Operating Systems Design with NachOS

Mohammed Almarakby, Eslam Mohammed, Sofiane Tanji, Arthur Chardon

Master Of Science In Informatics In Grenoble MOSIG
Master 1 Group 1

January 2020

1 Kernel Features

NachOS is an operating system (OS) written in c++ that runs MIPS based workstations, the OS accepts user programs and cross-compiles them to MIPS instruction set. Our implementation extends the functionalities of NachOS with the following features

1. I/O : A thread safe console implementation to interact with the OS
2. Multi-threading: Implementation for User-level threads for multi-threaded tasks with synchronization primitives.
3. Multi-Processing: Implementation for User-level processes for spawning and executing binaries.
4. File systems: Implementation for multiple file & directory management operations, ability to create, open, write and close several files, thread safe access to the files, ability to make, change and delete directories.
5. Networking: Implementation of critical section handler MUTEX locks and conditional variable synchronization primitive. Also, a reliable sending and receiving protocols were implemented for communication between NachOS machines.

2 User Program Specification

Here we describe the functionalities available for user programs.

2.1 PutChar

2.1.1 Name

PutChar - System call for writing a character on the shell.

2.1.2 Synopsis

```
#include "syscall.h"
void PutChar(char c);
```

2.1.3 Description

Upon calling PutChar system call, a trap for kernel is triggered and an exception(SC_PutChar) is raised that receives the character via register 4, then by calling the SynchPutChar through the synchconsole object we print the requested character.

2.1.4 Return Value

This call does not return any value.

2.1.5 Errors

This call does not return any error value.

2.2 GetChar

2.2.1 Name

GetChar - System call for scanning a character from the shell.

2.2.2 Synopsis

```
#include "syscall.h"
char GetChar(char c);
```

2.2.3 Description

When calling GetChar, an exception(SC_GetChar) is handled by reading the desired character from the shell and then writing it to NachOS memory through register 2. Of course all the casting necessary for adapting both types has been considered.

2.2.4 Return Value

The function returns the character requested.

2.2.5 Errors

This function does not return any error values.

2.3 PutString

2.3.1 Name

PutString - System call for writing a string on the shell.

2.3.2 Synopsis

```
#include "syscall.h"
void PutString(char c);
```

2.3.3 Description

The challenge in writing a complete string on the shell is that how could we transfer this string from NachOS emulator to Linux's shell. We chose to follow a character by character reading such that on calling the PutString system call we invoke the copyStringFromMachine routine with a fixed size no more than 255 bytes.

2.3.4 Return Value

This call does not return any value.

2.3.5 Errors

This call does not return any error value.

2.4 GetString

2.4.1 Name

GetString - System call for reading a string from the shell.

2.4.2 Synopsis

```
#include "syscall.h"
void GetString(char *s, int n);
```

2.4.3 Description

Contrary to PutString, this function reads the string from the main memory and fetch it back to MIPS environment through SynchGetChar for no more than n characters.

2.4.4 Return Value

This call does not return any value.

2.4.5 Errors

This call does not return any error value.

2.5 PutInt

2.5.1 Name

PutInt - System call for writing an integer value on the shell.

2.5.2 Synopsis

```
#include "syscall.h"
void PutInt(int n);
```

2.5.3 Description

In order to print an integer we used snprintf() inside the system call.

2.5.4 Return Value

This function does not return any value.

2.5.5 Errors

This function does not return any error value.

2.6 GetInt

2.6.1 Name

GetInt - System call for reading an integer value from the shell.

2.6.2 Synopsis

```
#include "syscall.h"
void GetInt(int* n);
```

2.6.3 Description

The call for this function reads a positive integer from the register using the GetChar system call itself.

2.6.4 Return Value

This function does not return any value.

2.6.5 Errors

This function does not return any error value.

2.7 UserThreadCreate

2.7.1 Name

UserThreadCreate - System call to create a new user thread.

