Operating System Project
NachOS
M1-MOSIG

Alisa Patotskaya, Claude Goubet, Riyane Sid Lakdar,
Udhayan Venugopal, Yulia Patotskaya

January 28, 2016

Contents

1 Key Features **2**

2 User Interface **2**
2.1 I/O System Call Description . . . 2
2.2 MultiThreading . . . 3
2.3 Filesystem . . . 3
2.4 Network . . . 3

3 Test Description **4**
3.1 I/O operations testing . . . 4
3.2 Multithreading, Multiprocessing & Virtual memory . . . 4
3.3 File systems . . . 5
3.4 Networking . . . 5

4 Implementation **5**
4.1 Console Input/Output . . . 6
4.2 Multithreading . . . 6
4.3 Virtual Memory . . . 7
4.4 File system . . . 8
4.4.1 Implenting Paths . . . 9
4.4.2 Open up to ten files concurrently . . . 9
4.4.3 Increase the File Size . . . 9
4.5 Network . . . 10

5 Group Project Management **10**

6 Conclusion **11**

Abstract

This project aimed to implement basic operating system features, ranging from I/O operations to File System in NACHOS(Not Another Completely Heuristic Operating System), in order to study the exact functioning of an Operating System.

1 Key Features

The current release of NachOS provided by our team includes new features improving usability, performance and security of the original system. The usability of console was improved by implementing input and output functionality supporting integers and strings, while the initial version could only work with chars. Reliable multiple threads access to shared resources, such as console and file stream, are provided by synchronization tools such as semaphores, locks and condition variables. Virtual memory represents more sufficient security level by the creation of isolated address spaces for multiprocess execution and by allowing an access to larger memory spaces. A file system architecture based on directory tree paradigm handling basic operations with files and directories and allowing to store and access data on disk was also developed. Finally, a network system has been implemented allowing separated machines to communicate through an unreliable communication stream.

2 User Interface

2.1 I/O System Call Description

PutChar(char)

- Signature: void PutChar(char)
- Description:To write the input character to the input stream.

GetChar(char)

- Signature: char GetChar()
- Description : To read a character from the input stream. item
- Returns: A character or EOF (-1) when reaching the end of the input stream

GetCharInt()

- Signature: int GetCharInt()
- Description: To get the integer value of the next character from the input stream.
- Returns: Integer value, if -1 then we reached EOF(the end of file)

PutString()

- Signature: void PutString(char*, unsigned)
- Description: To write the given string into the output stream.

GetString()

- Signature: void GetString(char*, int)
- Description: To read a string from the input stream.

PutInt()

- Signature: void PutInt(int)
- Description: To write an integer into the output stream.

GetInt()

- Signature: void GetInt(int*)
- Description : To read an integer from the input stream and store the result into the input pointer.
- Returns: an integer value

Halt()

- Signature: void Halt()
- Description : To stops the nachos virtual machine. Will print the performance statistics

2.2 MultiThreading

UserThreadCreate()

- Signature: int UserThreadCreate(void (void), void)
- Description: To create a new thread within the existing calling process. Thereby the new thread will start executing the input function.
- Returns: 0 when thread successfully created or -1 in case or failure in thread creation.

UserThreadExit()

- Signature: void UserThreadExit()
- Description: To terminate a thread or a process. Removes the thread data structure from nachos system.

UserThreadJoin()

- Signature: void UserThreadJoin(int)
- Description: To wait for the thread described by the input TID to finish.

ForkExec()

- Signature: int ForkExec(char, char)
- Description: To create a process independent from the caller and transfer the control to it.
- Returns: TID of the new process if successful or -1 in case of error.

2.3 Filesystem

Open()

- Signature: int Open(const char*, bool)
- Description: To open a file described by its name
- Returns: id of the opened in the current thread structure or -1 if the file does not exist.

Write()

- Signature: void write (char*, int, int)
- Description: To Write a specified number of bytes from the given buffer in the opened file described by its id

Read()

- Signature: int Read (char*, int, int)
- Description: To read a specified number of bytes (2nd argument) from the opened file described by its id (3rd argument) into the given buffer (1st argument).
- Returns: The number of bytes actually read if the file isn't long enough or if it is a I/O device and there are not enough characters to read.

Close()

- Signature: void Close (int)
- Description: To close the file described by its id.

