Niels Joubert CS162-PC
Nadeem Laiwala CS162-PD
Eugene Li CS162-KI
William Wu CS162-KJ
Akhil Dhar CS162-KK
March 21, 2008
Computer Science 162

CS162 Design Doc Phase 2: Multiprogramming

Task One - (30%) System calls for file system [Willy, Eugene] [Tests: Akhil]

In order to allow a user to safely interact with the file system, we need to give the user a standard set of system calls pertaining to file management. These calls include creating, opening, reading, writing, closing, and deleting (unlinking) a file. This interface closely reflects the syscalls available in a UNIX system.

Correctness Constraints

- User programs cannot pass invalid arguments to the kernel which might corrupt the kernel's internal state or that of other processes.
- The halt() system call can only be invoked by the root process. If other processes invoke halt(), the request to halt should be ignored.
- If a system call results in an error condition, it must return -1 and not throw an exception to the kernel.
- File descriptors 0 and 1 must refer to standard input and standard output, respectively
- A user process is allowed to close the file descriptors for standard input and output
- A process must be able to support at least 16 concurrently open files
- Each open file must be associated with a file descriptor. This file descriptor can be reused if the associated file is closed.
- File descriptors should be process specific, so different processes may have different file descriptors to the same file

Validating user inputs

- Pointers
    - The value of a pointer should be greater than zero and less than the maximum size of memory
- Strings/Char pointers
    - An input string must be null-terminated
    - The maximum length for a string is 256 bytes
- File descriptor
    - A file descriptor must be greater than zero and less than the size of the open files table
    - The file descriptor of a closed file is no longer valid

Implementation

Each process must keep a table of currently open files. This can be achieved by using an array of OpenFile objects. The index of an OpenFile object in the open file table will also serve as its file descriptor. This array will be initialized in the constructor for UserProcess. If we need to increase our array size to handle more OpenFiles, we will simply instantiate a new array of OpenFile objects

which has the twice the size of the original array, then copy the original array over. To initially assign a file descriptor to an OpenFile object, we will simply walk through the open file table and return the index of the first non-null entry. We do not personally need to keep track of file positions when we read and write.

A hashtable for openedFiles keeps track of all the processes that have a particular file opened. The key for this hashtable is the file name string, and values are the number of processes that have this file open. We are assuming here that every file has a unique file name and that different file names cannot refer to the same file.

Inside UserProcess.java

```
OpenFile[] openFileTable

public UserProcess()
    ...
    openFileTable[0] = OpenFile representing console's standard input
    openFileTable[1] = OpenFile representing console's standard output
    ...
```

The halt system call can only be called by the root process. Any other thread that attempts to call this method will be ignored and the method would immediately return. Therefore, halt() can be modified as follows:

```
handleHalt()
    if current thread is not the root process:
        ignore the request and return -1
    or else halt the machine and return 0
```

We need to modify handleSyscall in UserProcess to handle the additional cases we're supporting. We can do this as demonstrated by the following:

```
handleSyscall()
        ...previous code...
        case syscallCreate
            return handleCreat(a0)
        case syscallOpen
            return handleOpen(a0)
        case syscallRead
            return handleRead(a0, a1, a2)
        case syscallWrite
            return handleWrite(a0, a1, a2)
        case syscallClose
            return handleClose(a0)
        case syscallUnlink
            return handleUnlink(a0)
        ...default handling for illegal syscalls...
```

Create

Create should attempt to open the specified disk file and possibly create it if it does not exist. Its only argument is the name of the file that should be created. If a file is successfully created or opened, it should return the file descriptor. Otherwise, it should return -1.

```
private int handleCreat(charPtr)
    if the pointer is invalid, return -1
    file name = readVirtualMemoryString with address=charPtr and maxLength=256
    if the name of the file is bad, return -1
    openfile = ThreadedKernel.fileSystem.open(a0, true)
    if openfile is null, retrn -1
    file descriptor = the next available entry in the open file table
    openFileTable[file descriptor] = openfile
    return file descriptor
```

Open

Open should only attempt to open the specified disk file and not try to create it if it does not exist. Its only argument is the name of the file that should be opened. If a file is successfully opened, it should return the file descriptor. Otherwise, it should return -1.

```
private int handleOpen(charPtr)
    if the pointer is invalid, return -1
    file name = readVirtualMemoryString with address=charPtr and maxLength=256
    if the name of the file is bad, return -1
    openfile = ThreadedKernel.fileSystem.open(a0, false)
    if openfile is null, retrn -1
    file descriptor = the next available entry in the open file table
    openFileTable[file descriptor] = openfile
    return file descriptor
```

Read

The function Read attempts to read up to count bytes into a buffer from the file or stream referred to by the file descriptor. If this is unsuccessful, the file position becomes undefined and the method returns -1. If successful, the return value of this method is the number of bytes that are read.

