Project 5
File System Protection

Introduction

This project will implement simple *protection* in the xv6 file system.

Your goals are to:

1. Implement protection in the xv6 file system.
2. Understand how the file system abstraction is implemented in the xv6 file system.
3. Understand the meta information underlying the file system abstraction.
4. Gain an appreciation for *transactions* as they are used in xv6.

For this project, you will use a new Makefile flag, `-DCS333_P5`, for conditional compilation.

Overview

This project requires you to implement a *completely new* abstraction in xv6: file system protection. This is a very large area, so you will focus on a small subset of protection that will nevertheless give you insights into how to implement additional protection and security concepts.

You will implement file system protection, user programs for manipulating those protections, and add protection checking to the `exec()` system call.

New System Calls

You are required to implement three new system calls.

1. `int chmod(char *pathname, int mode);`
   The `chmod()` system call sets the mode, or permission, bits for the target specified by `pathname`. The return value is '0' on success and '-1' on failure. See the description for the `chmod` command below for details.
2. `int chown(char *pathname, int owner);`
   The `chown()` system call sets the user UID for the target specified by `pathname`. The return value is '0' on success and '-1' on failure. See the description for the `chown` command below for details.
3. `int chgrp(char *pathname, int owner);`
   The `chgrp()` system call sets the group GID for the target specified by `pathname`. The return value is '0' on success and '-1' on failure. See the description for the `chgrp` command below for details.

New Commands

Three new commands will allow the user to set the UID, GID, or mode for a file or directory. In order to simplify the testing of the new features in xv6, set the permissions and ownership for these commands as: owner = 0; group = 0; and mode = 0755. Additional information regarding these programs will be discussed in class.

1. `chown`: sets the owner (UID) for a file or directory.
   Usage: `chown OWNER TARGET`
   where `OWNER` is the numeric UID to set as the owner of `TARGET` and `TARGET` is the name of a file or directory.

2. `chgrp`: sets the group (GID) for a file or directory.
   Usage: `chgrp GROUP TARGET`
   where `GROUP` is the numeric GID to set as the group of `TARGET` and `TARGET` is the name of a file or directory.

3. `chmod`: sets the mode bits (permissions) for a file (or directory).
   Usage: `chown MODE TARGET`
   where `MODE` is a string of octal values specifying the mode bits to set for `TARGET` and `TARGET` is the name of a file or directory.

   The numeric `MODE` is four octal digits (0-7), derived by adding up the bits with values 4, 2, and 1. No digits may be omitted, and any digit can be the value 0. The first digit selects the set UID (1) attribute. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group; and the fourth for other users not in the file's group, with the same values. It is permissible for one of user / group / other to be 0, which indicates no access to the file, e.g., `chmod 0000 TARGET` is valid.

   Mode bits are evaluated in this order:

   (a) User. Check against the user permissions. If the UID of the invoking process and the UID of the file are the same, use these permissions.
   (b) Group: Check against the group permission. If the GID of the invoking process and the GID of the file are the same, use these permissions.
   (c) Other: If neither the user or group permissions are used, apply these permissions.
   (d) Setuid: If the permissions allow the invoking process to execute the file, set the UID of the process executing the file to be the same as the UID for the file.

`chmod` Examples

1. `chmod 0755 File`
   File is not setuid, user has all permissions, group and other have read and execute permission.

2. `chmod 1550 File`
   File is setuid, when the file is executed the process takes on the UID of File. Only the user and group may read or execute File. Others cannot access File.

3. `chmod 0055 File`
   File is not setuid and the user may not access the file. File may be read and executed by group or others.

Modified Commands

1. `ls`: The `ls` command must be modified to display the new fields. A new routine, `print_mode` is provided in the file `print_mode.c` to handle the formatting. Since the output is getting crowded, you are to provide a header that labels each column in the output. The source code file `print_mode.c` can included directly into the `ls.c` source code file.

   `mode`. The first column will display the mode bits for the file/directory/device. The first character indicates if the item is a regular file ('-'), directory ('d'), or device file ('c'). If the file is setuid, then the 'x' in the user permissions will be displayed as 'S'.

