CSCI 509 - Operating Systems, Fall 2019
Assignment 6: File-Related Syscalls

Due Date: Tuesday, November 19, 2019
Points: 225

1 Overview and Goal

In this assignment, you will implement the syscalls relating to files and file I/O: Open, Read, Write, Seek, Close, OpenDir, ReadDir and ChMode. These syscalls will allow user-level processes to read and write to ToyFS files stored on the BLITZ DISK file as well as changing their permission mode bits and getting the entries in a directory. The goal is for you to understand the syscall mechanism in more detail, to understand what happens when several processes operate on a shared file system, and to understand how the kernel buffers and moves data between the file system on the disk and the user-level processes.

2 Download New Files

The files for this assignment are available in:
`https://facultyweb.cs.wwu.edu/~phil/classes/f19/509/a6`
The following files are new to this assignment:

```
TestProgram4.h
TestProgram4.c
TestProgram4a.h
TestProgram4a.k
Program1.h
Program1.c
Program2.h
Program2.c
file1
file2
file3
file1234abcd
FileWithVeryLongName012345678901234567890123456789
```

The following files have been modified from the last assignment:

```
makefile
```

The makefile has been modified to compile TestProgram4, TestProgram4a, Program1, and Program2.

All remaining files are unchanged from the last assignment. Remember to check out "master" and then add these files to your "os" directory.

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 following:

Kernel.h:

```
INIT_NAME = "TestProgram4"
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512
MAX_NUMBER_OF_PROCESSES = 10
MAX_STRING_SIZE = 20
MAX_PAGES_PER_VIRT_SPACE = 25
MAX_FILES_PER_PROCESS = 10
MAX_NUMBER_OF_FILE_CONTROL_BLOCKS = 15
MAX_NUMBER_OF_OPEN_FILES = 15
```

UserSystem.k

```
HEAP_SIZE = 20000
```

4 The Task

Implement the following syscalls:

- Open (20 points, create 15)
- Read (40 points)
- Write (30 points, extend 15)
- Seek (15 points)
- Close (10 points)
- OpenDir (20 points)
- ReadDir (20 points)
- ChMode (20 points)

Also, update your code as necessary to account for the possibility of open files and for the "current working directory" concept. (15 points) You'll probably need to modify the following (may not be a complete list):

- (Kernel.h) ProcessControlBlock: uncomment the workingDir and the fileDescriptor array.
- ProcessControlBlock.Init: uncomment the fileDescriptor code.
- ProcessControlBlock.Print: uncomment the fileDescriptor code.

- StartUserProcess: set the working directory to be the root directory.
- Handle_Sys_Fork: copy information for open files and the working directory.
- ProcessFinish: add code to close all open files and the working directory
- Handle_Sys_Exec: use the current working directory as the directory for the call to fileManager.Open.
- Handle_Sys_Stat: use the current working directory as the directory for the call to fileSystem.NameToInodeNum.

Also, you will also need to add code to FileManager.Open and implement the methods SetMode and AllocateNewSector in the InodeData class as part of the work for some of the above system calls.

Note that files may be open in a process that invokes the Exec syscall. These files should remain open in the process after the Exec completes successfully. Does this require a change to your code?

Take a look at the assignment 4 document for details on the specifications of each of these syscalls. Please re-read that document thoroughly. In particular, the following sections of that document contain important information that you'll need to complete this assignment.

```
The "ToyFs" File System
The "toyfs" Tool
The FileManager
FileControlBlock (FCB) and OpenFile
```

5 The Working Directory of a Process

Before you can start working on the system calls for this assignment you will need to go back fix code from previous assignments to deal with the "current working directory" of a process. To the current point, we have ignored this idea.

In the original BLITZ system, there was no need for a "working directory" since there was only one directory in the entire system. With the addition of the ToyFS, file names are now more complex and there are multiple directories with files in them. Each process now needs the idea of a "current working directory" that is the directory used to lookup a simple file name. File names starting with "/" still will need to start at the root directory, but that is the current functionality of the existing code.

First, notice in the OpenFile object, there is a "numberOfUsers" data member. This used to count how many pointers in the system exist to this object so when an open file is "Closed", it knows how many more references to this open file exist. Only when all references to the open file are closed does the OpenFile get closed. This means that in working with the code to process the working directory, we need to use the "NewReference()" method of the OpenFile object every time we need to get a new reference to a OpenFile.

