OPERATING SYSTEM
NachOS TD 1: Concurrency and synchronization

Global instructions

The goal of this TD is to implement in Nachos some basic system calls.
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 What is the goal?

The goal of this supervised assignment is to set up a minimal input-output mechanism under Nachos, allowing the execution of the following `putchar.c` program:

```
#include "syscall.h"

void print(char c, int n) {
    int i;
    #if 0
    for (i = 0; i < n; i++)
    {
        PutChar(c + i);
    }
    PutChar('\n');
    #endif
}

int main()
{
    print('a',4);
    Halt();
}
```

What is the reasonably expected output considering `#if 0` transformed to `#if 1`? We will not activate the code at first, so that it can compile until we have implemented it.

Therefore, it is necessary to place this `test/putchar.c` program under the `test` folder. It is normal that it compiles again with errors, they will not interfere at first.

2 Asynchronous Inputs Outputs

NachOS offers a primitive version of I/Os through the `Console` class which is declared under `machine/console.h`. Read the comments carefully. Do not read `machine/console.cc`, since this is hardware simulation: it is the interface `machine/console.h` which provides us interesting things. The I/Os operate asynchronously by interruptions to:

to write a character we send a writing request using `Console::PutChar(char ch)`, then we wait until to be notified about the end of the request by the execution of the interruption handler `WriteDoneHandler`.

to read a character we are waiting to be prevent by the execution of the interrupt handler `ReadAvailHandler` that there is something to read, then we read it using the function `Console::GetChar()`

Explain why it is a mistake to try:

- ✓ to read a character before being warned that a character is available
- ✓ to try to write before being warned that the previous writing is complete.

Handlers are C functions, not C++, because they are shared by the console and the classes that use them. There is no reason to not do useful things between the moment when an handler "sends" a request and the moment when another one is warned about its end. The communication can be completely covered by computations.

Now, look at the implementation in `userprog/progtest.cc`. We first place ourselves in a simple case where one handler stops on waiting for the end of another one using semaphores:

```
static Console *console;
static Semaphore *readAvail;
static Semaphore *writeDone;

static void ReadAvailHandler(void *arg) { (void) arg; readAvail->V(); }
static void WriteDoneHandler(void *arg) { (void) arg; writeDone->V(); }
```

In order to wait, we take the semaphore. The notification handlers release them. As a result, if the character is already present during a reading request, it is immediately served.

```
readAvail->P(); // wait for character to arrive
ch = console->GetChar();
```

(2.1) Examine the `userprog/progtest.cc` program. Launch `./nachos -c` which executes the `consoleTest` procedure (see `threads/main.cc`). Understand what is happening.

(2.2) Modify `userprog/progtest.cc` to display "Goodbye" to end of file (EOF), in addition to the character 'q'. (Note: to test, simply type control-D at the beginning of the line).

(2.3) Modify `userprog/progtest.cc` to write "<x>" instead of "x" in the loop body (regardless of the "x" character).

(2.4) Verify that this also works with an input file and an output file. For example, `nachos -c in.txt out.txt` (See `threads/main.cc`).

3 Synchronous Inputs Outputs

The goal is to implement, above the Console layer, a **synchronous** input-output layer `SynchConsole`. The idea is that a **synchronous console** must encapsulate the entire semaphore mechanism to provide only two functions. This is implemented right next to the `Console` class.

(3.1) Create the userprog/synchconsole.h file as follows: you can copy/paste from the PDF.

```
#ifdef CHANGED

#ifndef SYNCHCONSOLE_H
#define SYNCHCONSOLE_H

#include "copyright.h"
#include "utility.h"
#include "console.h"

class SynchConsole:dontcopythis {
    public:
        SynchConsole(const char *readFile, const char *writeFile);
            // initialize the hardware console device
        ~SynchConsole(); // clean up console emulation
        void SynchPutChar(int ch); // Unix putchar(3S)
        int SynchGetChar(); // Unix getchar(3S)
        void SynchPutString(const char *s); // Unix fputs(3S)
        void SynchGetString(char *s, int n); // Unix fgets(3S)
    private:
        Console *console;
};

#endif // SYNCHCONSOLE_H
#endif // CHANGED
```

Notes:

- ✓ `#include "console.h"` works correctly thanks to the search path specified during the compiler call.
- ✓ the semaphores must be shared between the `SynchConsole` class objects and those of the `Console` class. They must therefore be C functions and not C++, unless using advanced C++ features (`SynchConsole` should actually be a `Console` subclass).

