OPERATING SYSTEM
NachOS TD 0: First steps

Global instructions

NachOS (Not Another Completely Heuristic Operating System) is a rather basic simulated mini OS, implemented in C++, in which you will implement some components of a real system.
You have to provide your source code files and a report of 3 to 5 pages according to the procedure described at the following address `http://adrien.krahenbuhl.fr/en/teaching`*. Respect the following rules:*

- ✓ *You do* **not** *have the obligation give answers in the question order.*
- ✓ *Some requirements are deliberately imprecise on several points.*
- ✓ *Some non-trivial implementation choices are therefore left to your discretion.*
- ✓ *Before beginning to code,* **read first of all the whole TD parts***: they contain both descriptive passages to explain some concepts of the lectures or about NachOS. You have also some* **actions** *indicating precisely how to design your implementation step-by-step.*
- ✓ *Warning: this TD requires the add of methods from your part. Think before coding, otherwise it will be harder!*
- ✓ *To be reversible, all your changes must be framed with:*
  ```
  #ifdef CHANGED
  ...
  #endif CHANGED
  ```
  By default, you already compile with `-DCHANGED`*.*
- ✓ *Changes not reported are strictly prohibited.*

Keep in mind than this TD is not about implementation but mostly spending time to the NachOS exploration.

1 Set up NachOS

To work in good conditions, you will use a tool to easily manage your work and the file versionning. You can choose your preferred versionning tool (by example Git or SVN).

1.1 Install the NachOS base

The archive of the NachOS base is located on the course web page. It consists in a modified version of the original NachOS sources, directly compatible with the x86 architectures and the other TDs. Download and extract it:

```
wget http://adrien.krahenbuhl.fr/courses/UnivBordeaux/M1-System/nachosPUF.tar.gz
tar xvf nachosPUF.tar.gz
```

1.2 Install `xgcc`, a MIPS compiler

You will need `xgcc`, a MIPS compiler, as explained in the next sections. To install it, the easiest way is to retrieve the version already compiled from the course web page. You can download it with this command:

```
wget http://adrien.krahenbuhl.fr/courses/UnivBordeaux/M1-System/xgcc.tgz
```

Then you need to install it int the `/opt` folder with the correct rights:

```
tar xvf xgcc.tgz
sudo cp -r xgcc /opt/
sudo find /opt/xgcc/ -exec chmod 755 {} \;
```

If you have a 64 bit system, you should consider installing a 32bit libc, which is called libc6-i386.

1.3 NachOS tests

Move to the `nachosPUF/code` folder in the copy you have just extracted, and build NachOS:

```
cd nachosPUF/code
make clean   # To get back into a standard state (be cautious!)
make         # To generate the dependency files and start
```

Normally, everything must be ok and you just have to run the tests. We will discuss about the meaning of these tests later, do not worry. It is just to see if everything is in order.

Test 1:
```
cd threads
./nachos
```

Test 2:
```
cd ../userprog
./nachos -x ../test/halt
```

If the execution outputs doesn't contain suspicious indications, all is ready to start the NachOS exploration.

2 Source code reading

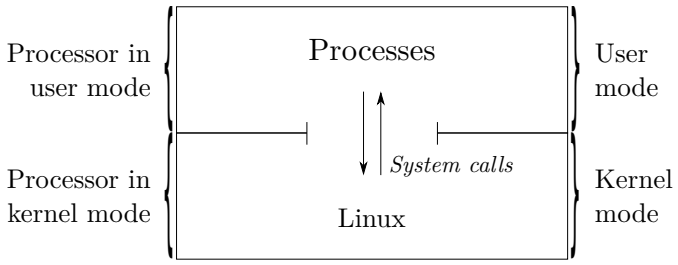
Important: Before starting this chapter, read the annexes concerning the tools for code reading.

2.1 Simulation principles

In a real mono-processor system, there are at least two methods of processor utilization:

- ✓ the **user mode**, in which are executed the instructions of a user program
- ✓ the **kernel mode**, in which are organized various required system tasks: communication with peripheral devices, resource management for memory and processors, ...

