Design Document

Project #2
Due Date: 3/9/2012

Group 2
Daeik Kim
Jonathan Child
Fred Knutson
Kevin Caton-Largent

Part 1: Implementation of various syscalls (create, open, read, write, close, and unlink)

According to various specifications and hints from the project guideline, it is essential that these implementations should be able to withstand any user programs and do not crash. Before venturing onwards, each syscalls should be assigned an integer value so that it can be referred to easily by the coder (This is given from syscall.h). Below are list of various implementations.

- **Create (integer value = 4)**
    a. Opens a new file, creates it if it does not exist.

b. The argument passed onto(a0) is the virtual memory address
c. As explained in the project guide, it returns -1 on failure or any errors.
d. If successful, it returns the new file descriptor.
e. *Pseudo code*
    i. `int handlingCreate(int a0) {`

```
    //Read virtual memory string with 256 bytes as its maximum size

    If (Reading returns null)
    return -1

   else{
     //Create file from calling user kernel with the string name
     //Create is true from filesystem open call
     //Add the created file to descriptorTable and return the index
     }
```

- **Open (integer value = 5)**
  a. Opens file at a given location in the file system.
  b. The argument passed (a0) is the memory address of a string containing the path to the file to open.
  c. Similar to handeCreate, if filename does not exist, it returns -1
  d. If successful, it returns the file descriptor of the newly opened file.
  e. *Pseudo code*
      i. `int handlingOpen(int a0) {`

```
      //Read virtual memory string with 256 bytes as its maximum size
        If (Reading returns null)
        return -1

        else{
        //Open file from calling user kernel with the string name
       //Create is false from filesystem open call
        //Add the opened file to descriptorTable and return the index
    }
```

- **Read (integer value = 6)**
  a. Reads data from a file descriptor into a buffer until given amount of bytes have been read.
  b. Parameter a0 is the file descriptor to read from.
  c. Parameter a1 is the memory address of a buffer to write the data to.
  d. Parameter a2 is the maximum number of bytes to read from the descriptor.
  e. If successful, returns the number of bytes read but returns -1 if there is an error.
  f. *Pseudo code*
      i. `int handlingRead (int a0, int a1, int a2) {`

```
  //Make sure a0 is nonzero, a0 is less than descriptor table size, and
  opened file from descriptor table with a0  does not return null
  //If it does not satisfy the above condition, return -1
  //Open file from descriptor table with a0 as parameter
  //Create array of type byte with length of a2
  //Read file with opened file to keep track of bytes read
```

```
If (bytes read is less than 0)
return -1
else if (bytes read is equal to 0)
return 0

//Create an array of type byte with length bytes read
//Write onto virtual memory to keep track of bytes written
If (bytes written is less than 0)
return -1
return (bytes read)
}
```

- **Write (integer value = 7)**
  a. Writes a buffer of a given length to a file descriptor.
  b. Parameter a0 is the file descriptor to write to
  c. Parameter a1 is the memory address of a buffer of date to write
  d. Parameter a2 is the length of the data to write
  e. If successful, returns the number of bytes written, but returns -1 if there is an error.
  f. *Pseudo code*
     i. `int handlingWrite (int a0, int a1, int a2) {`

```
//Similar to handlingRead, make sure a0 is nonzero, a0 is less than
descriptor table size, and opened file from descriptor table with a0
does not return null
//If it does not satisfy the above conditions, return -1
//Open file from descriptor table with a0 as parameter
//Read virtual memory and store it in string type variable

If (String from reading virtual memory is null)
return -1

else {
//Write file with opened file to keep track of bytes read
return (Number of bytes written)
}
}
```

- **Close (integer value = 8)**
  a. Closes an opened file descriptor, will return -1 if there is no opened file descriptor.
  b. Parameter a0 is the file descriptor to close
  c. If successful, returns 0
  d. *Pseudo code*
     i. `int handlingClose (int a0) {`

```
//Make sure a0 is less than descriptor size(to ensure the pointer
is in right position) and a0 is nonzero.  Also make sure opened
descriptor file isn't null.
//If it does not satisfy the above conditions, return -1

//Close the file descriptor
//Remove the file descriptor
//Return 0
}
```

- **Unlink (integer value = 9)**
    a. Deleted a file from system. If no processes have the file open, the file is deleted immediately.
    b. Parameter a0 is the memory address of a string containing the path to the file to unlink.
    c. If successful, returns 0 but will return -1 if an error occurs.
    d. *Pseudo code*
        i.
```
Int handlingUnlink (int a0) {

    // Read virtual memory string with 256 bytes as its maximum size

    If (Reading returns null)
    return -1
    else
    //Remove the file from file system.
    //Return 0
    }
```

Part 2: Implementation of Support for Multiprogramming

Here is a possible implementation of added support for multiprogramming. A suggestion was made in the project guideline to add a way of allocating machine's physical memory so that different processes do not overlap in their memory usage. The first change would be to implement a linked list of free physical pages.

