Design Document for Nachos

Phase 2: Multiprogramming

Justyn Bélanger, Tyler Jessup,

Clément Mindjimba, Alain Levasseur

COSC 3407 - Operating Systems

GROUP 04

March 23rd, 2018

Task I: Implement Six File System Calls

For task 1 of the second phase of the nachos project, I am tasked with delivering the implementation of a few system calls of the operating system. System call are methods that perform a function that the kernel performs on behalf of a user program because it is essential that the function be protected from being accessed by just another program. For this task particularly, I have a set of system calls that pertain to the file system and I/O of the computer. Mainly the system call: Create, Open, Read, Write, Close, Unlink. These are used to grant access to the file system and memory of the computer, so the user can open read and write to files as well as opening and closing them and deleting them from the file system. For this task, it is required that we keep a few static variable to keep track of the number of files open, the number of processes using each file and an array of which files are waiting to be deleted. For this we will use an array of files, an array of integers and an array of boolean to accomplish the task previously mentioned in that order.

Modifying the Halt System Call

A simple modification must be done in the halt system call to ensure that only a kernel thread can halt the system. For this a simple IF statement is used to check if the current thread is an instance of Uthread and if so then just return -1 indicating an error.

Implementing the toName Function

To reduce the amount of code that need to be written a function called toName will be implement that takes in an integer and returns a string. The integer is the starting location of the name of the file and the string is the name of the file.

String toName(int name) {

Get the file name (max 256 bytes)

Grab that whole section of memory and place in a byte array

Find the end character in the byte array

Convert name into string

Return name

}

Implementing the Create System Call

The create system call will have one argument which is the starting byte of the name of the file. This value will then be transformed into a name using the toName method at which point it will be checked if it is present in the files table. If it was not present, it will be created and the user will be notified with the index of the file. The same goes for if the file wasn’t found, it will simply create the file and return its index in the file table. If the function returns negative one, it means that the file table is full and a file cannot be created.

Int create(int name) {

Call toName with the name parameter

Check the file table for the file

IF it is in the file table

Return the index

Check for an open spot in the table

IF none

Return -1

ELSE found create file in the file system

Return the index

}

Implementing the Open System Call

The open system call is very similar to the create system call because the create system call has to perform the same function as its little brother. This is because the create system call has to make sure it's not creating something that’s already there. The difference is that the open system call cannot create a file in the file system. Neither of the function can open or create a stream, only a file on the disk.

Int open(int name) {

Call toName with the name parameter

Check the file table for the file

IF it is in the file table

Return the index

ELSE

Return -1

}

Implementing the Read System Call

The read system call is similar to the previous. The function must convert bytes to strings but this method has a few more parameters that have to be passed to it. It must receive the position of the first byte of the name of the file, the first byte of the address of the buffer that will store the data being read and the length of the data that is being read to know when to stop reading. It performs the read and returns the number of bytes read because it could be different than the amount the bytes requested to be read. Also, if it doesn’t have the file in the table is must return -1 to indicate to the user that the file was not available.

Int read(int name, int buffer, int size) {

Call toName with the name parameter

Check file table

IF not in table

Return -1

ELSE

Read the file and store it in a buffer for the process

Return the size of the file read

}

Implementing the Write System Call

The write system call is similar to the previous. The function must convert bytes to strings but this method has a few more parameters that have to be passed to it. It must receive the position of the first byte of the name of the file, the first byte of the address of the buffer that contains the data being written and the length of the data that is being written to know when to stop writing. It performs the write and return the amount of bytes written because it could be different than the amount the bytes requested to be written. Also, if it doesn’t have the file in the table is must return -1 to indicate to the user that the file was not available.

Int write(int name, int buffer, int size) {

Call toName with the name parameter

Check file table

IF not in table

Return -1

ELSE

Write to the file from what is in the buffer

Return the size of the file written

}

Implementing the Close System Call

As previous system calls, it must transform the byte number given into the name of the file. After this, it must check to see if the file exists in the table, if it does not it will simply return -1. If the file is found in the table the counter that keeps track of how many processes are using the file. If the amount is 0 or less the file must be closed and its value set to null in the table. If the file is awaiting for deletion, we must send the file name to unlink to make sure that there is proper cleanup of the file.

