S-38.3600_As2_LearningDiary_244536_YunfengHe

1. Fille Access Permission Problem:

Macros listed in figure 4.25 is used by fchmod(), also works when creating any type of file, such as regular files, directories, fifo, socket, etc. To make things simple and stupid, 0777 or 777 just work(all permitted. u=rwx, g=rwx,o=rwx)

Think of a 3 bits binary number, 1 represents execute permission, 2 represents write permission, 4 represents read permission. then, 3 reprensents wx, 7 rwx, 6 rw, 5rx.

Figure 4.26. Summary of file access permission bits

Constant	Description	Effect on regular file	Effect on directory
S_ISUID	set-user-ID	set effective user ID on execution	(not used)
S_ISGID	set-group-ID	if group-execute set then set effective group ID on execution; otherwise enable mandatory record locking (if supported)	set group ID of new files created in directory to group ID of directory
S_ISVTX	sticky bit	control caching of file contents (if supported)	restrict removal and renaming of files in directory
S_IRUSR	user-read	user permission to read file	user permission to read directory entries
S_IWUSR	user-write	user permission to write file	user permission to remove and create files in directory
S_IXUSR	user-execute	user permission to execute file	user permission to search for given pathname in directory
S_IRGRP	group-read	group permission to read file	group permission to read directory entries
S_IWGRP	group-write	group permission to write file	group permission to remove and create files in directory
S_IXGRP	group-execute	group permission to execute file	group permission to search for given pathname in directory
S_IROTH	other-read	other permission to read file	other permission to read directory entries
S_IWOTH	other-write	other permission to write file	other permission to remove and create files in directory
S_IXOTH	other-execute	other permission to execute file	other permission to search for given pathname in directory

The final nine constants can also be grouped into threes, since

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

mode_t umask(mode_t cmask);

used to set the file mode creation mask for a **process**, and returns the **previous** value. when calling umask(0); it means the calling process would then permit all access by default when creating or opening file. when calling umask(S_IRGRP |

S_IWGRP); group write and read permissions are masked(disabled) for a process, even when you create a file with these macros. When using Umask, 777 blocks all permission.

chmod() and fchmod();

chmod() function operates on the specified file, whereas the fchmod() function operates on a file that has already been opened.

2. Redirect perror message:

Method 1, duplicate the file descriptor?? test!!

```
ferr = open("stderr.log", O_WRONLY|O_CREAT, 0600);
if(dup2(ferr, STDERR_FILENO) == -1)
{
perror("stderr.log";
}
close(ferr);
```

Method 2, call strerror(errno); for example fprintf(fp, "sterror: %s\n", strerror(errno));

3. Something about select().

When open the FIFO as RDWR, select() blocks well, when open as RDONLY, select() won't block and always telling there's something you can read from the FIFO. Try explain why!

(listenfd = open(COMMONFIFO, O_RDWR) this would magically work!!

listenfd = open(COMMONFIFO, O_RDONLY); listenfd = open(COMMONFIFO, O_RDONLY | O_NONBLOCK); wouldn't work either. try to explain!!

When we `open` a FIFO, the nonblocking flag (`O_NONBLOCK`) affects what happens.

- In the normal case (`O_NONBLOCK` not specified), an `open` for read-only blocks until some other process opens the FIFO for writing. Similarly, an `open` for write-only blocks until some other process opens the FIFO for reading.
- If `O_NONBLOCK` is specified, an `open` for read-only returns immediately. But an `open` for

... an open for read-only blocks until some other process *opens* the FIFO for writing. ("*open" alone* is enough for the "ready" state of select)

Below are comments with wrong ideas. not the reason why select won't block

/*In the server side, I tried to use select as a timer, and listens if there's a incoming write from the other end of the FIFO. It turns out that as soon as the write end(the CLIENT end of the FIFO) is opened, select()would always return and telling that the fd is prepared to be read, even there's really nothing in the pipe buffer(I testified this by open the FIFO as write on the client and write nothing, then close the end). This cause a infinite loop, when there's nothing in the pipe as long as the pipe end is not shut. Even when the other end is shut, select() stills tell you there's something you can read!!!Because the mechanism of select is that it turns on a flag bit for a certain fd, when ready, the flag bit is set to 1, and then select will do nothing more about it!!!That's why select cannot be used as timer for a stream communication. The timer only works, when the detected fd is deleted from the set. if a fd which is ready is not deleted from fdset after select returns, the timer's value would remain untouched!!. In this sense, select()'s timer only time out the blocking time in total.(*the lines in red is wrong idea*)*/

