CSCI 509 - Operating Systems Internals
Assignment 7: More File-Related System Calls

Points: 205

1 Overview and Goal

In this assignment, you will implement four more syscalls relating to files: OpenDir, ReadDir, Dup, Pipe, ChMode, Link and Unlink. These will allow user programs a wider range of functionality making more programs possible, for example "ls" which will list files in the directories of the ToyFs. Pipes will also provide more practice in interprocess communication and synchronization.

2 Download New Files

Download the files for this assignment from the canvas assignment page.

The following files are new to this assignment:

```
TestProgram5.h
TestProgram5.k
```

The following files have been modified from the last assignment:

```
makefile
```

The makefile has been modified to compile TestProgram5 and add it to the DISK. All remaining files are unchanged from the last assignment. Remember to check out "main" and then add these files to your "os" directory on the main branch.

3 Kernel and System Constants

To make sure you have the correct constants as were used in the reference implementation, set your constants to the same as in assignment 6.

4 The Tasks

Implement the following syscalls in the following order:

- OpenDir (15 points)
- ReadDir (30 points)
- Dup (10 points)
- Pipe (70 points)
- ChMode (15 points)
- Link (20 points) Extra Credit
- Unlink (35 points) Extra Credit

In implementing these system calls, you may have to upgrade code you have already implemented. For example, Sys_Read will need to be upgraded to read from pipe. Currently, it just reads from a ToyFs file. Once you have a pipe, Handle_Sys_Read (and Handle_Sys_Write) will have to look at the file descriptor and call the correct object to do the read, the ToyFs object or the Pipe object.

5 Implementing Handle_Sys_OpenDir

Handle_Sys_OpenDir is very similar to Handle_Sys_Open. You should copy the filename to a local buffer and then call FileManager.Open. In this case, this system call does not have flags or mode. So your call should use O_READ for the flags and 0 for the mode. Also, in this case, you want to open a directory and an error should be returned if the call to Handle_Sys_OpenDir attempts to open anything other than a directory. You can do this by setting the 4^{th} parameter in your call to true "true". (This assumes you did a proper check for this in Assignment 6.) The return from FileManager.Open will return a file descriptor or -1 that you then return to the user.

There are several more items associated with the Sys_OpenDir system call that you need to remember in other system calls. You may need to add code to deal with these issues. They are:

- The Close system call must correctly close a file opened by Handle_Sys_OpenDir.
- Handle_Sys_Read and Handle_Sys_Write should fail on an open directory. The error should be E_Permissions.

- Handle_Sys_Seek should fail on any pointer other than 0. A 0 offset should reset a open directory to start at the first entry again. The error should be E_Bad_FD.
- Handle_Sys_ReadDir must work only on an open directory. It must return an error if a file is given as a parameter. The error should be E_Not_A_Directory.

6 The ReadDir System Call

Like other Handle functions, Handle_Sys_ReadDir needs to check the arguments. You need to check that the file descriptor is valid and the user pointer is valid. It must store data in the user page and the size of the buffer must be as big as the largest possible directory entry. This value may be computed by the function entSize(255) which is defined in the Syscall.h/k files. Once you have verified the arguments, you then need to call the ToyFS.ReadDir with the OpenFile and the user buffer pointer.

In ToyFS.ReadDir function, you can verify that the OpenFile is a directory. (This could be done in the Handle function, but it is a good check to do here just in case another place calls ReadDir.) The primary method needed here is the OpenFile.GetNextEntry. Call it with an argument of 255 and it will return a pointer to a dirEntry. GetNextEntry may return null that says there are no more entries in this directory. Once you have a valid entry, you can copy the string to the virtual address space using CopyBytesToVirtual. You need to use the function entSize and the size of the "ent.name" array to calculate the number of bytes to copy. This returns 0 success and -1 on any fail

7 The Dup System Call

The Dup system call is needed by several user programs for their proper operation. The prototype for the user level call is:

```
    Sys_Dup  (fd: int) returns int
```

The Handle_Sys_Dup is one of the few system calls where most processing is done by the Handle routine. First, make sure you are handed a valid file descriptor. If so, you need to find an empty file descriptor. Both of these should be handled by a helper routine. If it is a valid file descriptor and you can get a new file descriptor, all that needs to be done is to add a NewReference in the new file descriptor to the OpenFile referenced in the old file descriptor. Once that is done, you just return the new file descriptor number.

8 The Pipe System Call

The Pipe system call creates a pipe for use by the process. The prototype for the user level call is:

```
    Sys_Pipe (fds: ptr to array [2] of int) returns int
```

This syscall will create a new pipe and return the file descriptors which refers to the pipe in the array. If the pipe is successfully created, the system call returns 0. If something goes wrong, the system call returns -1. On success, the file descriptor of the "read end" is put into fds[0] and the file descriptor of the "write end" is put into fds[1].

