CSCI 509 - Operating Systems Internals
Assignment 6: File Related System Calls

Points: 185

1 Overview and Goal

In this assignment, you will implement five syscalls relating to files and file I/O: Open, Read, Write, Seek, Close, Stat and ChDir. Optional file creation and permission checking in Open are available for extra credit. These syscalls will allow user-level processes to read and write to ToyFS files stored on the BLITZ DISK file as well as find out information about a file on the DISK. 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 on canvas. The following files are new to this assignment:

```
TestProgram4.h
TestProgram4.k
TestProgram4a.h
TestProgram4a.k
Program1.h
Program1.k
Program2.h
Program2.k
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, Program2 and create a more complex DISK.

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

In 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
```

In UserSystem.k:

```
HEAP_SIZE = 20000
```

4 Bug Fix In Kernel.k

There is a lock bug in ToyFs.OpenLastDir. Please change the lines:

```
-- Get the entry
-- print("OpenLastDir: elName: ") print(&elName) nl()
entPtr = elDir.Lookup(&elName)
```

```
if entPtr == null
```

to:

```
-- Get the entry
-- print("OpenLastDir: elName: ") print(&elName) nl()
fileManager.fileManagerLock.Lock()
entPtr = elDir.Lookup(&elName)
fileManager.fileManagerLock.Unlock()
if entPtr == null
```

5 The Tasks

Implement the following syscalls in the following order:

- Open (30 points)
- Close (15 points)
- Read (40 points)
- Seek (20 points)
- Write (30 points)
- Stat (25 points)
- ChDir (10 points)
- Write/extend (15 points)
- Open/create (20 points extra)
- Open and Exec/permissions (10 points extra)

Also, you will need to implement the method **FileManager.Open** as part of implementing the syscall Open and implement the method **AllocateNewSector** in the InodeData class as part of the work for extending a file during Write. Also, **InodeData.SetMode** will also need to be implemented if you choose to implement the create feature of Open. Details will be given later.