Feedback from Riba

This is the basic nature of FIFOs. If there are no more writers the FIFO will indicate end-of-file. This will show up on select() as if the FIFO FD would always seem to be readable, yet all you read() out of it is 0, as in end-of-file. Simplest tweak is, like you discovered, to open the FIFO in O_RDWR-mode, so that there is at least one writer always. The other

way would be to set the O_NONBLOCK-flag when opening the FIFO.
The complete mechanism is as follows: if you do open() FIFO in O_RDONLY mode, the open() call blocks until the first writer opens the FIFO. If you open() FIFO in O_WRONLY mode, the open() call blocks until first reader opens the FIFO. This is a simple mechanism to prevent readers and writers from making premature actions on the FIFO.
For more in-depth info, Stevens' book chapter 15.5 and Kerrisk's book chapter 44.7 contain wealth of more data.

4.FIFO

1.Concept:

FIFO is another type of pipe, which have the same characteristics of a common pipe. Pipe is used for inter-process communication through fork(), where a common ancestor has created the processes and the pipe that connect them. FIFO on the other hand, is used as a way to communicate unrelated processes(programs). FIFO is also called named pipe, the basic idea is that FIFO is created just like a common file(except that it is in the form of a pipe), so that any program who knows the path of this file can open it and perform I/O. Thus, unrelated processes can communicate through FIFO by writing to and reading from it.

2.FIFO buffer length problem

Since FIFO(PIPE) use a internal buffer, it can only hold certain amount of data before the data is read out. To avoid overflowing the underlying buffer of a pipe, we shall know the length of the buffer. (it's typically like 4096Bytes and can be manually set).

```
#include <unistd.h>
long pathconf(const char *pathname, int name);
long fpathconf(int filefd, int name);
```

Calling fpathconf() at run time to show the length of a FIFO buffer can make sure that we always get the right size of the buffer, no matter if the buffer length has been modified or not.

At fifo's read end, when read returns 0, it shows that it encounter's an end-of-file.(the write end has been closed).

3.Something about opening FIFO

When a fifo is made, the read end shall be opened first, then the write end. Otherwise, the process who tries to open the write end of the fifo would encounter SIGPIPE. If two process and opening the write and read end of the fifo respectively, then synchronization must be done. if no synchronization is implemented, the process who tries to write and simply open the FIFO as O_RDWR, in this way, things would be fine

5. Semaphore and mutex

- The "toilet model" explains the difference between semaphore and mutex very well.
- **semaphore:** specified number of keys to the toilet. (Access control)"A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number."
- **mutex:** only one key to the toilet. "Mutex is typically used to serialize access to a section of re-entrant code that cannot be executed concurrently by more than one thread"
- The corresponding APIs for acquiring Semaphore and Mutex are reliable(atomic operations, unlike signal APIs). However, the use of semaphore and mutex can both rise race conditions. In addition to simply try to acquire a semaphore (different processes) and a mutex(among threads), certain way of synchronization could make things better, depending on the program task.

6.Thread:

- Detach mode makes sure a new thread's resources is well cleaned up when it returns. In this way, the main thread no

longer need to join the new thread and do the clean up for it.

- Detach doesn't mean that the new thread can still run on after the main thread returns. when the main returns, all threads are destroyed, naturally. since thread is only in the stack of the memory space of its belonging process.
- To make sure that the new thread can still run on after the main thread exit, use pthread_exit(); instead.
- When using thread, be careful about the called functions within the thread routine. make sure that they are thread safe, or put in another way, thread-reentrantable.

7.Daemon Creation

To check Daemon processes, use ps -axj

There's nothing too complex about creating a daemon process; one thing is to call umask(0) to clear the file mode creation mask to 0. this make sure that I/O operation performed by the daemon process won't encounter unpredicted permission problem. The second thing after the common procedure, is to close all file descriptors before turn into daemon. then open "/dev/null"(which would give you fd 0, stdin), then dup(0) twice to redirect stdout and stderror. Daemonize the process as soon as the program start, this surely help save trouble.

To understand daemonization, Know this:

process group leader(PID==a)
fork
process 1(gid==a)
fork
process 2(gid==a)
fork
process 3(gid==a)
....
.....(gid==a)
....
.....(gid==a/ PID==x)
setsid(create new session,session ID and Group ID ==x)
fork exit to daemonize a process.

