University of California Merced CSE150

Design Document For
Nachos Program (Phase 2)

Jonathan Srinivasan
Paulo Alberto Sousa
Erick Uriostegui
Johnathan Vastola
Alexander Woersching
Morgan Trembley

November 2020

1. Overview

Our goal is to implement a completely functional program. We are managing threads through different functions. With 6 main tasks for different functionality in our operating system.

2. Content

Task 1: System Calls
- create()
- open()
- read()
- write()
- close ()
- unlink()

Task 2: Multiprogramming
- loadSections()
- readVirtualMemory()
- writeVirtualMemory()

Task 3:System Calls
- exec()
- join()
- exit()

Task 4: Lottery Scheduler
- getTotalTickets()
- pickNextThread()

Task 1: System Calls

Task 1 consists of implementing the following functions: creat(), read(), write(), close(), unlink(). The purpose of these functions is to load file system information onto the thread processes, enhancing the program's ability to multitask by being able to run up to 16 thread processes at simultaneously.

handleSyscall() directs to the correct function to handle the request. We just extended the switch case that was already provided to accommodate the calls that had to be implemented.

UserProcess Scope vars: int MAX_NAME_LENGTH = 256, MAX_PROCESSES = 16,
fdStandardInput = 0, fdStandardOutput = 1
OpenFile[] fdTable = new OpenFile[MAX_PROCESSES]

getFD is a helper function for creat() and open() that finds a file descriptor that is not spoken for. If there are no available file descriptors it will return -1 and the function that called it will return -1 as a result.

```
getFD() {//helper function to return an available file descriptor
        flag = false;
        fd = -1;
        for (i = 2 to 16) {//since 0 and 1 are taken
            //flag stops fd from overwriting multiple times
            //if fdTable[i] is null, the descriptor is available
            if (fdTable[i] == null and !flag) {
                  fd = i;
                  flag = true;
            }
      }
      //returns -1 if no descriptors are available and an fd
      //otherwise
      return fd;
}
```

Create attempts to open a file with the name passed to it in the parameter. If there is no file with that name it will create and open a new file with that name. If, for some reason, the attempt fails, the function will return -1. Since file descriptors 0 and 1 are reserved for standard input and output, creat should only check for descriptors greater than 2.

```
creat(string name) {//attempts to create file of [name]
        flag = -1;
        fd = getFD();
        if (fd >= 2 and fd < MAX_PROCESSES) {
            file = attempt to open/create file with [name];
            if (file == null) {
                  //creation failed, do nothing
            } else {//link file to fd
                  fdTable[fd] = file;
                  flag = fd;
            }
        }
        //returns fd on success, -1 on fail
        return flag;
}
```

This function operates almost identically to creat except, when attempting to open the file, if there is not a file of the parameter name, it will not attempt to create and open a file with that name.

```
    open(string name) {//attempts to create file of [name]
            flag = -1;
            fd = getFD();
            if (fd >= 2 and fd < MAX_PROCESSES) {//0 and 1 reserved
                file = attempt to open file with [name];
                if (file == null) {//no file with [name]
                      //open failed, do nothing
                } else {//link file to fd
                      fdTable[fd] = file;
                      flag = fd;
                }
            }
            //returns fd on success, -1 on fail
            return flag;
    }
```

To accomplish read, we need to first grab the appropriate file using the passed in file descriptor and make the necessary error checks (both for the file descriptor and for the obtained file). We then simply use the file's built-in read function and store that data in a temp array buffer. A -1 return value from the read function indicates an error and we can just propagate that error/return value at this point as well. Otherwise, we write the read result to the process' address using writeVirtualMemory and return the same value that that function returns, which is the total amount read.

//attempts to read [count] number of bytes into buffer from file, returns number of bytes read.

```
    read(int fd, int address, int count) {
          if fd invalid: return -1;

          File file = fdTable[fd];
```

```
		if file invalid: return -1;

		byte[] buffer = new byte[count];
		readCount = file.read(buffer, count); //Use file's built-in
		                                         read function
		if readCount == -1: return -1; //Indicates an error while
		                                   reading

		//Write our result to the process' requested address
		return writeVirtualMemory(address, buffer);
		//since writeVirtualMemory will return the actual amt read,
		      we'll just return it as well
	}
```

Write will work similarly to read, but almost flipped. We start with the same process of grabbing and validating the file descriptor and its corresponding file. Here we instead first read the process' memory via a readVirtualMemory call to grab what it wants to write. We then have to validate that as well, checking if it returned a -1 or if the number of bytes written to our temporary buffer does not match the number of bytes that was requested by the process, both of which indicates an error while reading. Similarly to read, we then use the file's built-in write function to then write our buffer to the file. We do the final check to see if the number of bytes actually written matches the number of bytes requested to be written before we return that same written count.