   `name`. Name of the file or directory.

   `uid`. User identifier (owner) of the file or directory.

`gid`. Group identifier (owning group) of the file or directory.

`inode`. Inode number of the file or directory.

`size`. Size in bytes of the file or directory.

Sample Output

```
$ ls
mode              name     uid      gid      inode     size
drwxr-xr-x  .                0        0        1         512
drwxr-xr-x  ..               0        0        1         512
-rwxr-xr-x  README           0        0        2         1973
-rwxr-xr-x  cat              0        0        3         14884
-rwxr-xr-x  echo             0        0        4         13849
-rwxr-xr-x  forktest         0        0        5         9361
-rwxr-xr-x  grep             0        0        6         16812
-rwxr-xr-x  init             0        0        7         14750
-rwxr-xr-x  kill             0        0        8         13981
-rwxr-xr-x  ln               0        0        9         13879
-rwxr-xr-x  ls               0        0        10        19010
-rwxr-xr-x  mkdir            0        0        11        14010
-rwxr-xr-x  rm               0        0        12        13991
-rwxr-xr-x  sh               0        0        13        26855
-rwxr-xr-x  stressfs         0        0        14        14969
-rwxr-xr-x  usertests        0        0        15        69424
-rwxr-xr-x  wc               0        0        16        15470
-rwxr-xr-x  zombie           0        0        17        13615
-rwxr-xr-x  halt             0        0        18        13441
-rwxr-xr-x  MMchown          0        0        19        14464
-rwxr-xr-x  MMchgrp          0        0        20        14464
-rwxr-xr-x  MMchmod          0        0        21        14780
crwxr-xr-x  console          0        0        22        0
$ ls halt
mode              name     uid      gid      inode     size
-rwxr-xr-x  halt             0        0        18        13441
$
```

2. `mkfs`. File `mkfs.c`. This program is used to create the xv6 file system. It is used when the file system is built, which must be done before you can boot the kernel. It is invoked from the `Makefile`. Note that it needs our new flags to be added to its compilation line as appropriate.

   This program will set the default ownership (UID and GID) and default permissions (mode bits) for items in the file system.

Modified System Calls

1. `exec()`. File `exec.c`. The exec() system call will require two changes. The first change is that the file should not be read unless the process has execute permission for the file. The second change is when the new image is committed; if the file is setuid then the new uid will need to be set.

2. `fstat`. Deep into the implementation of the fstat() system call is the code where information from the inode is copied into the buffer provided to the fstat() system call. You will need to ensure that

the new information is copied as well.

inode / dinode

The necessary modifications to the in-memory and on-disk structures for the inode are seemingly simple but, if done incorrectly, can leave you struggling. In particular, there is a tight relationship between the size of the inode and the size of disk blocks in xv6. Because of this, we are providing a new definition for the inode/dinode structures for you.

Three new fields are required: uid; gid; and mode. Mode will be interpreted as a 12-bit binary vector. The mode bits for user/group/other will be interpreted as "read write execute" permissions (rwx). Bits 0-2 are the mode bits for "other"; bits 3-5 are for "group"; bits 6-8 are for "user"; bit 9 is the "setuid" indicator; and the remaining bits in the integer field are unused. There are several ways to set/check these values and as long as they make sense, no one is preferred. For information on using unions and bit fields in C, see sections 6.8 and 6.9 of the K&R book.

The number of direct inodes, (`NDIRECT`), needs to change to add in the two new integer fields[1].

```
#ifdef CS333_P5
#define NDIRECT 10
#else
#define NDIRECT 12
#endif
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)

#ifdef CS333_P5
union mode_t {
  struct {
    uint o_x : 1;
    uint o_w : 1;
    uint o_r : 1;  // other
    uint g_x : 1;
    uint g_w : 1;
    uint g_r : 1;  // group
    uint u_x : 1;
    uint u_w : 1;
    uint u_r : 1;  // user
    uint setuid : 1;
    uint      : 22;  // pad
  } flags;
  uint asInt;
};
#endif
```

