

Programming Project 7: File-Related Syscalls

Due Date: _____

Project Duration: One week

Overview and Goal

In this project, you will implement the syscalls relating to file I/O: **Open**, **Read**, **Write**, **Seek**, and **Close**. These syscalls will allow user-level processes to read and write to files stored on the BLITZ DISK file. 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.

Download New Files

The files for this project are available in:

<http://www.cs.pdx.edu/~harry/Blitz/OSProject/p7/>

The following files are new to this project:

```
TestProgram4.h
TestProgram4.c
Program1.h
Program1.c
Program2.h
Program2.c
```

The following files have been modified from the last project:

```
makefile
DISK
```

The **makefile** has been modified to compile **TestProgram4**, **Program1**, and **Program2**. The **DISK** file has been enlarged, since the previous version was too small to accommodate **TestProgram4**, **Program1**, and **Program2**.

All remaining files are unchanged from the last project.

The Task:

Implement the following syscalls:

Open
Read
Write
Seek
Close

Update your code as necessary to account for the possibility of open files. You'll probably need to modify these routines, too:

Handle_Sys_Fork
ProcessFinish

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 project 5 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 project.

The "Stub" File System
The "diskUtil" Tool
The FileManager
FileControlBlock (FCB) and OpenFile

Limitations to the "Stub" Filesystem:

New files cannot be added to the disk. We will not implement the **Create** syscall in this project. All files must already be on the DISK before they can be opened and accessed. (Files can be added to the BLITZ DISK with the **diskUtil** command.)

Also, we will not be able to change the size of a file. If the user-level program invokes **Write** with arguments that would cause the bytes transferred to be beyond the current end of the file, the bytes that are not before the file end should be transferred, but the bytes beyond the file end will not be transferred. The **Write** will return the number of bytes successfully written. For example, if the file size is 10 and

Write is invoked with a length of 5 on a file whose current position is 7, then 3 bytes should be transferred and the 2 bytes beyond the end of the file will not be transferred. The **Write** should return 3.

Implementing Handle-Sys-Open:

Here are some actions you'll need to take when 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`.
- Allocate an **OpenFile** object (see **FileManager.Open**).
- If this fails, return `-1`.
- Set the entry in the **fileDescriptor** array to point to the **OpenFile** object.
- Return the index of the **fileDescriptor** array element.

Suggestions Concerning the Algorithm For Handle-Sys-Read:

Here is a possible approach to implementing the **Read** syscall.

You'll need to 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`.

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

We'll discuss problems related to the **buffer** argument later.

When a user-level program invokes the **Read** syscall, it asks for a sequence of bytes to be fetched from a file on disk. Unfortunately, this sequence may span 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 **Read** syscall. 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 project.]

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.

When implementing the **Read** syscall, 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. 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 to 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.

[Setting the **Dirty Bit** correctly will be necessary if we implement virtual memory in a later project. 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.

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.

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 second loop will repeat the computations exactly and will also perform the reading operations.

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

A similar approach to **Handle_Sys_Read** can be taken for **Handle_Sys_Write**.

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.
- Deal with the possibility that the new current position is less than -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.

The User-Level Programs

The **p7** directory contains a new user-level program called:

TestProgram4

Please change **InitUserProcess** to load **TestProgram4** as the initial process. Then run it several times, once for each of the tests it contains.

After you have finished coding and debugging, please run each test once and hand in the output from each test. A separate document shows more-or-less what the correct output should look like.

Use the same code to execute all tests. Please hand in only one copy of your **Kernel.c** code and do not hand in any output that was produced by a different version of your code!

Do not change **TestProgram4**, except to uncomment one of the lines in the **main** function.

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 recopy **TestProgram4.c** from our directory, so that you get a fresh, unaltered version.

Please hand in hardcopy of **Kernel.c**. You only need to hand in...

Handle_Sys_Open
Handle_Sys_Close
Handle_Sys_Read
Handle_Sys_Write
Handle_Sys_Seek

and any other code you wrote / modified, like **Handle_Sys_Fork** and **ProcessFinish**.

Desired Output

There is some sample output, collected together into a separate file called **DesiredOutput.pdf** in the p7 directory.