8.Other problems encountered when coding the Assignment

1. When writing an integer to a buffer, use sprintf (buf, "%d", number); //I tried to write the client's process id to the server's listening FIFO, when I write directly, I got weird outcomes.

this code segment for example, only works when the value of an integer is smaller than 255!

```
{
    int new =100;
    char tmp = new;
    write(fd,&tmp,1); // only 1 byte is written to the fd, a char can nicely hold the value of an Int when the value is less than 255, any bigger, 8 digits won't be enough. if I simply code like this: write(fd, &new, sizeof(int)); it still create mess, since the order of how to put the bits is different in INT and a common CHAR buffer
    read(fd,buf,1);
    fprintf(stderr, "%d", (int)(*buf));
}
```

2.A problem with sprintf(buf,"%d",NUM); (char buf[20], int NUM = 581;)

An integer contains 4 bytes. when the value hold by the integer is say 3 digits number(for example 581), then sprintf(buf,"%d",NUM); would take only 3 bytes of buf's 20 bytes space. ('5' '8' '1' are written to buf respectively.) If the

value of NUM is a 4 digits, then sprintf takes 4 bytes of buf's space to save this number.

3. File lock problem:

Fcntl(); provide a way to put read or write lock on a file(fd). this is useful when more than one process wanna write to the same file. My Data Transfer program don't really use the file lock, since I consider fwrite() quicker. I made a test of two instances of my client writing the same file for the server to copy(a 102MB text file), and the copied file is not corrupted. This is because, the two files opened by the two clients under different folders are exactly the same. this means at the server two threads would write the same thing at the same offset of the file(regardless of who write first, it doesn't matter, it's like draw the same line at the place on a paper twice). if the two files with the same name are not the same in the first place, then the COPY file will be corrupted. Anyway, trying to make a copy of two files with the same name at the same time is really silly anyway. that's why, File lock is only advisory lock and not all the time needed!

4. SIGPIPE handling:

SIGPIPE is given to a client who try to write more data to a pipe when the read end of the pipe is already shut down. the default action on SIGPIPE is to terminate the process, which is fine to my client. signal(SIGPIPE,SIG_IGN); is typically used to set the action to "ignore".

5. Dealing with command-line argument: (getopt() is sort of complex for a simple program)

```
for (i =1; i<argc; i++){
    do something about argv[i];.....//writing a simple loop could be quite enough sometimes.
}
```

6.More on FIFO

Try fifo:

open a shell and type in "mkfifo /tmp/fifo","cat /tmp/fifo", cat blocks;
Then open another shell and "echo hello> /tmp/fifo". watch the output of the first shell.(cat display "hello" and return)
then "echo hi> /tmp/fifo", "cat /tmp/fifo" again in the first shell and "hi" will be displayed.
Fifo acts as a stream pipe holding the content, after reading, content gone.

FIFO opening

FIFO must be opened as read first, if a FIFO is opened as write first, it encounters SIGPIPE.

FIFO buffer overflow

pipe buffer is 4096, meaning that 4096/5 requests can be held by the buffer, surely the server will always read from the pipe as soon as possible. In my program, since I use signal to synchronize client request, only one request will be in the listening FIFO buffer at a time(server read one, create a working thread and then allows another write to the listening FIFO).

7. Pthread_exit() Vs return

use pthread_exit() in main and don't call return 0; then child thread will keep running. In the thread_routin, call return NULL, instead of pthread_exit() to avoid memory leak

8. Memory leak problem

memory leak was spotted at server side, but I really could not locate the problem at first. then I realize there might be something special about thread. first I checked that malloc and calloc are indeed thread-safe. Then I noticed I wrote pthread_exit(); to quit the new thread_routine. this is bad since pthread_exit() do not perform necessary clean up. use pthread_exit() to quit the main thread was my intension, so that when main listening thread times out, it would not affect working thread. But working thread shall just return. after I replace the pthread_exit() calls with return, memory leak problem is solved.
when server time out and returns, valgrind tells me there's mem leak, by malloc and calloc, but I really cannot tell what's wrong in the code, I freed allocated memorys, and malloc is thread safe when compiling with pthread. feedback

please!!!(mem-leak won't happen if there's only one thread working. multiple client caused this somehow. always 5 blocks and 936 bytes lost.) This problem is solved. When thread is done with it's copy job, I wrote pthread_exit(NULL), instead of calling return. This somehow create problems as pthread_exit() do not do any clean up at all. I need only to use pthread_exit() when the main thread return, to prevent from the timout mechanism shutting down other working thread prematurely. In the thread routine, use return NULL to return.

