NachOS Subproject 2 : Console Input/Output

Year 2010/2011

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

Note *Attention : this subject requires you to be extremely methodical. Think carefully before starting to code, otherwise it will be even more difficult !*

In order to manage your code modifications, you should use a version management software like SVN or enclose your modifications in the following way

```
#ifdef CHANGED
  ...
#endif // CHANGED
```

This makes your changes reversible. By default, compile with the option `-D CHANGED`.

The purpose of this practical exercise is to implement a few basic system calls within NachOS. Explanations are deliberately left vague on several points. Therefore, some important design decisions are left to your judgement.

Partie I. What is the purpose ?

The purpose of this practical exercise is to implement a minimal input/output mechanism within NachOS, making it possible to execute the following small program called `putchar.c` (what output can reasonably be expected ?)

```
#include "syscall.h"

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

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

You should put this program `test/putchar.c` in the test directory and edit the `test/Makefile` if necessary.

Partie II. Asynchronous input/output

NachOS offers a primitive version of input/output for the `Console` class, which is defined within `machine/console.cc`. Read the comments carefully. The input/output works asynchronously via interruptions :

- To write a character, you "post" a write request via `Console::PutChar(char ch)`, and then you wait to be informed of completion of the request through execution of the `writeDone` handler.
- To read, wait until you are informed that there is something to read, through the execution of the handler `readAvail`, and then you perform the actual read operation through the function `Console::GetChar()`.

It is a mistake to try to read a character before being informed that a character is available, or to try to write a character before being informed that the preceding write operation has been completed. Explain why.

Note that the handlers are C functions, not C++, because they are shared by the console and the classes that use it.

Note also that there is no reason not to perform useful tasks between the moment that you "post" the request and the moment at which you are informed of its completion. You can perfectly well overlap communications and calculations !

Now look at the implementation in `userprog/progtest.cc`. First, we take a simple case in which we restrict ourselves to waiting for completion via semaphores.

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

static void ReadAvail(int arg) { readAvail->V(); }
static void WriteDone(int arg) { writeDone->V(); }
```

To wait, we take the semaphore. The notification handlers release them. Consequently, if the character is already present when a read request is presented, you immediately have what you are waiting for !

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

Action II.1. *Examine the program `userprog/progtest.cc\`. Run `\|./nachos âĂŞc`, which executes the procedure `consoleTest` (see `threads/main.cc`). Ensure that you fully understand what happens.*

Action II.2. *Edit `userprog/progtest.cc` to take account of correction completion of input : end of file or, on a TTY terminal, `^D` at line beginning.*

Action II.3. *Change `userprog/progtest.cc` in order to write `<c>` instead of `c` in the loop.*

Action II.4. *Try and do this with an input and an output file. For example, `nachos -c` in out (see `threads/main.cc`.)*

Partie III. Synchronous input/output

The goal is to place a synchronous input/output layer `SynchConsole` above the `Console` layer. The idea is that a *synchronous console* must encapsulate the entire semaphore mechanism, so as to only provide two functions. This is implanted just beside the `Console` class.

Action III.1. *Using the file `machine/console.h`, create the file `userprog/synchconsole.h` as follows. Note that the `#include "console.h"` operates correctly because of the search path specified in the call to the compiler.*

```
#ifdef CHANGED

#ifndef SYNCHCONSOLE_H
#define SYNCHCONSOLE_H

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

class SynchConsole {
  public:
    SynchConsole(char *readFile, char *writeFile);
                                // initialize the hardware console device
    ~SynchConsole();                    // clean up console emulation

    void SynchPutChar(const char ch);   // Unix putchar(3S)
    char SynchGetChar();                // Unix getchar(3S)

    void SynchPutString(const char *s); // Unix puts(3S)
    void SynchGetString(char *s, int n);       // Unix fgets(3S)
  private:
    Console *console;
};

#endif // SYNCHCONSOLE_H

#endif // CHANGED
```

Note that the semaphores must be shared between the objects of `SynchConsole` class and those of `Console` class. Therefore, they must be C functions and not C++ functions, unless sophisticated features of C++ are used (in fact, `SynchConsole` must be a child class of `Console`). The file `userprog/synchconsole.cc` 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 ReadAvail(int arg) { readAvail->V(); }
static void WriteDone(int arg) { writeDone->V(); }
```

```
SynchConsole::SynchConsole(char *readFile, char *writeFile)
{
  readAvail = new Semaphore("read avail", 0);
  writeDone = new Semaphore("write done", 0);
  console = ...
}

SynchConsole::~SynchConsole()
{
  delete console;
  delete writeDone;
  delete readAvail;
}

void SynchConsole::SynchPutChar(const char ch)
{
  // ...
}

char SynchConsole::SynchGetChar()
{
  // ...
}

void SynchConsole::SynchPutString(const char s[])
{
  // ...
}

void SynchConsole::SynchGetString(char *s, int n)
{
  // ...
}

#endif // CHANGED
```

Action III.2. *Complete the code of the character functions of* `synchconsole.cc`*.*

Action III.3. *If necessary, complete the file* `userprog/Makefile` *as well as the* `Makefile.common` *file that is included. Each time the* `console` *is used, you should add*`synchconsole`*. Beware of circular dependecies ! Attention à ne pas faire de circuits de dépendances :* `synchconsole` *depends on* `console` *but not the contrary ! Note that you need to recreate the dependencies :* `make clean` *for security, followed by* `make`*.*

Action III.4. *Change* `threads/main.cc` *so as to add a* `-sc` *option for the test of the synchronous console(launching of a* `SynchConsoleTest` *function).*

Action III.5. *Add the definiion of this function at the end of* `progtest.cc`*. For example :*