User Kernel changes:
```
// add a linked list of free physical pages to keep track of the machine's allocated //
physical memory pages

LinkedList availablePages // keeps track of the available free pages
Lock availablePagelock // lock for synchonization purposes
```

It turns out that we ended up not needing the extra helper functions. The code that was needed was in the initialize function of the User Kernel class.

```
public void initialize (String[ ] args) {
// original code
super.initialize(args);

console = new SyncConsole(Machine.console());

Machine.processor().setExceptionHandler(new Runnable() {
       public void run() { exceptionHandler();}
       });
//---------------------------------------------------
//+++++++++++++++++++++++++++++++++++++++++++++++++++
availablePages = new LinkedList<Integer>();
avaiblePageLock = new Lock();
for (loop through total number of physical pages)
{
        availablePages.add(index);
}
//*****************************************************
}
```

UserProcess changes:

readVirtualMemory() transfers data from this process's virtual memory to the specified array. It handles address translation details. It does this by looping through the virtual memory and copying data from amount read to amount to read in virtual memory to the specified array.

```
readVirtualMemory()
{
        // original code
        Lib.assertTrue(offset >= 0 && length >= 0 && offset+length <= data.length);

                byte[] memory = Machine.processor().getMemory();

        //***************************************************************
        //+++++++++++++++ added pseudo code +++++++++++++++++++++++++++++
        int byteNum = 0
        do{
                // calculating page index from virtual address and byte number
                pageIndex = Processor.pageFromAdress(vaddr + byteNum)
                // checking the bounds of the page index to make sure that it is within the
proper bounds
                if (pageIndex < 0 or pageIndex >= pageTable length)
                        return 0
                // calculating page offset using virtual address and byte number
                pageOffset = Processor.offsetFromAddress(vaddr + byteNum)
                // calculating number of bytes left in page which is the page size - the page
offset
                amountLeftInPage = pageSize - pageOffset
                // calculating number of bytes to read which is the min of the amount left and
the length - byte number
                amountToRead = min(amountLeftInPage, length - byteNum)

                physicalAddr = pageTable[index].ppn*pageSize + pageOffset
                // transfering data from this process's virtual memory to all of the specified
array
                System.arraycopy(memory, physicalAddr, data, offset + byteNum, amountToRead)
                byteNum+=amountToRead
        }
        while (byteNum < length)

        return byteNum
}
```

writeVirtualMemory () transfers data from the specified array to this process's virtual memory. It handles address translation details. It does this by looping through the virtual memory and copying data from amount written to amount to write in in specified array to this process's virtual memory.

```
writeVirtualMemory()
{
        // original code
        Lib.assertTrue(offset >= 0 && length >= 0 && offset+length <= data.length);
```

```
                byte[] memory = Machine.processor().getMemory();

        //**************************************************************
        //+++++++++++++++ added pseudo code ++++++++++++++++++++++++++++
int byteNum = 0
// a check to see if the write could succeed before trying
do {
        pageIndex = Processor.pageFromAddress(vaddr + byteNum)
        if (pageIndex < 0 or pageIndex >= pageTable length or pageTable[index].readOnly)
                return 0
        pageOffset = Processor.offsetFromAddress(vaddr + byteNum)
        amountLeftInPage = pageSize - pageOffset
        amountToWrite = min (amountLeftInPage, length-byteNum)
        byteNum += amountToWrite
}while(byteNum<length)

//do the write
byteNum = 0
do{
        // same calculations as in readVirtualMemory
        // except array copy is different

        pageIndex = Processor.pageFromAddress(vadd + byteNum)
        pageOffset = Processor.offsetFromAddress(vaddr + byteNum)
        amountLeftInPage = pageSize - pageOffset
        amountToWrite = min (amountLeftInPage, length - byteNum)
        physicalAddr = pageTable[index].ppn*pageSize + pageOffset

        // this time it transfer data from the specified array to this process's virtual memory
        System.arraycopy(data,offset+byteNum, memory, physicalAddr, amountToWrite)
        byteNum += amounToWrite
}while(byteNum < length)

return byteNum
}
```

