Operating System Design
First steps with NachOS

Vincent Danjean Guillaume Huard Arnaud Legrand
Vania Marangozova-Martin Jean-François Méhaut

January 2013

Abstract

The aims of these initial sessions are to:

- provide initial exposure to the NachOS simulator by reading the source code, tracing execution via the debugging options, and gdb. In addition to starting-up the simulator, you will observe the execution of a user program that will give an example of a system call, followed by kernel thread switching and sequencing.

1 Testing NachOS

You can get NachOS code source in the web page `https://forge.imag.fr/frs/?group_id=300`.
The `nachos.tar.gz` file contains the compressed archive. Install this archive in your personnal workspace with the following command:

```
tar xvfz nachos.tar.gz
```

This archive contains a Git source repository, i.e. a place to store various version of your NachOS project. At installation time, this directory contains a first minimal version of NachOS. If you already know Git, you can use it to track and manage your modifications. Else, just work with the visible directories and files, and ignore the hidden `.git/` directory.

Change directory to `nachos/code` within the copy that you have just extracted, and build NachOS:

```
cd nachos/code

make clean      # To restore a standard state (careful!)

make            # To generate the dependency files
                # and start the compilation (this can take time...)
```

Normally, everything should proceed uneventfully. During the compilation, you can read the above advice and start the next section, which starts with a reading of the code.

All you need do now is to start the tests. We will come back to the significance of these tests later. Do not worry: it is simply a question of checking that everything is in order.

```
cd build
./nachos-threads
```

for the first, and for the second:

```
./nachos-userprog -x ./halt
```

If the output displayed from these two tests does not contain anything that gives you reason to believe that there was a problem, then everything should be OK.

2 Reading the source code of NachOS

2.1 Principles of the simulation

In a true single-processor system, there are at least two modes in which the processor can be used:

user mode in which it executes instructions from a user program;

system mode in which it organizes the various system tasks requested of it: communication with peripherals (hard disk, keyboard, graphic card, etc.), resource management (memory [swapping, allocation], processor [process sequencing]).

In reality, the system executes system mode and user mode alternately in the same way as with one processor.

In the case of the NachOS simulator, the operation is different: you need to draw a clear distinction between what is simulated and what is really executed. Being compiled in x86 assembler, system mode executes on the true microprocessor (i386 family).

However, user-level programs are compiled in MIPS assembler and emulated by a MIPS processor emulator. Because this simulator is like any other program, it runs on the real processor (i386) of the computer on which you are working.

The NachOS program is an executable like the others on your computer, which periodically has access to the real processor (i386) for executing instructions in system mode, or for simulating various hardware devices such as a MIPS processor (for running user programs in user mode), a terminal for keyboard or screen input/output, a hard disk, etc.

To differentiate between them, we use the term real memory for the memory of the process (address space) in which NachOS runs, and the term MIPS memory for the simulated memory associated with the simulated MIPS processor. The same principle is employed for the real processor and the simulated MIPS processor.

To correlate this generalized explanation with actual reality, we are going to read the source code to track the execution of the command:

```
nachos-userprog -x ./halt
```

`nachos-userprog` has been compiled with only the `USER_PROG` flag.

The behavior of this command is to initialize the NachOS system and execute the MIPS user program `../test/halt`, which requests the system to shutdown.

Before reading the code, you can (and should) consult the appendix on the use of tags, which can make life a great deal easier for you.

2.2 System initialization

Like any process, the NachOS executable has real memory (address space) subdivided into code, heap and stack areas. Naturally, the entry point to this program is the main function of the file `threads/main.cc`. The first aim of this section is to observe how a common place executable "transforms" itself into an operating system in which there is one single kernel thread.

The second aim (more delicate) is to distinguish between what corresponds to real execution in system mode and what corresponds to a hardware simulation (the MIPS processor, the hard disk, etc.)

Simulator allocation: The diagram below illustrates the resources of the NachOS process and the elements of the system that are initialized at the end of the `Initialize()` function. These elements are created by the C++ operator `new` that is the "equivalent" of `malloc` in C.