2.7.2 Synopsis

```
#include "syscall.h"
int UserthreadCreate(void f(void *args),void *args);
```

2.7.3 Description

The UserThreadCreate function requests a new thread in the calling process, the new thread is instantiated by invoking do_UserThreadCreate. UserThread-Create takes 2 arguments, *int f* which is the routine to be invoked by the thread requested and *int arg* is the argument that is needed by the routine *f*.

2.7.4 Return Value

On success UserThreadCreate returns thread ID, on error it return -1;

2.7.5 Errors

USerThreadCreate reutrns -1 if there are no available memory in the address space stack for a new thread.

2.8 UserThreadExit *Automatic*

2.8.1 Name

UserThreadExit - System call to exit a user thread on finishing the invoked routine.

2.8.2 Synopsis

```
#include "syscall.h"
void UserThreadExit();
```

2.8.3 Description

UserThreadExit is called to destroy the user thread created and release its resources after finishing the routine run by the thread. This routine is invoked automatically after the thread finishes execution.

2.8.4 Return Value

This function does not return to the caller.

2.8.5 Errors

This function does not produce errors.

2.9 UserThreadJoin

2.9.1 Name

UserThreadJoin - System call invoked by a thread to wait the finishing of execution of another desired thread.

2.9.2 Synopsis

```
#include "syscall.h"
void UserThreadJoin(int tid);
```

2.9.3 Description

UserThreadJoin is called when there is a dependency in execution between created threads. The calling thread execution is paused until the waited for thread is finished. We follow the implementation of POSIX threads where a thread can not be waited on by more than one other thread, otherwise an undefined behaviour occurs.

2.9.4 Return Value

This function does not return to the caller.

2.9.5 Errors

If the requested thread to wait on is not existing “whether not created or finished execution and already destroyed” it will raise an error for non existing parameter.

2.10 ForkExec

2.10.1 Name

ForkExec - System call invoked by a thread to spawn a new process executing the attached binary.

2.10.2 Synopsis

```
#include "syscall.h"
int ForkExec(char* s);
```

2.10.3 Description

ForkExec creates a new process that is totally independent of the the calling process. the system initiates a process by creating a new kernel thread that will drive the execution of the main program, and then creates a new separate address space and attach it to the thread. The created process does not inherit any kind of data from the calling process.

2.10.4 Return Value

This function does not return to the caller.

2.10.5 Errors

ForkExec returns two errors, the first one if there is no available space in the mainMemory for a new process address space to reside,and the second one if the executable file to be launched by the process is not able to load or corrupted.

2.11 ProcessTerminate

2.11.1 Name

ProcessTerminate - System call invoked at the end of every process spawned.

2.11.2 Synopsis

```
#include "syscall.h"
void ProcessTerminate();
```

2.11.3 Description

ProcessTerminate releases the calling process resources “address space, terminates the main kernel thread”, and signal the end of the program if the calling process is the last running one.

2.11.4 Return Value

This function does not return to the caller.

2.11.5 Errors

This function does not return any error value.

2.12 Sbrk

2.12.1 Name

Sbrk - System call invoked by a thread to extend the brk pointer.

2.12.2 Synopsis

```
#include "syscall.h"
int Sbrk(unsigned int n);
```

2.12.3 Description

When Sbrk is called, it extends data segment in the process address space by increasing the Brk pointer, to allocate new memory block for the heap segment. This is used when the user requests a dynamic memory area during the run time of the program.

2.12.4 Return Value

If there is enough memory to handle to the user’s request, Sbrk returns a pointer to the new allocated block of memory, If there is no enough memory, it returns -1.

2.12.5 Errors

Sbrk fails if there is no sufficient memory to handle the request, it returns -1;

2.13 UserSemaphoreCreate

2.13.1 Name

UserSemaphoreCreate - System call to create a user level semaphore.

2.13.2 Synopsis

```
#include "syscall.h"
sem_t UserSemaphoreCreate(char *name,int value);
```

2.13.3 Description

Reserves a semaphore from the list of semaphores available for a process, to allow the execution threads to have synchronization between there operation, initializes it with the passed value, this value is not allowed to go below zero.