2.4 Network

The aim of Network development is to allow communication between distant machines. Indeed, our kernel may execute a set of functions which allows one to send/receive messages through remote networks. The use of these functions is described in subsection 4.5. Although, not all of the network functions have been integrated to the set of system calls proposed to user programs. They can hence only be called by kernel functions. However, these features will be available to the user programs in our next release of the operating system.
The design of this future tools is described as follows:

OpenSocket()

- Signatre: int OpenSocket(int , const char*, int)
- Description: To create a socket linked to the local port given by the first parameter. This socket will allow communication with a remote machine described by its address, given by the second parameter, through the foreign port given by the third parameter.

- Returns: the socket id or -1 if the connection failed.

CloseSocket()

- Signature: void CloseSocket(int)
- Description: To close the input socket previously created by OpenSocket.

SendMsg()

- Signature: void CloseSocket(int, const char*, unsigned int)
- Description: To send a message given by the second parameter of size given by the third parameter, through the socket given by the first parameter.

SendMsg()

- Signature: bool CloseSocket(int, const char*, unsigned int)
- Description: To send the message given by the second parameter of size given by the third parameter, through the socket given by the first parameter. The message will be resent at fixed rate within a fixed time gap, until an acknowledgment will be returned by the foreign machine. Return true if an acknowledgment has succesfully ben returned and false otherwise. The input message may be delivered however the function returns false.

ReceiveMsg()

- Signature: int ReceiveMsg(int, const char*)
- Description: To wait until a message arrives through the first parameter socket. In case of success an acknowledgment is sent to the transmitter.
- Returns: the size of the message or -1 if the socket has been closed before a message has been received.

3 Test Description

A set of tests has been implemented in order to check the correctness of the our developed release. These tests can be found in the folder code/tests.

3.1 I/O operations testing

ioTest.c
To run: ./nachos-userprog -x ./ioTest
Test which checks that PutString() system call works correctly.
putcharint.c
To run: ./nachos-userprog -x ./putcharint
Test which checks the correctness of the execution of the GetCharInt() system call: reads one character and puts it into the console.
halt.c
To run: ./nachos-userprog -x ./halt
Test which checks the correct execution of the Halt() system call.

3.2 Multithreading, Multiprocessing & Virtual memory

Note: all of the tests below are executed in user mode thanks to the command ./nachos-userprog -d a -x <TestName>. This mode allows to visualize how the virtual and the physical memory correspond between each others. Also, by running the tests using ./nachos-userprog -frameProvider <algoNumb> -d a -x <TestName>, where algoNumb is the strategy number which will be used to find free frames: 0 - First free, 1 - Last free, 2 - Random free, 3 - Block free. You may run the tests below and control that no matter which allocation algorithm is being used, the virtual memory stays well managed.
multithreadingTest.c
To run: ./nachos-userprog -x ./multithreadingTest
This test was developed in order to check the correctness of the user available thread system calls, such as UserThreadCreate(), UserThreadJoin(). Four threads are created, running different writing functions. Besides, it eventually checks the correctness of PutChar() and PutInt() system calls.
multithreadingLongTest.c
To run: ./nachos-userprog -x ./multithreadingLongTest
This test runs four different threads, which each call PutString() and PutChar() system calls with long string arguments. Thus, this test will control that even with big input parameters, the system call programs are still working correctly since each thread has enough memory. This test also checks

UserThreadCreate() and UserThreadJoin() system calls. **userpagest.c**
To run: ./nachos-userprog -x ./userpagest
This test checks the correctness of the virtual memory management for threads and processes with our developed OS. Besides, it also tests the correctness of the used PutChar() and PutInt() system calls. This test executes two different processes, which's codes are located in the userpages0.c and userpages1.c source files. Each of these two processes will create several threads themselves and print messages using PutChar() and PutInt() system calls.

3.3 File systems

filesTest.c
To run: ./nachos-filesys -cp filesTest filesTest
./nachos-filesys -d f -x filesTest
Simple test, which checks that the Read(), Write(), Create(), Open() and Close() operations are working correctly for files.
fileStressTest.c
To run: ./nachos-filesys -cp fileStressTest fileStressTest
./nachos-filesys -d f -x fileStressTest
Stress test for file system. Makes sure that it is impossible to open more than 10 files at the same time. If the user is attempting to open more than 10 files, he will receive a corresponding error message.
fileReadWriteTest.c
To run: ./nachos-filesys -cp fileReadWriteTest fileReadWriteTest
./nachos-filesys -d f -x fileReadWriteTest
This test was done to unsure that it is impossible to open twice a file in read and write mode.
testFilseSys.sh
To run: Place yourself into the os_projectNACHOS/src/code/test directory and run the script with ./testFilseSys.sh
This test is designed to test file system operations such as sub directories creation, moving within directory hierarchy and importing files.