Int close(int name) {

Call toName with the name parameter

Check for file in the table

IF not found

Return -1

Decrement the number of processes using file

IF equal or less than zero

Close the file

Set value in the table to null

IF waiting for deletion

Send to unlink with the name parameter

Return 0

ELSE

Return -1

}

Implementing the Unlink System Call

The unlink system call is to delete a file in the file system. To ensure that nothing is done to compromise the file system, it checks the name of the file. If the name returns the amount of process using the file are greater than 0, then it will mark for deletion return -1 to mark that the file was not deleted. If the file is no longer used, it will be removed from the file table and deleted by the file system. If the delete is successful, it will return 0 otherwise it will return -1.

Int open(int name) {

Call toName with the name parameter

Find the index of the file in the table

IF there are still processes using the file

Mark for deletion

Return 1

ELSE

Remove it from the table

Marked for deletion equal to false

Delete the file store boolean

Return 0 for success

Return 1 for failure

}

Test Cases for Task I

To test and make sure that the implementation that was presented in this document works a series of tests will be performed. The test cases for this task are fairly simple, test each system call and make sure it does what it should. We will test the systems call with an empty file table, we will try to create a file when the file table is full, test the open and close system calls and also make sure to properly test the unlink system call. During all of the test, when testing a new system call, it will be attempted with an invalid name afterwards to test that the system doesn’t get fooled by a fake name.

Test Case 1: Test Open, Close, Read and Write with no files in the table

The first test is the simplest, simple call every system call and hope to receive an error. This meaning that the value returned is -1.

Test Case 2: Test Create, Write and Read (with full file table)

At the beginning there should be no files open to begin with so it is safe to assume that if a file is created and can be written to. The file was created properly. The write and read system calls will be tested by simply writing and read and making sure that the output is correct meaning that the data coded to be written is the same as the data read. Afterwards, the file system will be filed and an attempt will be made to create another file and this should result in an error shown as a return of -1.

Test Case 3: Test Open and Close

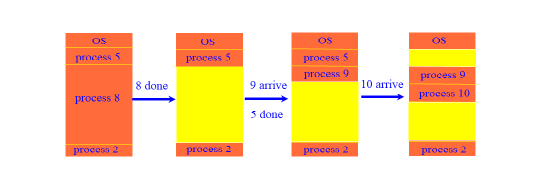
To test the open and close system call, a file will be open and attempted to be written to, if there is success when writing, it will be assumed that the system call performs its duty. Afterwards, the file will be closed and attempted to be written to and if successful, the test will result in a failure but if unsuccessful the test is classified as a success.

Test Case 4: Test Unlink

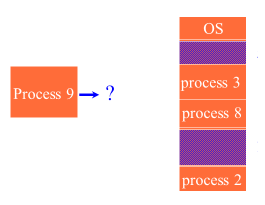
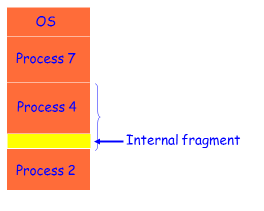
To test the unlink system call it will simply be called in the program and an attempt will be made to open the files and if it fails the test will be considered a success.

# Task II: Implement support for multiprogramming

The problem imposed by Task II is about a process' memory management and the complexities inherent to a processes memory allocation, and its reading and writing to physical memory. These complexities come from memory fragmentation and segmentation. Fragmentation occurs in memory both internally and externally when a process is allocated. This can be illustrated from the basic model of base and bounds, shown in Figure 1.0, Figure 2.0, and Figure 3.0:



**Figure 1.0:** *The basic problem of contiguous memory allocation with base and bounds. Image cropped from* [*http://www.cs.laurentian.ca/kpassi/cosc3407/schedule.html*](http://www.cs.laurentian.ca/kpassi/cosc3407/schedule.html) *[Lec 13]*



**Figure 2.0:** *Internal fragmentation of a process A with base and bounds* **Figure 3.0:** *External fragmentation of multiple processes with base and bounds*

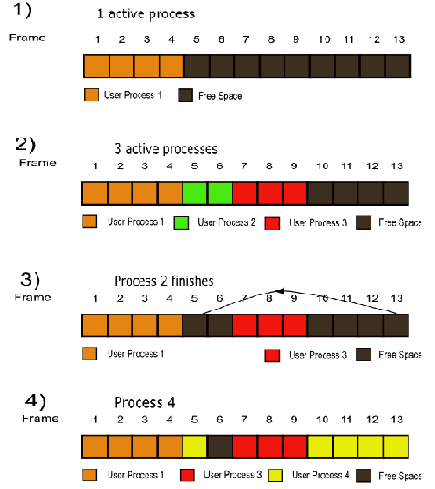
*Images cropped from* [*http://www.cs.laurentian.ca/kpassi/cosc3407/schedule.html*](http://www.cs.laurentian.ca/kpassi/cosc3407/schedule.html) *[Lec 13]*

To address these related problems, a memory management scheme is introduced to overcome these complexities and pitfalls. Rather than dealing with contiguous memory an address translation scheme is introduced to virtualise a process to be running independently on its own single machine. This scheme is the separation of a processes logical (or virtual) memory apart from its physical memory. The scheme uses a frame table for individual frames abstracted upon physical memory, and a segmentation and paging (even multi-level) scheme to map virtual-to-physical memory. This scheme is discussed next in Task II Solution.

