OPERATING SYSTEM
NachOS TD 2: Multithreading

Global instructions

The goal of this TD is to allow the execution of multi-threaded applications in NachOS.
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.*

1 Multithreading in user programs

In TD 0, you already played with the kernel threads of NachOS, especially in the `ThreadTest` function. The next step consists in allowing the user programs to create and manipulate user threads of NachOS helped by system calls that will use kernel threads to support the initial user threads. Note: In this first part, we will launch only one additional thread.

> Examine in details how NachOS threads work (kernel and user). How are allocated and initialized these threads? Where is located the stack of a NachOS thread, as kernel thread?

(1.1) Start your `putchar` program with some trace options seen in TD 0:

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

Observe especially at the beginning of the obtained output how a program is installed in the memory (helped by an object of type `Addrspace`), then launched, then stopped. Find in particular where this is done in the files `userprog/progtest.cc`, then `userprog/addrspace.cc`.

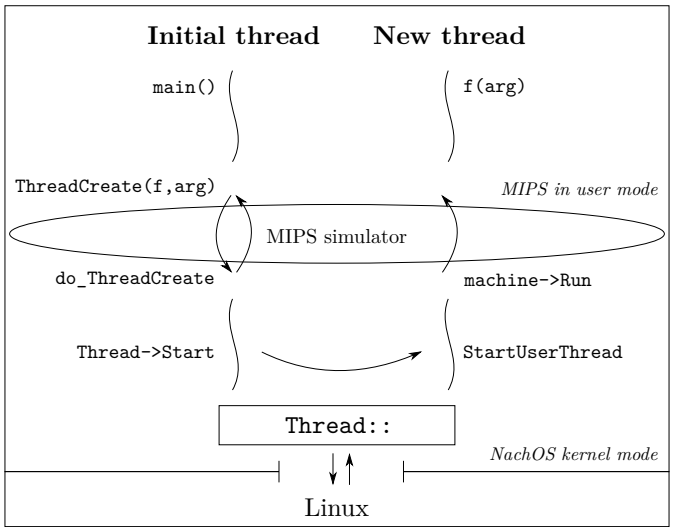
(1.2) Now, we desire that a user program can create threads able to execute functions of the program, i.e. execute a MIPS system call:

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

This call must start the execution af `f(arg)` in a new copy of the MIPS interpreter (i.e. a new instance of the interpreter executed by a new kernel thread). Herafter a diagram and a summary of what you are going to implement in the following quesions. Do not start coding, Read this summary first, and do not start coding until you reach the text of the next question, which explains how to start.

- ✓ On the `ThreadCreate` system call, the current kernel thread must create (in a new `do_ThreadCreate` function) a new thread `newThread`, initialize it and place it in the (kernel) thread queue by using:

```
NewThread->Start( StartUserThread, plop )
```

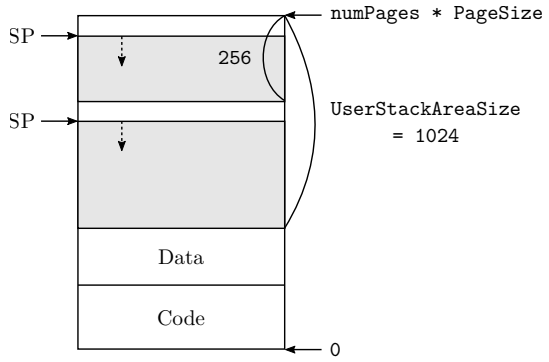


You will notice that the constructor set the space variable of this new thread `newThtread` to the same value as the current thread, in order than the new copy of the MIPS interpreter shares the same MIPS address space as the parent thread. The `Thread::Start` function allows you to pass only one parameter `plop` to `StartUserThread`: you can not directly pass both `f` and `arg`. It is up to you to find out how to do.

✓ When this new thread is activated by the scheduler, it run the `StartUserThread` function which you will create. This function initializes all MIPS registers in a similar way to the primitive interpreter (get inspired by the `AddrSpace::InitRegisters` function) and starts the interpreter (`Machine::Run`). You will need to initialize the stack pointer. Add for that an `AllocateUserStack` method to the `AddrSpace` class, returning the top address of this new stack. It is suggested, in a first time, to place it 256 bytes below the end of the virtual memory (i.e. below the stack of the main thread). The size of the virtual memory is `numPages*PageSize`. This is a basic method, you probably will have to do better in a second time. To be able to call this method, use `currentThread->space`.

✓ Finally, a thread executing code in user space must destroyed itself by a `ThreadExit` system call, which calls a `do_ThreadExit` function defined in a new source file `userprog/userthread.cc`. This function must call `Thread::Finish` to terminate the NachOS thread. The `Thread` object is already freed automatically by the scheduler.

The following diagram shows the address space with the basic method to allocate the second stack:



(1.3) Set up the system call interface

```
int ThreadCreate(void f(void *arg), void *arg);
void ThreadExit(void);
```

For what reason(s) could the creation of a thread fail? It will be thought later to return `-1` in this case.

(1.4) Write the function

```
int do_ThreadCreate(int f, int arg)
```

activated at the NachOS level when the calling thread execute `ThreadCreate`. You will have to work a lot on this function: put it in the `userprog/userthread.cc` file by placing only the declaration:

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

in the `userprog/userthread.h` file. Then add the inclusion of this file in `userprog/exception.cc`. This function is invisible elsewhere. Consider adjusting the `Makefile.common` to take into account `userprog/userthread.cc`.

