# NachOS Report

BODART Maxime, CIVADE Thomas,
DETROYAT Alexis and GROUSSON Dylan
M1 MoSIG

January 23, 2025

## Contents

## 1 Global Description

Welcome to NonOS, our custom NachOS build! NonOS is a multithreaded, multiprocess operating system that supports files up to 122 KB, based on the ext2 file system. It includes virtual memory implemented through simple redirection, networking capabilities with a pre-integrated reliable FTP protocol and process migration feature. Provided with a memory allocator as well as a shell to manipulate the file system and run your scripts, NonOS is the perfect operating system to run light applications.

## 2 User System Call Specifications

### 2.1 Proper Stop

- **Halt** - halting the machine:
  Signature: *`void Halt()`*
  Description: Halt and cleanup the whole machine (interrupting all processes and threads).
  Return value: none

- **Exit** - exiting the current thread:
  Signature: *`void Exit(int exitCode)`*
  Description: Exit properly the current thread with the *`exitCode`* value (exiting the whole process if this is the last process' thread).
  Return value: none

### 2.2 Input/Output

- **PutChar** - display a character in the console:
  Signature: *`void PutChar(char c)`*
  Description: Display the *`c`* character in the console.
  Return value: none

- **PutString** - display a string in the console:
  Signature: *`void PutString(char *c, int size)`*
  Description: Display the *`size`* first *`c`* string characters in the console. The displayed string size is capped at our intern limit (see the *`MAX_STRING_SIZE`* macro), above which the string is truncated.
  Return value: none

- **GetChar** - get a character from the console:
  Signature: *`int GetChar()`*
  Description: Return a character from the console in its proper ASCII format.
  Return value: the obtained character from the console (the ASCII value)

- **GetString** - get a string from the console:
  Signature: *`void GetString(char *s, int n)`*
  Description: According to the *`fgets()`* behavior, *`GetString`* store a maximum *`n`* long character string from the console in *`s`*.
  Return value: none

- **PutInt** - display an int in the console:
  Signature: *`void PutInt(int n)`*
  Description: Display *`n`* in the console.
  Return value: none

- **GetInt** - get an int from the console:
  Signature: *`void GetInt(int *n)`*
  Description: Store an int from the console into *`n`*.
  Return value: none

## 2.3 Threads

- **ThreadCreate** - create a new user thread:
  Signature: *`int ThreadCreate(void f(void *arg), void *arg)`*
  Description: Create a new user thread and makes it run the routine *`f`* with the argument *`arg`*.
  Return value: the ID of the new thread (positive integer) or -1 if the creation failed, which can be caused by a lack of memory in the address space

- **ThreadExit** - terminates a user thread:
  Signature: *`void ThreadExit()`*
  Description: Terminate a user thread by clearing its data entries, decrementing the number of threads and ending its execution.
  Return value: none

- **ThreadJoin** - waits for a thread to terminate:
  Signature: *`void ThreadJoin(tid threadId)`*
  Description: Block the calling thread until the thread *`threadId`* terminates.
  Return value: none

- **SemInit** - initialize a semaphoree:
  Signature: *`void SemInit(sem_t *sem, unsigned int v))`*
  Description: Allocate a new semaphore with value *`v`*.
  Return value: none, but set a semaphore identifier in the *`sem`* variable

- **SemPost** - increment semaphore value:
  Signature: *`void SemPost(sem_t *sem)`*
  Description: Increment the *`sem`* value. Doesn't do anything if the *`sem`* isn't initialized.
  Return value: none

- **SemWait** - decrement semaphore value:
  Signature: *`void SemWait(sem_t *sem)`*
  Description: Set the calling thread to sleep until the *`sem`* value is superior to 0, then decrement it. Doesn't do anything if the *`sem`* isn't initialized.
  Return value: none

- **SemDestroy** - de-allocate a semaphore:
  Signature: *`void SemDestroy(sem_t *sem)`*
  Description: De-allocate the given *`sem`* semaphore.
  Return value: none

## 2.4 Processes and Virtual Memory

- **ForkExec** - create a new process:
  Signature: *`pid_t ForkExec(char *s)`*
  Description: Create a new process running the executable *`s`*.
  Return value: pid of the new process