The first kernel thread: We are interested in the creation of the first kernel thread. To answer the questions, we also recommend that you read the files `threads/thread.h` and `threads/thread.cc`. How is this first kernel thread created? That is to say, where do its stack and its registers come from? What is the (future) role of the data structure allocated by the following line of code?

```
currentThread = new Thread("main");
```

Can the subsequent kernel threads be created in the same way?

2.3 Execution of a user program

The MIPS processor: In the code of the `Initialize()` function, identify the allocation of the MIPS processor, and read the code that initializes this object and then answer the following questions: How are the registers of this processor initialized? What variable represents the MIPS memory?

Go back to the `main()` function and verify that the `StartProcess()` function is called with the file name `./halt`. Scan the code of this function and identify the loading of the program into memory (simulated or real?), the initialization of the MIPS processor's registers and, above all, the start of execution of the MIPS processor via the function `Machine::Run`.

Read the code of the `Machine::Run` function and locate the function that executes a MIPS instruction. Observing this function will enable you to ascertain the name of the exception caused when an addition (assembler instruction `OP_ADD`) overflows (even if that does not prove crucial subsequently). Look at the end of this function and identify the register that contains the program counter.

The Halt system call: Once within the `Machine::Run` function, the simulation of a user program can only be interrupted in two ways: either an interrupt is set (refer to the `Interrupt::OneTick()` function), or else the user program makes a system call.

Examine the end of execution of the system call `Halt()` present at the end of the program `../test/halt.c`. The assembler instruction coding a system call in `OneInstruction()` is `OP_SYSCALL`. Observe the processing of this instruction, particularly when the system takes back control and the number of the system call (`SC_Halt` in this case) is passed by a register (which?) of the MIPS processor. Follow the code until execution of the `CleanUp()` function, of which the role is to de-allocate the entire simulator.

3 Using the NachOS system

3.1 Observing the execution of a user program

Change directory to the `test` directory, and look at the program `halt.c`.

Now change directory to `build/`. Run:

```
make halt
./nachos-userprog -x ./halt
```

User programs are written in `C` and then compiled into MIPS binaries, which are then loaded and executed by the NachOS machine, instruction by instruction.

Trace to understand: Try tracing the execution of the `halt` program:

```
./nachos-userprog -d m -x ./halt
./nachos-userprog -rs 0 -d m -x ./halt
```

You can also execute the MIPS simulator step by step:

```
./nachos-userprog -s -d m -x ./halt
```

Edit the `halt.c` program and insert some calculations — perform a few operations on an integer variable, for example. Run a step-by-step trace to check that it does indeed change something.

Do not forget that in this MIPS realm, you can only use `C` language functions and NachOS system calls. You do not have any library functions available (such as printf, etc.)

You can also easily generate the assembler version of a MIPS program. Start by removing the halt.o files and then run `make V=1`. Retrieve the generated line into an editor by copying and pasting; it should resemble the following:

```
/opt/NACHOS/nachos_gcc/cross/decstation-ultrix/bin/gcc -c \
-I../userprog -I../threads -G 0 -c -o test.o ../test/halt.c
```

Change the last `-c` into `-S`. This will generate the MIPS assembler code for `halt` in the file `halt.s`. Then run `halt` again step by step and, this time, follow the assembler code instructions!

Locate the assembler instructions that encode the calculations and the call to the `Halt` function. You will find the code for this `Halt` function in the file `test/start.c`, which enables you to correlate your investigation with the study of the end of the system call in the preceding section.

(Optional question) Why is the first MIPS instruction executed on the tenth clock *tick*? (Refer to the `Interrupt::OneTick()` function and the system initialization.)

3.2 Observation of kernel threads

We will test the NachOS system "alone", i.e. in a self-test configuration. In fact, this test consists of starting two internal threads within the kernel that take turns in displaying a line on-screen for 5 iterations. Change directory to `build`. Start NachOS from the *threads* flavor:

```
./nachos-threads
```

The compilation options for this directory are such that Nachos starts the function `ThreadTest` in `main.cc` (take a look for yourself). This function is defined in `threadtest.cc`. Look carefully at its definition.

Nachos has debugging options. For instance, try:

```
./nachos-threads -d +   # + = all possible options
```

You can see the *ticks* of NachOS's internal clock, the switching between threads, the management of interrupts, etc.