The Handle_Sys_Pipe function does the usual jobs. In this case, it validates the user pointer to the file descriptor array. Remember, this is an array of two integers and it has another integer that is the length of the array, so there must be 12 valid bytes and you will be saving data in the user memory. Then the handle function should call the FileManager.Pipe method. That method should do the work of getting the pipe ready to use and assigning the the file descriptor values to the user array.

FileManager.Pipe function will be where the pipe is "constructed." The FileManager.Pipe jobs are:

1. Get two new fd locations in the file descriptor array.
2. Get two new OpenFile objects to use. The kind of these OpenFile objects will be PIPE. One of the OpenFile objects will have flags contain O_READ and the other one has O_WRITE. By setting these properly, your tests in the Read and Write syscall processing will catch the wrong kind of an operation.
3. Get a new Pipe object. Notice, the OpenFile object has a reference to a pipe and you will need to assign the reference to this pipe to both of the OpenFile objects. Also, you will need to implement the FileManager.GetAPipe method to get the pipe.
4. Call the Pipe.Open function to get the pipe ready to use.
5. Create a file descriptor array and copy it to the User space.
6. And as usual, if an error occurs, clean up before returning an error code.

First, you will need to implement the FileManager.GetAPipe and FileManager.PutAPipe. The Pipe manager code is similar to the thread manager code. For this resource manager, you should not wait if a pipe is not available on the FileManager.pipeList, which is the list of free pipes. The not waiting is similar to GetAnOpenFile with false as the parameter. You will need to upgrade the

FileManager.Init method to initialize the pipe objects and add them to the free list. The PutAPipe function just puts the pipe back on the pipeList list.

Now for the Pipe object. The following is the definition of the Pipe object as distributed with assignment 3 Kernel.h. You do not have to use the class as defined, but it may cost you more time to invent your own class. The following description of how a pipe works assumes this definition?

```
class Pipe
    superclass Listable

    fields
        bufferFrame: int    -- Buffer frame, needs to be acquired at open time
        head, tail: int     -- Circular buffer
        numberOfUsers: int  -- Should start at 2 and go down on closes
        pipeMutex: Mutex
        charsInPipe: int
        readQueue: Condition
        writeQueue: Condition
        writer: Condition

    methods
        Init () -- Run once at kernel start up
        Open () returns bool
        Read (buffer: ptr to char, sizeInBytes: int) returns int
        Write (buffer: ptr to char, sizeInBytes: int) returns int
        Close ()
  endClass
```

The pipe acts as a FIFO. (This is a variant of the producer consumer problem from assignment 2, but doing variable amounts of producing and consuming.) Data written to the Pipe is to be read back in the same order. This requires that the Pipe contain a buffer to hold the characters between the write and the subsequent read(s). That brings up the question: “How many bytes should be in the buffer?” Only one byte is really necessary but more will allow greater efficiency for programs using pipes when a producer generates data faster than the consumer is reading. Using a frame is what the design handed to you uses. The field “bufferFrame” is to be used as the physical address of a frame in memory being used as buffer. You can get a single frame from the FrameManager using GetAFrame in the Open method. On final Close of your Pipe object, you need to return that frame back to the FrameManager. You use PutAFrame() to return the pipe buffer.

For the operation of the pipe, you will need some way to control the concurrency and synchronization between producers (writes on the write end) and consumers (reads on the read end). A writer adds data to the buffer. When the buffer is full, any process trying to write to it must be suspended. If the writer is writing more than can fit in the buffer, it should do a partial write so the buffer is completely full and then go to sleep. When the writer runs again, it does the same thing until it can write all of the characters to the buffer and return to the user program. This process of "add some characters to the buffer and sleep" can happen multiple times. A big write can fill the buffer, and then small reads can restart the writer when the buffer is nearly full.

A reader takes data out of the buffer and returns it to the user process. It will copy the minimum of the bytes requested and the number of bytes in the pipe buffer. When the Read method is running and there are bytes available, it needs to transfer the correct number of bytes and then return that number to the user. The Read method will put the reader to sleep **only** if there are no characters in the buffer when the Read starts. When a Read completes, it makes room in the buffer and if a writer is suspended, it must restart the writer. Multiple readers will wait only if the pipe is empty and multiple readers ask to read.

Notice the difference here between read and write. Once there is at least one character available, the read then proceeds to get what is available up to the maximum requested and then returns from the call. A reader will not block multiple times. A write must write the entire buffer before it returns and as such, it may block multiple times. Also, when one process starts writing to the pipe, no other process may write to the pipe until the original write is complete. This is very much like the fair waiting in the assignment 2 game and for the the frame manager. This is why the Pipe object has a "writeQueue" and a "writer" condition variables. There is only one active writer and the active writer waits on the "writer" if it can not complete a write. All other writers must wait on the "writeQueue" until the active writer is complete and then a writer on the "writeQueue" can be signaled to become the active writer.