It is possible, however, that the return value is a smaller number than the desired count number. This has a few possible interpretations. If the file descriptor is referring to a file on disk, then this scenario indicates that the end of the file has been reached. If the file descriptor is referring to a stream, then this scenario indicates that there are fewer bytes available right now than were requested, and more bytes may be available in the future. The read function always returns as much as possible immediately.

To safeguard against a request to read an arbitrarily large amount of data, we split up reading into 1 kilobyte blocks. We will read 1 kilobyte of data from the file into a buffer, and then write 1 kilobyte of data from the buffer into virtual memory. We continue this process until the desired amount of data has been read, or there is nothing left to read. Then we return the number of bytes successfully read.

```
private int handleRead(fileD, bufPtr, size)
    if invalid file descriptor or fileD==standard output or invalid bufPtr or size<0:
        return -1

    openfile = openFileTable[fileD]
    if openfile is null:
        return -1
```

```
    bytesRead = totalBytesWritten = bytesWritten = 0
    readSize = 0x400
    byte[] bytes = new byte[readSize]

    while totalBytesWritten is not equal to size:
        desiredBytes = Math.min(readSize, size-totalBytesWritten)
        bytesRead = read desiredBytes from openfile and put into buffer at bufPtr
        if number of bytes read is not correct:
            return -1

        bytesWritten = write bytesRead from bytes to VM addr at bufPtr+totalBytesWritten
        if you didn't read the number of bytes desired:
            break
        else if you didn't write the number of bytes read:
            return -1
        else
            increase totalBytesWritten by bytesWritten
    return totalBytesWritten
```

Write

Write will attempt to write bytes from the buffer to the file referred to by the file descriptor. Write should be called with three arguments. The first argument is the file descriptor of the file that you wish to write to, the second is a pointer to the buffer that contains what you wish to write, the third is the number of bytes that should be copied from the buffer to the file.

Write is able to return before the bytes that it wrote are flushed to the file or stream. Upon successfully writing to a file descriptor, it will increment the file position by the number of bytes written and return this number. It is possible that the number of bytes written is smaller than the number of bytes requested, either due to disk files being full or streams being terminated before all the data was transferred. In this case, write should return -1 to signify an error. Other situations where an error might occur are if the file descriptor is invalid, if the buffer is invalid, or if the network stream has been terminated by a remote host. In all these cases, return -1.

Similiar to reading, we split up writing into 1 kilobyte blocks to safeguard against a request to write an arbitrarily large amount of data. We will read 1 kilobyte of data from a buffer in virtual memory to a local buffer, and then write 1 kilobyte of data from the buffer into the file. We continue this process until the desired amount of data has been written and return the number of bytes read.

```
private int handleWrite(fileD, bufPtr, size)
    if invalid file descriptor or fileD==standard output or invalid bufPtr or size<0:
        return -1

    openfile = openFileTable[fileD]
    if openfile is null:
        return -1

    bytesRead = totalBytesWritten = bytesWritten = 0
    readSize = 0x400
    byte[] bytes = new byte[readSize]

    while totalBytesWritten is not equal to size:
        desiredBytes = Math.min(readSize, size-totalBytesWritten)
        bytesRead = read desiredBytes from VM addr at bufPtr+totalBytesWritten into
```

```
bytes
        if bytesRead is not equal to desiredBytes:
            return -1;

        bytesWritten = write bytesRead into openfile
        if bytesWritten is not equal to bytesRead:
            return -1
        else
            increase totalBytesWritten by bytesWritten
    return size;
```

Close

The function close will close a file descriptor, after which it no longer refers to a file or stream. The file descriptor may be reused after that.

There are several conditions to take into account. If the file descriptor refers to a file, then all data written to it by write() will be flushed to disk before close() returns. If the file descriptor refers to a stream, all data written to it by write() will eventually be flushed but not necessarily before close() returns.