- **ProcessJoin** - join a process:
  Signature: *void ProcessJoin(pid_t processId)*
  Description: Block the calling thread until the process *processId* terminates.
  Return value: none

- **Sbrk** - allocate new frames:
  Signature: *void *Sbrk(unsigned int n)*
  Description: Allocate *n* new frames for the calling process.
  Return value: the starting point of the new allocated space, 0 if the allocation failed

## 2.5 File System

- **Create** - create a file in the current directory:
  Signature: *int Create(char *name)*
  Description: Create an empty file named *name* in the current directory.
  Return value: 1 if the creation succeeded and 0 otherwise

- **Remove** - remove a file in the current directory:
  Signature: *int Remove(char *name)*
  Description: Remove the file named *name* in the current directory.
  Return value: 1 if the remove operation succeeded and 0 otherwise

- **Open** - open a current directory file in write and read:
  Signature: *int Open(char *name)*
  Description: Open the file named *name* in the current directory.
  Return value: the file descriptor if the open operation succeeded and -1 otherwise

- **Close** - close an opened file:
  Signature: *int Close(int fd)*
  Description: Close the open file descriptor *fd*.
  Return value: 0 if the close operation succeeded and -1 otherwise

- **Write** - write in an opened file descriptor:
  Signature: *int Write(char *buffer, int size, int fd)*
  Description: Write at most *size* characters from the *buffer* string in the open file descriptor *fd*.
  Return value: the written size (in bytes) if the write operation succeeded and -1 otherwise

- **Read** - read in an opened file descriptor:
  Signature: *int Read(char *buffer, int size, int fd)*
  Description: Read at most *size* characters in the open file descriptor *fd* and store it in the *buffer* string.
  Return value: the read size (in bytes) if the read operation succeeded and -1 otherwise

- **Seek** - seek in an opened file descriptor:
  Signature: *void Seek(int fd, int offset)*
  Description: Seek at the *offset* position starting at the current position in the open *fd* file descriptor.
  Return value: none

- **Mkdir** - create a directory in the current directory:
  Signature: *int Mkdir(char *s)*
  Description: Create an empty directory named *s* in the current directory.
  Return value: *1* if the creation succeeded and *0* otherwise

- **Rmdir** - remove a directory in the current directory:
  Signature: *int Rmdir(char *s)*
  Description: Remove the directory named *s* in the current directory, if it is empty.
  Return value: *1* if the remove operation succeeded and *0* otherwise

- **Listfiles** - list all the files in the current directory:
  Signature: *void Listfiles()*
  Description: List all the files in the current directory.
  Return value: none

- **Changedir** - change the current directory:
  Signature: *int Changedir(char *s)*
  Description: Change the current directory to the directory at the end of the path *s*.
  Return value: none

## 2.6 Network

- **StartFTPServer** - start an FTP server:
  Signature: *void StartFTPServer()*
  Description: Block the current thread as an FTP server until the machine *Halt*.
  Return value: none

- **SendFile** - send a file to an FTP server:
  Signature: *int SendFile(int farAddr, char *filename)*
  Description: Send the file *filename* to the machine *farAddr*.
  Return value: *0* if the sending operation failed, *1* otherwise

- **ReceiveFile** - receive a file from an FTP server:
  Signature: *int ReceiveFile(int farAddr, char *filename)*
  Description: Receive the file *filename* from the machine *farAddr*.
  Return value: *0* if the receiving operation failed, *1* otherwise

- **SendProcess** - migrate the current process to another machine:
  Signature: *int SendProcess(int farAddr, int stopAfter)*
  Description: Duplicate and send the current address space to the machine *farAddr*. If *stopAfter != 0*: stop the calling thread once the process is sent.
  Return value: *-1* if the migration operation failed, *0* for the sender (if *stopAfter == 0*), *1* for the receiver

- **ListenProcess** - set the current thread as a listener of process:
  Signature: *tid_t ListenProcess()*
  Description: Receive and create a new process.
  Return value: *tid* of the new process