3.4 Networking

Test Description Since the network features of our OS are only available at kernel level, it is impossible to create any user program to test this features. However, we have developed an external program (completely independent from our kernel) to lead our test.
In order to execute this test, you may execute the **ringTopology** implemented in **"ringTopology.cc"** in the directory "networkTest".

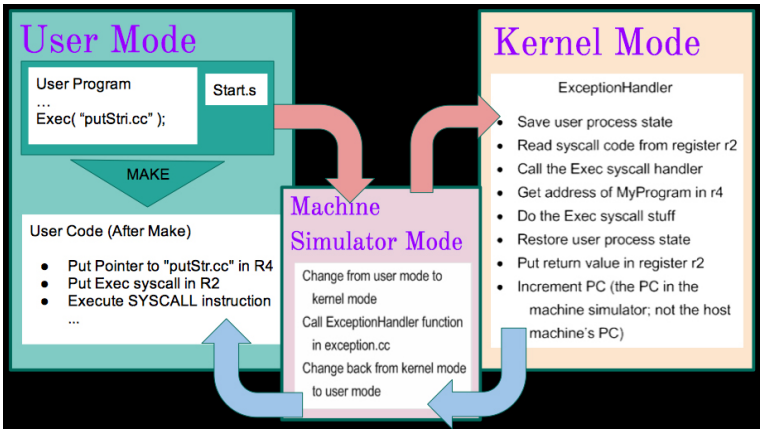
This program allows one to run a given number of independent processes (UNIX processes, and not nachos processes). Each one of these process will run a nachos program which will receive a message from another given process. Then, will forward it to an other one. To make this test stand, we had to design a synchronization system between the process to ensure that a transmitter will never send a message before the receiver has been created.

This program may be launched with several options (the "-help" option lists the available options). Two of these options allow us to test the main features of the network system:

- The option **"-reliability** $< p >$**"**, where p is the probability for a message to be successfully sent, allows to notice that whichever data loss accrues during the exchange of a message, the user's messages are still successfully forwarded.
- The option **"-nachosDebug"** allows to print the nachos information relative to the network functions. This option helps to identify the lost message and the corresponding message ressend. This option also allows to identify the acknowledgment received and the way the nachos uses them to stop sending again the corresponding message.

4 Implementation

NachOS architecture provides two standard modes for program execution: user mode and kernel mode. We use system call to switch from the user program into the kernel, and we use registers to transmit the data between these two modes. This causes an interrupt to ExceptionHandler in the kernel and verifies SyscallException, identifies the requested system call from the register.



4.1 Console Input/Output

We started by writing and reading only one character at the time from console. In order to avoid busy waiting while the console is printing the data, we have created a synchronized console. **Semaphores** will permit this waiting allowing the console to block and wake up threads if necessary. *PutChar* writes a character into the output stream. Once the console device is done writing the character, it invokes the user-supplied procedure writeDone(), to signal that the writing is done. *GetChar()* retrieves a character from the input stream. GetChar returns EOF if anymore new data is not available.

String read/write are handled with SynchPutString() and SynchGetString() which are based on already implemented functions working with chars. But now to transmit the data between user and kernel we do not put into the register the character itself but a pointer to the buffer string in memory. To provide console reading/writing in kernel mode correspondingly GetChar and PutChar are called for each char in string.

Integer read/write operations use previously designed reading string from console, after which we perform type casting to integer.

The problem of implemented SynchGetString and SynchGetChar system calls is that the 'y' character is confused with an end of file. That is why we have implemented new system call **GetCharInt**, which returns an integer value instead of char value. The point is that this function does not takes into account character value, but using information about file size and by this way recognizes end of file. If end of file is reached, this function will return -1.

To provide multithread shared access to input and output stream we use **Locks** ('writing' and 'reading') - to protect reading and writing operations as they are not atomic. Locks will be described below in more details.