If the descriptor is the last reference to a disk file that has been removed with unlink, then the file is deleted. This method returns 0 on success or -1 if an error occurred.

```
private int handleClose(fileD)
    if invalid file descriptor:
        return -1

    openfile = openFileTable[fileD]
    if openfile is null:
        return -1

    close the openfile
    openFileTable[fileD] = null
    return 0
```

Unlink

The function Unlink deletes a file from the file system given various conditions. If no processes have the file open, then the file is deleted immediately and the space it was using is made available for reuse. If any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed. Creat() and Open() will not be able to return new file descriptors for the file until it is deleted. This function returns 0 on success or -1 if an error occurred.

```
private int handleUnlink(charPtr)
    if charPtr is invalid:
        return -1
    file name = readVirtualMemoryString with address=charPtr and maxLength=256
    if file name is null:
        return -1

    file removed = ThreadedKernel.fileSystem.remove(file name)
    if file removed:
```

```
        return 0
    else
        return -1
```

Testing Strategy

We want to test our implementation for conformance to the correctness constraints we laid down with the following test cases:

- load a user process P1 and attempt to invoke the halt() system call
  --- attempt open() call with a file name of length > 256
  --- attempt open() call with an file that doesn't exist
  --- attempt to write() on an invalid file descriptor
  --- attempt to open() several valid files (F1, F2, F3, ... F16)
  --- attempt to write() to F3
- load another process P2
  --- attempt open() call on F1
  --- attempt close() call on F1
  --- attempt write() call on F1
  --- attempt unlink() call on F2
  --- attempt to read() from F3

Task Two - (25%) Multiprogramming Memory Management [Nadeem, Akhil] [Tests: Niels]

We must change Nachos from being restricted to one user process at a time to multiple user processes. In order to ensure this functionality, we need to design an efficient allocation scheme for assigning physical memory to processes without any overlap. Moreover, we need to increase functionality so that data can be copied between the kernel and the user's address space.

Correctness Constraints

- Once a block of memory has been allocated to a thread, it is made unavailable to all other threads.
- The memory allocated to a process is made available as soon as the process exits, be it normally or abnormally.
- Upon failing on an attempted read/write beyond the bounds of memory, the kernel should return 0 instead of throwing an exception.
- Upon failing on an access violation, the kernel should return 0 instead of throwing an exception.
- Memory does not have to be allocated in contiguous blocks.

Implementation

Memory allocation scheme of physical memory to processes:
The method for assigning physical memory to processes requires a global linked list - free list - of free physical pages within the UserKernel class. The underlying philosophy of the free list follows the K&R implementation. When pages need to be assigned we simply iterate through the list using first-fit method.

```
public class UserKernel() {

private static LinkedList<Integer> freeList;
```

```
private static Lock freeListLock;

   public UserKernel() {
      //...everything from before [including: super();]
      create a new integer linked list - freeList
      create a new lock - freeListLock
      traverse all page numbers in the range [0,
Machine.processor().getNumPhysPages()]:
         add each page (number) to freeList
   }

   private static int getNextAvailPhysPage() {
      acquire freeListLock for atomicity
      temp = -1
      if (freeList not empty)
         pop first ppn off freeList, store in temp
      release freeListLock
      return temp
   }

   private static void addAvailPhysPage(int ppn) {
      acquire freeListLock
      if (freeList doesnt already contain ppn)
         add ppn to end of freeList
      release freeListLock
   }

//...everything from before
}
```

```
public class UserProcess() {

   public UserProcess() {
      create LSLock for protected memory allocation while loading
   }

   public int readVirtualMemory(int vaddr, byte[] data, int offset, int length) {
      check for invalid parameters -> return 0
      memory = Machine.processor.getMemory()
      extract vpn and off from virtual address (vaddr)
      while (read not complete)
         if (vpn beyond bounds of pageTable)
            break
         index into pageTable[vpn] to obtain ppn translation
         turn on dirty flag
         construct physical address (paddr) from ppn and off
         calculate amount to be read on page: min(length, pageSize - off)
         copy this amount from physical memory to data buffer
         move to next virtual page
      return total amount read
   }

   public int writeVirtualMemory(int vaddr, byte[] data, int offset, int length) {
      check for invalid parameters -> return 0
```

```
        memory = Machine.processor.getMemory()
        extract vpn and off from virtual address (vaddr)
        while (write not complete)
            if (vpn beyond bounds of pageTable)
                break
            index into pageTable[vpn] to obtain ppn translation
            if (pageTable entry for vpn is read-only)
                break
            turn on dirty flag
            turn on used flag
            construct physical address (paddr) from ppn and off
            calculate amount to be written on page: min(length, pageSize - off)
            copy this amount from data buffer to physical memory
            move to next virtual page
        return total amount written
    }

    protected boolean loadSections() {
        acquire LSLock
        if (numPages > freeList.size)
            close process (Coff), "insufficient physical memory"
            return false;
        create pageTable: TranslationEntry array of size numPages
        traverse sections:
            use i to traverse pages within each section:
                obtain vpn from section.getFirstVPN() + i
                obtain ppn from getNextAvailPhysPage()
                create new pageTable entry with vpn, ppn - set valid(true), used(false),
dirty(false), readOnly(section.isReadOnly() for COFF pages, false otherwise)
        release LSLock
        return true
    }

    protected void unloadSections() {
        if (pageTable==null)
            return;
        use i to traverse pageTable entries
            addAvailPhysPage(pageTable[i].ppn)
    }
```

