**CSE 150 Operating Systems**

**Design Phase 2: Multiprogramming**



**Authors: Team Thimble**

Andrew Yfantis

Isaac Flores

Landon Taylor

Lenny Khodel

**TASK I:**

Implement the file system calls: creat,open,read,write,close, and unlink. The methods are under nachos.userprog.UserProcess.java. The file system has already been implemented for us, and we will simply allow user processes to access said file system. Syscall.h is under nachos.test, which is the documentation for the user’s program file system calls.

We must bullet-proof the Kernel from user program errors, meaning all cases that would corrupt the internal state of the kernel or that of other processes.

To fix halt() so that it can only be invoked by a root process, we will need to check if the process is a USERKERNEL class. We will probably do this by adding an extra variable that is set inside of the default constructor of the UserKernel class.

Class UserProcess{

TestCase: call halt using a userprocess

//halt == 0

Protected int handleHalt(){

If (thisprocess != root process)

Return 0

Halt() //void halt();syscall.h

assertNotReached(“halt didn’t work”)

return -1;

}

TestCase: create a file

\* Attempt to open the named disk file, creating it if it does not exist,

\* and return a file descriptor that can be used to access the file.

\* Returns the new file descriptor, or -1 if an error occurred.

//create == 4

Protected int handleCreate(int vMemAddr)

{

Savename= Read virtual memory String

If(savename==null) //doesn’t exit

Return -1

//Occupy page frame, create file. //int creat(char \*name); syscall.h

return (fileID)

}

TestCase: exceed openfile limit

\* Attempt to open the named file and return a file descriptor.

\* Returns the new file descriptor, or -1 if an error occurred.

//open ==5

//a0 is the memory address of a string containing the path to the desired file

Protected int handle open(int a0)

{

Savename = Read virtual memory string

If( savename==null)

Return -1

//open file //int open(char \*name);

Return (fileID);

}

TestCase: read from file too big

//returns number of bytes read

//read ==6

Protected int handleRead(int fileDescriptor, int MemAddrOfBuffer, int numofBytes)

{

If ( fileDescriptor<0 || memAddr <tableSize || fileFromTable==null)

Return -1

Open file

Create array of type byte with length numofBytes

//read file, track bytes read

// int read(int fileDescriptor, void \*buffer, int count); //syscall.h

Return bytesRead;

}

TestCase: write on bad argument

\* Attempt to write up to count bytes from buffer to the file or stream

\* referred to by fileDescriptor. write() can return before the bytes are

\* actually flushed to the file or stream. A write to a stream can block,

\* however, if kernel queues are temporarily full.

\*

\* On success, the number of bytes written is returned (zero indicates nothing

\* was written), and the file position is advanced by this number. It IS an

\* error if this number is smaller than the number of bytes requested. For

\* disk files, this indicates that the disk is full. For streams, this

\* indicates the stream was terminated by the remote host before all the data

\* was transferred.

\*

\* On error, -1 is returned, and the new file position is undefined. This can

\* happen if fileDescriptor is invalid, if part of the buffer is invalid, or

\* if a network stream has already been terminated by the remote host.

//Write==7

Protected int handleWrite (int fileDescriptor, int MemAddrOfBuffer, int count)

{

If ( fileDescriptor<0 || memAddr <tableSize || fileFromTable==null)

Return -1

Open file

//read virtual memory and store in string

If(string from vmem==null)

Return -1

//Write // int write(int fileDescriptor, void \*buffer, int count);syscall.h

If(write != count)

Return -1;

Return numBytesWritten;

}

TestCase: close a file

\* Close a file descriptor, so that it no longer refers to any file or stream

\* and may be reused.

\* If the file descriptor refers to a file, 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 (unless the stream is terminated remotely), but

\* not necessarily before close() returns.

\* The resources associated with the file descriptor are released. If the

\* descriptor is the last reference to a disk file which has been removed using

\* unlink, the file is deleted (this detail is handled by the file system

\* implementation).

\* Returns 0 on success, or -1 if an error occurred.