4.2 Multithreading

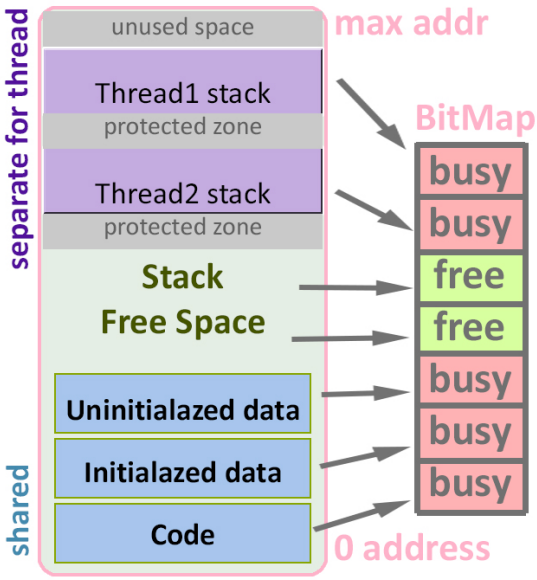
Multi-thread access **synchronization** is provided with Semaphores, Locks, Condition variables. For implementation of **Locks** we have used the concept of Semaphores, but the difference is that only one thread is allowed to enter the critical section. **Conditions** are used when we need to perform some action on triggering some event. Waiting threads are put into queue and woken up on signal or broadcast.

To optimize the use of the ware features of a computer, it is important for the user to run different programs concurrently.

For this purpose, we initially implemented a system to manage many **threads** within the same address space, sharing the same code global data and heap regions. This system makes the stack virtual region of each thread completely separated from the stack of the other threads. It allocates for each thread with a fixed size stack and within a contiguous virtual region. This virtual region may be used by a future thread as soon as the current one not exits.

But when running the thread demands allocation of more memory than the fixed size we provide, that leads to the erasing of nearby memory space, belonging to other threads.

That is why to guarantee data safety, we have bounded the upper and lower limits of each threads stack with an inaccessible region to ensure that each thread will not try to allocate memory out of his stack and corrupt other thread's stack.



From the point of security it is reasonable to execute the concurrent programs in separated address spaces. This is the sole reason for implementation of a **multi-processing system**. The user programs can run independent independently, as they are allocated in independent executable files. For each new program a completely different address space would be defined.

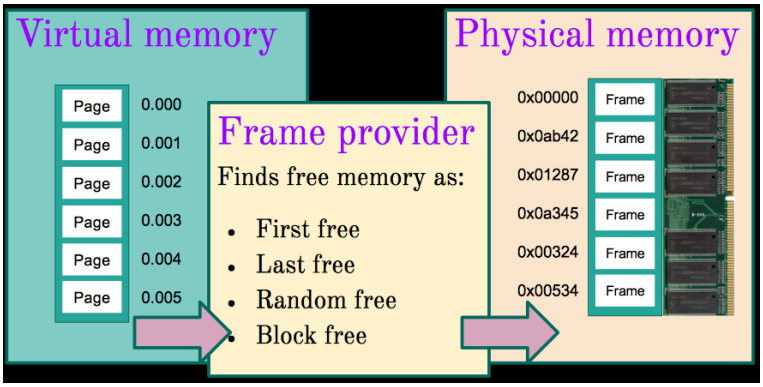
One of the most interesting features that we have implemented to manage our thread and processes is a system for garbage collection. Indeed, every time a thread finishes its execution, our system will check if no other thread or process is currently using the same address space. If so, the current address space will be marked as free for the virtual and physical memory systems.
The garbage collecting will run automatically no matter if the user calls the specific functions to close a thread (such as UserThreadExit) or not. So in case if an user forgets to explicitly remove the memory used for the thread and processes(function UserThreadJoin), the system runs memory free mechanism itself. Thus, no memory leakage is caused by the user, which is an advantage compared to most UNIX thread management systems.

Finally, the management of our several threads and processes has led us to improve the system. But we also need to ensure that thread that will call this turn off system call will not interrupt other running thread, hence interrupt probably be critical user tasks. Thus we first have implemented a turn off system which is synchronized with all the other user threads. This system waits for all the running threads to finish them task, allowing them to have access to all the existing features. Once all the threads have finished the tasks, our system will safely remove all its internal memory used since it is running, without disturbing any user thread's task.