Testing Strategy

The following cases will test our implementation for conformance to the correction constraints established earlier:

- load a user process, determine the allocated ppn
- check whether the ppn is on the freeList (correct behavior: not on list)
- attempt to read from a vpn that is larger than the NUMPAGES variable for the process (should fail, return 0)
- attempt to write to a negative vpn (should fail, return 0)
- find a virtual page that corresponds to read-only memory and attempt to write to it (should fail, return 0)
- exit process and check whether the associated ppn is on the freeList (correct behavior: on list)

Task Three - (30%) System calls for Multithreading - Exec, Join, Exit [Niels, Nadeem] [Tests: Willy]

For task three we implement the necessary system calls that will allow a process to spawn child processes, possibly join them, and exit from itself. For this we need to maintain a data structure to store the process tree of how processes relate to each other as parents and children. We also need to initiate allocation and reclaiming of memory resources, dealing with open files. Globally, we enforce unique process IDs, and keep track of the number of active processes. All arguments passed in as system calls are read using readVirtualMemory and readVirtualMemoryString from the given register addresses.

Correctness Constraints

- exec, join and exit must always fail gracefully without crashing the kernel in the event that something goes wrong.
- exit must terminate the calling process immediately
- exit must close all of the process' open file descriptors
- exit must set its child processes to not have a parent
- a process terminating through kernel termination should also set an exit value and close the process gracefully as if it exited itself.
- the last process that calls exit must cause the machine to halt via the Kernel.kernel.terminate() procedure
- a process must know of all its child threads and their execution status (including possibly exit code if a child has terminated)

Implementation

Data Structures

There are two ways of organizing the process tree. You can have each process maintain references to its parent and children, and store its own processID. Alternatively, you can have a global map from processID to UserProcess object, and have each process maintain the processIDs of its parent and children, as well as its own processID. We decided to go with the former, since this has less overhead in our UserProcess implementation. This scheme does depend on the UserThread staying on queues at all times during its execution, since we no longer have a reference to each process. If a process has no parent and it is not the root process, we cannot get access to it through a universal data structure. If we need this capability later, we can easily modify our implementation to use the second way of organizing processes.

Globally:
int processIDcounter, starting at 0: global processID counter that always contains the next processID to hand out
int activePIDs: global number of active processes, incremented on process creation, decremented on process exit
Lock sysCallMultiprocessing: global lock to synchronize access to these data structures

Per process:
UserProcess parent: Reference to parent process
ArrayList<UserProcess> children: List of references to child processes
int pid: your process ID
int exitStatus: your exit status as set from the exit syscall
int exitCode: your exit code that indicated whether you exited with an unhandled exception or not, by default 1 to indicate success.

```
boolean terminated: boolean indicating whether you have terminated or not
UThread myThread: a reference to your thread of execution
```

Exit

Exit should terminate the calling process immediately. This includes closing any open file descriptors owned by the process, and setting all its children to no longer have a parent. Exit should not remove information about a process that could potentially be needed in the future if the parent attempts to join on the terminated process.
If the caller is the last running process, it should terminate the machine itself.

```
handleExit(int status)
        acquire the sysCallMultiprocessingLock
        //Data Structure Integrity Maintenance:
        set completion boolean to true
        set my exit status according to the argument value
        for all my children
            delete their parent references pointing to me
        empty my list of references to my children

        unloadSections() to free memory that was in use (possibly zero out all
bytes)
        activePIDs-- to decrement the amount of active processes

        for all elements in the openFileTable:
            call the handleClose system call on each of the file descriptors I own
        (see Task One)

        if the activePID integer is zero, call Kernel.kernel.terminate() to halt the
        system
        release lock
```

We also make a slight modification to handleException to set the correct flags if an unhandled exception occurs:

```
handleException(int cause)
    ...

    default:
        exitCode = 0; //indicates that an error occured.
```