(1.5) Define in `userprog/userthread.cc` the function:

```
static void StartUserThread(void *plop)
```

called by the new NachOS thread created by `do_ThreadCreate`. Be very careful because you have no control over when this function is called: it depends on the scheduler. Again, you must pass to this function both the arguments `f` and `arg`. It is up to you to find out how to do. Add a lot of debugging informations: use the `DEBUG('x', "my debug %d\n", myvar);` macro to print the values you put into the registers, to be sure of your calculations.

(1.6) Define the behavior of the `ThreadExit()` system call by a `do_ThreadExit` function, also located in the `userprog/userthread.cc` file. For now, it just destroys the initial NachOS thread by calling `Thread::Finish`.

What should be done for its address space?

(1.7) Demonstrate on a small `test/makethreads.c` program how works your implementation, using a simple `PutChar` in the created thread (but not in the main thread), and paying attention to the important remarks below. If you have bugs, check what value of `PCReg`, `NextPCReg` and `StackReg` you are giving to your thread. Test different schedules.

Important: For the new user threads have a chance to run, the user main thread must not terminate, i.e. must not exit from the `main` MIPS function, as long as the user threads do not call `ThreadExit`. At first, make the `main` function waiting after the thread creation with an infinite empty loop.

Important: First, in your test programs, complete all your threads by calling the `ThreadExit()` system call systematically, so that they will kill themselves from the user mode. They will never "leave" their initial function `f`.

Important: As long as you have not yet locked out your console implementation, do not make displays from several threads.

Warning: NachOS must be launched with the `-rs` option to force the preemptive (and therefore realistic) scheduling of user threads:

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

By adding a parameter to the option, you modify the random sequence used for scheduling:

```
./nachos -rs 1234 -x ../test/makethreads
```

Note that scheduling kernel threads is not preemptive.

2 Multiple Threads per Process

The above implementation is still very primitive and can be improved on several points.
If you try to make writings, for example by the `putchar` function, from the main thread and from the created thread, you will probably get an `Assertion failed` error message. **Try it**. Indeed, the requests of the two threads for writing and awaiting of acknowledgment are mixed. It is necessary to protect the corresponding kernel functions by a lock. Use semaphores, or better, complete the implementation of the locks in `synch.cc` inspired by the one of the semaphores.

(2.1) Modify your implementation of the `SynchConsole` class to put the treatments performed by `SynchPutChar` and `SynchGetChar` in the critical section.

Can you use two different locks?

Note that these locks are private to this class. Demonstrate the functioning by a test program.

Should `SynchPutString` and `SynchGetString` also be protected? Why?

If a thread executes `Exit` (the `Halt` system call will no longer be used) or the main thread exits the main function, NachOS is stopped without giving a chance to another thread to continue its execution. To let the other threads run in the process, `main` can use `ThreadExit` to ending itself without ending the process.

Does NachOS actually end if both the thread created and the initial thread are using `ThreadExit`?

(2.2) Fix the termination by counting in `ThreadExit` the number of threads sharing the same address space `AddrSpace` (in the next TD, there will be several processes), in order the last thread calls `interrupt-> Halt()` to end NachOS. Show how it works using a test program.

Warning, maybe you will want include `synch.h` from `addrspace.h`. This can not work because `synch.h` includes `thread.h`, which itself includes `addrspace.h`. In the end you can not include anything before the other. Rather than include `synch.h`, simply write a partial declaration class, for example: `Class Semaphore;`.

So the `addrspace.h` will compile, since it only contains pointers to semaphores, it just needs to know that the `Semaphore` class exists.

Until now, a program can call `ThreadCreate` only once, because of the stack allocation which is too basic. This limitation must be lifted.

What would happen if the program started several threads instead of one? Try it to see.

(2.3) This may work, but add to your threads a `for` loop on a local variable (so on the stack) `volatile int i;` which displays an `a` at each loop, and count the number of `a`. Suggest a correction to launch some threads. Before despairing to observe only mysterious bugs, make sure that you give to the live threads different stacks of size at least 256 bytes, and that do not overflow in the data or code spaces, and that you take into account the stack of the main thread. Demonstrate how this works by a test program.

What happens if a program launches a large number of threads?

(2.4) Possibly review your mechanism of stack allocation, using, for example, the `Bitmap` class of `bitmap.cc` to store which stack slots are already in use. Pay attention to the stack of the main thread. Discuss precisely the different behaviors according to the scheduling.

3 Automatic Termination (bonus)

For now, a thread must explicitly call `ThreadExit` to end. This is not very elegant, and especially very error-prone.

> Explain what would happen if a thread does not call `ThreadExit`. How is this problem solved for the main thread (with `nachos -x`)? Look especially in the file `test/start.S`. What should be set up to use this mechanism for threads created with `ThreadCreate`?

Note: Your solution must be independent of the real loading address of the function and `ThreadExit`. It will be necessary to pass this address in parameter during the system call. It's your turn!

4 Semaphores (bonus)

(4.1) Rewind the access to the semaphores (`sem_t` type, `P` and `V` system calls) at the level of user programs. Demonstrate how they work by an example of consumer/producers at the user level this time.