### 2.7 User-Level Library

- **mem_init** - initialize the Heap:
  Signature: *void mem_init(size_t size)*
  Description: Initialize the heap by allocating *size* bytes.
  Return value: none

- **mem_alloc** - allocate a memory block:
  Signature: *void *mem_alloc(size_t size)*
  Description: Allocate a memory block of *size* bytes.
  Return value: A pointer to the allocated block in the heap, *NULL* if we cannot allocate a memory block

- **mem_free** - free a memory block:
  Signature: *void mem_free(void *ptr)*
  Description: Free the memory blocks that the pointer *ptr* points to.
  Return value: none

## 3 User Test Programs

### 3.1 Proper Stop

The following tests are in *test/Returns* directory.

- *exit.c* - call the syscall *Exit()* with the value 2:
  Description: Check if the syscall *Exit()* exits with the right value *2*. Should end improperly with an exit code *2*.
  Usages: *./nachos-final -f -cp Returns/exit exit -x exit*

- *exitautomatic.c* - exit the function automatically without calling any *return* or *Exit()*:
  Description: Check if we can return from a function without explicitly invoking its termination. Should end properly.
  Usages: *./nachos-final -f -cp Returns/exitautomatic exitautomatic -x exitautomatic*

- *return0.c* - exit the function by calling *return* with the value 0:
  Description: Check if we can return a value from a function and exit the function. Should end properly.
  Usages: *./nachos-final -f -cp Returns/return0 return0 -x return0*

- *returnn.c* - exit the function by calling *return* with the value -1:
  Description: Check if we can return from a function without explicitly invoking its termination. Should end improperly with an exit code -1.
  Usages: *./nachos-final -f -cp Returns/returnn returnn -x returnn*

### 3.2 Input/Output

The following tests are in *test/InputOutput* directory.

- *getchar.c* / *getint.c* / *getstring.c* - get a character / integer / string (infinitely) from the user and print it on the console:
  Description: Check if the syscall *GetChar()* / *GetInt()* / *GetString()* returns the right character / integer / string typed by the user.
  Usages: *./nachos-final -f -cp InputOutput/getchar getchar -x getchar* (same for *getint* and *getstring*).

- *putchar.c* - displays characters to the console:
  Description: Check if the syscall *PutChar(char)* displays the right character, should display *abcd.*
  Usages: *./nachos-final -f -cp InputOutput/putchar putchar -x putchar*

- *putint.c* - displays three integers to the console:
  Description: Check if the syscall *PutInt(int)* displays the right integer from small to large values.
  Usages: *./nachos-final -f -cp InputOutput/putint putint -x putint*

- *putstring.c* - displays four strings to the console:
  Description: Check if the syscall *PutString(str, size)* displays the string according to the specification of the function for a size *>*, *<*, or *==* to *len(str)* with a *const* char or not.
  Usages: *./nachos-final -f -cp InputOutput/putstring putstring -x putstring*

## 3.3 Threads

The following tests are in *test/Threads* directory.

- *threads.c* - create a lot of threads, and threads create subthreads:
  Description: Check if we can create and run properly each thread and if the errors are well handled when we create too many threads.
  Usages: *./nachos-final -rs 0 -f -cp Threads/threads threads -x threads*

- *threadsexit.c* - create a thread and Halt the machine in the thread:
  Description: Check if the syscall *Halt()* shutdowns the whole machine even inside threads.
  Usages: *./nachos-final -rs 0 -f -cp Threads/threadsexit threadsexit -x threadsexit*

- *threadsjoin.c* - Create threads sequentially by waiting for them:
  Description: Check if the syscall *ThreadJoin()* waits for the threads, should create 20 threads sequentially, and ensure the previous thread has ended before starting the next one.
  Usages: *./nachos-final -rs 0 -f -cp Threads/threadsjoin threadsjoin -x threadsjoin*

- *threadsputchar* - create 2 threads that write to the console:
  Description: Check if the syscall *PutString()* is exclusive between threads.
  Usages: *./nachos-final -rs 0 -f -cp Threads/threadsputchar threadsputchar -x threadsputchar*

- *incr.c* - create 2 threads that increment an integer:
  Description: Check if the semaphores syscalls respect mutual exclusion, should return *10000*.
  Usages: *./nachos-final -rs 0 -f -cp Threads/incr incr -x incr*

- *producerconsumer.c* - implement the concurrent programming producer-consumer
  Description: Check if the semaphores syscalls are well done, should produce and consume in the same order.
  Usages: *./nachos-final -rs 0 -f -cp Threads/producerconsumer producerconsumer -x producerconsumer*

## 3.4 Virtual Memory and Processes

The following tests are in `test/VM` directory.

