Project #2 Multiprogramming

Hamel Ajay Kothari (de), Maliena Guy (dd), Jack Wilson (dx), Bryan Cote-Chang (ed)

Section 106, TA: Kevin Klues

Task 1: FileSystem Calls

This section implements the file system calls creat, open, read, write, close, and unlink. We use the help of an array, called fileTable, which holds OpenFile objects, to hold our file descriptors, which contain details of open files. Each user process has its own unique fileTable. We support up to 16 concurrently open files per process, where each active file has a unique file descriptor, which is kept track of in the fileTable array.

Correctness Constraints

- There should be nothing a user program can do to crash the operating system, besides by explicitly calling halt()
- Only the first process in the system may call halt()
- Maximum length for strings passed as arguments to system calls is 256 bytes
- If a user process does anything illegal, causing the Processor to throw an exception, we don't crash. We instead return -1 (ensure this using try-catch blocks)

Declarations

- *OpenFile[] fileTable:* A table that holding file descriptors (explained in the Overview section as well)
- *static int nextPID:* Initially 0. Incremented in the constructor every time a process is instantiated. Used to set the process id, PID, of each process.
- *int PID:* Each instance of the class has a PID value that is set when it is instantiated. In the constructor, PID is set to the value of nextPID at that moment. PID is used in handleHalt() to ensure that the root process is the one calling handleHalt().
- *static Lock lock:* Used to prevent mis-assignment in the statement 'PID=nextPID'.

Description

First off, inside the UserProcess constructor, we initialize the variables that were described in the declarations section. I won't detail what each variable does again, because I already did that in the declarations section, but the code will look as follows:

```
public UserProcess(){
  .... code already given to us ....
  fileTable = new OpenFile[16];

  lock.acquire();
  PID = nextPID++; // Set PID, then increment nextPID
  lock.release();
}
```

- Helper Functions:

  1. *boolean isValidFN(String file_name):* Make sure the file name isn't null and doesn't have a length of 0
  2. *boolean isValidFD(int fd)*: Make sure a file descriptor is nonnegative, exists in the fileTable, and isn't greater than 16 (fileTable.length). The code is written below:

     ```
     private boolean isValidFD(int fd){
       if (fd < 0 || fd >= fileTable.length || fileTable[fd] == null){
         return false; }
       return true;
     }
     ```

  3. *int openFile()* (extracted reused code from handleCreate and handleOpen). The *openFile()* method pseudocode is below:

     `int openFile(int f, boolean shouldCreate):`

     (a) int fd = 0;
     (b) Begin try block {
     (c) Find an opening in the fileTable, by incrementing fd and indexing into the fileTable, until either fileTable[fd] == null, or fd ¿= fileTable.length
     (d) If no opening is found, return -1. Otherwise, fd is set to the open index
     (e) *String file_name = readVirtualMemoryString(f, 256);* // read in the file_name
     (f) textitif !isValidFN(file_name) return -1; // Make sure the file_name is valid
     (g) *OpenFile file = ThreadedKernel.fileSystem.open(file_name, should_create).*
     This opens the file, where should_create is either true or false, and set it handleOpen() and handleCreate()
     (h) if the file is null, return -1
     (i) Otherwise, set fileTable[fd] = file
     (j) end try block }
     (k) catch (Exception e) return -1;
     (l) if execution didn't fall into the catch block, return fd

- Then it's really easy to implement *handleCreate* and *handleOpen*:

  ```
  private int handleCreate(int f){
    return openFile(f, true);
  }
  private int handleOpen(int f){
    return openFile(f, false);
  }
  ```

- *handleHalt():* Do nothing and return if this isn't the root process. Otherwise call Machine.halt()

  ```
  private int handleHalt(){
    if(PID != 0) return -1;
    Machine.halt();
    Lib.assertNotReached(Machine.halt() did not halt machine!");
    return 0;
  }
  ```

- *handleRead(int fd, int buffer, int size):*

  1. *if (!isValidFD(fd) || size < 0) return -1;* // This makes sure fd $\geq$ 0, fd $<$ fileTable.length, and fileTable[fd] == null

