Nachos

Project [MOSIG - M1]

Antoine Faravelon, Lucas Felix, Miratul Khusna Mufida, Hugo Guiroux, Simon Moura

January 30, 2014

Contents

1 Introduction

Nachos study was one of the one-month project proposed in M1-MOSIG. This is our final report on this subject.

First, we will expose the main features of our implementation of the kernel.

Then, we will explain our implementation choices, step by step, and discuss our organization of the past month.

Finally, we give, in an appendix section, the user documentation of system calls we made followed by malloc documentation.

2 Features of our kernel

The nachos kernel offers most of the basic features awaited from a kernel :

- Synchronized Input/Output
- User Multi Threading
- Virtual memory
- Concurrent multi-processes
- A File System
- TCP/IP like protocol

2.1 Interesting features

First, all the network features have been ported to the user space. A user program can use sockets and transfer content through the network using the means offered by our nachos kernel.

For these same threads, the kernel also give access to facilities to synchronize them (semaphore).

Processes have also been implemented with some managing functions. Indeed, though the kernel does not offer any processes hierarchy, a process can wait for the end of another one and then catch its exit value.

In user space, we provide a minimal shell which gives access to a prompt and the ability to launch programs with jobs management.

Finally, dynamic memory allocation has been implemented for the user space. User can allocate dynamically on the heap instead of the stack.

After that, the user has access to a whole range of system calls that will be detailed in the appendix. We consider our kernel fully-tested and we ensure no memory leak or bad memory accesses.

3 User documentation

User documentation has been moved in appendix for more readability.

4 User tests

4.1 Motivation

Testing has been an important phase in our whole development. In order to insure that our implementation was still working after any changes, we made a lot of regression tests.

It means that, as soon as we added a new feature which was compiling, we were able to check if it didn't generate any bugs or if it broke something we made before.

The idea was to create these tests in parallel with the implementation of new parts of the kernel. Thus, we tried to test everything we implemented on top of the given Nachos kernel.

4.2 The way it works

Our tests are divided in two directories, *regression* and *test*. The structure is the following :

- $*.c$ files, in test directory, contains some programs which helped us test user-level features and system calls
- $x.desc$ files contains descriptions about the matching $x.sh$ scripts
- $x.sh$ are bash scripts calling either C programs or launching directly one nachos command line option

4.3 How to run them

To run all tests, just compile the kernel then enter the following command : **make regression**.

It will run tests for all steps. Those which are marked as "ok" didn't raised any error, the others did.

In order to not be lost while tests are running, we specified the step we were working on at the beginning of the test name. For instance, $step5\_*$ tests are related to file-system.

To create our tests we proceeded the following way :

- First we create some basic test, as unitary as possible trying to cover each execution flow (including error handling)
- Then we create more complex tests using execution scenarios

We finally made 132 tests for all nachos parts.

4.4 Examples

4.4.1 Consumers/Producers

We implemented a producer/consumer test in user-mode using circular buffer. We used two semaphores and one mutex. The mutex protects the critical section where buffer is manipulated. The two semaphores are used as counter :

- One for the number of empty slots inside buffer
- One for the number of currently available items inside buffer

We do this with five producers and five consumers with buffer of size three.

4.4.2 The tiny shell

A shell was implemented with jobs support. Program can be launched using ForkExec. These programs run either in foreground or background using & at the end of the command line.

The "jobs" command list all currently background running processes.

The "fg" command is used to switch a background process to foreground.

These tests allow to check ForkExec and WaitPid syscalls as well as processes management.

4.4.3 Stress process

In the aim of testing our implementation limits on processes and threads, we tried to create 150 processes, each of them with 40 threads. It allows us to check if Nachos does not misbehave and handle limits properly. Thus, running this test, we see that when the bounds are reached, bad request for new process/threads creation are rejected.

5 Implementation choices

5.1 Synchronized Console

For consistency inside outputs, we needed to implement synchronization mechanisms. To do that, we choose to use the **Monitor** concurrent programming model.
Each method of the synchconsole was encapsulated using two semaphores :

- one for read **read** (GetInt, GetString, GetChar)
- one for **write** (PutInt, PutString, PutChar).