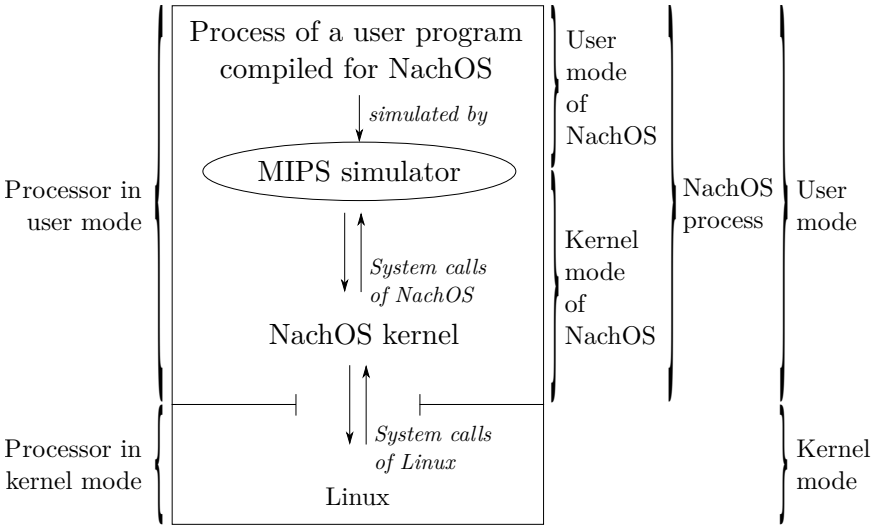
In fact, the system alternates executions of kernel and user modes in the same way on a processor, as illustrated below:



In the case of the NachOS simulator, the operating is different: you must absolutely distinguish what is simulated from what is really executed. Compiled in x86 assembler, the NachOS system is running on the real processor of your computer. On the other hand, the NachOS user-level programs are compiled for a MIPS processor (another processor architecture than x86) and executed by a MIPS processor simulator. This MIPS processor simulator is a program like any other of your computer: it is therefore executed on the real processor of your computer.
The `nachos` program is an executable like the others on your real computer which has from time to time the real processor to execute instructions in kernel mode or in user mode, as the MIPS processor simulator. All this is summarized in the following diagram:

> Give an example source file of a MIPS user program, and an example source file of the NachOS kernel. What is the programming language used each time?

To distinguish them quickly, the process memory (address space) in which NachOS is executed will be called the **real memory**, and the simulated memory associated with the simulated MIPS processor will be called the **MIPS memory**. There will be the same for the **real processor** and the **MIPS processor** that is simulated.



To link this general discussion to the NachOS implementation, read the source code associated to this command:

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

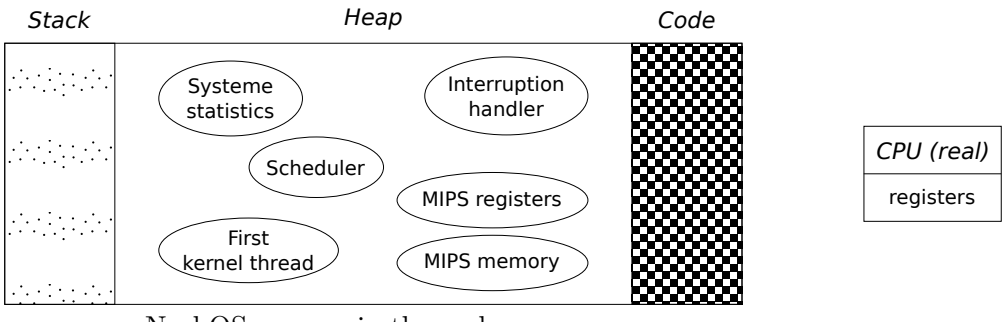
`nachos` having been compiled with the only flag $USER\_PROGRAM$, the behavior of this command is to initialize the NachOS system and run the MIPS user program `halt` that requires the system shutdown.

2.2 System initialization

Like any process, the `nachos` executable has a **real memory** subdivided into code, heap and stack areas. The entry point of thCODE is the `main` function of the file `threads/main.cc`.

> The first objective of this section is to observe how an usual executable "transforms" itself into an OS on which it exists a single thread kernel.
> The second objective (which is more tricky) is to distinguish what corresponds to a real execution in kernel mode from what corresponds to a hardware simulation (MIPS processor, hard disk ...).

Simulator allocation



NachOS process in the real memory.

The above diagram shows the resources of the NachOS process and the system elements, initialized at the end of the function `Initialize()`. Previously, they are created by the C++ operator `new` which is "similar" to `malloc` in C.

Specify in the above schema the names of the C++ classes that define these elements.
Specify if these elements belong to the kernel mode or the hardware simulation.

Notes: During the next NachOS TDs, you will not have to modify the files concerning the hardware simulation: this is a course about systems, not architecture. It is possible to buy memory module at a very competitive price by increasing the `NumPhysPages` macro.

The first kernel thread

We focusing on the first kernel thread. We advise you to read the files `threads/thread.h` and `threads/thread.cc`.

How is this first kernel thread created?
Where does its stack and its registers come from?
What is the (future) role of the data structure allocated by the instruction:

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

Why is it necessary to call the `Start` method for the next kernel threads?