//close ==8

Close( int fileDescriptor)

{

If( fileDescriptor ==0 || fileDescriptor > DescriptorSize)

Return -1

//close file //int close(int fileDescriptor);

//remove file descriptor

return 0;

}

TestCase: create a file, close, unlink

\* Delete a file from the file system. If no processes have the file open, 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. However, creat()

\* and open() will not be able to return new file descriptors for the file

\* until it is deleted.

\* Returns 0 on success, or -1 if an error occurred.

//unlink ==9

Unlink (int memAddrString)

{

//read virtualMemString

If( reading == null)

Return -1;

Remove from system //int unlink(char \*name);

Return 0;

}

}//end class

**TASK II:**

Implement support for multiprogramming. Allocate physical memory so that processes do not overlap. There are no dynamic memory allocation needs. Assuming 8 pages for a processes stack is okay.

First we need to keep track of available physical memory pages, for example a linked list. This would be done in the UserKernel Class. The processor class (inside of nachos.machine) is already made to handle translation/page-table needs.

For the UserProcess class, we need to change readVirtualMemory() to correctly calculate where the virtual memory is to copy to an area in the physical memory.

**//returns #ofbytes copied**

**public** **int** readVirtualMemory(**int** vaddr, **byte**[] data, **int** offset,

**int** length) {

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*original code\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Lib.*assertTrue*(offset >= 0 && length >= 0 && offset+length <= data.length);

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

// for now, just assume that virtual addresses equal physical addresses

**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;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*end original code\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

We also need to change writeVirtualMemory() where physical memory will be copied to the disk using virtual memory details.

**//returns #ofbytes copied**

**public** **int** writeVirtualMemory(**int** vaddr, **byte**[] data, **int** offset,

**int** length) {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*original code\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Lib.*assertTrue*(offset >= 0 && length >= 0 && offset+length <= data.length);

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

// for now, just assume that virtual addresses equal physical addresses

**if** (vaddr < 0 || vaddr >= memory.length)

**return** 0;

**int** amount = Math.*min*(length, memory.length-vaddr);

System.*arraycopy*(data, offset, memory, vaddr, amount);

**return** amount;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*end original code\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

LoadSections() allocated memory for this process and loads the mips sections into memory.

**protected** **boolean** loadSections() {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*original code\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**if** (numPages > Machine.*processor*().getNumPhysPages()) {

coff.close();

Lib.*debug*(***dbgProcess***, "\tinsufficient physical memory");

**return** **false**;

}

// 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)");

**for** (**int** i=0; i<section.getLength(); i++) {

**int** vpn = section.getFirstVPN()+i;

// for now, just assume virtual addresses=physical addresses

section.loadPage(i, vpn);

}

}

**return** **true**;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*end original code\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

UnloadSections() releases allocated memory from LoadSections().

**protected** **void** unloadSections() {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*original code\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*end original code\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

**TASK III:**

Implement system calls: exec, join, exit.

TestCase: execute file normally

TestCase: execute file without completion

//on error, return -1. On pass, return child process’s PID

/\*\*

\* Execute the program stored in the specified file, with the specified

\* arguments, in a new child process. The child process has a new unique

\* process ID, and starts with stdin opened as file descriptor 0, and stdout

\* opened as file descriptor 1.

\*

\* file is a null-terminated string that specifies the name of the file

\* containing the executable. Note that this string must include the ".coff"

\* extension.

\*

\* argc specifies the number of arguments to pass to the child process. This

\* number must be non-negative.

\*

\* argv is an array of pointers to null-terminated strings that represent the

\* arguments to pass to the child process. argv[0] points to the first

\* argument, and argv[argc-1] points to the last argument.

\*

\* exec() returns the child process's process ID, which can be passed to

\* join(). On error, returns -1.

\*/

Int Exec( int virtualFileAddr, int numOfArguments, string[] args)

{

Create new process //save the PID

Read virtual memory string //get file name vAddr

Read virtual memory //run file

// int exec(char \*file, int argc, char \*argv[]);syscall.h

If ( newprocess.execute () ) //if correctly executes

Return PID

Return -1;

}

TestCase: child exits normally

TestCase: join with non-child

TestCase: child exits with exception

//if child exits normally return 1. Exits with unhandled exception return 0. If PID doesn’t match up with

/\*\*

\* Suspend execution of the current process until the child process specified

\* by the processID argument has exited. If the child has already exited by the

\* time of the call, returns immediately. When the current process resumes, it

\* disowns the child process, so that join() cannot be used on that process

\* again.