As GetString/GetInt and PutString/PutInt call respectively GetChar and PutChar, we either need recursive semaphores or make internals sub-routines _GetChar/_PutChar which are called by GetString/GetInt/GetChar, letting the synchronization mechanism inside these calling methods.
The second choice was made for simplicity.

To conclude with synchronized console implementation choices, we had to handle error inside GetChar and GetInt :

- For GetChar, the choice was made to return an integer instead of a char. By convention, **EOF** is -1.
- For GetInt, we cannot used a long int because the return value of a syscall is inside a register which is of size 4. Thus the address of an int is needed by GetInt. This int will be filled with the getting number and the return value of GetInt is used to handle error (see user documentation for more information).

5.2 Multi-Threading

Concerning stacks management, the first kernel thread (created at *Nachos* start) use the UNIX-process stack. Other kernel threads use heap-allocated stacks. For user thread, each thread as his own stack, from bottom of the memory up to the top (limitated by a constant MAX_STACK_SIZE). All user thread stacks are handled inside StackMgr component. StackMgr handle all stacks requests inside an AddrSpace. If UserThreadCreate is called and no space for stack is available, the function returns -1.

A word about thread hierarchy : there is none. This mean that every thread belongs to a process. A thread can create other threads, but the behavior (e.g exit) does not influence others.
A user thread can exit in two ways :

- calling UserThreadExit(void *ret) where *ret* is a generic pointer to the returned data
- returning from the thread function (also a generic pointer)

There is an exception for the first thread (the one with the main function). As pthread implementation, calling UserThreadExit with main thread let other threads finish. After the last thread end, it will exit the process.

As threads are light-weight processes, calling Exit in any threads kill the process with all threads belonging to it. This is the same if the main thread reach the return of the main function. It will implicitly call Exit. Thus does not wait for threads termination.

It is noticeable that thread id (tid) are unique to a process during is lifetime. A tid will never be re-used. This allows us to implement UserThreadJoin as discussed below. Even after a thread termination, his AddrSpace keeps track of his statue and keeps a Semaphore to join on it as well as the thread return value if available.

UserThreadJoin(unsigned int tid, void **retval) allows one thread to wait for another.
Inside retval (if not NULL) you can get the value return either by UserThreadExit or by the return value of thread function. If that thread has already terminated, then UserThreadExit returns immediately.
If multiple threads simultaneously try to join with the same thread, only the first one will be able to join on it. For others, syscall will return error code -2.
If threads join multiple times on the same thread (not at the same time), each call will be successful and if *retval* is not NULL, it will be filled with the return value of thread.

About semaphores, they are available to user program using UserSemaphoreCreate syscall. This returns a semaphore id process-specific. To wait for resource being available, UserSemaphoreP is here. To notify resource availability, it's UserSemaphoreV. Finally, UserSemaphoreDestroy destroys a semaphore. Semaphore id are process-unique and not re-used. If a UserSemaphore$[P|V]$ is used after the semaphore was destroyed, these syscalls will return -1. All semaphores are destroyed at the end of a process.

5.3 Virtual Memory

To handle virtual memory, all requests for memory page allocation/deallocation was made through a FrameProvider. This component has his own frame placement policy. Every time a process (via his AddrSpace) ask the FrameProvider for physical pages when it need it (Stack, Heap or Code/Data segments). We implemented different strategies to see if our model was coherent : random, first and last pages availaible.

A heap management was implemented. Stack and heap have fixed max size and we let a page between both in order to throw a page fault when an access is done outside the bounds. A HeapMgr which embed a top heap pointer manage requests for new heap pages. Heap and stack pages are dynamically allocated using FrameProvider.

5.4 Multi-Processes

ForkExec creates new kernel thread that will execute user program in a new address space. The creation of the new process is divided between the calling thread and the called thread which will begin its execution with a special initialization function. Management of processes is done by the ProcessMgr component which manage pids and provides the ability to join on a certain thread using Waitpid function.

There is no processe hierarchy, only the number of currently running threads is counted and when the last process exits the machine is halted.

When a process halt it does not affect the others since they are all independent programs with there own virtual memory code, data, and stacks.

Each pid is unique, so when using Waitpid it is possible to know and prevent the process from waiting on a dead process or on itself. As such it is possible to avoid deadlocks that could happen otherwise.