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
```

Most system calls need to validate their arguments. You have been doing this already and you will need to do it again. You should have already written the **Valid_User_Pointer** function. There are two more helper functions you will need to complete for this assignment. They are **Valid_User_FD** and **Get_Open_FD**. **Valid_User_FD** should make sure the passed fd indexes a valid fileDescriptor in the ProcessControlBlock of the current process. This means that it should have a pointer to an openFile if the index is in bounds. **Get_Open_FD** should return the first fileDescriptor that is not open, that is it should have a null pointer in the file descriptor. The second parameter specifies where to start in the array. Most calls will use 0 as the starting point, but one useage will need two new file descriptors, so it can start start at number other than 0. If there is no available file descriptor, it should return a -1. You should implement these when you need them to complete a system call.

Finally, you should enable interrupts for most system calls. For most Handle_Sys_* routines, it should be the first thing to do. Quick ones like GetPid don't need to do this, but things like Open, Read, Write and so forth, it is important to have interrupts enabled.

6 The Working Directory of a Process

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. File names not starting with "/" need a starting directory for the relative name. This is where the "working directory" is used.

In assignments 4 and 5, you were instructed in how to use the fileSystem.rootDirectory to initialize the working directory and then to get a "new reference" to the working directory in the Fork

system call. You should also be closing the working directory when a process exits. In this assignment you will implement "ChDir" which will allow a process to change the working directory. There are two places in this kernel where the working directory is very important. First, in the Exec system call when opening a file for execution, the starting directory must be the working directory. You will need to verify you have done this. Also, in processing the Open system call, the working directory is the starting location for any file search.

7 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.

As usual, Handle_Sys_Open does not do a lot of work. It should already copy the filename into a local buffer and return an error if it can't copy the file name. The remainder of the job is handled by the routine FileManager.Open so just call fileManager.Open and return what it returns. The 4^{th} parameter of FileManager.Open allows FileManager.Open to be used for both Sys_Open and for Sys_OpenDir. For Sys_Open the "isDir" parameter is "false".

The routine FileManager.Open has some jobs not directly related to the ToyFS file system. The implementation of FileManager.Open does the following things:

- Validate the "flags" parameter. It must have at least O_READ or O_WRITE defined. If neither of them are set, the flags parameter is a "Bad Value" and you should return -1.
- Locate an empty slot in this process's fileDescriptor array. You have a function "Get_Open_FD" defined, but not implemented. This is a common job needed in at least three places in the kernel. Instead of duplicated code, you should implement this function. It looks through the fileDescriptor array in the current PCB for a descriptor that is not used, specifically it has the null pointer. It then returns the proper index or -1 if the fileDescriptor array has no null pointers.
- If there are no empty slots, return -1.
- Ask the ToyFS to open the file. (You should read the ToyFS.Open code.) The "dir" parameter to ToyFS.Open must be the current working directory of the process.
- If this fails, return -1.
- Check to make sure you have correctly opened a directory or a file based on the isDir parameter.
- If the wrong kind of file was opened, close the file, set the error to "Not A Directory" or "Permissions" and return -1.

- Set the entry in the fileDescriptor array to point to the OpenFile object. Do not use NewReference here because you are not duplicating the pointer but just saving the reference given to by fileSystem.Open().
- Return the index of the fileDescriptor array element.

For extra credit, you may complete the implementation of the Open system call by upgrading ToyFs.Open. If you choose to do these, do them after your required elements have been completed. There are two specific jobs you may do.

1. File Creation: Flags has two possible flags related to file creation. "O_MAYCREATE" says that if a file does not exist, a new one may be created. "O_CREATE" says that a new file must be created and if an existing file is found that an error must be returned after the file is closed. File creation will be discussed later.
2. Permissions: Each file has permissions in the inode: read, write and execute. These need to checked against the flags to make sure if O_WRITE is in flags, the file has write permission and if O_READ is in flags, the file has read permission. If the permission is not there for a given flag, the error is "permissions" and open needs to fail. Remember to release your FCB when an error is being processed. Also, in ExecNewProgram, you also need to check if the file has execute permission.

8 Upgrading the ForkNewProcess and ProcessFinish

You will need to upgrade the code in **ForkNewProcess** to handle copying the file descriptors of processes. Once you have open files in a process, a fork needs to copy the file descriptors to the child's PCB. Since you are copying an OpenFile pointer, you need to use the **NewReference** method. (Using **NewReference** increments a number in the OpenFile so that it knows how many times it should see a file close before actually closing the OpenFile.) This copying is just a simple loop that copies every valid open file in the file descriptor array and makes sure the non-open file descriptor entries are null. You should have a comment in your code saying where you should do this.

Also, you will need to upgrade your code in FinishProcess to close all files that remain open at ProcessFinish time. Depending on your ForkNewProcess code, you may want to set the fileDescriptor to null after closing an open file. Also, you must close the working directory open file as the exiting process is no longer using the working directory.

9 Implementing Handle_Sys_Close

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. The FileManager.Close method is mostly implemented for you already so you could use files for execution. The Handle_Sys_Close system call has three primary jobs.

1. Make sure that the file descriptor parameter is a valid file descriptor. This is again done by implementing a helper function that can be used in other system calls. “Valid_User_FD” should make sure that the file descriptor number correctly indexes the fileDescriptor array and that the fileDescriptor array has a valid pointer. If not, Valid_User_FD should set the error to “E_Bad_FD”. As Sys_Close has no return value, any error just returns. The error value as set by Valid_User_FD can still be retrieved by the user.