2. Initialize a new byte array, called data, where data.length = size
3. *int read = fileTable[fd].read(data, 0, size);*
4. if read $\leq$ 0, then return 0
5. *return writeVirtualMemory(buffer, data, 0, read);*

- *handleWrite(int fd, int buffer, int size):* Very similar to *handleRead*
  1. *if (!isValidFD(fd) || size < 0) return -1;* // This makes sure fd $\geq$ 0, fd < fileTable.length, and fileTable[fd] == null
  2. Initialize a new byte array, called data, where data.length = size
  3. *int read = readVirtualMemory(buffer, data, 0, size);*
  4. if read $\neq$ size, then return 0
  5. Otherwise return fileTable[fd].write(data, 0, size);
- *handleClose(int fd):*
  1. *if (!isValidFD(fd)) return -1* // This makes sure fd $\geq$ 0, fd < fileTable.length, and fileTable[fd] == null
  2. begin try block {
  3. close file using *fileTable[fd].close()*
  4. set *fileTable[fd] = null*
  5. end try block }
  6. catch(Exception e) return -1
  7. If execution didn't fall into the catch block, return 0
- *handleUnlink(int addr):*
  1. begin try block {
  2. String file_name = readVirtualMemoryString(addr, 256);
  3. if !isValidFN(file_name) return -1 // Makes sure the file_name isn't null and doesn't have a length of 0
  4. *if (ThreadedKernel.fileSystem.remove(file_name)) return 0*
  5. Else return -1
  6. end try block }
  7. Catch(Exception e) return -1
- *handleSyscall(int syscall, int a0, int a1, int a2, int a3):*

```
switch(syscall):
   case syscallCreate: return handleCreate(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: if(a0 < 0 || (str = readVirtualMemory(a0, 256)) == null){
                         return -1;
                       else return handleUnlink(readVirtualMemoryString(a0, 256));
```

Testing Plan

We can test this by creating multiple user processes and then attempting the following:
Create a file, open it, read it, write from buffer to the file, close it, and finally, unlink it.
We also will try opening a file which is nonexistant, seeing if it is successfully created and then opened.
We can also write a certain number of bytes to a file from a buffer and see what is returned (should be that number of bytes). We can then try reading that file and make sure that same number is returned.
We additionally can test the edge cases specified in the Correctness Constraints section, like having a non-root process attempt to call halt(), and see if this is handled correctly (-1 is returned, and nothing happens).

Task 2: Multiprogramming

Correctness Constraints

- Each process can only use it's own virtual memory space(each physical address is only available to one process at a time)
- Program is loaded correctly ordered and contiguous in the virtual memory space

Declarations

- freePages - A LinkedList in UserKernel which holds a TranslationEntry for each free page.

Description

The first step is to create a linked list containing all of the free pages that have yet to be allocated to a program. This will be stored in the UserKernel class. Each page will be represented by a TranslationEntry.

```
for (int i = 0; i < Machine.processor().getNumPhysPages(); i++)
    freePages.add(new TranslationEntry(0,i,false,false,false,false));
```

For *UserProcess* the modifications that we will make to the constructor consist of removing the current code and creation of the page table.

For read/writeVirturalMemory we use the processors translation method to get the physical address bassed on the virtual address.

- *readVirtualMemory*:

```
Processor.setPageTable(pageTable);
int paddr = Processor.translate(vaddr,1,false);
if (paddr < 0 || paddr >= memory.length)
    return 0;
int amount = Math.min(length, memory.length-paddr);
System.arraycopy(memory, paddr, data, offset, amount);
return amount;
```

- *writeVirtualMemory*:

```
Processor.setPageTable(pageTable);
int paddr = Processor.translate(vaddr,1,false);
if (paddr < 0 || paddr >= memory.length)
    return 0;
int amount = Math.min(length, memory.length-paddr);
System.arraycopy(data, offset, memory, paddr, amount);
return amount;
```