2.13.4 Return Value

Returns the Id of the semaphore.

2.13.5 Errors

Returns -1 if the semaphore creation failed.

2.14 UserSemaphoreDestroy

2.14.1 Name

UserSemaphoreDestroy - System call to release a user level semaphore.

2.14.2 Synopsis

```
#include "syscall.h"
void UserSemaphoreDestroy(sem_t UserSemaphore);
```

2.14.3 Description

Releases a semaphore that is utilized from the list of semaphores allowed for a process. This operation is also performed automatic at the end of the process execution to release all the un-used semaphores.

2.14.4 Return Value

Returns back the status of the operation, if the semaphore is not available "not created from the first place", the operation and returns false, otherwise it returns true.

2.14.5 Errors

Returns False if the semaphore is not created.

2.15 UserSemaphoreProberen

2.15.1 Name

UserSemaphoreProberen - System call to perform wait operation on a user level semaphore.

2.15.2 Synopsis

```
#include "syscall.h"
void UserSemaphoreProberen(sem_t UserSemaphore);
```

2.15.3 Description

Performs the usual semaphore waiting.

2.15.4 Return Value

This function does not return to the caller.

2.15.5 Errors

Returns if the semaphore is not created.

2.16 UserSemaphoreVerhogen

2.16.1 Name

UserSemaphoreVerhogen - System Call to perform post operation on a user level semaphore.

2.16.2 Synopsis

```
#include "syscall.h"
void UserSemaphoreVerhogen(sem_t UserSemaphore);
```

2.16.3 Description

Performs the usual semaphore posting.

2.16.4 Return Value

This function does not return to the caller.

2.16.5 Errors

Returns if the semaphore is not created.

2.17 Create

2.17.1 Name

Create - user-level function to create NachOS files.

2.17.2 Synopsis

```
bool Create(const char *name, int initialSize);
```

2.17.3 Description

Similarly to UNIX 'create', it creates a file in the NachOS filesystem. This file has fixed size (initialSize).

2.17.4 Return Value

Return true if everything goes ok, otherwise, return false.

2.17.5 Errors

Returns false if a file already exists in the directory or if there's no free space.

2.18 Remove

2.18.1 Name

Remove - user-level function to remove a file from the NachOS filesystem.

2.18.2 Synopsis

```
bool FileSystem::Remove(const char *name, unsigned int tid);
```

2.18.3 Description

Similarly to UNIX 'unlink', it deletes the file "name" in the NachOS filesystem. This is done

2.18.4 Return Value

Return TRUE if the file was deleted, FALSE if the file wasn't in the file system.

2.18.5 Errors

Returns false only if the file can't be find in the current directory.

2.19 List

2.19.1 Name

List - user-level function to list a NachOS directory.

2.19.2 Synopsis

```
void List();
```

2.19.3 Description

Similarly to UNIX 'ls', it prints all the directories and file in the current directory (always including . and ..)

2.19.4 Return Value

This function does not return to the caller.

2.19.5 Errors

This function does not produce errors.

2.20 ChangeDirectory

2.20.1 Name

ChangeDirectory - user-level function to change working directory

2.20.2 Synopsis

```
bool ChangeDirectory(const char *name, int threadId);
```

2.20.3 Description

Similarly to UNIX 'cd', it changes the current working directory to the directory specified by 'name'. 'name' must be a directory inside the current working directory.

2.20.4 Return Value

Returns true if it succeeded in changing the current working directory, false otherwise.

2.20.5 Errors

Returns false only if the directory can't be find in the current directory.

2.21 RemoveDirectory

2.21.1 Name

RemoveDirectory - user-level function to remove a NachOS directory.

2.21.2 Synopsis

```
bool RemoveDirectory(const char *name, int threadId);
```

2.21.3 Description

Similarly to UNIX 'rm -rf', it deletes an empty NachOS directory.