It is possible to display just a part of this information. Instead of `+`, you can put:

`t` for information pertaining to NachOS system threads;

`a` for information pertaining to MIPS memory;

`m` for information pertaining to the NachOS machine;

`t` for information pertaining to thread management;

`i` for information pertaining to interrupts;

`n` for information pertaining to the NachOS network;

`f` for information pertaining to the NachOS filesystem;

`d` for information pertaining to NachOS disks.

Some parts of NachOS (disks, etc.) are not handled for the moment. So the corresponding flag will not display any additional message.

Now edit the file `threadtest.cc`. Compile and run the program as usual, and then watch carefully.

Add the start-up of an additional thread to the function `ThreadTest()`. Does it still work?

The semantics for the `fork` method of the `thread` object have nothing to do with that of the Unix `fork` function. What does the `fork` method in NachOS do? When are the NachOS threads created (memory allocated, structures initialized, etc.)?

Now comment-out the following line:

```
currentThread->Yield();
```

Recompile (`make`) and then examine what happens. What does this tell you about the preemption of system threads by default?

Uncomment the line again. You can run NachOS and coerce a certain degree of preemption, via the option `-rs <n>`. In addition, the parameters passed make the thread interlacing random (but *reproducible*!) — the number does not have any particular signification and is just used for initializing the random generator.

```
./nachos-threads -rs 0
./nachos-threads -rs 1
./nachos-threads -rs 7
```

What happens? Now use this option in tandem with the option `-d +`. How many clock ticks are there now?

This point is rather difficult to understand. Verify your intuition by commenting-out the line:

```
currentThread->Yield();
```

Your conclusions?

3.3 Introduction to the sequencer

The aim now is to understand part of the sequencer's operation, using the above experimentation as a basis.

Explicit context-switch: What happens exactly during a call to the `Yield()` function? Take a look at the source of this function in the file `code/thread/thread.cc`. At what point does a thread re-emerge from this function?

The Scheduler class: Examine the methods of the Scheduler class called by the `Yield()` function. What are the respective roles of the functions `ReadyToRun()`, `FindNextToRun()` and `Run()`?

At the heart of context-changing: In which function of the Scheduler class does one find the instruction that really causes a context change (i.e. a switching) between two processes? Find the source of the corresponding low- level function. What does it do?

3.4 Step-by-step execution of NachOS

Go back to the `build` directory and run NachOS within the `gdb` debugger.

```
gdb nachos-threads
[...]
(gdb) break main
Breakpoint 1 at 0x804d6fa: file main.cc, line 84.
(gdb) run
Starting program: nachos-threads  [...]
Breakpoint 1, main (argc=1, argv=0x8046db0) at main.cc:84
84          DEBUG('t', "Entering main");
(gdb)
```

You can move forward with the commands `s` (atomic step), `n` (next instruction in the current function), `c` (continue until next breakpoint), `r` (run: run or rerun the execution of the program), etc. In particular, try:

```
(gdb) break ThreadTest
[...]
(gdb) cont
[...]
```

This places a breakpoint in the `ThreadTest` function, and then tells NachOS to continue execution until it gets to a breakpoint.

Edit the `SimpleThread()` function, recompile, re-run, think, etc.

Run `SimpleThread()` step by step. What happens when you press `n` (*next*) when gdb is ready to execute the `yield` method? How can you insert a breakpoint to take back control within gdb whenever NachOS executes a context change?

3.5 Exercise

Edit the `yield` method so that it only performs a context change once every two calls.

Restart the `nachos-threads` program, and observe its execution (with the NachOS option `-d` and/or with `gdb`).

Now we will test the NachOS system threads, i.e. the *kernel-level* threads. Change directory to `build`. Run NachOS:

```
./nachos-threads
```

The compilation options of this flavor now cause NachOS to run the `ThreadTest` function in `main.cc`. This function is defined in `threadtest.cc`. Look carefully at its definition.

Now edit the file `threadtest.cc`. Compile and run the program as usual, and then watch carefully.

Add the start-up of another thread in the `ThreadTest()` function. Does it still work?

Now comment-out the line:

```
currentThread->Yield();
```

Recompile (`make`) and then examine what happens. What does this tell you about the preemption of system threads by default?