9.System Limits:

Figure 2.8. POSIX.1 invariant minimum values from <limits.h>

Name	Description: minimum acceptable value for	Value
_POSIX_ARG_MAX	length of arguments to exec functions	4,096
_POSIX_CHILD_MAX	number of child processes per real user ID	25
_POSIX_HOST_NAME_MAX	maximum length of a host name as returned by gethostname	255
_POSIX_LINK_MAX	number of links to a file	8
_POSIX_LOGIN_NAME_MAX	maximum length of a login name	9
_POSIX_MAX_CANON	number of bytes on a terminal's canonical input queue	255
_POSIX_MAX_INPUT	space available on a terminal's input queue	255
_POSIX_NAME_MAX	number of bytes in a filename, not including the terminating null	14
_POSIX_NGROUPS_MAX	number of simultaneous supplementary group IDs per process	8
_POSIX_OPEN_MAX	number of open files per process	20
_POSIX_PATH_MAX	number of bytes in a pathname, including the terminating null	256
_POSIX_PIPE_BUF	number of bytes that can be written atomically to a pipe	512
_POSIX_RE_DUP_MAX	number of repeated occurrences of a basic regular expression permitted by the regexec and regcomp functions when using the interval notation \{m,n\}	255
_POSIX_SSIZE_MAX	value that can be stored in ssize_t object	32,767
_POSIX_STREAM_MAX	number of standard I/O streams a process can have open at once	8

Appendix (some trash, don't bother!!)

some links

http://hi.baidu.com/best_wenhouhou/item/47b458e4d714e8b2c10d75fa

http://lxr.linux.no/

http://www.cppblog.com/elva/archive/2008/09/10/61544.html

在使用上 perror 和 strerror 应注意的地方

在调用某些库函数出错时，通常需要将错误信息打印出来，有的程序用 perror(print error)打印，有的用 strerror 打印，到底他们之间有什么区别呢？man 手册里面写的很清楚，perror 是将 errno 对应的错误消息的字符串打印到标准错误输出上，即 stderr 或 2 上，若你的程序将标准错误输出重定向到/dev/null，那就看不到了，就不能用 perror 了。而 strerror 的作用只是将 errno 对应的错误消息字符串返回，要怎样处理完全由你自己决定。通常我们选择把错误消息保存到日志文件中，即写文件，所以通常可以用 fprintf(fp, "%s", strerror(errno))将错误消息打印到 fp 指向的文件中。其中 perror 中 errno 对应的错误消息集合跟 strerror 是一样的，也就是说不会漏掉某些错误。

#include <stdio.h> // void perror(const char *msg);

#include <string.h> // char *strerror(int errnum);

#include <errno.h> //errno

errno 是错误代码，在 errno.h 头文件中

void perror(const char *s)

perror 是错误输出函数，在标准输出设备上输出一个错误信息。

参数 s 一般是参数错误的函数

例如 perror("fun")，其输出为：fun：后面跟着错误信息（加上一个换行符）

char *strerror(int errnum);通过参数 errnum(也就是 errno)，返回错误信息

以下是测试程序：

```
//程序名：errtest.c，环境为 linux
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
int main(int argc,char *argv[]){
FILE *fp;
char *buf;
if((fp=fopen(argv[1],"r"))==NULL)
{
perror("perror");
printf("sterror:%s\n",strerror(errno));
exit(1);
}
perror("perror");
errno=13;
printf("strerror:%s\n",strerror(errno));
fclose(fp);
return 0;
}
```

==========

编译为 errtest

如果输入这样的命令格式：./errtest 111.c（其中 111.c 不存在）

输出为：

perror: No such file or directory

sterror:Illegal seek

就是两个都是输出到屏幕上来了。而且 sterror 函数通过 errno 得到错误代码

如果命令格式为：./errtest 111.c > out.c（其中 111.c 不存在）

把输出重定位到 out.c 文件中，会发现屏幕输出为：

perror: No such file or directory

就是说函数 perror 始终输出到标准输出设备上。而 printf 输出到文件中了