Implementing the Solution

In the implementation of UserProcess three methods will have to be adapted for modifications to this new memory management scheme: 1. readVirtualMemory (), 2. writeVirtualMemory () and 3. loadSections (). The main modification comes with the newly introduced virtual address space (which maps logical memory to physical memory) and the introduction of the FrameTable, Segment Table, and low and high level page tables.

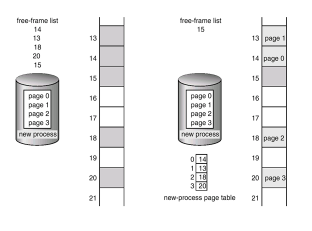
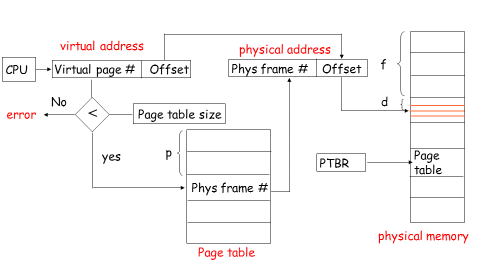
The FrameTable is an abstraction upon physical memory that separates physical memory into a set of frames. When a process is allocated in physical memory it is allocated the required amount of frames for its process and when reading and writing to memory it modifies the data within these frames. This process is summarized in Figure 4.0:



**Figure 4.0:** *A process A requiring x frames of physical memory. The frames are allocated dynamically but also not contiguously, which introduces the complexities of gaps in the memory pool as external (and also internal) fragmentation*

*Image cropped from* [*http://www.cs.laurentian.ca/kpassi/cosc3407/projphases.html*](http://www.cs.laurentian.ca/kpassi/cosc3407/projphases.html) *[Phase 2 Design Tutorial]*

The FrameTable is implemented by a dynamically allocated linked list of frames (globally accessible to all UserProcess(s) in the UserKernel), and each frame containing an associated reference to the higher-level page tables of this UserProcess. The FrameTable is later used to map the virtual addresses of the logical space (implemented by segmentation and paging) of the virtual address space translated to the physical address space and hence physical memory. The FrameTable is initially allocated to hold a max capacity of 8 frames (arbitrary). When a process requests an allocated frame for its virtual pages, the FrameTable allocates new frames to that UserProcess where the frames are available as shown in Figure 4.0, but more specifically in Figure 5.0. The implementation of this virtual-to-physical address translation is also shown generically in Figure 6.0.

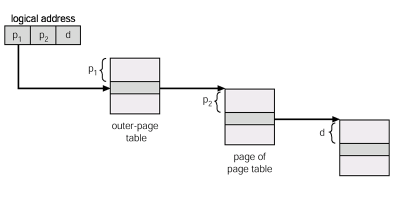
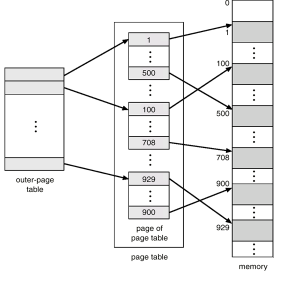


**Figure 5.0:** *A process request to allocate pages based on free frame availability*

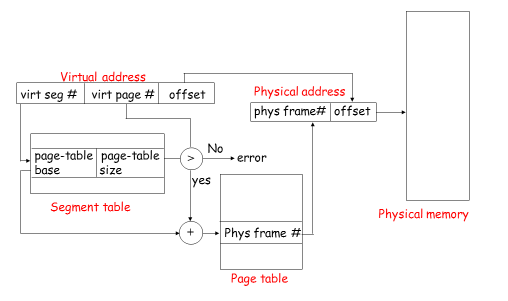
**Figure 6.0:** *The address translation of virtual addresses in the paging scheme (or logical/virtual) space to physical address space*

*Image cropped from* [*http://www.cs.laurentian.ca/kpassi/cosc3407/schedule.html*](http://www.cs.laurentian.ca/kpassi/cosc3407/schedule.html) *[Lec13]*

This scheme can be optimized yet again through another layer of abstraction using multi-level paging. The same idea persists yet the address translation is able to be made sparser through the extra level of abstraction. Together with segmentation, multi-level paging and translation through the frame table can achieve the desired results for the virtual-to-physical memory translation. This hybrid approach is finalized in Figures 7.0, 8.0, and 9.0.