[1]Necessary for implicit assumptions built in to the file system. These assumptions are made explicit at two locations in `mkfs.c` which uses this assert twice:
`assert((BSIZE % sizeof(struct dinode)) == 0);`

```
// On-disk inode structure
struct dinode {
  short type;            // File type
  short major;           // Major device number (T_DEV only)
  short minor;           // Minor device number (T_DEV only)
  short nlink;           // Number of links to inode in file system
#ifdef CS333_P5
  ushort uid;            // owner ID
  ushort gid;            // group ID
  union mode_t mode;     // protection/mode bits
#endif
  uint size;             // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

The `stat` command also uses the inode structure but due to the way that include files are used in the kernel, the union will have to be renamed to `stat_mode_t`. This could also be handled with compiler directives, but since the rest of the kernel does not use these, you will follow the existing style which, unfortunately, favors the renaming approach[2]. However, since both the `inode` and `dinode` structures already have a similar dependency, it isn't terrible as long as you carefully document the dependencies in the code and your project report.

You will need to provide **DEFAULT values** for uid, gid, and mode. Define them in `param.h`. Use the same defaults for uid and gid that you used for creating the first process at boot time. You can use the same define as you used for that purpose. Be sure to provide a comment in `param.h` that indicates which defines are for the first process and which for default file system permissions. A comment such as

```
// DEFAULT_UID is the default value for both the first process and files
// created by mkfs when the file system is created
#define DEFAULT_UID 0
```

would be good.

The *default mode* should be '0755'. This is interpreted as

0: The file or directory does not have the set UID bit set.

7: The file owner has full permissions to the file: read, write, and execute.

5: The file group has permissions read and execute but not write.

5: Others have permissions read and execute but not write.

Other values for uid, gid, and mode defaults could be chosen, but these are reasonable since they will allow for ease of testing. Since the default uid and gid are 0 for both processes and files, the default behavior for the system will be read, write, and execute for the first process at boot time for all files and directories. This means that, by default, the first process after boot will actually be able to execute programs, etc. Discuss with course staff if you wish to explore alternates with the file system defaults.

File System Transactions

The xv6 file system uses a log to help maintain file system consistency. Operations to disk are *very slow*. As a result, all file system operations are logged before being committed to the disk. In this way, if the

[2]Having two structures with different names that *must* have the exact same contents is *very* problematic. This technique should be avoided if at all possible. It is being used here simply as an expedient to reduce the amount of work required by the student. You should include comments pointing out this requirement in your code.

commit process is interrupted, the actions may be replayed from the log. This also means that there must be a way to ensure that the log is not corrupted. The xv6 file system uses a *transactional* approach. All operations that result in a file system modification are wrapped in a transaction. The `begin_op()` call indicates the start of a file system transaction and the `end_op()` call indicated that the transaction is complete.

You should look for code in the file system that uses transactions to better understand how to use these calls in your code.

Required Testing

An automated test suite is provided for testing *system calls*. User command testing needs to be developed by the student.

Students are required to understand the automated testing and correctly explain the output for each test. Student may write additional test programs if they determine there is a need, but for system call testing, the automated tests are deemed sufficient[3]. Do not modify the automated test program.

The remaining required tests are

1. That the `chmod`, `chown`, and `chgrp` user commands change the mode, UID, and GID, respectively, of a file when passed in valid parameters (other than the defaults).
2. That the `chmod`, `chown`, and `chgrp` user commands do not modify anything when passed in an invalid filename.
3. That the `chmod`, `chown`, and `chgrp` user commands do not modify anything when passed in an invalid numeric argument. These would be the mode digits for chmod, the UID for chown, and the GID for chgrp.
4. That the `ls` command prints out the correct information, including correct permissions for the mode bits.
5. That any changes made to the file system persist after the xv6 kernel is shut down. Specifically, you are required to boot the xv6 kernel, change the UID, GID and mode of a file to values other than the defaults, shut down the kernel, reboot the kernel without remaking it or doing a `make clean`, and show that the file's changes are still there.

[3]Suggestions for improvement for the automated tests are encouraged.