CS 4513 Distributed Computing Systems | WPI, B Term 2005
Craig E. Wills | Project 1 (35 pts)
Assigned: Friday, October 28, 2005 | Due: Tuesday, November 15, 2005

Introduction

Operating systems maintain a number of attributes about a file. Typically file systems support attributes such as access permissions, size and last modification time. On a Unix/Linux system, information about the standard set of attributes for a file can be obtained using the *stat()* system call. The standard Linux file system, *ext2*, supports additional attributes for a file that can be set. See the Linux man pages for the commands *lsattr* and *chattr* to see more details on these additional attributes.

In addition to supporting standard file attributes, the Be Operating System supports user-defined attributes [1] [1]. This feature allows users to attach (attribute,value) pairs to a file. For example, if each mail message a user receives is stored in its own file, then a "From" attribute could be attached to each file. The presense of this feature allows all files with a given attribute value to be easily identified. User-defined attributes also allow information to be stored about a file in addition to its content. For example, the icon position of the file in a window system, or the URL source of a downloaded Web document. In this project you will be adding user-defined file attributes to the Linux file system and creating user-level programs to set and use these attributes.

Creating User-Defined File Attributes

The first part of the project involves no kernel modifications. You will be creating a user-level routine with the following name and definition:

```
ret = SetAttribute(char *filename, char *attrname, char *value, int size)
```

The *SetAttribute()* routine will take the name of a file, an attribute name, an attribute value and the size of the attribute value. The attribute value for a file cannot be stored with the contents of the file, but needs to be stored in a separate location. For this project we will store all user-defined attributes in an "attribute" directory, where each file within the directory corresponds to a user-defined attribute and the contents of an attribute file are the value of the attribute.

We will use the file name in creating the name of the attribute directory. For a given file, the attribute directory is created by prepending the file name with a dot (".") and appending the file name with `_attr`. Thus the attribute directory for the file `foo` will be `.foo_attr`. You can create directories as needed with the *mkdir()* system call. Your routine should return an error if attempting to set an attribute for a file already beginning with ".". It is possible to set attributes for directories as well as regular files.

An example use of *SetAttribute()* is:

[1] The Be File System and book were written by a WPI alumnus Dominic Giampaolo.

```
ret = SetAttribute("foo.c", "Programmer", "Craig Wills",
                                    strlen("Craig Wills"));
```

This routine creates the attribute directory `.foo.c_attr` and the attribute file `.foo.c_attr/Programmer` with the contents of “`Craig Wills`”. You should use the system calls *open()*, *close()*, *read()* and *write()* for accessing attribute file contents as other I/O routines may not be available in the kernel. The file permissions for the attribute file should match those of the file. The routine should return a -1 if the file does not exist or the attribute file cannot be created. If the routine succeeds then it should return the number of bytes written to the attribute file. This value should be the same as the last argument.

As a convenience in using and testing attributes, you should write a program, which compiles to a command called *setattr*. It has the following syntax:

```
setattr attr=value file(s)
```

Two example uses for this command are:

```
% setattr Type=header foo.h
% setattr "Programmer=Craig Wills" *.c *.C *.h
```

The first example sets the user attribute “`Type`” for the file `foo.h` and the second example sets the attribute “`Programmer`” for all C/C++ source and header files in the current directory. Note that the shell will expand the wildcard “`*`” and your program will iterate through this list of files and call *SetAttribute()* for each file. Also note that any spaces in the attribute value will requires the use of quotes for the shell to treat it as one argument.

This implementation for user-defined attributes is similar to that in the Be Operating System in terms of storing the attributes in an attribute directory as regular files. However, in the Be Operating System the inode of the attribute directory is stored in a file’s inode rather than actually using a named directory. Also the BeOS optimizes “small attributes” by storing their value in the same disk block as the inode of the file. See [1] for more details on the Be OS file system.

Listing User-Defined File Attributes

Once you can create user-defined attributes you need a means to retrieve and display these attributes for a file. The system routine you will need to write is given as follows:

```
ret = GetAttribute(char *filename, char *attrname, char *buf,
                                                    int bufsize)
```

where an example use of this routine is

```
char buf[1024];
ret = GetAttribute("foo.c", "Programmer", buf, 1024);
```

The routine stores the value of the attribute in `buf` and returns the number of bytes stored (maximum is the given buffer size). The routine returns a -1 if the file or its attribute cannot be found.

You should also create a command called *listattr*, which has the following syntax:

```
listattr attr file(s)
```

Two example uses for this command are:

```
% listattr Programmer *.C
file1.C Programmer=Craig Wills
file2.C Programmer=Craig Wills
% listattr ALL *.h
bar.h Programmer=Craig Wills
foo.h Programmer=Craig Wills
foo.h Type=header
```

The first example prints the value of the user attribute "`Programmer`" for all C++ source files. The output shows file name, attribute name and attribute value. The second example introduces a special attribute named "`ALL`". When used with *listattr*, the output should be the values of all attributes associated with each file. This special attribute is implemented as a special case by *GetAttribute()*, where passing "`ALL`" as the attribute will return a colon-separated list of all attribute *names* for the file. In the case of `bar.h`, the value of the attribute "`ALL`" is "`Programmer`" while for `foo.h`, the value of the attribute "`ALL`" is "`Programmer:Type`". The *listattr* will need to iterate through this list of attribute names and retrieve individual attribute values. You can use a routine such as *strtok()* to parse the colon-separated string.

