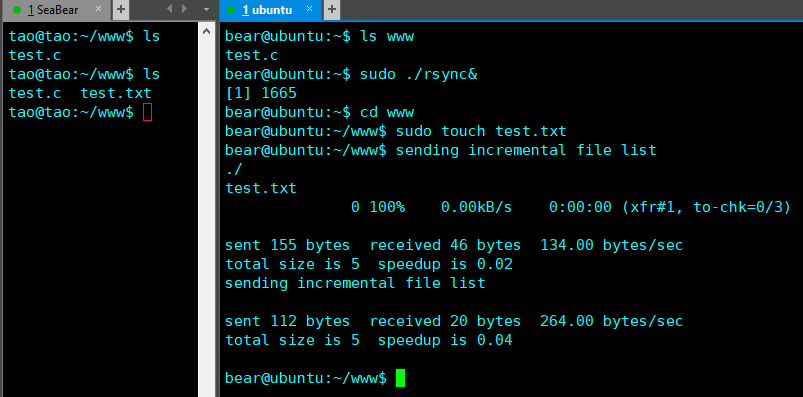


可以看出从服务器将test.c文件同步到了本地，rsync服务安装配置成功。要实现客户端向服务端实时同步，需要在客户端编写实时同步脚本如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #!/bin/bash  #param  host=47.107.142.38 #服务端主机ip  src=/home/bear/www/ #需要同步的客户端目录  dst\_module=www #module  user=tao #用户  rsync\_passwd=/etc/rsyncd/rsyncd.secrets #存放密码的文件路径  /usr/bin/inotifywait -mrq --timefmt '%d/%m/%y %H:%M' --format '%T %w %f' -e modify,delete,create,attrib ${src} | while read file # inotifywait监听文件目录是否发生操作  do  /usr/bin/rsync -avzP --delete --progress ${src} ${user}@${host}::${dst\_module} --password-file=${rsync\_passwd} #向服务端发送同步  echo "${file} was rsyncd" >>/var/log/rsyncd.log 2>&1 #写入日志  done |

脚本的功能是使用inotifywait监听同步目录的修改、删除、创建等操作后向服务端发送同步数据，达到实时同步的效果。存放密码的文件只需要单独写入密码即可，如：123456。另外需要注意的是要修改文件权限为可读。

最终结果:



可以看出，在后台启动同步脚本之后，在同步目录中创建了test.txt文件便自动同步到了服务端。若想实现双向同步，则两台主机都需配置开启rsync服务，两边都需要写同步脚本，与上述方法相同，不再赘述。

## 问题记录和实验总结

问题记录：

1.open创建文件的权限与设置不符。

解决：open打开文件若文件不存在则创建文件，设置文件权限时需要在8进制前加个0例如：0644。

2.rsync访问登陆失败。

解决：把存放密码的文件（rsyncd.secrets）设置权限为可读。便可成功登陆。

3.rsync服务开启后连接失败。

解决：由于本人使用两台阿里云服务器搭建rsync同步，在开启服务端口时需要在阿里云的安全组里开启对应的端口才能正常连接。

本次学习了对linux文件系统的基本操作方法，学会使用库函数与系统调用来实现文件的打开、读写等操作。通过库函数fopen、fread、fwrite与系统调用open、read、write的对比使用，初步理解了库函数与系统调用之间的区别。可以自己动手时间简单的文件复制等功能，在拓展部分学习的inotify API实现对文件系统的监控，以及利用rsync结合inotify-tools时间主机之间的文件同步，在配置过程中掌握了简单的inotify API使用方法，更加熟悉了linux文件系统的操作。