**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()

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

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

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.

**TASK III:**

Implement system calls: exec, join, exit.

**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.