For our *loadSections* method we first create a pageTable that can hold all the necessisary pages for the program and the stack. We then pull each TranslationEntry from the free pages and assign it the virtual page number that is associated with each section in the coff. We then add 8 more pages to the table for the stack.

```
if (numPages+8 > UserKernel.freePages.size())
    coff.close();
    Lib.debug(dbgProcess, "\tinsufficient physical memory");
    return false;

pageTable = new TranslationEntry[numPages+8];
for (int s=0; s<coff.getNumSections(); s++)
    CoffSection section = coff.getSection(s);
    Lib.debug(dbgProcess, "\tinitializing " + section.getName() + " section (" + section.getLength() +
    for (int i=0; i<section.getLength(); i++)
        int vpn = section.getFirstVPN()+i;
        (lock)
        pageTable[vpn] = UserKernel.freePages.remove();
        (unlock)
        pageTable [vpn].vpn = vpn;
        pageTable [vpn].valid = true;
        if (section.isReadOnly())
            pageTable [vpn].readOnly = true;
        section.loadPage(i, pageTable [vpn].ppn);

for (int i = 0; i<8; i++)
    (lock)
    pageTable [numPages+i] = UserKernel.freePages.remove();
    (unlock)
    pageTable [numPages+i].vpn = numPages+i;
    pageTable [numPages+i].valid = true;
```

For our *unloadSections* method we need to release our pages and clean out translation entries:

```
for (int i = 0; i<pageTable.length; i++)
    pageTable[i].vpn = 0;
    pageTable[i].valid = false;
    pageTable[i].readOnly = false;
    pageTable[i].used = false;
    pageTable[i].dirty = false;
    (lock)
    UserKernal.freePages.add(pageTable[i]);
    (unlock)
```

Testing Plan

Start up the virtual machine and load in multiple user programs that constantly modify their data and see if the programs have their data protected between context switches.

Task 3: System Calls

Correctness Constraints

We can break our correctness constraits into 3 parts for the 3 handlers we have to write: *exitHandler*, *execHandler* and *joinHandler*.

For *exitHandler*:

- Process is terminated immediately.
- Ensure all file descriptors are closed.
- Ensure that the status is appropriately set.
- Check to see if the parent of our thread is waiting for us to join and wake it up.
- This thread is *finish()*ed

For *execHandler*:

- Ensure that the correct program is executed.
- Check that a new child process is created with a unique ID.
- Ensure that the correct child process ID is returned or -1 if the process creation fails.

And finally for our *joinHandler*:

- Check for correct process suspension.
- Check if correct child is processed.
- Ensure that parent is put to sleep and woken up appropriately.
- Ensure that the current process disowns the child it called join on after completion.

Declarations

In order to keep track of the state of our process and its children in this part we will need to declare the following variables:

- *exitCode* - the return code of our process, determined by whether it exited normally or with an exception
- *parentProcess* - the process which spawned this process.
- *waitingParent* - a boolean which keeps track of whether the parent has called *join* on our thread.
- *childProcesses* - a list which maintains the children of our given process.

Description

We can begin our implementation for this part with handler methods for each of our syscalls:

- exitHandler:

```
exitHandler(int exitCode)
    (disable interrupts)
    this.exitCode = exitCode // Specified to be -1 if abnormal exit
for(int fd : openFiles)
        closeFile(fd)
    if(waitingParent)
        parentProcess.wake() // Puts our parent process's thread back on the ready queue
    unloadSections()
    (restore interrupts)
    this.finish() //to kill our thread an schedule it for deletion
```

- execHandler:

```
execHandler(string file, int argc, string[] argv)
    (disable interrupts)
    newProcess = newUserProcess()
    if(!newProcess.execute(file, argv))
        return -1;
    newProcess.parentProcess = this
    this.childProcesses.add(newProcess)
    (restore interrupts)
    return newProcess.getProcessId
```

- joinHandler: Handles the joining between a parent and child process, it returns the exit status of the child process. Implementation is listed under the method's requirement.

```
joinHandler(int pid, int status)
    (disable interrupts)
    if(pid not in childProcesses)
        return -1
    childProc = getProcessForPid(pid)
    if(!childProc.finished)
        childProc.waitingParent = true;
        KThread.sleep()
    childProcesses.remove(childProc)
    (restore interrupts)
    if(childProc.exitStatus < 0)
        return 0
    else
        status = childProc.exitCode
        return 1
```

Then we also want to add to the exception handler, a call to *exit()* with a negative exit code or something similar to ensure that an unhandled exception cleans up all resources and wakes waiting threads.