**Figure 7.0:** *An extra layer of abstraction for* *multi-level paging to overcome page* **Figure 8.0:** *Address translation of multi-level* *paging sparsification*



**Figure 9.0:** *The hybrid approach in the implementation of the union of segmentation, paging and multi-level paging and their virtual-to-physical address translation*

*Image cropped from* [*http://www.cs.laurentian.ca/kpassi/cosc3407/schedule.html*](http://www.cs.laurentian.ca/kpassi/cosc3407/schedule.html) *[Lec13]*

In essence, the virtual addresses are links to the segment table and the page table which are virtualised for a machine independent layer for an individual user process. By obtaining from the page table the physical frame number of the frame within the FrameTable, and combining this with the offset from the base of the frame, locates the physical address in physical memory where the data is stored. Upon this locating, physical memory can be read from or written to via the UserProcess in writeVirtualMemory () and readVirtualMemory (). Finally, upon a process's exit () call routine, all of the User Process’s memory must be deallocated, and all frames released back to the FrameTable as available for future User Processes.

Finally, when a section is loaded, this new memory management scheme will override the existing scheme which assumes that all virtual addresses are exactly physical addresses. As well, the necessary synchronization primitives will be needed through various elements of this translation to ensure cooperation threads running virtualised in parallel in the OS do not make the address translation state go out of sync.

Implementing Implemented Attributes and Functionalities

Many existing implemented attributes and functionalities will be used and such included are:

//Processor Parameters

//Some such parameters give necessary constraints and physical data access

/\*\* Size of a page, in bytes. \*/

public static final int *pageSize* = 0x400;

/\*\* Number of pages in a 32-bit address space. \*/

public static final int *maxPages* = (int) (0x100000000L / *pageSize*);

/\*\* Number of physical pages in memory. \*/

private int numPhysPages;

/\*\* Main memory for user programs. \*/

private byte[] mainMemory;

//Processor Methods:

//Some such existing functionality is needed for Task II:

//trivial specification

public int getNumPhysPages()

//returns the processes data from the physical memory

public byte[] getMemory()

//construct a new virtual address

public static int makeAddress(int page, int offset)

//obtain the offset index from a virtual address

public static int offsetFromAddress(int address)

//obtain a page index from a virtual address

public static int pageFromAddress(int address)

Implementing Address Translation

//translate a virtual address into a physical address and mark the page as written to or dirty

//some key points of this method are noted

private int translate(int vaddr, int size, boolean writing)

// calculate virtual page number and offset from the virtual address

int vpn = *pageFromAddress*(vaddr);

int offset = *offsetFromAddress*(vaddr);

//obtain the virtual page

entry = translations[vpn];

//obtain the physical page number

int ppn = entry.ppn;

// set used and dirty bits as appropriate

entry.used = true;

if (writing)

entry.dirty = true;

//locate the physical address translated from the virtual address

int paddr = (ppn\**pageSize*) + offset;

//read from physical memory a certain number of bytes

private int readMem(int vaddr, int size)

//the read value

int value = Lib.*bytesToInt*(mainMemory, translate(vaddr, size, false),size);

//write to physical memory a certain number of bytes

private void writeMem(int vaddr, int size, int value)

Lib.*bytesFromInt*(mainMemory, translate(vaddr, size, true), size,value);

//Need functionality of Translation Entry and its interpretations i.e. save the record

//fields listed in full below

//Allocate a new translation entry with the specified initial state.

//vpn the virtual page numben.

//ppn the physical page number.

//valid the valid bit.

//readOnly the read-only bit.

//used the used bit.

//dirty the dirty bit.