- `exec.c` - launch 5 processes that execute several times the test `incr.c`
  Description: Check if the syscall `ForkExec()` creates and launches the new process well, should print 5 times the result of incr (`10000`).
  Usages: `./nachos-final -rs 0 -f -cp Threads/incr incr -cp VM/exec exec -x exec`

- `stress.c` - launch 20 times 20 processes that execute the test `incr.c`
  Description: Stress the performance to test if the process structures are well freed.
  Usages: `./nachos-final -f -m 0 -rs 0 -cp Threads/incr incr -cp VM/stress stress -x stress`

- `threadexitprocesses.c` - launch a program and Halt
  Description: Check if the `Halt()` in a subprocess shutdowns everything, even the main process. Shouldn't print "`Hello, World!`" at the end.
  Usages: `./nachos-final -rs 0 -f -cp Threads/threadsexit Threads/threadsexit -cp VM/threadexitprocesses threadexitprocesses -x threadexitprocesses`

- `tinyshell.c` - launch a tinyshell that can execute the asked program
  Description: Check if we can infinitely create a process, by waiting for its end with the `ProcessJoin()` syscall.
  Usages: `./nachos-final -rs 0 -f -cp [program] [name] -cp VM/tinyshell tinyshell -x tinyshell`

- `alloc.c` - test the memory allocator
  Description: Check if the memory allocation is properly freed and the block is reused. Must display 3 - 3 because the first pointer has been freed.
  Usages: `./nachos-final -f -m 0 -rs 0 -cp VM/alloc alloc -x alloc`

## 3.5 File System

The following tests are in `test/FileSys` directory.

- `CreateSim.c / MkdirSim.c / OpenSim.c / WriteSim.c` - create / mkdir / open / write a/in a file simultaneously with several threads
  Description: Check the concurrency issues of the syscalls `Create() / Mkdir() / Open() / Write()` to make sure they do their jobs without having an inconsistency in the file system.
  Usages: `./nachos-final -rs 0 -f -cp FileSys/CreateSim CreateSim -x CreateSim` (replace `CreateSim` by the wanted file (`MkdirSim`, `OpenSim` or `WriteSim`))

- `test.c` - create and write a message and read the message of the created file
  Description: Check if the syscall `Seek()` works by seeking at the start of the message written, and if the content of the file is coherent. Should print "`Hello, World!`".
  Usages: `./nachos-final -rs 0 -f -cp FileSys/test test -x test`

- `WriteCloseSim.c` - open a file and create two threads, one of which closes and the other writes in a file
  Description: Check if the closing thread waits for the writer thread to be done, should not print anything if it works.
  Usages: `./nachos-final -rs 0 -f -cp FileSys/WriteCloseSim WriteCloseSim -x WriteCloseSim`

## 3.6 Network

The following tests are in `test/FTP` directory.

- `FTPlisten.c` - Start an FTP server:
  Description: Used to run other FTP tests.

- `testreceive.c` - Receive a file using FTP:
  Description: Check if an FTP client can receive a file. Should display that the transfer is a success and the content of the directory with file.txt.
  Usages:
  (server) `./nachos-final -f -disk SERVER -m 0 -rs 1 -cp FTP/FTPlisten FTPlisten -cp ../network/test/client1/file.txt file.txt -x FTPlisten [-l 0.8]`
  (client) `./nachos-final -f -m 1 -rs 1 -cp FTP/testreceive testreceive -x testreceive [-l 0.8]`

- `testsend.c` - Send a file using FTP:
  Description: Check if an FTP client can send a file. Should display the content of the server composed of its executable and file.txt.
  Usages:
  (server) `./nachos-final -f -disk SERVER -m 0 -rs 1 -cp FTP/FTPlisten FTPlisten -x FTPlisten [-l 0.8]`
  (client) `./nachos-final -f -m 1 -rs 1 -cp ../network/test/client1/file.txt file.txt -cp FTP/testsend testsend -x testsend [-l 0.8]`

The following tests are in `test/Migrate` directory. These are reliable, but because of the really high number of messages exchanged, this can take a lot of time (`l > 0.98`).