2. Call FileManager.Close function with the OpenFile reference stored in the fileDescriptor array.
3. Setting the proper entry in the fileDescriptor array to null to show the file is now closed.

10 Implementing Handle_Sys_Read and reading the file

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 assignment 8, 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. Assuming you implemented Valid_User_FD, this should be easy. The next check is to see if the buffer parameter is valid. Again, your utility routine should do the job. The final check is to make sure the file was opened with the correct open flag that allows reading. For example, if the file was opened without O_READ flag, the read system call should fail. This is a “permissions” error. This is just a check of the OpenFile flags field which must have been set properly in the Open system call processing. Also, you need to make sure the user did not pass an open directory to read. You can only read a directory with the Sys_ReadDir system call. This can be done by checking the OpenFile kind field.

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.

Since the only kinds of files our Blitz OS has to open to this point in the assignments is a ToyFS file, read must call the ToyFs.ReadFile method to read from the ToyFs file. ReadFile will return the proper value for the system call to return, so the final job of Handle_Sys_Read is to call the ToyFs ReadFile method and return the result. Note, ReadFile takes an OpenFile which means that Handle_Sys_Read needs to index the process fileDescriptor array to give ReadFile the proper OpenFile pointer.

10.1 Implementing ToyFS ReadFile

No code is provided for ReadFile. This method takes an OpenFile pointer, userBuffer that is assumed to be correct and points to writable memory and the number of bytes to be fetched from the disk file. Several issues arise in this process. First, this sequence of bytes in the file may be located in several sectors in the disk file, so several calls to DiskDriver.SynchReadSector will be needed to read all the data. Next, the userBuffer may cross memory page boundaries and thus will not be able to write the data across the page boundaries as the pages will not be adjacent in memory.

To start with, consider the DiskDriver. 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 FileControlBlock.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, .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. This would happen if you write to the file and then read from the file in a different sector.

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. FileControlBlock.SynchRead cannot deal with this; it expects its target area to be one contiguous region in memory and it expects a physical address.

This means that you cannot simply call FileControlBlock.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 user page. You could loop over each page in the user buffer and call FileControlBlock.SynchRead to read each of these chunks in a loop.

Below is some pseudo-code showing one way such a 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 FileControlBlock.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. Note: The variable "offset" will be non-zero only on the first iteration of the loop. All other iterations of the loop, you are starting your read at the beginning of the user page and the amount will be to read will be the smallest of PAGE_SIZE, the number remaining bytes to read or the number of remaining bytes in the file.

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

  -- 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" and "referenced". Think carefully about why a "read" would mark the page dirty and referenced. 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 but just marked referenced.

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!]

While the above code checks for errors in the user buffer, your Handle_Sys_Read function should have checked the user buffer and if there was an error, not call ReadFile. It still doesn't hurt to make sure a bad user buffer does not cause a kernel error. In an OS, it may be possible ReadFile is called from code other than Handle_Sys_Read. That is not the case for our OS, but in general that may be the case. Notice, by checking for a bad user buffer in Handle_Sys_Read, the user buffer is totally unchanged if some part of the buffer is valid and some is out of the virtual address range.

If the above code was called without the buffer check, it could modify part of the user buffer. 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.

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

The final thing you need to do in ReadFile is to update the current position in the file control block. This is necessary so the next read (or write) will start reading (or writing) at the next byte in the file. This is the same information that will be changed by the Sys_Seek system call. Since your loop is keeping track of the current location in the file, your code should update this information and then return the number of bytes read. Remember, an any error case should not update the

current position in the file.

11 Implementing Handle_Sys_Write

A similar approach to Handle_Sys_Read can be taken for Handle_Sys_Write. Of course, you would be using the FileControlBlock.SynchWrite method in a very similar loop. The complicating factor for Sys_Write is the point where you are writing past the end of the existing file. In Sys_Read your code may reduce the "chunk size" so it won't try to read past the end of the file. You can't do the same thing here. 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.) Also, remember to update the current position in the file even if you do not extend the file.

The real problem for extending a file is when you need to allocate a new block to the end of the file to hold the data about to be written. Long writes could possibly need to allocate many new data blocks, but since you are breaking your writes up into block size calls to FileControlBlock.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
   && bytePos < inode.fsize
      -- No need to read the sector first
      self.Flush()
elseIf relativeSectorInBuffer != sector
      -- Read the sector before we do a partial write
      if !self.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 to the buffer. It also makes sure to call ReadSector if a new sector is needed. The second argument to ReadSector controls whether the code will allocate a new disk sector if there is no sector already allocated for the requested logical sector. If a new sector is needed, 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 case is if the logicalSector is a sector in the direct pointers in the inode. (e.g. logicalSector is less than 10.) The second case is if the logicalSector is not in the direct pointers and you must use an indirect block. (logicalSector is 10 or greater.) Note that 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.