loadSections() allocates memory for this process, and loads the COFF sections into memory. It does this by looping through each section and updating the page table entries with the correct physical page number. It also updates their statuses. It then sets each page table entry to read only and loads the page from the current segment in virtual memory into physical memory.

```
loadSections()
{
        // added pseudo code
        UserKernel.availablePageLock.acquire
        //--------------------------------------------------
        // original code
        if (numPages > Machine.processor().getNumPhysPages()) {
                coff.close();
                Lib.debug(dbgProcess, "\tinsufficient physical memory");
                // added pseudocode
                UserKernel.availablePageLock.release
                //-----------------------------------------
                return false;
                }
        //++++++++++++++++++++++++++++++++++++++++++++
        //                      added pseudocode
        // initialize a new page table and place the available pages from the user kernel into
the page table
        pageTable = new TranslationEntry[numPages]
        for (loop through page table)
                pagetable[index] = new
TranslationEntry(index,UserKernel.availablePages.remove,true,false,false,false)
        Userkernel.availablePageLock.release
        //-------------------------------------------

        // load sections
        for (int s=0; s<coff.getNumSections(); s++) {
                CoffSection section = coff.getSection(s);

                Lib.debug(dbgProcess, "\tinitializing " + section.getName()
                                + " section (" + section.getLength() + " pages)");
                //*************************************************************

                // +++++++++++added pseudocode+++++++++++++
                // goes through each section and updates the page tables to read only
                // also loads the pages from the page table

                // orginal code
                for (int i = 0; i < section.getLength(); i++)
                {
                        int vpn = section.getFirstVPN()+i;
                        // added pseudo code
                        pageTable[vpn].readOnly = section.isReadOnly
                        section.loadPage(i, pageTable[vpn].ppn)
                        // -----------------------------
                }

        }
return true
}
```

unloadSections() releases any resources allocated by loadSections(). Since this function was empty need to add functionality. This is so that resources would be properly released when this function is called.

```
unloadSections()
{
        UserKernel.availablePageLock.acquire
        // adds the pages from the page table back to the User Kernel and clears the page
table contents
        // this effectively releases any resources allocated by loadSections()
        for (loop through page table)
        {
                UserKernel.availablePages.add(pageTable[i].ppn)
                pageTable[index] = null
        }
        UserKernel.availablePageLock.release

}
```

Part 3: Implementation of various syscalls(exec, exit, and join)

In part three of this project we have to implement the exec, exit, and join syscalls.

Exec takes 3 parameters; the file name of the program to execute, an integer number of arguments, and an array of arguments. Exec starts off by creating a new UserProcess by calling the method newUserProcess located in the UserProcess class. Once this child process is created, it's process id will be acquired by the parent and stored in the parents list of childrens id's. The next step is to convert the arguments that were passed in, into usable variables using the readVirtualMemory and the readVirtualMemoryString methods. After this is done, the execute method inside the UserProcess class will be called with the newly converted variables passed in to fork a new UThread to run the program. Exec will then return. If execute returns true, the program ran correctly and the child's process id will be returned. If execute returns false, exec will return -1 to signify that it did not run properly.

```
handleExec(String file, int argc, String[] argv) {
UserProcess offspring = newUserProcess(); // Create the new UserProcess
this.children[numberChild] = offspring.ID; // Add the child's ID to the
                                            // parents array of children

// Increment the numberChild variable
numberChild++;

// Mapping the virtual addresses to the main memory adresses
String filename = readVirtualMemoryString(vaddress, file.length);
int count = readVirtualMemory(vaddress, byte date);
String[count] vector;
for(i=0 ; i<count ; i++) {
vector[i] = readVirtualMemory(vaddress, argv[i].length);
}
if(offspring.execute(filename, vector))
return offspring.pID; // The program executed properly so we
// return the child's process ID
else
return -1;
// If program failed to run correctly we return -1
}
```