- `migratelisten.c` - Start a thread listening for a process migration:
  Description: Used to run other Migrate tests.
  Usages: `./nachos-final -f -m 0 -cp Migrate/migratelisten migratelisten -x migratelisten`

- `simplemigrate.c` - Send a process twice:
  Description: Test the `SendProcess` syscall, and termination or not of the calling thread. Should display 3 strings in the receiver, and only one in the sender.
  Usages: With `migratelisten.c` launched: `./nachos-final -f -m 1 -cp Migrate/simplemigrate simplemigrate -x simplemigrate`

- `threadsmigrate.c` - Send a multithreaded process:
  Description: Test the `SendProcess` syscall with multithreaded processes and user semaphores. Should produce and consume in the same order. The receiver should start depending on when the process has been sent.
  Usages: With `migratelisten.c` launched: `./nachos-final -f -m 1 -rs 3 -cp Migrate/threadsmigrate threadsmigrate -x threadsmigrate`

## 3.7 Global

The following test is in *test* directory.

- *shell.c* - launch a shell
  Description: Launch a shell that allows to test almost every syscall of our system. To know the commands that we can use, you can launch the command "*help*".
  Usages: *./nachos-final -m 1 -rs 0 -f [-cp <file> <name_file>] -cp shell shell -x shell*
  (for FTP feature) *./nachos-final -f -disk SERVER -m 0 -rs 1 -cp FTP/FTPlisten FTPlisten [-cp <file> <name_file>] -x FTPlisten*

# 4 Implementation Choices

## 4.1 Threads

There are two different thread data structures. On the one hand, the *thread_id* field in *thread_info_t **threadsInfos* is reusable and unique to each process throughout the whole system (declared in *system.h*). This is used to manage thread data in the system. On the other hand, the *thread_id* field in *thread_info_t **localThreadsInfos* is not reusable and local to each process (in *addrspace.h*). It is used to join threads at user level. We leave an empty slot of 2 bytes at the bottom of the stack (high addresses) and we allocate a slot of 2 pages for each thread stack in the virtual memory. Exiting a main thread only terminates the execution of the current process if no other thread is currently running : there is a parent-child relationship between the main thread and the other ones. A thread can only be joined by threads of the same process, and multiple threads can join a same thread.

## 4.2 Processes and Virtual Memory

We use reusable process IDs to handle processes. We wrote a memory allocation library at user level called *mem_alloc.c/.h*. Its initialization method *mem_init* calls the *sbrk* system call to allocate physical pages, then the user can call *mem_alloc* and *mem_free* to handle memory blocks. We use the "first fit" policy in order to allocate memory blocks in the heap. We use several synchronization primitives to avoid concurrency issues, including the provided *Semaphore* class, as well as the *Lock* and *Condition* classes that we implemented ourselves in *synch.cc*.

## 4.3 File System

For a way simpler navigation in the directory tree, we extended the file header content. In this way, we added a *type* field that represents which type of file it represents (*DATA_FILE*, *DIRECTORY*, *ROOT*) this is mainly in order to differentiate files and directories for the *Read / Write* operations.

In order to reach higher file sizes in the file system, it was mandatory to implement one level of undirected data block. This architecture partially follows the **ext2** file system with a multi-level (one level of indirection) index in an unbalanced tree. Indeed, the file header (corresponding to an inode in a classic file system) will give us access to *NumDirect - 1* direct data blocks and *1* undirected data block that contains references to other data sectors in the disk.

At the moment, a file header corresponds to `NumDirect` =

$$\frac{sectorSize - fileMetadata}{sectorAddressSize} = \frac{128 - 3 * 4}{4} = 29$$

data blocks, so with our indirection the file size can go up to

$$sectorSize * \left(NumDirect - 1 + \frac{NumDirect * sectorSize}{sectorAddressSize}\right) = 128 * \left(28 + \frac{29 * 128}{4}\right) = 122368$$

bytes.