Testing Plan

In order to test this section we can write a few C programs which test out our syscalls. The first two programs which will be helper programs will print a message (which can be used to identify them when debugging) and have one exit normally which the other intentionally fails.

Next we will write a third program which will trigger execution for the other two and examine their exit codes.

Task 4: Lottery Scheduler

Correctness Constraints

- Effective priority of a ticket holder is the sum of all the donated tickets plus it's own, not the max.
- Our solution should work for an incredibly large number of tickets in the system, ie. no array storage of tickets.
- Increase priority should give a process one more ticket and decrease priority should do the opposite.
- The total number of tickets cannot exceed $Integer.MAX\_VALUE$, hence the maximum prority is $Integer.MAX\_VALUE$ and the minimum value is increased to 1.

Declarations

In our LotteryScheduler class we will have to make the following declarations:

- *outstandingTickets*
- *MINIMUM_PRIORITY* the minimum priority (number of tickets for a thread), 1.
- *DEFAULT_PRIORITY* the default priority value to start with, 1.

Description

We can create a new LotteryThreadState class which extends our ThreadState from our PriorityScheduler class to deal with our LotteryScheduler's thread state information. We will also refactor our *recalculateEffectivePriority()* method in ThreadState, extracting the logic where we check the effective priorities from the owned queues to gather donations into a *gatherDonations()* method. Then we can override that method in our LotteryThreadState to return the sum of the donated priorities. So, in our original thread state the method would look like this:

```
gatherDonations(int currPriority)
    int newEffective = currPriority;
    for(PriorityQueue owned : ownedQueues)
        ThreadState ts = owned.pickNextThread()
        if(ts != null)
            int tsEffective = ts.getEffectivePriority()
            newEffective = Math.max(newEffective, tsEffective)
    return newEffective
```

And in our LotteryThreadState will implement the method as follows:

```
gatherDonations(int currPriority)
    int newEffective = currPriority;
    for(PriorityQueue owned : ownedQueues)
        for(ThreadState ts : owned.stateQueue)
            newEffective += ts.getEffectivePriority()
    return newEffective
```

Now to get this thread state to be put into use and to satisfy the correctness constraints we need to reimplement the following methods: *getThreadState()*, *setPriority()*, *increasePriority()* and *decreasePriority()*

The *getThreadState()* method should work in a pretty similar method to the one in PriorityQueue, but instead of creating a ThreadState object in the absence of a thread state for a given thread we create a LotteryThreadState.

For the *setPriority()* we must ensure that we're not adding more tickets then would be availible in the system and we're setting the priority to a value greater than one. So it would look like the following:

```
setPriority(thread, priority)
    assert(Machine.interrupt().disabled())
    LotteryThreadState ts = getThreadState(ts)
    int oldPriority = ts.getPriority()
    assert(priority >= priorityMinimum && (outstandingTickets - oldPriority + priority) <= Integer.MAX_
    ts.setPriority(priority)
```

For the *increasePriority()* and *decreasePriority()* methods we also perform similar logic to the logic used in PriorityQueue, but instead of checking against the maximum priority for the increase method we perform the same check above, ensuring that the number of outstanding tickets will be less than the max outstanding, and we check against our minimum for the decrease method.

Testing Plan

We can test this part in a similar manner to the PriorityQueue. As a simple sanity check we will create 5 dummy threads with 3 LotteryQueues (created via a LotteryScheduler). Then we will waitForAccess on *q1...q3* using *t1...t3* respectively, and we will give access to each of those queues by calling acquire for *t2...t4*. Now *getEffectivePriority()* on *t4* should be 4, since it gets the donation from *t3* which gets a donation from *t2* which in turn gets a donation *t1* thus summing all the donations you should get a total effective priority of 4. Then we can set the priority of *t1* to 6 and we would expect the priority of *t4* to be 9. Then if we have our 5th thread, *t5*, wait on *q1*, and set it's priority to be 10, we expect *t4* to have an effective priority of 19. Finally if we call *q1.nextThread()* we expect *t4* to have an effective priority of 1 again, and *t1* to have an effective priority of 13, taking it's own priority plus it's donation from *t5*.