//attempts to write [count] number of bytes from buffer into file, returns number of bytes written

```
	write(int fd, int address, int count) {
		if fd invalid: return -1;

		File file = fdTable[fd];
		if file invalid: return -1;

		byte[] buffer = new byte[count];

		//Get what the process wants to write
		bufferWriteCount = readVirtualMemory(address, buffer)

		//Check and return an error -> such as not being able to
		      write the entire buffer
		if bufferWriteCount == -1 or bufferWriteCount != count:
		      return -1;
        //Use file's built-in write function

        writeCount = file.write(buffer, count);

        //Check for any errors when it was writing (ie, only a
              portion was written)
        if writeCount != count: return -1;

        return writeCount;
}
```

Close takes a file descriptor and attempts to close a file with that descriptor. If it can not do so, it will return -1. If the descriptor is in bounds and closes successfully, it will return 0.

```
close(int fd) {//attempts to close file with [fd]
        if (fd >= 2 and fd < MAX_PROCESSES) {//expected range
            fdTable[fd].close();
            fdTable[fd] = null;
            return 0;
        } else {
            return -1; //unexpected parameter value
        }
}
```

Unlink attempts to remove a file with the name that is passed to it. If successful, it returns 0, if not, it returns -1.

```
unlink(string name) {//attempts to unlink file with [name]
        flag = attempt to remove file of [name];
        if (flag) {
            return 0;
        } else {
            return -1; //unsuccessful removal
        }
}
```

Task 2: Multiprogramming

For task 2 we have to make sure that our code works for multiple user processes. In order to assure this happens, we have to modify UserProcess.loadSections(), UserProcess.readVirtualMemory(), and UserProcess.writeVirtualMemory. For loadSections() we have to allocate the number of pages that the process needs and this will depend on the size of the page. We have to make sure the process is loaded into the correct physical memory pages by adding a page table. We should return an error if we don't have enough page available for the process. For readVirtualMemory() we will be transferring data to the process's virtual memory and this memory will be transferred from the byte array. For writeVirtualMemory() we check to see that any byte isn't copied before the page is read.

```
loadSections(){
      if(numPages > Machine.processor().getNumPhysPages()) {
      coff.close();
      if no sufficient physical memory
      return false;
      }

      for(int s=0; s<coff.getNumSections(); s++) {
      CoffSection section = coff.getSection(s);
      Allocate the number of pages

      for (int i=0; i<section.getLength(); i++) {
      int vpn = section.getFirstVPN()+i;
      TranslitionEntry = pageTable[vpn];
      section.loadPage(i, vpn);
            }
      }

      return true;

}
```

For read we will be transferring data from the byte array to the process's virtual memory. It will be able to track the bytes that were transferred from the byte array to the process's virtual memory and then return them. We will never return an error message. Instead, we will return the number of bytes that were transferred even if there were zero bytes transferred. If this is the case, then we will return 0.

```
readVirtualMemory((int vaddr, byte[] data, int offset,int length){
      if (offset >= 0 && length >= 0 && offset+length <= data.length);
      byte[] memory = Machine.processor().getMemory();
```

```
        TranslitionEntry = pageTable[vpn];
        if(vaddr < 0 || vaddr >= memory.length)
        return 0;

        int amount = Math.min(length, memory.length-vaddr);
        System.arraycopy(memory, vaddr, data, offset, amount);

        return amount;


    }
```

Write will have the same parameters as read and will also be formatted in a similar way. However, write makes sure that the bytes are copied only if the page is in read only and if the page is in read only before the bytes start to get copied. If the bytes are copied, then that part of the memory will be saved in an array. Just like we did in read, we should always return the number of bytes transferred even if it is zero. This should always happen rather than giving an error message.

```
    writeVirtualMemory(int vaddr, byte[] data, int offset,int length){
        if (offset >= 0 && length >= 0 && offset+length <= data.length);
        byte[] memory = Machine.processor().getMemory();

        TranslitionEntry = pageTable[vpn];
        //make sure it meets the requirements in order to be copied
        if !Translation.valid while on TranslitionEntry.readOnly())
        return 0;

        int amount = Math.min(length, memory.length-vaddr);
        System.arraycopy(data, offset, memory, vaddr, amount);

        return amount;

    }
```

Task 3: System Calls

For task 3, we implemented the following functions: `exec()`, `join()`, `exit()`. We keep track of this process by using a positive global integer called positiveID. It is used by uniquely identifying a new process. (`static private int processID;`)