For example, in Handle_Sys_Fork you will need to add code to deal with the file descriptors and the working directory. You must not just copy the pointers, but use the NewReference() method to

get that pointer. That way, when the child terminates and closes its file descriptors and its working directory, the OpenFile is not closed unless it is the last reference to that OpenFile.

6 Implementing Handle_Sys_Open

First, you should review the assignment 4 instructions about the FileManager with special attention on the fileDescriptor array. Then you should start with Sys_Open implementation.

Here are some actions you'll need to take when to implementing the Open syscall:

- Copy the filename string from virtual space to a small buffer in this routine.
- Make sure the length of the filename does not exceed MAX_STRING_SIZE
- Locate an empty slot in this process's fileDescriptor array.
- If there are no empty slots, return -1.
- Ask the FileManager to open the file. (You should read the FileManager.Open code.) The "dir" parameter to open must be the current working directory of the process.
- If this fails, return -1.
- Make sure it is not a directory that was opened.
- If so, close the file and return -1
- Set the entry in the fileDescriptor array to point to the OpenFile object.
- Return the index of the fileDescriptor array element.

To complete the implementation of the Open system call, you need to upgrade FileManager.Open to do processing of the Open flags and mode arguments. There should be a comment in the correct location to alert you where to process the flags. First, the flags specify either reading or writing and the permissions on the file must allow what the flags ask to do. The biggest job is dealing with the O_CREATE and O_MAYCREATE flags. The O_CREATE flag requires a new file to be created and thus, if a file exists, an error must be returned. (It might be a good thing to check this earlier when the name is converted into an inode number. Also, you should delay implementing the file creation code until after everything else is completed.)

7 Implementing Handle_Sys_OpenDir

This is very similar to open, except that you need to make sure that the file you are opening is a directory rather than a regular file. Notice, you don't get any open flags and thus, all you need for opening a directory is O_READ permissions. OpenDir should fail if MODE_READ permissions are not set for the directory. Other specifications are:

- This system call must prepare a file descriptor that can be closed by the Close system call.

- Sys_Read and Sys_Write should fail on an open directory.
- 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.

Note: You will need to add code to Kernel.h/.k for the system calls that are found in Syscall.h but have no code in your kernel code. You may choose to add these one at a time as needed or add all system calls at one time and leave "unimplemented" code in your kernel.

8 Suggestions Concerning the Algorithm For Handle_Sys_Read

The Read system call must be implemented in multiple places. The Handle_Sys_Read code should not do any actual reading. The job of Handle_Sys_Read is to verify the correctness of the arguments passed via the system call and determine the proper subsystem that needs to actually perform the read. For this assignment, the proper subsystem is ToyFs as accessed by the variable fileSystem. In the next assignment, you will need to expand this to deal with a serial device and a pipe.

Handle_Sys_Read should begin by checking that the fileDesc argument really is valid. The user must provide a legal index into the fileDescriptor array and the file referenced must have been previously opened. If not, the Read syscall should return -1 with the error E_Bad_FD.

Once you have a valid fide descriptor, your next check is to make sure the file was opened with the correct open flag that allowes reading. For example, if the file was opened with only O_WRITE, the read system call should fail. This is a "permissions" error.

You'll also need to check that the requested number of bytes (the sizeInBytes argument) is not negative. If it is, then return -1 and error E_Bad_Value.

We'll discuss problems related to the buffer argument later. This argument must be checked for being a valid pointer and that the entire buffer specified is valid and writable.

When a user-level program invokes the Read syscall to read a disk file, it uses the ToyFS filesystem. Read must call the ToyFs.ReadFile method to read from a ToyFs file. The call asks for a sequence of bytes to be fetched from a file on disk. Unfortunately, this sequence may be located in several sectors in the disk file, so several calls to DiskDriver.SynchReadSector will be needed to read all the data.

Each call to DiskDriver.SynchReadSector will read an entire 8K sector from the disk into memory. The caller will provide the address of where in memory to read the sector, which we can refer to as the "sector buffer" area in memory.