Waitpid allows to catch the exit code of the waited process.

5.5 File-System

5.5.1 Sub-Directory and Current Directory

To handle sub-directory we added an attribute inside each entry of the directory table. This attribute ($isDir$) allows to distinguish between file and directory. A function called ExpandFilename handles all ".", ".." to compute the absolute path. This allows us to avoid creation of two special files inside each directory, saving two entries.

A notion of CurrentDirectory was implemented attached to a process. Each process has his own directory and relative path are computed taking into account this one. All paths are computed using ExpandFilename.

5.5.2 Open files table

Each process has his own files table which is a unique identifier and a FileInfo structure. This file structure allows checking which thread own the file and which file name was already opened to apply some restrictions.

5.5.3 Max File Size and Dynamic Size

By default, each file was limited to $\frac{SectorSize-2*4}{4} * SectorSize$ bytes (default $3.75K$). Either we need to extend the $SectorSize$, or add another indirection level. Now the $\frac{SectorSize-2*4}{4}$ data sectors of a FileHdr points to DataBlockHdr. Each of these DataBlockHdr points to $\frac{SectorSize}{4}$ data block (the real content). The maximum size is now of $(\frac{SectorSize-2*4}{4} + \frac{SectorSize}{4}) * SectorSize$ bytes ($122kB$).

When a file is created, it has a size of 0. At each write, if needed, file size is expanded allocating Data Sector and DataBlock sector.

5.5.4 Concurrent Accesses

The SynchFileMgr component manages all concurrent accesses to a same file (represented by the sector of his FileHdr). Attached to this sector, a ReaderLock and a WriterLock are used to implement a reader/writer schema for all file accesses across all processes and threads.

5.6 Network

We chose to implement a socket user interface in our Nachos kernel. These sockets work in connected mode with robust transmissions.
As in TCP sockets, a server creates a listening socket and accept new connections on it. This allow to join a server on one port (defined by convention as 80 for HTTP servers) with many clients, and each of them will then have a personal socket with the server. On its side, the server can communicate with each client separately on different sockets. Our kernel provides 16 ports which can handle 16 sockets each, so the kernel can handle 256 connections at the same time.

5.7 Low level transmission

The lower layer of our network implementation is the transmission of one mail. This part has to be robust and then we can construct a higher level protocol on it.
For every single mail, the emitter sends the mail, wait a confirmation and retry if there is no. The number of tries is limited and the send function will return an error if this limit is reached. The receiver can receive many times the same mail before the emitter receives a confirmation. To prevent the duplication of a mail, we include an id in each of them. The receiver will take the first iteration of a mail and just confirm the other. The id we use is an integer that is always incremented, so the socket will stop to work after 2^{32} mails sent. This limit is hard to reach, so we did not deal with it, but it is possible to make the id go back to 0 when it happens without blocking the network. A confirmation mail contains the id of the mail it confirms, so a confirmation mail cannot be used later by the emitter to confirm another mail.

5.8 Protocol

Then we constructed the connected protocol over the robust transmission. The connection establishment use a three-way handshake.
We used the mailboxes of the initial Nachos code as ports, but now each mailbox contains an array of socket because many sockets can be connected within a same port (for instance the port 80 of HTTP servers). The number of connections a port can handle is limited to 16 because an array was simpler to implement and 16 is sufficient (but it can be replaced easily by a list).

The postal worker dispatch the mails he receives. He know th concerned mailbox thanks to the mail header and then find the good socket by comparing the origin of the mail to the ones contained into sockets.
Here is the schema of the classes used in network :