Let us consider the synchronization between reads and writes. First, read will only block if the buffer is empty. Therefore, when a writer puts characters in the buffer, it must restart a reader even if the writer must block. Also, when a reader is done reading, if there are still characters in the buffer it needs to restart a blocked reader. Also, when a reader takes out data from the buffer, it needs to restart the active writer because there is now room in the buffer for the writer to continue. Finally, as stated above, when a writer is complete and ready to return to the user, it needs to restart the first writer waiting in the write queue.

To access the read and write methods on the pipe, you need to upgrade your Handle_Sys_Read and Handle_Sys_Write functions. At this point, your Handle functions should just call the ToyFs ReadFile or WriteFile methods. You now need to change that code to look at the file type and then choose which method to call. A read on a FILE should still call ToyFs.ReadFile, but if the read is on a PIPE, you need to call the Read method on that pipe. The upgrade should be similar in both of these Handle functions.

A difficult issue is that of closing the pipe. Remember, that the OpenFile object has a "number of users" field. The NewReference method increments this number when used to copy a pointer to that open file. Pipes are most often used with a fork. Typical uses are "Sys_Pipe(); Sys_Fork(); ..."

where the parent and the child then communicate via the pipe. When a fork is processed, it will copy the file descriptors, incrementing the “number of users”. This includes any file descriptors that are pipes and with pipes, it is expected that there will be multiple users of the pipe OpenFile objects.

When an OpenFile object is “closed”, it decrements the “number of users” and only if that goes to zero is the OpenFile object fully closed and recycled. We expect that a “pipe Openfile” may be closed several times and still have a reference in some file descriptor of some process. Again, that is what the “number of users” field indicates. The statement “when all write ends of a pipe are closed” means that the OpenFile associated with a pipe has the “number of users” go to zero and that OpenFile is closed.

When the “number of users” does go to zero, if the OpenFile is a FILE or a DIRECTORY, the FCB will be released. For the PIPE case, instead of releasing an FCB, it must close the pipe object. You will need to update the FileManager.Close code that closes an OpenFile. When the OpenFile is a PIPE, the valid field will be the pointer to the pipe object, not the FCB. When the OpenFile “number of users” goes to zero, we then need to call the Close method on the Pipe object.

Next, we look at the operation of the Pipe Object. The Pipe object also has a “number of users” field and should be set to 2 in the Pipe.Open method. Since two OpenFile objects will point to a Pipe object, both OpenFile objects must be closed before we can reuse the Pipe object. When the pipe system call is being run, a Pipe object is allocated and the Pipe.Open method is called. It needs to set up the pipe for use. This includes getting the frame for the buffer and initializing all the fields for use. (The Mutex and Condition fields should be initialized in the Init() method and should not be initialized on every Open.) This also sets up for correct processing of Pipe.Read and Pipe.Write that does the core of the work for processing reads and writes to the pipe.

When one end is closed completely, that is the OpenFile that represents a PIPE is closed, the Pipe.Close method should be called. This now should change how the Pipe works. Notice, at this point only one OpenFile now has access to the Pipe object and it is either a “write end” or a “read end”. So the Pipe object will not be getting both read or write calls. Only one kind of call can make it to the Pipe object. So the Pipe just needs to know how many OpenFile objects point to it. So the first call to Close can just decrement the “number of users” to one. But there is one other job that needs to be done before the return. Any process waiting on the pipe must be restarted. Give that one end is closed, it will only be blocked readers or blocked writers. Once waiting processes have been restarted, the Close method can return.

Now, the Pipe object has only one user. This must change the operation of the Read and Write methods. And again, we notice that only Reads or Writes may happen after the first Pipe.Close call so we don’t have to keep track of which end of the pipe was closed. Writes at this point have no reader to read any data, so writes must return an error. Reads must return data in the buffer if there is data there. Finally, a Read when there is no data in the buffer and there is only one user of the pipe just returns the value zero. This indicates and “end-of-file” rather than an error.

On the second call to Pipe.Close, which can be detected by the number of users, the frame must be freed and the pipe must be recycled. Setting “bufferFrame to -1 is an easy way to indicate that the bufferFrame field is not pointing at a valid frame. Remember to put the pipe back on the free list by calling the FileManager PutAPipe method.

9 The ChMode System Call

This system call should also be relatively easy to implement. It is similar to the Sys_Stat you had to implement in assignment 6, but in this case you are setting the permission bits rather than getting them. There are three places that need work to implement this system call.