\*

\* processID is the process ID of the child process, returned by exec().

\*

\* status points to an integer where the exit status of the child process will

\* be stored. This is the value the child passed to exit(). If the child exited

\* because of an unhandled exception, the value stored is not defined.

\*

\* If the child exited normally, returns 1. If the child exited as a result of

\* an unhandled exception, returns 0. If processID does not refer to a child

\* process of the current process, returns -1.

\*/

Int Join( int processID, int status)

{

If( ! this.isparent(pID)) //check to see if I am parent.

Return -1;

//proceed with KThread join // int join(int processID, int \*status);syscall.h

If( process terminates correctly)

Return 1;

else

Return 0;

}

TestCase: normal exit

TestCase: check for children have parents

TestCase: check for all files closed

\* Terminate the current process immediately. Any open file descriptors

\* belonging to the process are closed. Any children of the process no longer

\* have a parent process.

\* status is returned to the parent process as this process's exit status and

\* can be collected using the join syscall. A process exiting normally should

\* (but is not required to) set status to 0.

\* exit() never returns.

void Exit( int status)

{

Close all file descriptors

Forall ( this. Children )

Children.hasParent == false

//clean up the virtual memory space

//add space taken up to the free physical pages linkedList

}

**TASK IV:**

Implement a lottery scheduler. The major difference is the mechanism used to pick a thread from a queue. Lottery scheduler must implement priority donation. Instead of donating priority, waiting threads transfer tickets to threads they wait for.

The owner’s ticket count Is the sum of its own tickets and the tickets of all waiters, not the max.

Do not keep an array containing an entry for every ticket.

Real tickets in the system is guaranteed not to exceed Integer.MAX\_VALUE, instead of 7.

Lottery scheduler is based on priority scheduler. priority scheduler picks the ready thread with the highest priority, but lottery picks a random number and picks a winner based on that.

instead of the ready queue used in priority scheduler, we use a binary tree.

partial sum tree:

each node contains number of tickets and number of threads

numTickets = left.numTickets + right.numTickets

numThreads = left.numThreads + right.numThreads

the root will contain the total number of tickets in all ready threads, and the total number of ready threads

to simplify things, we have implemented our tree with a hard limit

up to 1024 threads may be ready at a given time

this can be changed any time, except when nachos is running

this allows us to find the winning thread in O(log(n)) time, while still maintaining O(n) storage requirements

node: contains parent, children, number of tickets, number of threads, and thread

only leaf nodes have a thread, only root has null parent

add(Thread thread): this method adds a thread to the ready tree

starting at root, pick the branch with the least threads

at root, height = maxHeight = log(max number of threads)

as you go down, height decreases

when height = 0, the thread is added to the node, which is a leaf node

iteratively bump down the tree until you reach a leaf node

create new nodes if necessary(if current.right = null or current.left = null)

leaf node now contains a thread and updates partial sums in parents, up to root

updateData(): starting at a leaf node, increases the number of tickets and number of threads in the node, if necessary

go up to root, each node above a leaf contains the partial sum of its children

chooseWinner(): this method of the scheduler picks a random number between 0 and the total number of tickets

similar to add

generate a random number between 0 and the total number of tickets

starting at current = root, compare winning number to number of tickets in left child

if winner > current.left.numTickets

current = current.left

else

winner -= current.left.numTickets

if we didn’t do this, the picker would be biased to the later threads. this is equivalent to giving each node a range of ticket numbers instead of simply a number of tickets

current = current.right

when current is a leaf node, return current.remove()

remove: set all data to 0, update parents

return the thread in this node

lottery scheduler contains tree.

the tree contains nodes

all methods of tree and nodes are private or protected

this prevents anyone except the scheduler from manipulating them