Implicit File Attributes

In addition to attributes explicitly set by the user for a file, your *GetAttribute()* routine also needs to support implicit attributes for a file. These attributes are `Mode`, `Uid`, `Gid`, `Size`, `Atime`, `Mtime` and `Ctime`. Values and descriptions for these attributes can be obtained using the *stat()* system call and its man page. If one of these implicit attributes is given to *GetAttribute()* then your routine should get the appropriate value, which will be an integer and convert the value to a string to be stored in the buffer. The conversion from integer to string can be done using the *sprintf()*. For example, the following example stores the value of `num` in `buf` as a string.

```
int num = 5623;
char buf[128];
sprintf(buf, "%d", num);
```

These attributes should also be included in the list of names when the attribute name "`ALL`" is given. You should support the attribute "`ALLUSER`" to only return a list of user-defined attributes. Thus the expected output for the *listattr* command with the "`ALL`" attribute is:

```
% listattr ALL foo.h
foo.h Uid=507
foo.h Gid=100
foo.h Size=340
foo.h Atime=1004105714
foo.h Mtime=1004105712
foo.h Ctime=1004105712
foo.h Programmer=Craig Wills
foo.h Type=header
```

You are welcome to convert the times to string format using *ctime()* instead. Note that your *SetAttribute()* routine should return an error if a program tries to set any of these attributes for a file.

System Calls

Completion of the *setattr* and *listattr* commands using user-level routines *SetAttribute()* and *GetAttribute()* is worth 15 of the 35 points for the project. For an additional 10 points on the project you need to take the *SetAttribute()* and *GetAttribute()* routines and push them into the kernel as system calls.

Their system-call definitions are the same as their respective user-level routine definitions. These should be straightforward system calls to add to the Linux Virtual File System, although note that calls to other system calls within the kernel need to be prepended with "*sys_*" so that making a call to *mkdir()* is done using *sys_mkdir()*. You should pay careful attention to the Fossil Lab Web page on "Adding a System Call to Linux" and at how similar system calls are implemented, particularly in transferring data to and from kernel space.

Once you have added these system calls, your *setattr* and *listattr* should perform exactly the same after recompilation with the system calls in place. You should not link in your user-level versions of these routines at this point.

Note: When writing kernel code, you will want to print messages to stdout, as you do in *printf()*. Since many parts of the kernel may not have access to the stdio library, kernel developers wrote their own version of *printf()* called *printk()*. *printk()* basically behaves the same as *printf()*, in terms of formatting. Furthermore, *printk()* also writes messages to the log file `/var/log/messages`, so you can view output there in case your modified OS crashes. You might add prefixes to your *printk()* messages, such as ”CEW: ” or ”Fossil: ” so you can more easily pick out your messages from the log file. Be careful! If you have *printk()* messages in a part of the kernel that is accessed frequently it can fill up your log file quickly. When this happens, your system can become unstable. Check the size of your log file (using *ls -l*) and the disk space that is free (using *du*) frequently.

You are advised to take a conservative, incremental strategy for developing these system calls (and any kernel work). First focus on putting *printk()* statements into some of the key parts of the

Linux virtual file system (for calls such as *stat()*, *open()*, *read()* and *write()*) to build up confidence as to where to add your modifications.

Removal of Attributes

Once your Linux kernel supports the creation and retrieval of user-defined attributes the next obvious step is to be able to remove attributes and to handle attributes correctly when files themselves are removed. For removal of attributes, you need to create another system call with the following interface:

```
ret = RemoveAttribute(char *filename, char *attrname)
```

This system call will remove a specific attribute for a given file and delete the attribute directory if this is the last user-defined attribute. Obviously none of the implicit file attributes can ever be deleted and attempts to do so should return an error. If the attribute is "`ALLUSER`" then the call should remove all attribute files and the attribute directory for the given file.

You should also create a command called *rmattr*, which has the following syntax:

```
rmattr attr file(s)
```

This user-level command will invoke the *RemoveAttribute()* call for each of the files in the list.

You also need to ensure that when a file is removed (system call *unlink()*) that any user-defined attributes for this file are also removed. You will need to modify the *unlink()* system call to take this action. Thus the command "*rm foo.c*" will remove the file *foo.c* and any attributes associated with the file.

You also need to investigate other commands that remove or rename files. Such commands are *mv* and *cp* (there are others, but at the minimum you need to handle these). You can use the utility *strace* to trace the list of system calls invoked for a command. This utility will help you in tracking which other system calls need to be modified to correctly handle attributes.

Additional Work

Completion of all portions up to this point define the basic objective for this project. These portions are worth a total of 30 out of the 35 points for the project. For the final five points of the project, you need to implement one of the following directions for additional work:

1. *Improve the kernel implementation of user-defined attributes.* For this direction you could modify the inode structure of the WPI file system (it is not suggested you do so for all file systems) so that the inode of the attribute directory entry is included as part of the file's inode. To avoid expanding the inode size, you could remove the `a_time` entry and replace its space with the inode. You could also look to see how the *ext2* file system organizes the inode to include its support of attributes.

2. *Create a new command queryattr to allow flexible queries for files based on their attributes.* The syntax for the command is as follows. It should output the files from the given list that match the query.

   ```
   queryattr querystring file(s)
   ```

   Examples of the types of queries you might support are (be sure to include a description of the exact syntax and features for your queries):

   ```
   % queryattr ’Size>1000’ *.c *.C
   bar.c
   foobar.C
   % queryattr ’Mtime>1 day’ *
   bar.c
   foo.c
   % queryattr ’Programmer=C*’ *.c *.C *.h
   foo.c
   foo.h
   % queryattr ’Mtime>1 day && Programmer=C*’ *
   foo.c
   ```

3. *Create an application that uses user-defined file attributes.* This application should *clearly* take advantage of attributes in a way that would not be easy if user-defined file attributes were not supported.

Submission of Assignment

Details on submission of the project will be provided near the project completion date.

References

[1] Dominic Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann, 1999.