2.3 User program execution

MIPS Processor

Locate the MIPS processor allocation in the code of the `Initialize()` function and read the initialization code for this object to answer the following questions:

How are the registers of this processor initialized?
What variable is MIPS memory?
Return to the `main()` function and verify that the `StartProcess()` function is called with the name of the `../test/halt` file.

Scroll through the code of the `StartProcess()` function to recognize:

- ✓ the loading of the program into memory (simulated or real?),
- ✓ the initialization of the registers of MIPS processor
- ✓ the launching of the execution of the MIPS processor by the `Machine::Run()` function.

Read the code of the `Machine::Run` function and locate the function that executes a MIPS instruction.

What is the name of the exception thrown when an addition (assembly instruction `OP_ADD`) overflows?
Observe the end of this function: which is the register containing the program counter?

System call Halt

Once in the `Machine::Run` function, the simulation of a user program can only be interrupted by two ways: (1) if an interruption is triggered (see the `Interrupt::OneTick()` function) or (2) if the user program makes a system call.

Observe the end of the `Halt()` system call, at the end of the program `../test/halt.c`. The assembler instruction encoding a system call in `OneInstruction()` is `OP_SYSCALL`.

Observe the processing of this instruction, in particular the moment when the system attempts to tackle the problem and the passage of the system call number (here `SC_Halt`) through a register (which?) of the MIPS processor. Follow the code until the execution of the `CleanUp()` function, whose role is to deallocate the whole simulator.

3 Running NachOS step-by-step

In case of a bug, you will always be able to follow the execution of NachOS, using a debugger such as `gdb`. By default, NachOS is compiled with the `-g` option (see the `Makefile`). We may prefer a more graphical debugger, that could be `gdb -tui`, `ddd`, `tdb`, ...
Go back to the `threads` directory and start NachOS in the gdb debugger. Use the `-args` option in gdb, in order to pass options defined after `nachos` to NachOS, not to gdb.

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

You can progress with these commands:

- **s** atomic step
- **n** next instruction in the current function
- **b** define a breakpoint
- **c** continue until the next breakpoint
- **r** (run), (re)-start the program execution
- **p** (print), display the variable values

Try for example:

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

This places a breakpoint on the `ThreadTest` function, then tells NachOS to continue the execution until it reaches a breakpoint. Notes that to set a breakpoint on a class method, you have to specify the name of the class, for example: `break Thread::Start`.
You can also specify a specific line, with for example `break main.cc:100`. If a NachOS execution ends by an assertion error, run it in gdb to find out where is the problem and see the call stack. Try for example to add `ASSERT(FALSE);` at the beginning of `SimpleThread`. You can observe this behavior:

```
$ ./nachos
Assertion failed: line 30, file "threadtest.cc"
zsh: abort ./nachos
```

Then, you can run the following commands:

```
$ gdb --args ./nachos
[...]
(gdb) run
Starting program: /home/samy/enseignement/SE/nachos/code/threads/nachos
Assertion failed: line 30, file "threadtest.cc"
Program received signal SIGABRT, Aborted.
[...]
(gdb) backtrace

#0 0x00007ffff730d165 in *__GI_raise (sig=<value optimized out>)
```

```
at ../nptl/sysdeps/unix/sysv/linux/raise.c:64
#1 0x00007ffff730ff70 in *__GI_abort () at abort.c:92
#2 0x0000000000404697 in Abort () at ../machine/sysdep.cc:432
#3 0x00000000004032ee in SimpleThread (arg=0x0) at threadtest.cc:30
#4 0x00000000004033ae in ThreadTest () at threadtest.cc:53
#5 0x0000000000401203 in main (argc=1, argv=0x7fffffffdad8) at main.cc:104
```

You see here that it is a direct call in `ThreadTest` that implied the drop, not the created thread. Use `Frame 3` to go back to the `SimpleThread` backtrace. Note that you can use `p` to display variables. You can also use `up` and `down` to go back up and down in the call stack.

Remove the `ASSERT` instruction and run `SimpleThread()` step-by-step. Use `display num`: gdb shows the evolution of the variable `num`.

What happens when you press `n` (next) when gdb is ready to execute the yield method?

Use `watch currentThread` and restart the program. The execution stops every time the `currentThread` variable changes, so you know who changes it. For each gdb command, you can use `help the_command` to have a summary.