4.3 Virtual Memory

The usage of stack has one serious gap, as for each thread only one fixed size piece of memory is given. This problem leads either to the erasing of nearby memory space or to deny to run a thread from the side of OS. So the next step of the development is to increase the memory space and providing independent address space for all the processes via paging memory management. The memory required for each process could be splitted into pages, so that for each process there could be allocated as much space as it needs.

Our physical memory is splitted into **frames** of fixed size, each of which has physical address. Virtual memory is divided into **pages** which has its virtual addresses. The translation between virtual and physical addresses is contained in page table. When a scheduler runs a process it loads translation table and requests for virtual memory. The OS use Page Table to find corresponding physical address.



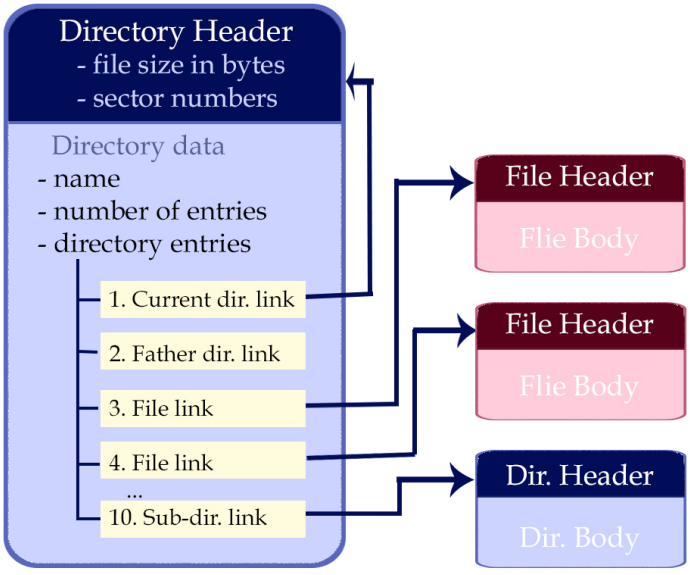
For this purpose we have implemented a **Frame provider**, which selects a free chunk of memory in correspondence to virtual address using one of four algorithms: First free, Last free, Random free, Block free. A user can select one of this algorithms depending on performance expectations, such as security, economy, simplicity. So each process can see and access to only his own memory space, completely separated from other threads.

4.4 File system

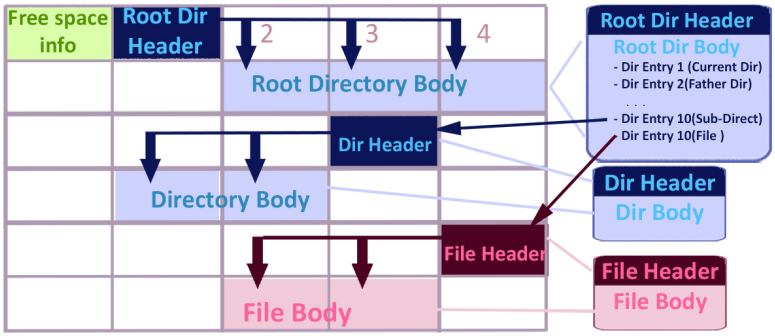
Our disk implementation is based on bitmap, where information about a files and directories stored on its sectors. The sector 0 contains information about the free space on disk that is available for allocation. In sector 1 we can find the root directory's header. Further we will use an emulation provided by NachOS with DISK file as hard disk .

In the new releasation of NachOS we have improved single root directory file system storage principle into the directory tree concept. The root of file system tree is a root directory, the nodes are sub-folders, the leaves are empty folders or files. For this purpose we designed **Create sub dir** function.

The implementation of sub-directory abstraction is based on the class structure of a file. In fact each sub-directory is represented on disk as file. It has, in the same way as a regular file, a **header** and a **body** (data region), which are placed in disk sectors. But the contents of sub-directory file has a restricted structure, which represents the list of files and folders inside of the directory.



Access to file or sub-directory is done via link to **header**. Each header fits inside an unique sector. In this header you will find meta-data such as the number of sectors allocated for the file/sub-directory's data and a table containing the indexes of file body sectors.



