CMSC 412 Project #6

Users and File Protection

Due Tuesday, May 13, 2014, at 5:00pm

Introduction

The purpose of this project is to add the notion of users and access control lists to the Geek OS kernel. This will allow different processes to access only files they are allowed to see.

Users and I/O Protection

This project will also introduce the idea of user's and I/O protection to the OS. Adding the system calls `SetAcl`, `SetSetUid`, and `SetEffectiveUid` will do this. The key idea to adding users is simply to have a field in the user context data structure that identifies the current user that is running this process. The user will be represented by an integer called the uid. There is a special user (uid 0) that is the super user. The super user (and only the super user) may change the user id of a running process via the `SetEffectiveUid` system call. If the uid of a process is 0 (superuser) when this system call is made, the uid is changed to the passed uid. The `GetUid` call returns the current uid of the process. When the shell starts, it runs with uid 0, i.e. super-user privileges. When a process is spawned, it normally will inherit the uid of the parent process. There may be instances where a process may need more rights than its parent (think of the passwd command in unix). In order to allow this, there is a special bit, setUid stored in the directoryEntry. If this bit is enabled (set to 1) for the executable, the process should inherit the uid of the owner of the **executable file** rather than the parent process. The function `SetSetUid` is used to set this bit for a file. Each file in the file system can be "owned" by exactly one user.

The uid is also used by the file system to check if a particular user is able to perform a particular operation on a given file. A process running with the uid of the superuser can open any file regardless of the ACLs. If the uid is any other value, the I/O operation will only succeed if the uid has appropriate privilege based on the ACL of the file (or directory). For the Open system call, if the file exists the current uid must have the requested access level (i.e. Read privilege if the file is being opened with read access). If the file does not exist, then the user must have write access to the parent directory. The read, write, seek, stat, and close system calls do **not** require checking the ACL since the check is made on the Open call. The createDirectory call requires write access to the parent directory. The delete and setAcl calls require write access to the named file.

The `SetAcl` system call sets the file permissions for the passed uid on the named file. If the permissions are 0, this should delete any access that uid had to the passed file. If the request is for a new uid to have some privilege for a file, and the ACL table for that file is full, you should return -1 as an error code. This code should also be used for invalid permissions or for non-existent files. The SetAcl system call can also be used to change the owner of a file. However, the effective id of the process making the SetAcl system call must be 0 if you are changing a file's ownership. The files owner is defined as the uid of the zeorith entry of the ACL table for that file.

You could add the file protection information to any GeekOS file system. To keep this project independent of project #5, you will add the file permission code to the PFAT file system supplied with GeekOS. All ACL information must be stored on disk as part of the file's meta data (i.e. in the directoryEntry defined in …/include/geekos/pfat.h).

No new project files will be provided for this project. You should use start from a fresh checkout of the project.

New System Calls

The following system calls will need to be added to your operating system.

User Function	**Return success/ failure**	**Reasons for failure**	**Comment**
SetAcl(char *name, int uid, int permissions)	0/-1	• `name` does not exist • illlegal value for `uid` (it must be greater than 0) • illegal value for `permissions` • the ACL table for `name` is full	• The permissions values are flags and may be OR'ed together in a call. For example: • O_READ • O_WRITE • O_READ|O_WRITE • O_OWNER • 0 (zero) • O_OWNER may not be combined with any other flags.
SetSetUid(char *name, int setUid)	0/-1	• `name` does not exist • illegal value for `setUid` (it can only be 0 or 1)	• This call will set the setUid bit in the directoryEntry structure for the file corresponding to `name`
SetEffectiveUid(int uid)	0/-1	• current Uid is not superuser	Set the user id for the current process
GetUid()	uid /-1		Return the user id of the current process

Testing

You will need modify the implementation of stat to return the ACL information and setuid information as part of the fileStat.

You should write some new user mode programs that test out your new system calls. The examples in the directory src/user should provide a starting point. We have included programs `setuid.c` and `setacl.c` to help you get started.

Since the PFAT file system has only read only files, you will need to create some additional files to test with as part of the process to build the diskc.img. If you look in the …/build/user directory, each file in this directory (that ends in .txt) will end up on the pfat file system. To include a sub-directory, use the –d command line option. Look in the Makefile.common file (diskc.img rule) for more information.