The data requested by the user-level process will not necessarily begin or end on a sector boundary. So some buffering and movement of the data will be necessary. Take a look at FileManager.SynchRead, which will perform a lot of the work you'll need to do when implementing the ReadFile method. In particular, it will call DiskDriver.SynchReadSector to read a sector into the sector buffer and then it will copy the desired bytes from the sector buffer to wherever they should go. (This is called the "target address.")

Each FileControlBlock already has a sector buffer associated with it. This is an 8K memory region that was allocated from the pool of memory frames when the FileManager was initialized. [You do not need to allocate any memory buffers in this assignment.]

If the requested sequence of bytes spans several sectors, FileManager.SynchRead will read as many sectors as necessary (re-using the one sector buffer) and, after each disk I/O completes, it will copy bytes in several chunks to the target area. (Also, if the sector buffer is dirty before it starts, it will first write the old sector out to disk.)

When a user requests a sequence of bytes to be read from a file, the user provides a pointer to a "user-space target area," which tells the kernel where to copy this data to. User-level code often refers to its target area as a "buffer" but be careful to avoid confusing this area with the "sector buffer," which is part of the FileControlBlock in kernel space. The user will supply a virtual address for the user target area by supplying values called buffer and sizeInBytes to the Read syscall which will then passed to ReadFile..

When implementing the ReadFile method, you'll need to translate the target address from a virtual address into a physical address, so you can know where in memory to move the data.

Unfortunately, the user target area may cross page boundaries in the virtual address space. In general, the pages of the user's address space will not be in contiguous memory frames. FileManager.SynchRead cannot deal with this; it expects its target area to be one contiguous region in memory.

This means that you cannot simply call FileManager.SynchRead once to get the job done. You'll need to break the user target area into chunks such that each chunk is entirely within one page. Then, you can call FileManager.SynchRead to read each of these chunks in a loop.

Below is some pseudo-code showing how this loop might work. (This loop would be in the ToyFS.ReadFile method.) It works by breaking the entire sequence of bytes to be read into several chunks. Each chunk is entirely on one page and does not cross a page boundary. It computes the length and starting address of each chunk. It translates the starting address into a physical address. Then it calls FileManager.SynchRead to read the chunk and moves on the next chunk.

The key variables are:

virtPage Virtual address into which to read the next chunk (virtual address page number)

offset Virtual address into which to read the next chunk (offset into the page)

chunkSize The number of bytes to be read for this chunk

nextPosInFile The position in the file from which to read the next chunk

copiedSoFar The number of bytes read from disk so far

At the beginning of each loop iteration, virtPage and offset tell where the first byte in the next chunk should go. Each iteration computes the size of the next chunk, does the read, and then adjusts all the variables. nextPosInFile tells where in the file the next chunk to be read begins. Initially, it will be given by the current position in the file, but will be adjusted after each chunk is read.

```
virtAddr = buffer asInteger
virtPage = virtAddr / PAGE_SIZE
offset = virtAddr % PAGE_SIZE
copiedSoFar = 0
```

```
nextPosInFile = ...

-- Each iteration will compute the size of the next chunk
-- and process it...
while true

  -- Compute the size of this chunk...
  thisChunkSize = PAGE_SIZE-offset
  if nextPosInFile + thisChunkSize > sizeOfFile
    thisChunkSize = sizeOfFile - nextPosInFile
  if copiedSoFar + thisChunkSize > sizeInBytes
    thisChunkSize = sizeInBytes - copiedSoFar

  -- See if we are done...
  if thisChunkSize <= 0
    exit loop

  -- Check for errors...
  if (virtPage < 0) or
     (page number is too large) or
     (page is not valid) or
     (page is not writable)
    deal with errors

  -- Do the read...
  Set "DirtyBit" for this page
  Set "ReferencedBit" for this page
  Compute destAddr = addrSpace.ExtractFrameAddr (virtPage) + offset
  Perform read into destAddr, nextPosInFile, chunkSize bytes

  -- Increment...
  nextPosInFile = nextPosInFile + thisChunkSize
  copiedSoFar = copiedSoFar + thisChunkSize
  virtPage = virtPage + 1
  offset = 0

  -- See if we are done...
  if copiedSoFar == sizeInBytes
    exit loop

endWhile
```

Note that we are marking the page in the user's virtual address space as "dirty." Think carefully about why a "read" would mark the page dirty. When implementing the Write syscall, the operation

will not cause the page to become dirty!