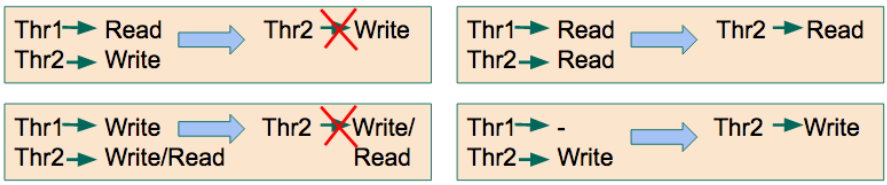
For the case of allocation of the file object on the disk the contents of a file is placed in the **body** sectors. But, in the case of a sub-directory, the body sectors are filled with the structures, which would be interpreted by corresponding method as a Directory object. After **FetchFrom** call, the Directory object will contain information about the folder (such as name and number of entries in the folder) and a list of Directory entries. Directory entries will contain the name of files and subfolders, identifier if the entry is a sub-directory, and number of sector where the header of entry file is stored. That is why we can know whether the target child is a file or a sub-directory before loading it. Each Directory can contain up to 10 Directory entries in the list. The first Directory entry in an array contains the link to current directory, the second - to father directory. So that we can place 8 more object in each sub-directory.

4.4.1 Implenting Paths

To provide parallel and safe access to files system, we have applied thread tools developed before. For the navigation the directory tree **ChangeCurrentDir** was designed. The information about current directory is stored inside of each thread, which browses file system. Therefore to navigate by input path between folders we parsed input string. As in every data section of a directory is given information of its adjacent directories, we can navigate through directories using its name.

4.4.2 Open up to ten files concurrently

In current OS release all user threads can open up to ten files at the same time. For this purpose inside of each thread we have an array of references to OpenFile objects. During execution of Open system call user will recieve ID as a result from 0 to 9 - an index of OpenFile object in this array.
But this logic is not enough to provide safe parallel access to shared files between threads. We have stated the file access policies for concurrent threads:



For this purpose we have designed a table which contains information in which mode file is opened and sector number where file is stored (every OpenFile object has its sector as an attribute). This table is stored in FileSystem data structure. Before opening a file in some particular mode we loop on the table and check if it is allowed to open file in this mode. We have the next rools: if user wants to open file in writing mode, we should check that file is not opened for reading writing. If user wants to open file in reading mode, we should check that file is not opened for writing. Besides, we do not give user ability to open more than 10 files at the same time.

Check wheither we can open a file and adding into table described above is atomic operation, no interruptions are allowed.

4.4.3 Increase the File Size

In order to optimize the maximum file size we had to optimize the use of the file header. Indeed, the header still has to fit inside one sector. The only way to do this is by using indirect links. The idea is to add two integers inside the header object (hence reducing the data sectors tables by two cells). One of

these integers will represent a sector containing indirect links, the other will point to a sector containing double indirect links.
An indirect link sector is a data sector which only contains a table of sector, hence we already multiplied the maximum size of a file by more than two.
Moreover, our next release will also contain the model of double indirect linking, where the sector will be filled with sectors pointing to indirect sectors. This model is still in development since only allocation is yet operational.
In order to have a clearer idea about the efficiency of this optimization we can refer to table bellow.

Model	NachOS values
Regular	$30 \times 128 = 3840$ Bytes
Simple indirection	$29 \times 128 + 32 \times 128 = 7808$ Bytes
3584 + 4096 + Double indirection	$28 \times 128 + 32 \times 128 + 32 \times 32 \times 128 = 138752$ Bytes

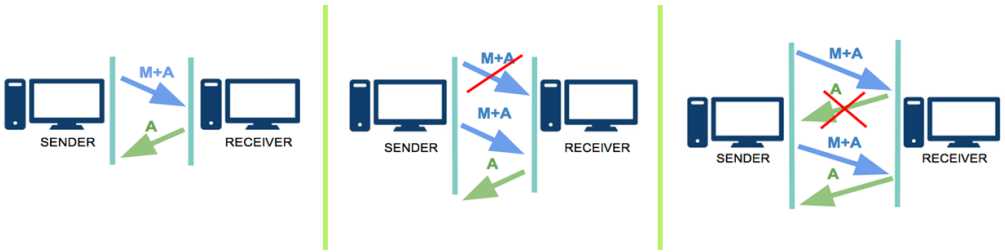
4.5 Network

To implement our network system, we have considered the used network as "unreliable": A sent message may be lost at any moment and the transmitter has no way to know it. Thus, to remove (or at least limit) the impact of this random data lost on the user, we have designed two network protocol layers over the physical network layer.