2.21.4 Return Value

Returns true if it succeeded in removing the specified working directory, false otherwise.

2.21.5 Errors

Returns false if the directory to be removed can't be find or if it is not empty.

2.22 MakeDirectory

2.22.1 Name

MakeDirectory - user-level function to create a NachOS directory.

2.22.2 Synopsis

```
bool MakeDirectory(const char *name, int threadId);
```

2.22.3 Description

Similarly to UNIX 'mkdir', it creates an empty NachOS directory.

2.22.4 Return Value

Returns true if it succeeded in making the specified directory, false otherwise.

2.22.5 Errors

Returns false if the directory to be created is a basic directory ('/', '.' or '..'), already exists, if there is not enough space for the file header or in directory or on disk or if there are already more than 10 directories created.

3 User Tests

3.1 I/O:

1-getchar,getint,getstring,putchar,putstring: Testing several functionalities to interact with the console

3.2 Multi-threading:

1. makethreads: Creates multiple threads and perform I/O and arithmetic to test basic user thread functionalities.

2. makethreads_testJoin: Creates multiple threads and calls thread join on different threads.

3. severalthreadstest: Create a large number of threads to test how the system will handle a large number of threads request, every thread performs I/O and arithmetic operations.

4. nestedThreadJoinTest: create a thread, and inside the thread we create another thread, and we wait for both threads using join,here we test nested creations and the ability to differentiate between threads when using join.

5. usesemaphoretest: create multiple threads and assign to all the same routine, and we use a lock to enclose this routine operations using a semaphore, here we test the user level semaphore functionality

6. Producerconsumertest: The usual producer consumer routine to test the semaphore functionality for a thread safe buffer.

3.3 Multi-processing:

1. testforkexec: All in one test for a multi-processing environment. we initiate multiple processes using ForkExec "the number of processes is dependent on the mainMemory size", each process executes a program that creates multiples threads, each thread invokes a routine that performs I/O and arithmetic operations, and is enclosed by semaphore for synchronization, this test utilizes most of the main functionalities in the third and fourth stage.

2. sbrktest: A test to for the Sbrk functionality, extending the program break by shifting brk and allocating new empty frames.

3. testdynamicmem: A test for dynamic memory allocation from the heap, we use sbrk to initiate a heap free memory block, and then we follow that we a series of mem_alloc and free operations to test the capacity of the allocated heap block.

3.4 File System:

1. filesysTest : A routine creating/removing directories and creating/opening/removing files.

3.5 Networking:

1. MailTest_10msgs: A routine for sending and receiving data between 2 machines 10 times.

2. MailTest_ring: A ring topology test for sending and receiving data between n machines sequentially.

4 Implementation

4.1 I/O:

All of the I/O operations “for e.g. SynchGetChar & SynchPutChar” are done through the class *SynchConsole*, which encloses all of the console class operations with semaphores that are used as locks, this ensures that the console operation are thread safe and thread interleaving produced by scheduling won’t affect the displayed or the acquired characters/integers/strings.

4.2 Multi-Threading:

In this part we discuss the implementation choices for the user-level threads. We use 1:1 model to map every user-level thread to kernel-level thread, we assign to every thread memory in the stack worth of two pages of memory, we manage the user stack using the bitmap instance to assign the thread a place in the stack. we do not re-use the ids for the threads, since it produced a problem with the thread join functionality. the initialized threads are tracked using an array with a size pre-initialized to MAX_THREAD_NUMBER_ALLOWED, this array allows us to track the initialized threads and facilitates further action like thread exit and thread join. To implement thread join, we attach a semaphore with every thread created, when a thread requests a join on another one, it waits on this semaphore, and only when the waited for thread finishes execution semaphore is posted on. In this version of the user thread implementation, we do not protect the threads from writing to each others stack part, but in future development, we can achieve that by adding a *stack fence post* value at the end of stack area reserved for the stack for each thread, and at the end of the execution of the thread we can test if the thread passed this fence or not.