When a user-program reads data from disk to memory, the page in memory is changed. So the page that receives the data from disk must be marked as dirty. When a user-program writes data from memory to disk, the page in memory is unchanged. The page would not be marked dirty.

You also need to set the Referenced Bit for all pages accessed, regardless of whether the user is invoking the Read or Write syscall, since in either case, the page has been used.

Normally, the MMU will set the Dirty Bit and Referenced Bit when appropriate, but this will only occur when paging is turned on. When the kernel handles the Read and Write syscalls, it will be accessing the pages directly, using their physical memory addresses. This will bypass the MMU, so the kernel code will need to change the bits explicitly.

[Although it doesn't effect our OS, setting the Dirty Bit correctly will be necessary if we implement paging in a later assignment. A failure to set the dirty bit correctly might occasionally result in data that gets lost when a page that should be copied to disk is not copied. A failure to set the referenced bit may have repercussions for the page replacement algorithm, causing it to malfunction. The resulting thrashing or poor performance might be very, very difficult to debug and fix!]

It is possible that the user will provide bad arguments to the Read syscall. For example, the user target area may lie outside of the virtual address space or may be in a page that is not writable. In such a case, the kernel should detect the error and return -1. Furthermore, if such a problem with the parameters occurs, the kernel should not perform any disk operation. In other words, the user target area should be entirely unchanged. (User arguments should be checked for being valid before calling ToyFs.)

This is not how the above code works. If, for example, the user target area runs past the end of the virtual address space, the above code may read several chunks successfully before encountering the error and aborting. In fact, if proper checks have been made earlier, the above code could assume that all requested addresses are valid.

To perform correctly, you'll actually need to do the work using two loops. You'll execute one loop to completion, then execute the second loop. Both loops will be just like the one shown above, except that the first loop will not actually perform the reading. The sole purpose of the first loop is to check the user target area and return from the syscall if problems. The first loop should be done in the syscall handler code. The second loop will repeat the computations exactly and will also perform the reading operations. The second loop will be in the ToyFs code for ReadFile.

FileManager.SynchRead will never fail; as coded it always returns true.

9 Implementing Handle_Sys_Write

A similar approach to Handle_Sys_Read can be taken for Handle_Sys_Write. The complicating factor for Sys_Write is the point where you are writing past the end of the existing file. Many of the write tests write over existing data in the file. There is one set of tests that checks to see if you can "extend the file". Extending a file to use the entire last block of the current files is easy. All you do is continue to write until the write is finished and then you need to update the inode's fsize field to accurately show the end of the file. (Remember to mark the inode as "dirty" since you changed data in the inode.) The real problem for extending a file is when you need to allocate

a new block to hold the write data. Long writes could possibly need to allocate many new data blocks, but since you are breaking your writes up into block size calls to FileManager.SynchWrite, you will be just extending one block at a time. Also SynchWrite already has the current code:

```
bytesToMove = Min (numBytes, PAGE\_SIZE - offset)
if offset == 0 && bytesToMove == PAGE\_SIZE
      -- No need to read the sector first
      open.fcb.Flush()
elseIf fcb.relativeSectorInBuffer != sector
      -- Read the sector before we do a partial write
      if !fcb.ReadSector (sector, true)
          fileManagerLock.Unlock()
          return false
      endIf
endIf
```

