NachOS: Multithreading

Year 2010-2011

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

The purpose of this subproject is to enable the execution of multithreaded user programs under NachOS.

Partie I. Implementing *User* Threads

The purpose of this practical exercise is to make NachOS threads available in user programs. In our context, each "user" thread will be directly supported by a NachOS thread.

Action I.1. *Examine NachOS threads in detail. How are these threads created and initialised? Where is the stack of a NachOS kernel thread? And where is the stack for the copy of the MIPS interpreter (i.e. the user thread) ? What are the functions* `SaveState` *and* `RestoreState\ of \|userprog/addrspace.cc` *for?*

Action I.2. *Start the* `putchar` *program with the following trace options:*

```
nachos -s -x ../test/putchar
```

for a step-by-step execution ,

```
../userprog/nachos -d + -x ../test/putchar
```

for a detailed trace (refer to `threads/system.cc` *for the other options, in particular* `-d t`*).*

Following the stepp-by-step execution of the prorem, examine how it is installed in memory(using an `AddrSpace` *object), how it is launched and then stopped. Take in particular a look at* `userprog/progtest.cc` *and at* `userprog/addrspace.cc`*.*

Now we want a user program to be able to create threads, i.e. perform a system call

```
int UserThreadCreate(void f(void *arg), void *arg)
```

This call must start the execution of `f(arg)` in a new copy of the MIPS interpreter (in other words, a new instance of the interpreter executed by a *kernel thread*).

- On the system call `UserThreadCreate`, the current kernel thread must create a new thread `newThread`, initialise it and place it in the threads queue (in the kernel) via the call

  ```
  newThread->Fork(StartUserThread, f)
  ```

  In the process, it sets the `space` variable of this new thread `newThread` to the same address as itself, so that the new copy of the MIPS interpreter shares the same MIPS address space.

  Note that because the `Thread::Fork` function only takes one single parameter, you cannot pass `f` and `arg` directly by these means. It is up to you to figure out what to do!

- When it is finally activated by the sequencer, this new thread executes the function `StartUserThread`. This function initialises backups of registers of a new copy of the MIPS interpreter in the same way as the primitive interpreter (`Machine::InitRegisters\ and \ |Machine::RestoreState\ functions), and starts the interpreter (\| Machine::Run`).

  Note that you will have to initialise the stack pointer. I suggest that you place it 2 or 3 pages below the pointer to the main program. Of course, this is an empirical value! You will probably have to do better at a later stage...

- To finish, a user thread must simply destroy itself by a `UserThreadExit` system call, which invokes a function called `do_UserThreadExit` in a new source file `userprog/userthread.cc`. This function activates `Thread::Finish` at NachOS level. Note that the MIPS function `UserThreadExit` never returns a value, like Unix's exit system call

Action I.3. *Insert the system calls*

```
int UserThreadCreate(void f(void *arg), void *arg)
```

et `void UserThreadExit()`.

For what reason(s) can the creation of a thread fail? In this case, return -1

Action I.4. *Write the function*

```
int do_UserThreadCreate(int f, int arg)
```

activated at NachOS level during the call to `UserThreadCreate` *by the calling thread. You will have to work hard on this function: put it in the file* `userprog/userthread.cc`*, only placing the declaration*

```
extern int do_UserThreadCreate(int f, int arg);
```

in the file `userprog/userthread.h`*. Inlcude this file in* `userprog/exception.cc`*.*

Thus, the function wil be invisible elsewhere. Remember to edit the `Makefile` *to take into account this new file. You will probably have to restart the entire compilation process so as to build the correct dependencies:* `make clean; make`*.*

Action I.5. *In* `userprog/userthread.cc` *define the function*

```
static void StartUserThread(int f)
```

This function is executed by the thread created by `do_UserThreadCreate`*.* Be careful because you don't control the moment when this function is to be executed, it depends on the scheduler... *Also, note that you need to pass the argument* `arg` *in another manner (serialization). It is up to you to figure out what to do!*

For the time being, we are going to consider that a thread ends with a systematic invocation of the system call `UserThreadExit()` (therefore, it never "exits" from the initial function).

Action I.6. *Define the behaviour of the system call* `UserThreadExit()` *by a function called* `do_UserThreadExit`*, also placed in the file* `userprog/userthread.cc`*. For the time being, it simply destroys the current NachOS thread by the call to Thread::Finish. What should you do for its address space?*

Note that the main program must not call the `Halt` function as long as the user threads have not called `UserThreadExit`! Therefore you have to make it wait artificially... How? It is up to you to figure out the solution!

Attention ! Nachos must be started with the option `-rs` to coerce the pre-emptive sequencing of user threads:

```
nachos -rs -x ../test/makethreads
```

By adding a parameter to the option, you change the random sequence employed for the scheduling:

```
nachos -rs 1 -x ../test/makethreads
```

Note that the scheduling of kernel threads is not pre-emptive. Why is this?

Action I.7. *Use a small program called `test/makethreads.c` to demonstrate the operation of your implementation. Test different sequencings. What happens if the option `-rs` is not specified? Explain.*

Partie II. Several Threads per Process

The implementation above is still very primitive, and could be improved in several ways. If you try to perform write operations (for example, via the `putchar` function) from the main program and from the thread, then you will probably get an `Assertion Violation` error message. (Try!) Indeed, the write and read requests from the two threads are mixed! Therefore, you must protect the corresponding kernel functions via a lock.

Action II.1. *Change the `SynchConsole` class in order to synchronize read and write operations. Can you use two different locks? Note that these locks are private to this class. Demonstrate its operation via a test program.*

For the time being, the user has to guarantee that the main program does not call the `Halt` function as long as the thread has not called `UserThreadExit`.

Action II.2. *What happens if the initial thread exits from the program (i.e. with a call to `Halt`) before the threads with which they co-habit have called `UserThreadExit`? Correct this behaviour, ensuring that there is a synchronisation with regard to `Halt` and `ThreadExit` calls — for instance, by counting the number of threads sharing the same address space (`AddrSpace`). You will undoubtedly have to use a semaphore at NachOS level used by all threads sharing the same address space. Demonstrate the operation via a test program.*

For the time being, a program only calls `UserThreadCreate` once. You should remove this limitation.

Action II.3. *What happens if the program starts several threads, and not just one? Run a test and explain what you observe. Suggest a correction allowing a large number of threads to be started. Demonstrate its operation via a test program.*

Action II.4. *What happens when a program starts a great number of threads? Discuss in detail the various behaviours as a function of the scheduling.*

Action II.5. *Implement the system call `UserThreadJoin` used by a user thread in order to wait the finish of another user thread. You will need to manage thread identifiers. Demonstrate your solution with a test program.*

Partie III. BONUS: Automatic Termination

For the time being, a thread has to explicitly call `UserThreadExit`. Similarly, the main program must explicitly call `Halt`. Obviously, this is not a very elegant solution and, above all, it makes errors very likely!

Action III.1. *Explain what would happen if a thread did not call* `ThreadExit`*. How is this problem resolved for the initial thread (with* `nachos âĂŞx`*)? In particular, look at the file* `test/start.S`*. What needs to be implemented to use this mechanism in the case of threads created with* `UserThreadCreate`*? Important: your solution must be independent of the true address at which the function is loaded. Therefore, you need to pass this address as a parameter when the system call is made... Over to you!*

Why not now implement a producer/consumer schema at user level?

Action III.2. *Extend the access to semaphores (*`type sem_t`*, system calls* `P` *and* `V`*) to user program level. This time, demonstrate their operation through a producer/consumer example at user program.*