For the users, open files are maintained in a proper table directly in the file system (global for the whole system) in order to limit the maximum amount of files that are open at the same time. Also, we protected the I/O operations in files by preventing file from being open again. Additionally, we maintain locks for each open file in order to protect these operations.

## 4.4 Network

To assure the reliability of the network system, every received message is automatically followed by an acknowledgement (`ACK`) notification that goes the other way around. After having sent a message, the sender waits on a condition variable and it will only be woken up in two cases : either an `ACK` notification is received, or too much time has passed, and the sender has to re-emit a message. This is handled by scheduled timer interruptions. We use message identifiers and acknowledgement identifiers, so we know which message ID the current machine should be waiting for. If the received ID is lower than expected, it means that the sender considered that the message got lost and re-emitted it, so we can ignore it. If the received ID is higher than expected, it most likely highlights a concurrency issue, because it means that several threads sent a message to the same mailbox.

We try to emulate the mechanism of UNIX sockets to establish connections : every machine has a default listening mailbox called `LISTEN_BOX` and bound to 0. When another machine tries to connect to this machine through this listening box, we allocate a dedicated communication mailbox that will then be used for data transmissions. Messages are signed with specific network types included in mail headers : `CONN` for connection attempts, `DATA` for data transmissions, or acknowledgments. When trying to disconnect, a machine will wait on a condition variable for `DISCONNECT_TEMPO` ticks, in case there are incoming re-emissions because of an `ACK` that got lost. If it hasn't received any new message in between, the machine considers it can safely close the connection on its end.

## 4.5 File Transfer Protocol

The file transfer protocol relies on a new layer, called FTPHeader, that we included under both the PacketHeader and MailHeader, that is, at the beginning of the data payload. This header includes specific types dedicated to FTP connections, communications and feedbacks which are specified in `FTPType`. The file transfer system is linked to the NachOS physical file system through a `FileHandler` class that contains static wrappers of file system methods, in order to handle errors. We designed an FTP client-server architecture : a server endlessly loops inside a listening method (`ServerRoutine`) to accept client connections, then creates a new thread for every new client attempting to connect. Client threads are then forked on a client handling method (`ClientHandler`) that receives their requests and treats them.

### 4.6 Process Migration

Only the main thread can perform a process migration, because of limitations due to the virtual memory implementation in NachOS. The sender machine copies the entire memory address space, as well as the semaphore table, threads info and registers. We can choose whether the calling (main) thread should stop running after the transfer or not, thanks to an additional argument inside the *`SendProcess`* system call. However, we still cannot stop the current thread if there are other threads currently running because of the parent-children relationship of our main threads. The thread copy requires synchronization mechanisms to make sure that no thread is currently trying to execute a system call during the process transfer, which would lead to disastrous bugs. Such mechanisms are directly implemented at user level through semaphores.

## 5 Project Organization

We tried to follow the suggested planning as much as we could. We spent the first 2 days (Monday 16/12 and Tuesday 17/12) of the project installing and getting used to NachOS, as well as implementing the basic I/O system calls (step 1 and 2). From Wednesday 18/12 and until Friday 20/12 on, we worked on threads (step 3). After the holidays, we started implementing the virtual memory (step 4) until the end of the week (from Monday 06/01 to Friday 10/01). We dedicated the last two weeks to both the file system and the network (from Monday 13/01 to Tuesday 21/01). We prepared the report and the defense from Wednesday 22/01 to the end of the project.

Every member in the group worked on its own on step 2 in order to get used to the NachOS basics, since this step is well guided. Thomas and Maxime started thinking about implementing step 3 first, then Alexis and Dylan joined them to implement the stack memory allocation for threads. Step 4 was mainly done by Thomas and Maxime, while Alexis and Dylan were writing the intermediate report. Finally, we worked on the two last steps in parallel : Dylan and Maxime designed step 5, while Thomas and Alexis elaborated step 6. We spent the whole working time at the IM2AG building, so we could easily communicate and share ideas using black boards.