To perform automatic termination of user threads, we passed the name of the routine with the other arguments to the method *THREAD::Fork()*, and we write the function name to the *retadr* register, so that when the thread finishes execution, it invokes thread exit and releases its resources.

If the execution of the main thread is faster than the others, we prevent the main thread from executing *Halt()* before the other threads finish, we block the main thread by a waiting on a semaphore on the first user thread created and we only post on it when all the threads are released.

To handle a large number of thread creation requests, we had to choose between two options, the first one is to stop the program from execution if it reaches a certain limit of threads, or if there is no available place in the user stack in the address space. The second one “and the one we chose” is to block the main thread from creating new threads, until there is a place available in the user stack for a new thread. By this design instead of limiting the user to some number of threads per process, we are giving the user the freedom to choose the number of threads but at the expense of the speed of the program execution, therefore, the user needs to be conscious about the thread creation number in his program. And of course if our implementation is used in a thread intensive environment “OS hosting a server”, we can have more memory to expand the user stack size “more than 2 page of 128 *bytes* each”.

For the User level semaphores, we choose to assign a certain number semaphores for each address space, defined by MAX_SEMAPHORES_ALLOWED, we track these semaphores by using a bitmap, when a user requests a semaphore via a system call, we check in the bitmap if there is a free semaphore to be used. The rest of the implementation is just a direct mapping to the P and V operations. The only difference is that we inhibit releasing a semaphore if it is used by threads or if its value is different than the initialization value, this protects the program from waiting without posting and an inevitable segmentation fault.

4.3 Multi-processing

For extending nachos to create and handle multiple processes, we implemented a partially linux similar fork and exec, the parent process spawns a child process by creating a kernel thread that drives the process creation and code execution. We do not re use the process ids, we create a new address space and attach it to the new kernel thread, the new address space reads and write in the virtual memory using *AddrSpace::ReadAtVirtual*.

We implement the class *FrameProvider* to handle the allocation of main memory frames, for each frame allocated we delete its content entirely and we have two options for allocation, either to allocate frames in order or to allocate the frames randomly, all the *FrameProvider* operations are protected by lock to ensure thread access safety.

For extending the heap for dynamic memory allocation by the user, we updated the address space implementation to allocate pages in the page table reserved for the dynamic allocation but we do not allocate counter-part page frames and we set the valid bit to false. When the user program requests a chunk of memory during runtime using the sbrk syscall,it allocates the frames and returns a pointer to the start of the block, and increases the brk pointer to point to the new program break. Every process has a predefined number of heap pages defined in *MAX_BRK_PER_ADDRSPACE* as 100 frames of memory, The Sbrk syscall is responsible for checking for this limit and requesting the page instead of *AllocPageFrame*.

Before a process terminates, we first check if there is a working user thread and let it finish, and if the process calling the terminate is the last process, we invoke *Halt()* to terminate the program. A process can not terminate before the spawned children processes exit, this in not handled in the case of nested processes with multiple nesting levels, but this is assured by the main process, since we track the number of working processes, we only can *Halt()* if it is the last process as mentioned.

4.4 File System

This part is dedicated to emphasize the implementation details of the filesystem. So, besides the original functions the user was able to do such as listing, reading or removing files, we implemented the functions to create a new directory, to remove one or to navigate between directories. This can be achieved respectively by the commands *-mkdir*, *-rm*, *-cd*.

Directory hierarchy : We first distinguished directories and files so that we could handle directories properly. We added the two directories (. and ..)

(current and parent) and specified the root as being both. Then we implemented the functions ChangeDirectory, RemoveDirectory and MakeDirectory as user-level functions to browse the directory hierarchy.
Opened files and concurrency : We built two tables for open files, one for the whole filesystem and one per thread. We changed RemoveDirectory so as to return false in case the directory is opened by a thread (for this, we use the PerThreadTable). We didn't have to for the Remove function because files can be opened by one thread at a time only.
Beware, path names are not implemented yet so you need to switch from a directory to another step by step.