The userprog/synchconsole.cc file must therefore have the following structure:

```
#ifdef CHANGED

#include "copyright.h"
#include "system.h"
#include "synchconsole.h"
#include "synch.h"

static Semaphore *readAvail;
static Semaphore *writeDone;

static void ReadAvailHandler(void *arg) { (void) arg; readAvail->V(); }
static void WriteDoneHandler(void *arg) { (void) arg; writeDone->V(); }

SynchConsole::SynchConsole(const char *in, const char *out) {
    readAvail = new Semaphore("read avail", 0);
    writeDone = new Semaphore("write done", 0);
    // console = ...
}

SynchConsole::~SynchConsole() {
    delete console;
```

```
    delete writeDone;
    delete readAvail;
}

void SynchConsole::SynchPutChar(int ch) { /* .... */ }
int SynchConsole::SynchGetChar() { /* ... */ }
void SynchConsole::SynchPutString(const char s[]) { /* ... */ }
void SynchConsole::SynchGetString(char *s, int n) { /* ... */ }

#endif // CHANGED
```

(3.2) Complete `synchconsole.cc` regarding the operations on the characters. As documented in `synchconsole.h`, they have a behavior equivalent to the Unix functions `putchar` and `getchar`). Do not complete `SynchPutString` and `SynchGetString` yet.

(3.3) Complete the `Makefile.common` file. Whenever `console` appears, `synchconsole` should also appear.

(3.4) Modify `threads/main.cc` to add a synchronous console test `-sc` option that starts the `SynchConsoleTest` function.

(3.5) Add to the end of `progtest.cc` the definition of this function, for example:

```
#ifdef CHANGED
void SynchConsoleTest (const char *in, const char *out)
{
    char ch;
    SynchConsole *test_synchconsole = new SynchConsole(in, out);
    while ((ch = test_synchconsole->SynchGetChar()) != EOF)
        test_synchconsole->SynchPutChar(ch);
    fprintf(stderr, "EOF detected in SynchConsole!\n");
}
#endif // CHANGED
```

and consider including `<synchconsole.h>` to get the declaration of the `Synchconsole` class. Note that the `fprintf` is done by Linux, not by Nachos!

(3.6) Add in `SynchConsoleTest` the display of "<" and ">" as in the previous section.

4 `Putchar` System Call

The goal is now to set up a system call `PutChar(char c)` taking a character `c` as argument in user mode then throws a `SyscallException`. This causes the switch into kernel mode and the execution of the standard handler `ExceptionHandler`. It must:

- ✓ retrieve parameters from the MIPS world
- ✓ call the `SynchPutChar` function
- ✓ return to the calling program

by taking care to increment the program counter. This explains why a system call is so expensive. In order to reduce this cost, the Unix input-outputs are **buffered**: `fprintf` (in man 3) is much less expensive than `write` (in man 2) on each character, because there is a calling system only for each line, not for each character.

The first task is to set up the system call:

(4.1) Edit the `userprog/syscall.h` file to add a "`#define SC_PutChar ...`" system call and the declaration of the corresponding "`void PutChar(char c)`" function. This is the Nachos user function: in Unix, the corresponding function is `putchar` (in man 3).

(4.2) It is now necessary to define the code of the `PutChar(char c)` function. Since this must cause a trap, it must be written in assembly code.

(4.3) Edit the `test/start.S` file to add the assembly code definition of `PutChar`. You can copy `Halt`, taking care to retrieve all the lines concerning `Halt`. Note that the system call number is placed in register `r2` before the call of the "magic" `syscall` instruction. It is the compiler that takes care to place the first argument `char c` in the register `r4`. This register `r4` is a 32-bit integer register: the character is therefore implicitly converted: `r4 = (int)c`. You can then activate the code snippet of `putchar.c` calling `PutChar`: this should compile.

It is now necessary to set up the handler which is activated by the `syscall` interrupt:

(4.4) Edit the `userprog/exception.cc` file. Observe the `switch` instruction in function

```
ExceptionHandler( ExceptionType which )
```

There is already the case of the system call `SC_Halt`. Add the case for `SC_PutChar` (surrounded by `#ifdef CHANGED` of course). Implement it using your `SynchConsole`.

(4.5) Already get used to putting a `DEBUG('s',"PutChar n");` at the beginning of the case, to be able to easily display which system call is executed by the program by simply using option `-d s` of `nachos`. There will be many possible exceptions, you will have to retrieve parameters from the MIPS world every time and write the results in the MIPS world. Note the presence of `UpdatePC` to increment the instruction counter: by default, the current statement is re-enabled upon returning an exception (especially for page faults).