4 Running NachOS in valgrind

It will probably happen that you make a mistake implying an overflow of allocated (or non allocated) spaces. You will get a `segmentation fault` during a call to `malloc`. In this case, use `valgrind`:

```
valgrind ./nachos
```

It will show you something like:

```
==7104== Invalid write of size 1
==7104== at 0x400511: main (test.c:4)
==7104== Address 0x517a04a is 0 bytes after a block of size 10 alloc'd
==7104== at 0x4C221A7: malloc (vg_replace_malloc.c:195)
==7104== by 0x400504: main (test.c:3)
```

Here, you are out of the table: you wrote just after its end.
It can also show you something like this:

```
==7310== Invalid read of size 1
==7310== at 0x4005BA: main (test.c:5)
==7310== Address 0x517a045 is 5 bytes inside a block of size 10 free'd
==7310== at 0x4C21DBC: free (vg_replace_malloc.c:325)
==7310== by 0x4005B1: main (test.c:4)
```

Here, you access to a table that has been released.

5 Using the NachOS System

5.1 Observation of a user program execution

Move under the `test` folder, and watch the `halt.c` program. Use `make halt` to compile it. Then go to `userprog` and run it:

```
./nachos -x ../test/halt
```

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

Tracing to understand

Try to trace the execution of the `halt` program:

```
./nachos -d m -x ../test/halt
```

Moreover, you can run the MIPS simulator step-by-step:

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

Modify the program `halt.c` to introduce some calculation, for example by doing some operations on a global integer variable (to prevent that gcc realizes that it is a fake computing and optimizes it). Trace step-by-step to see that the behavior change. Do not forget that in this MIPS world, you only have access to C functions and the NachOS system calls. No library function (`printf`, ...) is available.

It is also possible to easily generate the assembly version of a MIPS program, for example with `make halt.s`. Now, run `halt` step-by-step, following the assembly code instructions! Locate the assembly code instructions encoding the computing and the call to the `Halt` function. The code of this function can be found in the file `test/start.s` which allows to make the link with the study of the end of the system call in the previous part.

Optional question: Why is the first MIPS instruction is executed at the tenth clock tick? (Put a breakpoint on the `Interrupt::OneTick()` function to locate this first call.)

5.2 Kernel thread observation

We will test the NachOS system "alone", i.e. in self-test configuration. This test consists in starting two internal kernel threads displaying one line on the screen during 5 iterations. Go to the `threads` and run NachOS:

```
cd threads
./nachos
```

What are the compilation options of this folder?

With these compilation options, NachOS launches the `ThreadTest` function of `main.cc` (take a look!), defined in `threadtest.cc`. Carefully examine its definition.
Nachos has debugging options. Try for example:

```
./nachos -d + # this option means ``all possible options''
```

You can see the internal clock ticks, the thread switching, the interruption handling, ... This is due to the calls to `DEBUG`, similar to `printf`, activated only when the `-d` option is set. It is possible to display just a part of these debugging informations. Instead of `-d +`, you can specify:

- `-d t` for the kernel threads
- `-d a` for the MIPS memory
- `-d m` for the NachOS machine
- `-d i` for the interruptions
- `-d s` for the system calls

Some parts of NachOS (discs, ...) were not still manipulated until now. The corresponding flags will not display any additional messages.

Let's edit the file `threadtest.cc`, compile it, restart execution and watch carefully. Add the start of an second thread in the `ThreadTest()` function:

Is it still working?
What does the `Thread::Start()` method in Nachos?
When are created the NachOS threads? (memory allocation, structure initialization, ...)

Now, comment the following line `currentThread->Yield();`, recompile then examine what happens:

What can you deduce about the preemption of the default kernel threads?

Restore the previously commented line. You can run NachOS by forcing a certain degree of preemption with the `-rs <n>` option. Moreover, the seed passed as parameter makes a random (but reproducible!) thread interleaving. The number `n` has no particular meaning: it is used to initialize the random generator.

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

What happens?
Pair up this with the `-d +` option. How many clock ticks are there now?

This point is quite difficult to understand. Check your intuition by commenting the line `currentThread->Yield();`:

Your conclusions?

5.3 Discovering the scheduler

The objective of this last section is to understand a part of the scheduler work based on previous experience.

The change of explicit content

What happens precisely when you call the `Yield()` function?

Inspect the code of this function in the file `code/thread/thread.cc`.

When does a thread come out of this function?

Scheduler Class

Examine the methods of the `Scheduler` class called by the `Yield()` function.

What are the respective roles of the `ReadyToRun()`, `FindNextToRun()`, and `Run()` functions?

In the heart of the context switch

In which function of the `Scheduler` class is the actual instruction responsible of a context switch between two processes?
Find the source of the corresponding low level function. What is it doing ?

5.4 Exercise

Change the `yield()` method to make a context switch only once every two calls.
Restart the `nachos` program in the folder `threads` and observe the execution (with the `-d` option and/or with `gdb`).