Modifying UserProcess

**public** UserProcess() {

//existing constructor source

//allocate FrameTable in UserKernel

//Frame table manages dynamic links for gaps in the memory pool automatically

//i.e. solves external fragmentation

//allocate SegmentTable

//allocate top level page table

//allocate low level page table

//link FrameTable to UserProcess

//link pageTable to FrameTable using virtual addresses

}

Implementing readVirtualMemory

public int readVirtualMemory(int vaddr, byte[] data, int offset, int length)

{

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

//old implementation vs. new below

//validate bound constraints

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

return 0;

//until requested memory read is exhausted

while(bytesRead>0)

{

//compute virtual segment number, virtual page number and offset from vaddr

//from segment table retrieve page table base and size

//find page table index by adding page table base and virtual page number

//index into page tables

//retrieve physical frame number

//compute physical address (physical frame number + offset)

//index into frame table

//locate physical address in physical memory

//read a size of bytes using readMem

//add bytes read to amount

//remove amount of bytes read from bytesRead

//move to next physical address

}

return amount;

}

Implementing writeVirtualMemory

public int writeVirtualMemory(int vaddr, byte[] data, int off, int length)

{

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

//old implementation vs. new below

//validate bound constraints

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

return 0;

//until requested memory write is exhausted

while(bytesWrote>0)

{

//compute virtual segment number, virtual page number and offset from vaddr

//from segment table retrieve page table base and size

//find page table index by adding page table base and virtual page number

//index into page tables

//retrieve physical frame number

//compute physical address (physical frame number + offset)

//index into frame table

//locate physical address in physical memory

//write using writeMem

//add bytes written to amount

//remove amount of bytes read from bytesWrote

//move to next physical address

}

return amount;

}

Implementing the loadSections

protected boolean loadSections() {

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;

//modify to manage new memory management hierarchy

section.loadPage(i, vpn);

}

}

Test Cases for Task II

*Test 1:* Ensure that the UserProcess constructor configures the set up i.e. test Frame, Segment and Page Table configurations.

*Test 2:* Ensure all address translations are correct i.e. virtual segment number, virtual page number, physical page number, page table base, physical frame number, and physical address.

*Test 3:* Ensure that amounts read/written are correct and return, and that even on error the process returns to state normally not abnormally terminated.

*Test 4:* Ensure loadSections() is modified correctly to adapt to new memory management scheme.

# Task III: Implement three process system calls

For this task we must add a process ID to each process to differentiate between them. This introduces two new variables: PID and NPID. PID is the Process ID of a specific process whereas NPID is a number count of all the processes that have been created to ensure no two processes have the same ID. It is assumed that NPID will never overflow to keep implementation simple. Because each process gets its own ID the constructor must also be updated to implement the changes:

UserProcess() {

PID <- NPID

increment NPID

existing constructor code

}

Another new variable to be added is an array to keep track of how many children a process has. This array will be referred to as 'children' inside the pseudo-code portion of this report. This is particularly useful for the join system call because a process must not join to another process unless it is their child.

Implementing the exec system call

The exec system call handles the creation and execution of a new user process. A lock must be acquired before a new UserProcess is created because the PID might end up being duplicated if two processes are created before NPID is incremented. Then it must read the file name from the address it was given. It must also read the arguments from memory through the use of the passed in array of pointers. It then returns the process ID of the new process.

exec(int fileNameAddress, int argumentCount, int[] argumentAddress){

acquire lock

newProcess <- create a new UserProcess

release lock

fileName <- String from virtual memory at location fileNameAddress

for each argument specified by argumentCount

args[index] <- String from virtual memory at index

call execute with fileName and args as arguments

Return process ID of newProcess

}

Implementing the join system call

Must determine if the process ID to join is in fact a direct child process. If it is not a direct child, children of children do not count, the method will return with an error code. When a child PID is selected the method will call the join method located in the UserThread which is inherited from KThread in order to join the parent thread to the child thread.

int join(int PID) {

if PID is a child process

call UThread.join

else return -1

}

Implementing the exit system call

void exit(int status) {

send the exit status to the parent process

call UThread.finish

if numProcesses is zero

halt the machine

}

Test Cases for Task III

Test Case 1: Test execution of two new processes

For the first test case we will create two new processes. The first process will be a simple process that will calculate a sum and then exit. The second process will be created using a name that does not exist. This will test for user error in the loading of a process name and should return with an error process ID of -1.

Test Case 2: Test join on a child process

In this test case we must create a child process that will simply run an endless loop in order to keep it running. The parent will be joined to the child.

Test Case 3: Test join on a non-child process

For this test case three processes will be created. The first process will create a child. This child will exec a second process and then attempt to join the parent process. The join method will return with a -1 as an error. After the error is returned the original parent will attempt to join the child of the first child. This too will return an error code of -1.

Test Case 4: Test exit

We will need two processes for this test case. The first process will be used to ensure that exit works by simply making the system call and checking if the process thread has been terminated. The second process will be the only process left and should halt the machine when it is exited after the first process.