1. Handle_Sys_ChMode: Here you get the file name and then call the ToyFS ChMode method.
2. ToyFS ChMode: This is where you find the file. You can get the inode number and see if there is an existing FCB for the file. If so, you can use the existing FCB to call the InodeData SetMode method. If not, you will need to get InodeData, read the inode from the disk and then call the SetMode method.
3. InodeData SetMode: This is place that actually sets the bits. Make sure that setting the mode bits do not change the type of file. Remember that changing the mode makes the inode “dirty” and it should be written back to the disk.

10 The Link System Call

The Handle_Sys_Link function just gets the two names from the user program and then calls the ToyFs.Link method, returning the value returned by ToyFs.Link.

The ToyFs.Link method has two parameters, the original name (oldname parameter) and the new name where the link should be put (newname parameter). Link just adds a new entry in a directory that references the same inode and thus the link count on the inode should be incremented. The job of ToyFs.Link includes the following steps:

1. Verify that oldname is valid and not a directory and get the inode.
2. Extract the last element of newname which is the actual directory entry name. (No slashes (/) are allowed in names in a directory.)
3. Open the directory into which the link name needs to be put.

4. Add the entry to the directory which checks for duplicates.
5. Update the inode with a new link count.

Also, the Link system call will need to check for write permission to the "new" directory before adding the new entry. (This is very similar to file create except you already have the inode.) Any errors should leave the inode unchanged and the new directory unchanged. A successful call returns 0. The return value for an error is -1.

11 Unlink

The Unlink system call removes a directory entry in some directory. Unlink first needs to identify the directory which contains the entry that needs to be removed. Once it is determined that the process may remove the entry and the entry is valid (e.g. a valid file name) unlink then removed the entry and decrements the link count in the inode for the removed file. (You should only be changing inode information via the FCB class.) The jobs are essentially the following:

1. Get the FCB for the file that will be unlinked.
2. Open the directory in which the link will be removed.
3. Remove the entry.
4. Decrement the link count via the FCB. (Remember to set the inode data to dirty.)
5. Release the FCB.

The complicating feature of adding this system call is that it is now possible to end up with a link count of 0 in some inode. It is at this point that the space for the file must be reclaimed. When a file has a link count of zero and the file is closed for the last time, it can't be opened again or looked up. We must then delete the file. The includes freeing all data blocks associated with the file. This includes all blocks in the direct block entries (non-zero entries are allocated blocks), all non-zero entries in the indirect block (if there is one) and the indirect block itself. Once all blocks are freed, you then can free the inode.

In review, any time you want to use a ToyFS file, you should have a FCB. Even if two processes open the same file, they should have one FCB. Therefore, if a file is open, there should be a unique FCB for that file with the "number of users" showing how many times it is being used. As long as there is a FCB for the file, the file must not be deleted. Consider the Release method of the FCB class. It currently does:

```
    ----------  FileControlBlock . Release  ----------

      -- Must be called with fileManagerLock locked.

      method Release (lock: ptr to Mutex)
        fcbLock.Lock()
         numberOfUsers = numberOfUsers - 1
         if numberOfUsers <= 0
            -- Delete the file?  -- NOT IMPLEMENTED
            -- Final close, mark unused and release any indirect frame
            relativeSectorInBuffer = -1
            inode.FreeIndirect ()
            inode.number = -1
            fcbLock.Unlock()
            fileSystem.fsLock.Lock()
            fileSystem.fcbFreeList.AddToEnd (self)
            fileSystem.anFCBBecameFree.Signal ( &fileSystem.fsLock )
            fileSystem.fsLock.Unlock()
         else
            fcbLock.Unlock()
         endIf
         endIf
        endMethod
```

As you can see, there is a placeholder comment about deleting the file. You need to add code to this method to detect that a file needs to be deleted and then free all data blocks associated with this file. Finally, you then can free the inode. Since a FCB is used for both a FILE and a DIRECTORY, this code should work to delete both files and directories, although in this assignment you won't be implementing RmDir.

12 The User-Level Programs

The a7 directory contains a new user-level program called:

```
TestProgram5
```

Please change INIT_NAME to be TestProgram5 as the initial process. TestProgram5 contains 6 separate test functions. It is best to get them working in the order they appear although they can be done in any order.

After you have finished coding and debugging, please create a file called “a7-script” that contains the output from a run of each test in TestProgram5. A separate document (A7-Desired-output.txt) shows more-or-less what the correct output should look like. Use the same kernel code to execute all tests.

Please hand submit on canvas a cover sheet that includes a list of any problems still existing in your code when you turned it in. As usual, once you have completed assignment 7 and everything is working like you want it, then create the “a7” branch and push it.

During your testing, it may be convenient to modify the tests as you try to see what is going on and get things to work. Before you make your final test runs, please run your tests with an unmodified version of TestProgram5.k.