All this only happens if the synchronous console already exists when the request is sent. You must therefore create it at system initialization:

(4.6) Edit the `threads/system.cc` file. Add a global definition

```
#ifdef CHANGED
#ifdef USER_PROGRAM
SynchConsole *synchconsole;
#endif
#endif
```

- ✓ Add the creation of the synchronous console in the `main` function just before the call to `StartProcess()` by using `NULL`, `NULL` as parameter to simplify.
- ✓ Add its destruction in the `Cleanup()` function before the destruction of the machine.
- ✓ Update the `system.h` file accordingly to declare this object. The `#ifdef USER_PROGRAM` is done only when you want to run a user program, i.e. you compile from `userprog`.

Launch `make` from `code`, and verify that `putchar` works.

Note: It is possible that the `./nachos -c` and `./nachos -sc` tests now behave strangely. According to your implementation, the standard input could be used at the same time by the `synchconsole` allocated for system calls and test console. You don't need to fix this.

5 From characters to strings

For now, you can only write one character at a time. Writing a chain comes down to making a series of character writings. The only problem is that we only have a user pointer to the string, not a kernel pointer.

(5.1) Take a look at the Linux part: complete the `SynchPutString` method of the `SynchConsole` class, which works on a Linux string.

(5.2) Write a procedure similar to `strcpy`:

```
int copyStringFromMachine(int from, char *to, unsigned size)
```

which copies a string character-by-character from the user world (MIPS) starting from the address `from` to the kernel world starting from the address `to`, using the `ReadMem` method, by stopping when it read a '`\0`'. At most "`size` " characters must be written, so this will typically be the size of the kernel buffer. A '`\0`' must be forced at the end of the copy, in last position, to ensure the system security.

The number of characters written must be returned. Beware of the `int *value` argument of `ReadMem`: why can not we just pass a pointer pointing inside the buffer `to`?

The choice of the file in which write this function is yours, several places are appropriate. You must motivate your choice in the report. Note: such a function could be used in the next TDs).

(5.3) Add the `PutString` system call, which uses `copyStringFromMachine`, then `SynchPutString`. A local buffer of size `MAX_STRING_SIZE` can be used, declaring this constant in the file `threads/system.h`.

Why would not it be reasonable to allocate, to a buffer, the same size than the MIPS string? Depending on how you allocated the buffer, make sure that the buffer is released after the system call is complete.

Show on some examples the behavior of your implementation, especially if the string is too long (and try to correct).

6 But how to stop?

What happens if you remove the call to `Halt()` at the end of the `main` function of `putchar.c`? Decrypt the error message and explain it. What to change to avoid this error?

You will no longer need to explicitly call `Halt()` in your programs.

How to take into account the return value `return n` of the `main` function if this is declared as a integer value? Look for `test` "who" call the `main` function.

7 Reading Functions

(7.1) Complete the `GetChar` system call. The register used for a return value, at the end of a function, is the register 2. the value read in the console must be placed in this register.

What do you do in case of end of file?

(7.2) Complete the `SynchGetString` method of the `SynchConsole` class.

(7.3) As for `GetChar`, complete the system call `void GetString(char * s, int n)` on the `fgets` model. Read the manual for end-of-line characters and overflows. Follow the same principle than `PutString`: a `copyStringToMachine` function must be defined.
Warning:

- ✓ You must absolutely ensure that there is no array overflow in the kernel.
- ✓ You may need to deallocate all temporary allocated structures to avoid memory leaks.
- ✓ Even if for now there is no user threads, try to take into account concurrent calls now. what would happen if multiple threads simultaneously called this function?

(7.4) (Bonus) Set up a system call `void PutInt(int n)` that writes a signed integer using the `snprintf` function to get the decimal writing. Same in the other direction with `void GetInt(int *n)` and the `sscanf` function.

8 Bonus: a `printf`

Calling successively `PutString` and `PutInt` is fastidious. You would like to be able to call `printf`, as usual.

> Why is it a bad idea to set up a `Printf` system call?

You will simply integrate a function `printf` in the user space:

(8.1) Download the Linux 2.2 sources from `http://www.kernel.org/pub/linux/kernel/v2.2/`, get the file `lib/vsprintf.c`, and comment the inclusions of the `linux/*.h` files. The `stdarg.h` file is provided by the MIPS cross-compiler.

(8.2) By implementing the `isxdigit`, `isdigit`, `islower`, `toupper` and `strnlen` functions and by modifying `test/Makefile` to automatically include `vsprintf.o` during the linking of your MIPS programs, such as `start.o`), but by excluding it from the PROGS list (using the `filter-out` function of `make`), you should be able to obtain a working `vsprintf` function then implement a `printf`. Use `man va_start` to initialize the `va_list`.