4.5 Networking

For the first step in the Networking, we made all the changes in the file *nettest.cc* such that we re-implemented the function *MailTest* with the required 2 flavors; *MailTest_10msgs* and *MailTest_ring*. First, a simple *for* loop that sends the same message with their acknowledgments 10 times between 2 machines. The second one depicts the *ring* network between *n* machines by checking the *farAddr* of the machine if it i the first one in the network or not otherwise it sends/receives messages and acknowledgments.
Regarding the *reliable* transfer step between the machines, we figured out how to structure the delieverables by creating a *history* list that holds all the messages sent from the *postOffice* followed by the semaphore *notReceived* that waits for the message inside the *Get* method and signals when the acknowledgment is received. We implemented 3 main procedures:

- *TimeOutHandler*: an interrupt handler that resend the message again after *TEMPO* amount of timer ticks.
- *reliableSend*: a wrapper for *Send()* that checks first if the *history* is empty or not and accordingly it will be added if this is the first time to be sent or increment the number of trials this mail has been sent (of course after checking the number of maximum trials *MAXREEMISSIONS*). After sending the mail, the function schedules an interrupt that will execute the aforementioned handler and lastly, waits for an acknowledgment from the receiver machine.
- *reliableReceive*: guarantee the reliability of receiving an incoming message form a sender machine by sending an acknowledgment using the sender data.

Two mandatory changes in the supported version of *Mail* were applied, the *numTrials* which describes the number of trials for this message to be sent and the *carry* that helps the receiver to accept variable size messages. When *carry* is 0 means that this mail is completely sent as one package otherwise means the left or remaining chunk of bytes to be sent again. On the sender side, we would have implemented a few logical conditions that organize the flow of sending (either send this chunk completely or if it is larger than the *MaxMailSize* then send the maximum affordable amount of bytes and update the *carry* variable of this mail). The receiver would first check on the *carry* value to decide whether this message will be concatenated into a pre-received data or not.

5 Organization

For the organization of the work, we decided to split the steps and work in pairs in order to work efficiently and deliver faster. Although we faced multiple problems in the organization from a team member, we managed to finish -with a slow progress- most of the required main functionalities in different steps. We used Git and Gitlab which facilitated the management and the merging of different parts of the projects.

5.1 Step 1:

For the first step, all of the team members engaged in the codebase in order to get acquainted with the project and the requirements. After that we tried to divide the work according to the preferences for each member such that everyone works in a step he is comfortable with.

5.2 Step 2:

For the first part, all of the team dived in directly in the inner workings of the system calls and tried to finish the first part, after that Sofiane and Eslam continued the rest of this Step and the rest of the team jumped to step 3.

5.3 Step 3:

Mohammed and Arthur started to work on the multi-threading step, After multiple days Mohammed was able to finish most of the requirements, with no collaboration from Arthur's side. So after a meeting with professor Vincent we decided to split the team differently and have Mohammed and Eslam working on what is left from stage 3 and initiate stage 4 and the bonuses and Sofiane and Arthur jump directly to Stage 5 to gain more time.

5.4 Step 4:

After finishing Stage 3, Mohammed and Eslam Started to Work on Stage 4, It took more than the assigned time but they were able to finish until the Sbrk part, they tried to merge the implementation of the dynamic memory allocator but it caused large number of bugs and the team was behind schedule at this point because the file system was not completed yet.

5.5 Step 5:

Sofiane and Arthur started Stage 5, after a week Sofiane made progress with the implementation in the step, but the team figured out that Arthur was not able to advance at all in the part assigned to him without notifying anyone, so with one week left before the defense, the team decided that Sofiane will continue in this step since he was already acquainted with file systems and his implementations, While Mohammed and Eslam jump to Step 6 when they finish step 4.

5.6 Step 6:

After finishing step 4, Eslam and Mohammed Started to work in Step 6, but they weren't able to advance after the fixed size reliable send and recieve part as a result of what happened in step 5, So they Started to finalize the parts that were done and work on the documentation and the presentation.