This code is making sure that partial writes are added to the file and current data is not lost. Therefore, the current data must be read before adding new data. fcb.ReadSector when given true for the second argument will try to allocate a new disk sector if there is no sector already allocated. ReadSector calls inode.AllocateNewSector(logicalSector) to get a new sector allocated. This is the primary routine you need to implement to extend files. You have two cases to deal with. The first is if the logicalSector is a sector is in the direct pointers in the inode. The second is if the logicalSector is not in the direct pointers and you must use an indirect block. (ToyFS allows for single and double indirect blocks, but you need only implement the single indirect blocks. You may generate a fatal error if a file needs a double indirect block.) First, you must allocate a free data block on the disk. You do this by calling the method ToyFs.AllocDataBlock(). The return value from this method is the pointer you need to store in the direct or indirect pointer locations. You must then update the number of blocks allocated in the inode and remember to set the inode as "dirty" so it will be saved shortly. For a direct block, you just enter the pointer in the direct block and you are done. For a block in the indirect block, you have a more difficult job. If this is the first time the indirect block is needed (inode.indir1 == 0), you will need to allocate an indirect block and initialize it to all zero pointers. Notice, you have a couple of methods in the InodeData class already implemented for you: GetIndirect() will read in the indirect block into a frame and set the indSec field to point to this frame. WriteInode() will write a "dirty" inode back to disk. Next, you need to store your read data block pointer to the correct entry in the indirect block. SaveIndirect() is a method that saves this data. Any time you add to the indirect block, it should be saved. Finally, you should update the SynchWrite code given above to make sure it calls ReadSector anytime it needs to allocate a new data block and ToyFS.WriteFile needs to update the file size in the inode and write back the inode to the disk before returning from this system call.

This code uses the FileManager.GetAFrame method to get a frame in which to store the indirect block. So your access to the indirect block ends up having to use the physical address of this frame. Also, when a file is closed after the last user is done using the file, any frame used to hold an indirect block is released by calling FileManager.PutAFrame. (If you have not implemented this method, please do it now.)

To help in your debugging, you can print the inode and it will show you your direct and indirect pointers.

10 Implementing Handle_Sys_Seek

Here are some actions you'll need to take when to implementing the Seek syscall:

- Lock the FileManager since we will be updating shared data (the OpenFile.currentPos)
- Check the fileDesc argument and get a pointer to the OpenFile object.
- Make sure the file is open. (Recall that a null entry in fileDescriptors means "not open.")
- Deal with the possibility that the new current position is equal to -1, which has a special meaning. (Seek to the end of the file.)
- Deal with the possibility that the new current position is less that -1. (Zero is okay.)
- Deal with the possibility that the new current position is greater than the file size.
- Update the current position.
- Return the new current position.

Remember to unlock the FileManager, regardless of whether you are making a normal return or an error return.

11 The Close System Call

The Close system call should just "undo" the work of the Open and OpenDir system calls. This should be an easy system call to implement. Remember to "flush" any modified data. (Any previous document's reference to "CloseDir" should now be read just as "Close".)

12 The ChMode System Call

This system call should also be relatively easy to implement. You will need to look up the file in a similar way as Stat did. Once you have an inode (as part of a FileControlBlock), you call the SetMode method of the InodeData class. You should check the value of the mode passed and use only the meaningful bits. You also need to implement the SetMode method of the InodeData class. Remember that changing the mode makes the inode "dirty".

13 The ReadDir System Call

The primary methods used to implement this system call are found in OpenFile. Read the methods Lookup and GetNextEntry. These should help you to figure out how to return the needed data for ReadDir.

14 Implementing file creation

You should have already implemented extending a file. This capability should allow you to just have to create an empty file and then let the existing code add data to the file. The primary methods you need to use to implement file creation is OpenFile.AddEntry() and ToyFs.AllocInode(). AddEntry() adds a new entry to a directory. Note, you must have the directory open, at least in the kernel, to add a new entry. Often this is the current working directory, but may be any directory in the file system. And you need a new allocated inode before you add a new entry in the directory.

15 The User-Level Programs

The a6 directory contains a new user-level program called:

`TestProgram4`

Please change INIT_NAME to be TestProgram4 as the initial process. TestProgram4 contains 25 separate test functions. It is best to get them working in the order they appear. Clearly, many of the tests can't be successful if you can't open a file.

After you have finished coding and debugging, please create a file called "a6-script" that contains the output from a run of each test in TestProgram4. A separate document (DesiredOutput.pdf) shows more-or-less what the correct output should look like. Use the same kernel code to execute all tests.

Please hand in a cover sheet that includes a list of any problems still existing in your code when you turned it in. If you want comments on your code, please hand in a "diff -u -U10" of Kernel.k between assignment 5 and 6. As usual, once you have completed assignment 6 and everything is working like you want it, then create the "a6" 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 TestProgram4.c.

16 Desired Output

There is some sample output, collected together into a separate file called DesiredOutput.pdf in the a6 directory. It does not contain output from all the tests.