NACHOS Files:

We will be using NACHOS files exclusively. NACHOS Files are similar to Unix files except they are stored on a virtual disk that is implemented as one big Unix file. The interface to `Create, Open, Close, WriteAt` and `ReadAt` to those files are defined in `filesys/filesys.cc`, `filesys/openfile.cc` and `filesys/filehdr.h` . You may want to look through those files when getting ready to call these functions for the first time.
The following steps are known to work well. However, you could choose to implement these system calls any way you like.

Guide to an I/O Enabled NACHOS:

1. Create a `SysOpenFile` object class that contains a pointer to the file system's `OpenFile` object as well as the systems `(int)FileID` and `(char *)fileName` for that file and the number of user processes accessing currently it. Declare an array of `SysOpenFile` objects for use by all system calls implemented in part 2.
2. Create a `UserOpenFile` object class that contains the `(char *)fileName` , an index into the global `SysOpenFile` table and an integer offset represeting a processes current position in that file.
3. Modify the `AddrSpace`'s `PCB` to contain an array of `OpenUserFiles`. Limit the number to something reasonable, but greater than 20. Write a method (in `PCBManager`) that returns an `OpenUserFiles` object given the fileName.

4. Implement `Open(char *fileName)`;. This function will use an `OpenFile` object created previously by `fileSystem->Open` (fileName). Once you have this object, check to see if it is already open by some other process in the global `SysOpenFile` table. If so, incriment the `userOpens` count. If not, create a new `SysOpenFile` and store it's pointer in the global table at the next open slot. You can obtain the `FileID` by looking up the name in your `SysOpenFile` table.

   Then create a new instance of an `OpenUserFile` object (given a `SysOpenFile` object) and store it in the `currentThread`'s `PCB`'s `OpenUserFile` array.

   Finally, return the `FileID` to the user.
5. Implement a function to Read/Write into `MainMem` and a buffer given a staring virtual address and a size. It should operate in the same way `AddrSpace::ReadFile` writes into the main memory.
   It may help to put the section of the code in `ReadFile` into a "helper" function called `userReadWrite()` that is general enough that

both `myRead()` and `myWrite()` can call. It need only be parameterized by the type of operation to be performed (Read or Write).

When called by `Write()`, It will read from the `MainMem` addressed by the virtual addresses. It writes into the given (empty) buffer. `Write()`, will then put that buffer into an `OpenFile`. When called by `Read()`, It will write into `MainMem` the data in the given (full) buffer that `Read()` read from an `OpenFile.`

6. Implement `Write(char *buffer, int size, OpenFileId (int)id)`; First you will need to get the arguments from the user by reading registers 4-6. If the `OpenFileID` is == `ConsoleOutput (syscall.h)`, then simply `printf(buffer)`. Otherwise, grab a handle to the `OpenFile` object from the user's openfile list pointing to the global file list. Why can't you just go directly to the global file list?... because the user may not have opened that file before trying to write to it. Once you have the `OpenFile` object, you should fill up a buffer created of size '`size+1`' using your `userReadWrite()` function. Then simply call `OpenFileObject->Write(myOwnBuffer, size)`;
7. Implement `Read(char *buffer, int size, OpenFileId id)`; Get the arguments from the user in the same way you did for `Write()`. If the `OpenFileID` == `ConsoleInput (syscall.h)`, use a routine to read into a buffer of size '`size+1`' one character at a time using `getChar()`. Otherwise, grab a handle to the OpenFile object in the same way you did for `Write()` and use `OpenFileObject->ReadAt(myOwnBuffer,size,pos)` to put n characters into your buffer. pos is the position listed in the `UserOpenFile` object that represents the place in the current file you are writing to. With this method, you must explictly put a null byte after the last character read. The number read is returned from `ReadAt()`.

   Now that your buffer is full of the read, you must write that buffer into the user's memory using the `userReadWrite()` function. Finally, return the number of bytes written.
8. Test your system calls using your own test programs. Now that both projects 2 and 3 are finished, all Nachos test programs should work.

Steps for the filesystem system calls:

1. `Create`: We need to translate the file name passed in just as you did for the Exec system call, so it may help separate this section into a “helper” function called getString(). After doing this we can use fileSystem's Create function to create the file. This can be found in filesys/filesys.cc. Initially we create the file with size of 0. An implementation for this system call is shown bellow.

```
void SystemCall_Create ()
{
   printf (“System Call: [%d] invoked Create.\n”, currentThread->space->getPID());
   int virtualAd = machine->ReadRegister(4);
   char *fileName = new char [256];
   currentThread->space->getString (fileName, virtualAd);
   fileSystem->Create (fileName, 0);
}
```

The subroutine getString() defined in addrspace.cc file, can be implemented as follow:

```
void AddrSpace:: getString (char *str, int virtAd)
{
   int i = 0;
   int physAd = myTranslate(virtAd);
   bcopy (&(machine->mainMemory[physAd]), &str[i],1);
   while (str[i] != '\0' && i != 256-1)
   {
       virtAd++;
       i++;
       physAd = myTranslate(virtAd);
       bcopy(&(machine->mainMemory[physAd]), &str[i], 1);
    }
   if (i != 256-1 && str[i] != '\0')
   {
       str[i] = '\0';
    }
}
```

2. Open : You will need to translate the file name passed in just as you did in the Exec system call. Next you will want to add a new process open file to your list of your process' open files. This will in turn check to see if the system has this file open yet. If it does, the system file table increments the counter for that file. If it does not, the system will add a new system open file which will open the file using fileSystem's Open function. This can be found in filesys/filesys.cc.
3. Close : Using the file id passed in you will remove the open file from your process' list of open files and it in turn will let the system file table know to remove the file if necessary (ie. `if the count > 0 then count--` , otherwise remove file from list and close it). To close the file you can just clear out the reference in your system open file table to the open file. There is no actual close function that needs to be called. As far as the user is concerned you have closed the file.
4. Read : The main thing you need to do is translate the logical buffer address that the user passed in (what you are to read into) page by page and read into the translated address in memory each time you translate a page. To read each page you will call a read function in your structure for the list of your process' open files. This will get the correct open file and handle the moving of the offset in the file. The system open file table is called from the process open file and asked to read. This is a synchronized read (since many processes could be calling this system open file's read). You need to use locks and call OpenFile's ReadAt function. This is in filesys/openfile.h). Remember to keep track of the actual number of bytes written (ReadAt returns this) and let the user know how many were actually read.
5. Write : You will also need to translate the logical buffer provided by the user argument page by page. You will write each page separately as you translate them. Just as Read does, Write will keep track of the offset in the process' open file and will ask the system open file table to write to the desired file. This will occur like Read except you will be calling OpenFile's WriteAt function.