Join will run similarly like the KThread.join() method. The difference with this one will be that this version of join will only allow a parent to join with its own child. Unless the process id passed into join is the id of a child process of the current process, it will be unable to join with the requested process and return -1. If the requested id is the child of the current process, it will continue with the join, and will return 1 if the child executes to completion, or it will return 0 if execution exits too soon.

```
handleJoin(pID, status) {
if(!this.isParent(pID)) // A method that will check if the
// current process is a parent of pID
return -1;
// pID is not a child of the current
// process, we return -1
// Proceed with the join just like in KThread
Process gets condition lock;
Process sleeps on a condition lock;
When child is finished{
wake all sleeping on condition lock}
}
```

Finally the last syscall exit will handle the termination of processes. Exit will first terminate any threads that are currently running that were started by this process. The finish method inside the KThread class will be used to terminate the UThread (UThread extends KThread) of the process. It will then clean up all the resources by freeing up it's memory and closing all open files in use. The close syscall can be used to close all the current filedescriptors associated with the current process. Once this cleanup sequence is completed, the exit status will be determined`and transferred to it's parent process. Finally, if the current process is the last exisitng process, we will terminate it using Kernel.kernel.terminate(); and set the status to 0 and return.

```
handleExit(int a0) {
// Store our exit status in case the parent needs it for join()
if(parent != null)
// Gets the exit status of childeren

unloadSections();
//if there is coff, close it
//checks for open files and closes them
runningProcesses--;
if(runningProcesses == 0)
       Kernel.kernel.terminate();
return a0;
}
```

Part 4: Implementation of Lottery Scheduler

/* This is my summary of what we need to do.
Lottery Scheduler
Pass in priority of threads and waiting threads from priorityScheduler
totals up (sums) the priorities and makes sure that the sum is less than Integer.MAX_VALUE
When increasePriority is called, number of tickets goes up by one
When decreasePriority is called, number of tickets goes down by one
*/

In all essence, this is a fairly simple class to implement. Instead of finding what has been waiting the longest and what has the highest priority, everything has a randomly equal chance (depending on what priority each thread has). Due to the nature of this, there will have to be checks to make sure the bounds do not exceed the max value allowed by integers as well as 0. If a thread has a priority of 0, it still needs to be given one lottery ticket in order to give it a chance to be chosen, otherwise it will remain forever alone. In addition to the lower bounds, the main difference between this and priority scheduler is how the priority/number of tickets is chosen. In Priority Scheduler, the max priority is used, so if a single thread that is waiting is 7, the entire wait queue has a priority of 7 until that is chosen and completed. In Lottery Scheduler, a sum will be taken. Because of this, there needs to be 2 values, one that holds the total sum and one that will be for an individual thread. Due to current uncertainties, I have a few ideas on how to choose the next thread, the easiest being an iteration through the different queues, starting with 1, and checking to see if a random number (between 1 and the total sum) is less than the current priority plus the sum of the previous priorities.

Unfortunately, I see this as being a slightly inefficient process, so I will continue to work on making a better one that will be easier and simpler to implement.

What will be needed:
```
sum() {
       // Will sum the number of lottery tickets
       // Will be taken in a similar fashion to getEffectivePriority in PriorityScheduler
       //and
       // instead of finding max, will just sum total
       // Make sure it is less than Integer.MAX_VALUE
       // This will only do partialSum, need to go through all the different active
       //threads to
       // get total

       for i = 0 to sizeof(linkedList) {
              if (totalSum += priority > Integer.MAX_VALUE)
                     totalSum = Integer.MAX_VALUE
              else
                     totalSum += priority
       }
}

calculateTicket() {
       // Will calculate a random number between 1 and the sum of all tickets in integer
       //value
       // Will choose whatever class is in the range
       // Will have a while loop, each time checking the range of the thread's tickets
       // Will check to see if the current max (the previous sum and the current

       random from 1 to totalSum

       while current total < totalSum {
              if random <= partialSum + currentSum
                     return currentThread relating to partialSum
                     // This will probably be done using an iterator to move through the
                     //list
              else
                     currentSum += partialSum
       }
}

increasePriority() {
       // Will increase number of tickets by 1
       partialSum++
}

decreasePriority() {
       // Reduce number of tickets by 1
       partialSum--
}

totalSum() {
       // Will call getThreadList in priority scheduler

       for (i = 0 to sizeof(listofQueues) {
              for (j = 0 to sizeof(listofthreadsinQueue) {
                     totalSum += priority
              }
       }
}

linkedList // will be all the threads
totalSum  // total sum of all the threads combined - needed to implement calculate ticket
          // properly
partialSum // sum of a single thread, will be recalculated

// Will be adding/changing what these variables do in time, just as a rough approximation //of
what they will be.
```