Uncomment the line again. You can run NachOS and coerce a certain degree of preemption, via the option `-rs <n>`. In addition, the parameters passed make the thread interlacing random (but *reproducible*). Try:

```
./nachos -rs 5
./nachos -rs 1
```

- What happens?
- Try it in tandem with the option `-d +`. How many clock ticks are there in each case?
- What additional hardware is used?
- Verify your intuition by commenting-out the line:

  ```
  currentThread->Yield();
  ```

- What are your conclusions?

You can also trace NachOS at the lowest level by using `gdb`. By default, the option `-g` is used for compilation (look at the `Makefile*` files).

```
gdb nachos-threads
[...]
(gdb) break main
Breakpoint 1 at 0x804d6fa: file main.cc, line 84.
(gdb) run
Starting program: nachos-threads  [...]
Breakpoint 1, main (argc=1, argv=0x8046db0) at main.cc:84
84          DEBUG('t', "Entering main");
(gdb)
```

You can move forward with the commands `s` (atomic *step*), `n` (*next* instruction in the current function), `c` (*continue* until next breakpoint), etc. Also try:

```
break ThreadTest
```

Edit the `SimpleThread()` function, recompile, re-run, reflect, etc.

Appendix: Tools for code reading

The code of NachOS is divided into many files and, when you read it, you have to skip from file to file. The *tags* enable you to locate automatically the definition of a function from a place at which it is called.

To do this, you first have to build a glossary of function definitions. The glossary will differ according to the editor used. `Emacs` uses the program `etags`, while `vim` uses `ctags` to build such a glossary and store it in a file (`TAGS` for emacs and `tags` for vim).

Afterwards, the editor you are using will read this file (when it is found) and ascertain the position of the definition of a function, and will know how to open the corresponding file.

Using the vim editor

Building the glossary

To create the glossary of NachOS functions, enter:

```
cd Mon_Nachos/nachos/code
ctags */*.cc */*.c
```

You should see a `tags` file created in the current directory (`Mon_Nachos/nachos/code` in the above case).

Reading the code

We will use this glossary for reading the code of NachOS using the `vim` editor. To prevent accidental write operations, we will use it in read-only mode (refer to the `-R` option), invoking it with the following command:

```
vim -R
```

The editor works in two modes: insertion mode, in which you can edit the content of the file, and command mode in which you do things such as opening and closing files (compare it with what you enter on the `emacs` command line). You read source code files in command mode. To switch to command mode, press **Esc**.

Basic commands

Command	**Result**
`Echap`	Invokes command mode (or remains in command mode if command mode is already active)
`:help` *subject*	Displays help about a subject (`:help tags` for example)
`:q!`	Coerces an exit from the editor without saving
`:new` *a_file*	Opens *a_file*
`:ts` *a_function*	Offers a list of functions with the name *a_function*, and goes to the chosen definition
`:ta` *a_function*	Goes to the definition of the first function in the glossary that has the name *a_function*
`:tags`	Displays the stack of current tags (similar to the function call stack)
`:pop`	Pops (i.e. removes) the first tag from the stack

By default, `vim` searches in the glossary in the current directory. This is why you are advised to start `vim` from the directory containing the glossary. Otherwise, you can specify the name of the glossary to be read by the command.

`:set tags=`*`repertoire/le_fichier_dictionnaire`*

As in `emacs`, in `vim` there are keyboard shortcuts to the most commonly-used commands. Therefore, instead of the command `:ta` *`a_function`*, you could also position the cursor on an occurrence of *a_function* in the code and press `Ctrl-]`. Similarly, the shortcut to `:pop` is `Ctrl-t`.

Using the emacs editor

To build the glossary, enter:

```
cd Mon_Nachos/nachos/code

etags */*.cc */*.c
```

To use it, start `emacs` in the same directory.

To search for the definition of a function, position the cursor over its name and enter `Meta-.` ("Meta - dot"). To return to the previous definition, enter `Meta-*`.

When conducting the first search for a tag, emacs prompts you for the glossary file name. In this case, it is `Mon_Nachos/nachos/code/TAGS`.

The grep command

If you want to search for something other than a function definition (such as a call to this function, or a variable name), the `grep` command is very useful.

```
cd Mon_Nachos/nachos/code

grep -r ``currentThread'' */*.cc */*.h */*.c
```

This displays lines containing the variable `currentThread` in all source files.