Join

Join is a simple wrapper around the join implementation in the kernel's threading system.

```
handleJoin(int pid, int stat)

        acquire the sysCallMultiprocessingLock

        Child reference
```

```
        for each of your children
            if this child is the given pid, break
        if we didnt find a child
            release lock
            return -1 to signal error
        if the child's terminated flag is set
            write the child's exitStatus to the virtual memory block pointed to by stat
            release the lock
            return the child's exitCode

        release the sysCallMultiprocessingLock
        theChild.myThread.join()

        write the child's exitStatus to the virtual memory block pointed to by stat

        return the child's exitCode
```

Exec

Exec should add the newly created process as a child to the calling process. The newly created process must have its parent pointer to the calling process.

```
handleExec(int file, int argc, int argv)

        newProc = newUserProcess()
        String fileS = readVirtualMemory from file pointer handed as argument
        String args[] = new String[argc]
        byte[] addr = new byte[4]

        for i from o to argc
            readVirtualMemory at argv+4*i into addr
            int strAddr = convert bytes inside of addr into an integer
            args[i] = readVirtualMemory from strAddr (this is to align the pointers)
            if args[i] is null
                release sysCallMultiprocessingLock
                return -1 //signals that an error occured and we can't launch the
        program

        if (!newProc.execute(fileS, argS)) //arg checking happens in here
            decrement activePIDs
            release sysCallMultiprocessingLock
            return -1

        set newProc.parent = this
        add newProc to my children

        return newProc.pid


public UserProcess()
        ...
        pid = processIDcounter;
        processIDcounter++;
```

```
execute(String name, String[] args)
        ...
        increment activePIDs
        set myThread reference = new UThread(this process)
```

Hooking calls into Nachos framework

To make the connection between a system call from a user program to one of the routines described above, we modify the handleSyscall function as follows:

```
public int handleSyscall(int syscall, int a0, int a1, int a2, int a3) {
    switch (syscall) {
    //...other cases from before
    case syscallExit:
        handleExit(a0);
    case syscallJoin:
        handleJoin(a0, a1);
    case syscallExec:
        handleExec(a0, a1, a2);
    default:
        //...everything from before
    }
}
```

Securing Halt to only work for the root process

We already have the necessary information to prevent the Halt system call from working for anything but the root process. Since we assign processIDs from 0 in increasing order, we check to see whether the caller's PID is zero. In all other cases, halt returns with no effect.

Testing Strategy

Java Tests:
These can be accomplished inside our java code by faking system calls for processes.

Unit testing the UserProcess constructor:
- pass it a correct argument string, check that the COFF file gets loaded and the userthread created.
- pass it an incorrectly formatted argument string, assert that an error gets thrown
- pass a file that does not exist, assert that an error gets thrown

Data Structure Test:
- construct a process that does not spawn anything. call join and asserts that nothing happens
- construct one process that spawns a new process. check that the child and parent pointers are set appropriately
- construct a process that spawns multiple new processes. check that the child and parent pointers are set appropriately
- construct a process that spawns a whole tree of processes. check child and parent processes throughout the tree
- call exit for a leaf process. asserts that the process is ended but not removed. call join on this process and asserts that it returns the correct status code immediately.
- call exit for the process just above the previously killed process. assert that the child is killed and ready for garbage collection

- call exit for a process inside the above tree. asserts that the child and parent pointers change appropriately.

Task Four - (15%) Lottery Scheduler [Niels, Eugene] [Tests: Eugene]

Lottery Scheduler is an extention of the Priority Scheduler algorithm we designed for Phase One. Lottery Schedulers differ from Priority Schedulers in two important ways:

1. Lottery Schedulers donate tickets, which sum together to give a thread a total amount of tickets, rather than a maximum amount of donated priority.
2. Lottery Schedulers make a uniform, random selection amongst all the tickets held by waiters on a queue when the next owner is picked.

We can achieve these differences by extending the PriorityScheduler class to a LotteryScheduler class. The only methods we need to replace is the methods responsible for priority donation and for picking the next thread off a queue of blocked threads.

Lottery Scheduler's algorithm to pick the next thread is more extensive than a simple selection of the thread with highest priority. Each thread has a certain amount of tickets, dependent on their intrinsic ticket value and the total tickets donated to them. The algorithm to select the next thread makes a random selection amongst all the tickets owned by threads waiting on a queue. This number will fall into a range of tickets belonging to a particular thread. Thus, the probability of a thread being picked is dependent on the amount of tickets he owns (incidentally, tickets are indistinguishable, thus no metadata about ticket specifics needs to be stored, only the total amount of tickets a thread owns.)