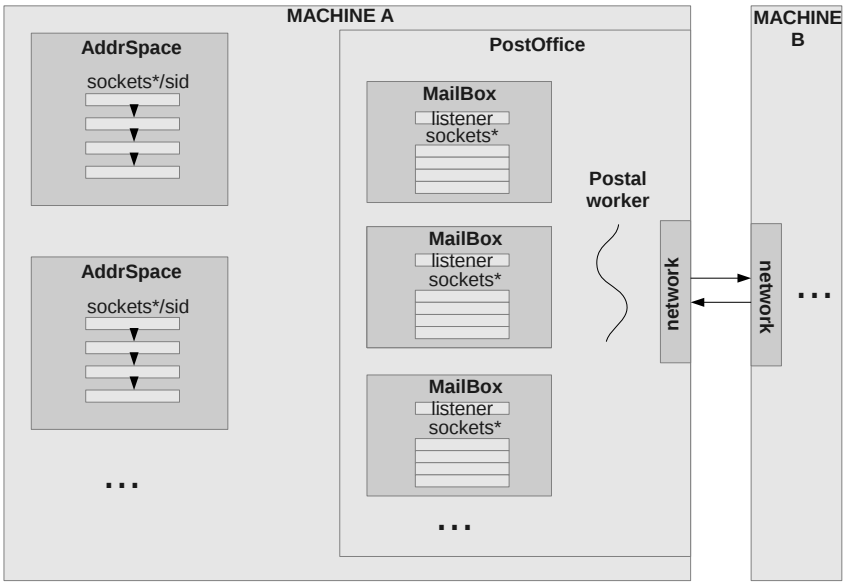

Figure 1: Each socket can be accessed from its address space and the mailbox which handle it.

6 Organization

6.1 Implementation

We decided to work everyday at the university so as to make communication between members of the group easier. Allowing us to solve bug faster, avoiding misunderstanding of the specification and cutting the work dynamically (reevaluating planning in real time when needed to).
For each step, before beginning anything, we all read and understood the subject. Then we discussed with each other, establishing a "todo list" and finally we distributed the work.
During the implementation, we often made summary of our work to explain the implementation of every part of the code to the rest of the team.
To make the group work easier we used a git repository. This allowed us to share our work more efficiently. This tool can manage code and handle merges automatically or at least make them easier. Thus, we were able to share the code immediately when needed.

6.2 Validation

Following the principle of the test driven development, a large set of automated regression tests were developed. It was used to validate the code or point out one or multiple bugs/regressions in a fast and easily understandable way.
A failure on a test will indicate what the test was about, thus allowing us to point out the error easily. And by doing the test before or during the implementation this automated test can allow to easily monitor the state and the advancement of the part currently being implemented.
Indeed during the beginning of the implementation, one of the members would implement the tests while the other began coding which allows to have tests ready as soon as the program compiles.
Valgrind has also been extensively used to remove memory leaks and wrong memory access. Thus, suppressing a significant number of possible bugs in the program. The program should now be free of any illegal memory accesses and memory leaks.

6.3 Part 1 to 4

For the parts 1 to 4, we followed the subject and all worked on the same part. We usually split into two teams and divided the work whole step into multiple parts. In these teams, we practiced extreme programming so as to compensate for the fact that all the work could not easily be split between each member. Thus, making the debugging phase faster. During the complicated phases, having two people working on the same problem often allowed us to solve it faster and to avoid many bugs in the code that a single person would not notice while coding.
Also, when working in pair (or triple) the tests were not done by the person who was currently coding. Thus they were not made with the weakness of the program in mind and tested uniformly the program without forgetting some part or another.

6.4 Part 5 and 6

The parts 5 and 6 were independent, thus, we decided to make them in parallel. The group was separated in two teams, each team working on one step. For the rest, the same philosophy as for the first parts was applied. That is, each team worked following the extreme programming principle.

The repository was split in branches in order to make the work easier for both team. Thus, we were able to work on the same files without causing neither, unnecessary conflicts, nor possibly bugs that would disturb the team not causing them.

We used tags to mark the last commit of a given step.

6.5 Last two days

On the last days, one person took care of the merging of the two branches. The others made a "todo List" for the report and presentation, then everyone took one part of it to prepare.

7 To be continued

There still exists some unimplemented functions in our kernel due to lack of time.
For the network, the migration of process and thread safe sockets has not been yet implemented. The process migration is still to be defined, but the sockets just miss the actual implementation of the necessary locks.

About file-system, optimization of input/output and robustness bonuses were not implemented.
In another hand, thanks to the regression tests, it is possible to say that, a priori, every implemented parts should be fully working. Though the set of test can not be complete and as such it is possible that there is still exists bugs that were not tested.

.1 Syscalls

NAME : **halt**

- SYNOPSIS : void Halt()
- DESCRIPTION : Halt is a system call which power off the system.