The first protocol (over the physical protocol) is in charge to root a message to the approprite machine and mailbox (port).

The second one is in charge to detect a lost of data during a transmission and to fix it. To do so, this protocol adds a unique acknowledgment to each user message. This message and acknowledgment couple will be ressent untile the acknowledgment is returned.

To implement this protocol, we first had to identify as much transmission scenarios as possible (finding all the scenarios is a problem with no solution).



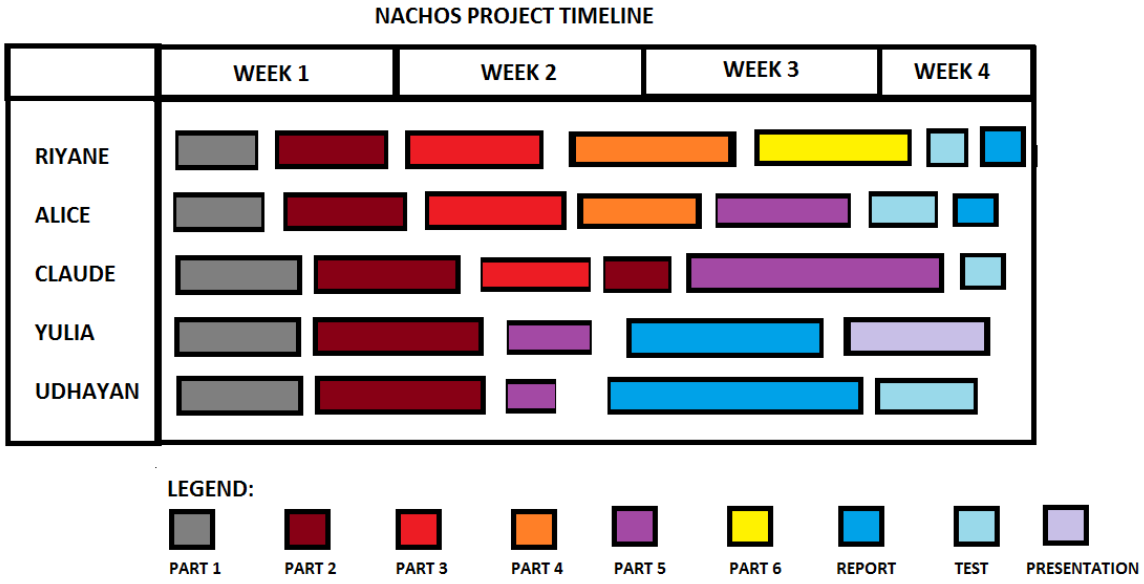
Then we had to implement appropriate structure to store a history of the exchanged messages. This structure allows to show for each received massage which scenario we are considering, and then to react consequently.

Within any distributed system, the network manager needs to be consistently observing the the network state. This permanent and continuous action allows to deliver an input message to the user as soon as it arrives. Within our system, this task is executed by a dedicated kernel thread: postmanThread. This thread is in charge to observe the network. When a message arrives, this thread will react according to the type of the message and to a specific scenario.

Our network system is also using an other thread to periodically send again a message that has not received an acknowledgment.

5 Group Project Management

Before starting off with implementing any part, we thoroughly read the nachos documentation and the list of functionality that has to be developed during this project. We initially worked all together for the part from 1 to 3 and for the part from 4 to 6, we have splited out tasks among us as they can be worked concurrently with out relying on the other part completion. Simultaneously we were compiling the report as it is vital about our project. Then we performed suitable tests to make sure our functionality works as recommended. We made sure that every one knows about what other were working individually and vice versa.



6 Conclusion

Current release provides useful features regarding each fundamental part of operating system architecture and provides solutions for several shortcomings of initial NachOS version. An improved I/O console has been implemented for the ease of usability to perform the basic Input and Output mechanism. Secondly, multi-thread access synchronization and virtual address space abstraction are being added to safeguard the critical resource of the code. Also overcame the limitation of restricted file size(3.75 KB) and ease of file handling with the basic file operations. Also a reliable networking protocol has been deployed to send data in terms of packets without any loss between two machines. We should admit an advantage of current thread management strategy in compressing to most UNIX systems.
To conclude, we hope that the result of our work is aims to provide the further system developers and users with more reach and reliable set of functionality feature.