The steps to do in AllocateNewSector are:

1. You must allocate a free data block on the disk. You do this by calling the ToyFS method AllocDataBlock(). The return value from this method is the pointer you need to store in the direct or indirect pointer locations. A negative return value means there are no more disk blocks available on the disk and you need to return false.
2. 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.
3. For a direct block, you just enter the pointer in the direct block, set the inode as dirty and you are done except writing the inode back to the disk as the inode has changed.
4. For a block in the indirect block, you have a more difficult job.
5. If this is the first time the indirect block is needed, detected if inode.indir1 == 0, you will need to allocate an indirect block. This is done by getting yet another data block, saving the block number in inode.indir1, getting a frame to hold the indirect sector and saving that frame pointer in the field “indSec”, adding one to the block count and then zeroing the frame which sets all indirect pointers to zero. This changes the inode so you need to write the inode.
6. If this is not the first time to use the indirect block, you can make sure your InodeData has read the indirect block from the disk by calling self.GetIndirect().
7. In both cases, you now need to set the correct entry in the indirect block to the block number of the block allocated at the start of the of this process. At this point, you have either created a new indirect block or updated an existing indirect block and need to write the indirect block back to disk. Fortunately, there is an InodeData method called SaveIndirect that does this for you.

You need to remember that anytime you change the inode data or the data in the actual indirect block you need to write that information back to disk. Also, if you run into an error, you need to undo any operations already done, for example, you get a disk block for the file, but you run out of disk blocks for the indirect block. Before you return an error, you need to return the original block you got so you don’t lose the block because the block pointer was not stored correctly.

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

12 Implementing Handle_Sys_Seek

The Handle_Sys_Seek system call needs only to validate the passed file descriptor. If it is valid, you have an OpenFile in the file descriptor array and you need only call the **Seek** method on the OpenFile.

To implement the **OpenFile.Seek** method, there are several steps:

- Verify the file is a regular file or a directory. Those are the only files on which one may seek.
- Lock the FileManager since we will be updating shared data. (**OpenFile.currentPos**)
- Validate the new position. The value -1 means at the end of the file. Zero means at the beginning of the file. Zero is the only value valid for a directory. If the new position is less than -1 or greater than the file length, it is an error.
- If there are no errors, update the current position to the correct value represented by the new position.
- Release the fileManagerLock.
- Return the new current position as updated in the last step.

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

13 Implementing Sys_Stat

This system call just gets information about a file and has similar problems to other system calls in that you need to get the file name from the user process. This adds the requirement of having to copy data back to the user process that is not the return value of the system call. This is similar to Sys_Read in that data is copied into the user address space. In this case, the user passes a pointer (logical address) of a "statInfo" record. (This is defined in Syscall.h.) The size is just the size of the statInfo record. The system call looks up the file and then fills in the information in the statInfo record and then copies it to the user space.

You should have code to get the file name and verify the "statBuf" as being a valid user buffer. Make sure your call to Valid_User_Pointer has toStore set true. All that remains is to call the "fileSystem" Stat function with the correct parameters and return that return value.

To finish Sys_Stat, you need to implement "fileSystem.Stat()". The outline of the Stat method is:

- Find the inode number for the file. This is done by calling the ToyFs.NameToInodeNum method. If it returns a number 1 or greater, that is the inode number. If it returns 0 or less, there was an error looking up the inode number and you must return -1.
- Get the inode's FileControlBlock (FCB). The **ToyFs.GetFCB** is the method you need to call. It always returns a valid pointer. (There is an incorrect comment in the code that says that GetFCB may return null.)
- Create a statInfo record using the FCB inode data you just got. (Should have a locally declared statInfo.)
- Release the FCB.
- Copy the statInfo record to the user space. If there was an error, return -1.
- Return 0 as the value of a successful system call.

14 Implementing Handle_Sys_Chdir

The ChDir system call changes the working directory to a new directory. This is done in two places, Handle_Sys_Chdir and ToyFs.Chdir. As usual, the Handle function just gets the name passed in by the user code and then calls ToyFs.Chdir. The rest of the work is done in ToyFs.Chdir.

In ToyFs.Chdir, use the ToyFs.Open method to open the name passed in from the user. If the Open is successful, you still must check to make sure the new OpenFile is a directory. If there are no errors, they you close the existing WorkingDirectory and then save the pointer to the newly opened directory as the new working directory. You shouldn't use the NewReference method here because this code opened the file. When another ChDir happens or the process exits, this working directory must be closed and if there are no more users, it then recycles the OpenFile object.

15 Implementing file creation

(Extra Credit) 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. A rough algorithm is:

1. Open the last directory in the file name
2. Allocate a new inode
3. Get a FCB
4. Get an OpenFile
5. Set up the inode
6. Add the new entry in the directory opened above
7. Initialize the OpenFile with proper information
8. Save anything that was changed

Remember, errors may occur at any point and you need to "undo" any actions taken to this point. For example, if you can't add the new entry, you have an allocated inode, a FCB and an OpenFile that all need to be released or freed.

16 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 and 3 test functions for the extra credit work that is available. 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. You should be able to get all the standard tests working.

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 sample output is provided on the canvas assignment page that 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 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.k.