NAME : **Exit**

- SYSNOPSIS : void Exit(int status)
- DESCRIPTION : The exit syscall is used to quit the process. It will not shut down the machine unless there is no other process running.

NAME : **PutChar**

- SYSNOPSIS : void PutChar(char c)
- DESCRIPTION : PurtChar is a function which will write the character c to the console.

NAME : **GetChar**

- SYSNOPSIS : int GetChar()
- DESCRIPTION : GetChar is a function that read a character from the input buffer.
- RETURN :
  - The character itself or EOF(-1) if there is none in the buffer

NAME : **PutInt**

- SYSNOPSIS : void PutInt(int i)
- DESCRIPTION : PutInt is a function which is used to write the integer i to the console.

NAME : **GetInt**

- SYSNOPSIS : int GetInt(int* p)
- DESCRIPTION : GetInt read an integer and put the result inside input integer pointed out by p.
- RETURN :
  - 0 if there is no error
  - -1 when the input cannot be read as an int
  - -2 when the address p cannot be written by the process

NAME : **PutString**

- SYSNOPSIS : void PutString(const char s[])
- DESCRIPTION : PutString writes the string s to the console. If the string given is longer than MAX_STRING_SIZE then the remaining part is not printed in the console and put end of string ('
  0') at the end of the buffer.

NAME : **GetString**

- SYSNOPSIS : char *GetString(char *s, int n)
- DESCRIPTION : It reads at most $n-1$ characters in the console.
- RETURN :
  - s if there is no error
  - otherwise return NULL (no data read)

NAME : **UserThreadCreate**

- SYSNOPSIS : int UserThreadCreate(void f(void *arg), void *arg)
- DESCRIPTION : This function creates a new thread which will execute the function *f* with the argument *arg*.
- RETURN :
  - 0 on success
  - -1 if no space left for stack
  - -2 if MAX_TOTAL_THREADS has been reached launched (20 by default)

NAME : **UserThreadExit**

- SYSNOPSIS : void UserThreadExit(void *ret)
- DESCRIPTION : This function is use to destroy the current thread and puts the return value in *ret*. When the main thread call UserThreadExit, other threads continue to run. The last thread to end will call Exit. When a thread function reach a return statement, it will be converted to this syscall with return value as *ret*.

NAME : **UserThreadJoin**

- SYSNOPSIS : int UserThreadJoin(int tid, void **retval)
- DESCRIPTION : This function is used to join another thread (eg : wait for the thread of *tid* to terminate). If multiple threads try to join on the same thread, only the first one will be able to join on it. The function will return an error for the others. If *retval* is not NULL, it contains the return value of exit thread, either by calling UserThreadExit or by reaching the end of the thread function.
- RETURN :
  - 0 on success
  - -1 if bad *tid*
  - -2 if another thread is already joining on the same thread *tid*

NAME : **UserSemaphoreCreate**

- SYSNOPSIS : int UserSemaphoreCreate(char* name, int value)
- DESCRIPTION : Initialize and return a semaphore id named *name* with an initial value *value*. It does not create a semaphore with the id of a previously destroyed semaphore, as identifier are unique.
- RETURN :
  - Return the *id* of the semaphore freshly created

NAME : **UserSemaphoreP**

- SYSNOPSIS : int UserSemaphoreP(int id)
- DESCRIPTION : Takes the lock on the semaphore pointed by *id*.
- RETURN :
  - 0 on success
  - -1 if error (semaphore does not exist)

NAME : **UserSemaphoreV**

- SYSNOPSIS : int UserSemaphoreV(int id)
- DESCRIPTION : Release the semaphore pointed by *id*.
- RETURN :
  - 0 on success
  - -1 if error (semaphore does not exist)

NAME : **UserSemaphoreDestroy**

- SYSNOPSIS : int UserSemaphoreDestroy(int id)
- DESCRIPTION : Destroy the semaphore pointed by *id*.
- RETURN :
  - 0 on success
  - -1 if error (semaphore does not exist)

NAME : **AllocPageHeap**

- SYSNOPSIS : int AllocPageHeap()
- DESCRIPTION : AllocPageHeap asks for a new page on heap.
- RETURN :
  - -1 if no more page for heap
  - new page address otherwise