This new algorithm introduces a problem in the case of deadlock. Deadlock is caused by a circular wait, which will bring out lottery scheduler crashing to a halt as tickets are donated around the circle indefinitely.

Correctness Constraints

- All threads have an intrinsic amount of tickets they own.
- A thread's total number of tickets is the sum of its own intrinsic amount of tickets and the total amount of tickets of all threads directly or indirectly waiting on it.
- The minimum number of tickets a thread has is one ticket.
- A thread does not lose tickets when it waits on another thread.
- Total tickets should never exceed Integer.MAX_VALUE.

Implementation

We write our implementation in a LotteryScheduler class that extends PriorityScheduler. LotteryScheduler contains two inner classes: LotteryQueue that extends PriorityScheduler's PriorityQueue, and ThreadState, that extends PriorityScheduler's ThreadState. We override the necessary methods to achieve the new functionality.

Priority Ticket System

LotteryScheduler extends PriorityScheduler

PriorityScheduler enforces a strict range of possible priority. We need to change the limits of this range to our new range of possible tickets. We override this in the child class by setting priorityDefault, priorityMinimum and priorityMaximum to the appropriate values.

LotteryScheduler's newThreadQueue method should now return LotteryQueue classes rather than PriorityQueue classes, thus we override this method as appropriate.

We also replace getThreadState to return a object of type LotteryScheduler.ThreadState so that we work with the correct type of ThreadState throughout the scheduler system.

LotteryQueue extends PriorityQueue
LotteryQueue needs to implement a different algorithm for picking the next thread. Upon running this algorithm, we implicitly assign tickets consecutive numbers in the range from 0 to the amount of tickets held by all threads on the queue. We pick a random number in this range, then associate that number with the thread holding that ticket. We achieve this by overriding the pickNextThread method as shown below:

```
pickNextThread()
    if waiters.empty()
        return null
    int sum=0, i=0;
    Sum the total amount of tickets that belong to all my waiters.
    random = Pick a random number between 0 and that sum.
    sum=0;
    i=0;
    walk the waiters list
        add to sum the effective priority of each waiter
        if we reach the waiter where this sum is bigger than the random nr we
selected
        return this waiter
```

Ticket Donation System

The priority donation system of Phase One needs to be modified to donate the total amount of tickets rather than find the maximum value of priorities. This is achieved by overriding PriorityQueue's recalculateDonation() method and ThreadState's pullDonationsUpToMe(). We also override getEffectivePriority() to return the total tickets I hold rather than just the maximum of my own priority and the priority I have received.

LotteryQueue:

```
recalculateDonation()
    if we do not transfer priority, return.

    reset the total received tickets to zero.
    for each thread waiting on this queue
        increase receivedTickets by the amount of tickets received from that thread
by calling donator.getEffectivePriority
    if the total amount of tickets that this queue now donates is different from the
previous amount, call owner.pullDonationsUpToMe()
```

ThreadState:

```
getEffectivePriority()
    return the total received tickets plus my own amount of tickets.
```

```
pullDonationsUpToMe()
    reset my total received tickets to zero.
    for each donator in my list of owned resources
```

increase receivedTickets by the amount of tickets they donate
if my total amount of received tickets change and i am waiting on a queue, send the new tickets to the queue i'm blocked on by calling sendDonationUpToWaitFor(), originally written for PriorityScheduler

Loop Handling System

The LotteryScheduler has the distinct possibility of donating tickets in an endless loop. This can occur if two threads enter deadlock - one thread waiting on another, which is waiting on it. Although infinite loops in your scheduler is "A Very Bad Thing", a deadlock in your kernel is what is known as "A Much Much Worse Thing", thus we are more concerned with preventing deadlock in the kernel that avoiding deadlock in the LotteryScheduler.

Testing Strategy

We want to test our implementation for conformance to the correctness constraints we laid down with the following test cases:

LotteryQueue and ThreadState:
- test randomness of pickNextThread()
- test that getDonation and getEffectivePriority return the correct value:
--- initially
--- after acquiring the queue (shouldnt change)
--- after adding to the queue (should be the amount of the one thread)
--- after adding to the queue multiple times (should be total of all thread)
--- after doing nextThread (should be old total - value of new owner's tickets)

Whole System:
- test a long linear list of waiting and owning threads for correct priority donation
- test a whole tree
- test a loop to see that it terminates correctly