To begin the first system call, we must use `exec(file, argc, *argv)` to execute the program contained in the specified file into a new child process. The main purpose of the `exec()` function is to figure out whether the program was executed with or without errors. It starts by acquiring a lock to prevent any interrupts from occuring while executing the system call. Then check if the `argc` parameter is a positive integer. Next, `processID` will be set to a unique positive integer, `stdin` is initialized to 0 and `stdout` is initialized to 1. We then put them in a waiting queue and wait till they are ready to be run. The purpose of this is to make sure multiple processes, such as read and write, are not working at the same time. We use `readVirtualMemoryString()` to run the program. If the program runs without any errors, we return `processID`, or else return -1. We then release the lock.

```
exec(){
        acquire lock

                if(argc > 0)
                        new unique processID
                        get args from argv
                        readVirtualMemory(), passing *argv and argc

                if(error occurs)
                        release lock
                        return -1

        release lock
        return processID
}
```

The main purpose of the `exit()` to terminate the current thread, cleaning up any states associated with the process, and making sure child processes don't have a parent. First, acquire the lock in order to prevent interrupts. Then make the child parent process to null. Next, set the current process status to finished and set the status to 0. Finally, release lock. Running processes also need to be checked in the running queue. If there are processes still running, we let them run and clean up all the states in the meantime. Once the processes come to an end, we halt the machine by using the following command: . Ultimately, it depends on the number of states we may add or need.

```
exit(){
      acquire lock;

            current process is set to finished;
            child parent process is made to null;
            status = 0;

      release lock;
}
```

The `join()` is meant to take in a process ID and the address of the process' exit status code, and join the child's thread using KThread.join(). In order to complete this the method first needs to store the UserProcess of the process ID passed. This is done by having a HashMap of all the children stored within each UserProcess, which makes it easy to find each child process based on their unique process ID's. Assuming there is such a child process with the passed process ID, the thread of that child process will be joined by calling KThread.join(). Once the thread finishes, the child's status needs to be handled. Depending on how the child process exited, the child's status is either transferred to the parent, assuming the child exited correctly, or it is not transferred because the child did not exit properly. `join()` also returns different integers depending on how the method was executed. If `join()` returns 1, the method was executed with no problems. If it returns -1, then most likely the process ID passed into the function was not a child of the parent. If it returns 0, then an unhandled exception was hit.

`halt()` was very slightly modified to only be executed if the process calling it was the root process. This can be handled by checking if the process ID of the process that is executing the `halt()` is equal to 0, thus indicating that it was the first process to be created.

```
join(pid, statusAddress) {//attempts to join a parent process to a child
process
        if(statusAddress < 0) return 0;
        // child is null if the pid is not a child of the parent
        child = find pid among all children;
        if (child == null) return -1; //not a child

        child.thread.join(); // calls join to thread
        remove pid process from children;
```

```
  // the process will exit with
        byte[] statusBytes = status value stored at statusAddress;
        writeVirtualMemory(statusAddress, statusBytes);
        if(child exited correctly and statusBytes were written properly)
                  return 1;
        else return 0; // Something went wrong
}
```

```
***halt***(){
        if(pid == 0){
              Machine.halt();
              return 0;
        }
        else{
              return -1; // Something went wrong
        }
}
```

Task 4: Lottery Scheduler

Goal:

A scheduler that chooses threads based on a lottery mechanism. Similar to priority scheduling, we will implement priority inversion and donation as well as additional functionality. The most significant change in functionality for lottery scheduling is instead of normal priority levels, we use a number of tickets. Transferring tickets should add to a thread's current pool and all threads perform donation if enabled, being taken from the donor if not owner. Further, we set a base ticket value to 1 ranging up to the maximum integer value.

Implementation:

When threads are added to the queue managed by the scheduler, we give them a default number of tickets(1). Donation and calculation are done by recalculating effective priority of all threads in a queue owned by the current thread. If we request effective priority, we must update by the number of donated tickets, continuing until a thread has effective priority equal to regular priority. This should return the previous level and continue

Selecting a thread using this process:

```
getTotalTickets() {//helper function for picking next thread
        int totalTickets = 0;
        for (NewThreadState t : waiting)
            totalTickets +=t.getEffectivePriority;
        return totalTickets;
    }
pickNextThread() {
      donationUpdate();
      Newthread ret =null;
      int totalTickets= getTotalTickets();
      if(totalTickets>0){
            rand=rand(totalTickets)
            for(every thread in lotteryqueue)
                  rand-=thread.totalTickets
                  if(rand <=0)
                        ret=getNewThreadState(thread);
                        break;
            }
      return ;
}
```

Weighing is based on the principle that threads with more tickets have a higher chance of reducing random to 0. With proper implementation, the lottery scheduler can handle any number of independent queues with varying amounts of tickets implementing donation or not. Finding the next thread is simple by checking for the total tickets and looping to find a new thread.

Testing:

- Test functions for proper output, and random effectiveness
- Get Effective Priority should return correct donation priority
- Ensure threads do not add themselves to a queue more than once