NAME : **FreePageHeap**

- SYSNOPSIS : int FreePageHeap()
- DESCRIPTION : FreePageHeap gives back a new page for heap.
- RETURN :
  - The new heap top address

NAME : **ForkExec**

- SYSNOPSIS : unsigned int ForkExec(char *s)
- DESCRIPTION : ForkExec creates a new process that execute the program stated in the argument *s*.
- RETURN :
  - *pid* of the newly created process in case of creation success
  - -1 if more than MAX_PROCESS processes have been created (by default 30)
  - -2 case of an invalid executable

NAME : **Waitpid**

- SYSNOPSIS : int Waitpid(unsigned int pid, int *retval)
- DESCRIPTION : Waitpid wait on the process which pid is given as argument. If *retval* is not NULL, the exit code of the process is put at address *retval*.
- RETURN :
  - -1 if process does not exist
  - -2 if process is dead
  - -3 if waiting for itself
  - 0 otherwise

NAME : **Open**

- SYSNOPSIS : int Open(const char* filename)
- DESCRIPTION : Open try to open file $filename$ taking into account current directory, returning a unique identifier
- RETURN :
  - -1 if file can not be opened
  - -2 if MAX_OPEN_FILES are already opened (default 10)
  - id $\in [0; MAX\_OPEN\_FILES[$, unique identifier used for a future syscall

NAME : **Close**

- SYSNOPSIS : int Close(int id)
- DESCRIPTION : Close try to close file with identifier *id*.
- RETURN :
  - -1 if file *id* does not exists
  - 0 otherwise

NAME : **Create**

- SYSNOPSIS : int Create(const char *filename)
- DESCRIPTION : Create file $filename$ taking into account current directory.
- RETURN :
  - -1 if creation failed
  - 0 otherwise

NAME : **Read**

- SYSNOPSIS : int Read(int id, char *buffer, int numBytes)
- DESCRIPTION : Try to read at most *numBytes* inside file *id* and store result in buffer. buffer should be large enough to fit *numBytes*.
- RETURN :
  - -1 if file does not exists
  - the real number of bytes read otherwise

NAME : **Write**

- SYSNOPSIS : int Write(int id, const char* from, int numBytes)
- DESCRIPTION : Try to write inside file *id* at most *numBytes* bytes stored in from memory.
- RETURN :
  - -1 if the file does not exists
  - the real number of bytes written otherwise

NAME : **Seek**

- SYSNOPSIS : int Seek(int id, int position)
- DESCRIPTION : Move at position *position* inside file *id* relative to the beginning of the file.
- RETURN :
  - -1 if the file does not exists
  - 0 otherwise

NAME : **Remove**

- SYSNOPSIS : int Remove(const char* name)
- DESCRIPTION : Delete file named *name*.
- RETURN :
  - -1 if the file does not exists
  - -2 if the file is opened by another process
  - 0 otherwise

NAME : **GetCurrentDirectory**

- SYSNOPSIS : char *GetCurrentDirectory(char *result)

- DESCRIPTION : Write the current process directory (absolute path) inside buffer *result*.
- RETURN :
  - address of *result* (never fail, can be ignored)

NAME : **SetCurrentDirectory**

- SYSNOPSIS : int SetCurrentDirectory(const char* dirname)
- DESCRIPTION : Set the current directory to *dirname* of current process. *dirname* can be relative path to current directory.
- RETURN :
  - -1 if *dirname* does not exists
  - 0 otherwise

.2 Malloc

An implementation of $malloc/free$ has been provided as on UNIX. *realloc* and *calloc* are also available.

Same rules and same prototypes as for UNIX version (see man malloc). Our version dynamically allocate and release pages for heap.

In context of multi-threads, you need to call *memory_init* into the main thread before forking thread to initialize synchronization structure. This is not needed (but not wrong) for no-multi-threads applications (only the main thread).

Three allocations strategies has been implemented :

- FIRST_FIT
- BEST_FIT
- WORSE_FIT

To use this library, include at top of your program :

```
1  #define BEST_FIT 1
2  #include "mem_alloc.c"
```

By default, the strategy is FIRST_FIT and the first line (define) is unnecessary. The mem_alloc.**c** is needed as Makefile only compile one file as a binary.