```
#ifdef CHANGED

void
```

```
SynchConsoleTest (char *in, char *out)
{
  char ch;

  SynchConsole *synchconsole = new SynchConsole(in, out);

  while ((ch = synchconsole->SynchGetChar()) != EOF)
      synchconsole->SynchPutChar(ch);
  fprintf(stderr, "Solaris: EOF detected in SynchConsole!\n");

}

#endif //CHANGED
```

Note that the `fprintf` is executed by Linux and not by Nachos!

Action III.6. *Add code to the `SynchConsoleTest` function as in the previous section.*

Partie IV. `PutChar` System Call

The purpose is now to implement a system call, `PutChar(char c)`, that takes as an argument a character c in user mode, and then sets `SyscallException` interrupt. This causes a switch to kernel mode and the execution of the standard handler. This handler has to call the function `SynchPutChar` and then return control to the calling program, taking care to increment the program counter. You will understand now why a system call is so costly. This is why Unix input/output is *buffered* : `fprintf(3)` is much less costly than `write(2)` for each character, because there is a system call for each line but not for each character .

With `PutChar` implemented, the above NachOS user program should work.

The first task is to insert the system call.

Action IV.1. *Edit the file `userprog/syscall.h` and add a system call `#define SC_ PutChar ...` and the corresponding `void function PutChar(char c)`. Here it is the user function Nachos : in Unix terms, it is `putchar(3)` (Enter the command `man 3 putchar` to check.)*

Now you need to write the code for the function `PutChar(char c)`. As it should cause a *trap*, this code has to be written in assembler.

Action IV.2. *Edit the file `test/start.S` and add the assembler definition for `PutChar`. You can copy this from `Halt`. Note that you put the number of the system call in register `r2` before calling the "magic" instruction `syscall`. The compiler puts the first argument `char c` in the `r4` register. This register is a 32-bit integer register, so the character is converted implicitly : `r4 = (int)c`. You must remember to perform the reverse conversion after the extraction.*

Now you have to insert the handler that is activated by the `syscall` interrupt.

Action IV.3. *Edit the file `userprog/exception.cc`. Convert the function `ExceptionHandler (ExceptionType which)` into a C/C++ switch because, of course, there will be many possible exceptions.*

Be careful to remember to increment the instruction counter : by default, the current instruction is reactivated on returning from an interrupt (think about page errors).

```
void
ExceptionHandler(ExceptionType which)
```

```
{
    int type = machine->ReadRegister(2);

#ifndef CHANGED // Noter le if*n*def

    if ((which == SyscallException) && (type == SC_Halt)) {
        DEBUG('a', "Shutdown, initiated by user program.\n");
        interrupt->Halt();
    } else {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }

#else // CHANGED

    if (which == SyscallException) {
      switch (type) {

      case SC_Halt: {
        DEBUG('a', "Shutdown, initiated by user program.\n");
        interrupt->Halt();
        break;
      }

      case SC_PutChar: {
        ...
      }

      default: {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
      }

      }

      UpdatePC();

    }
}

#endif // CHANGED
```

But all this will only work if the synchronous console already exists when the request is issued. Therefore, it must be created during the system initialization.

Action IV.4. *Edit `threads/system.cc`. Add a global declaration.*

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

```
#endif
```

Then add the creation of the object to the C function `Initialize(int argc, char **argv)`, and add its destruction to the C function `Cleanup()`. Update the file system.h as appropriate. Note the `#define USER_PROGRAM` : this change is only made when you want to execute a user program, i.e. you compile it from `userprog`.

Change directory to test and run putchar... What happens ?

Partie V. From Characters to Strings

For the time being, we can only write one character at a time. Writing a string involves a sequence of character-writing operations. The only problem is that you only have a MIPS pointer to the string, and not a Linux pointer.

Action V.1. *Write a procedure*

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

that copies a string from the MIPS mode to the Linux mode. No more than `size` characters are copied. A
`0` should be coerced at the end of the copying operation, in the last position, to ensure system security.

Action V.2. *Complete the code of the `SynchPutString` system call. You could use a local buffer of `MAX_STRING_SIZE` size, declaring this constant in the file `threads/system.h`. Ensure that you release the buffer once the system call has been completed.*

Action V.3. *Use a few examples to show how your implementation behaves, particularly in the case of a string that is too long.*

Partie VI. How To Stop ?

Action VI.1. *What happens if you remove the call to `Halt()` at the end of the main function of `putchar.c` ? Decode the error message and explain. What can you do not to call the `Halt()` function explicitly in your programs ? What can you do to take account of the value of `return n` from the main function, if it is declared to be an integer value ?*

Partie VII. Read Functions

Action VII.1. *Complete the system call `SynchGetChar`. The register used for returning a value at the end of a function is register 2 : this is where you have to place the value read from the terminal. Remember that a register is an integer, so do not overlook any necessary conversions. What will you do in case of an end of file ?*

Action VII.2. *Do the same thing for the system call void `SynchGetString(char *s, int n)`, modeling yourself on `fgets` (read attentively the manual on the management of end-of-line characters and overflows). Important : 1) You must absolutely guarantee that there is no overflow at the kernel level. 2) You should de-allocate all allocated temporary structures to prevent memory leaks. 3) You must take into account concurrent calls : what happens if several threads call this function at the same time ?*

Action VII.3. *Insert a system call void `SynchPutInt(int n)` that writes a signed integer using the function `snprintf` to obtain the external decimal write operation. Do the same for the other direction, with void `SynchGetInt(int *n)` and the `sscanf` function.*

Partie VIII. BONUS : EOF (End of File) Detection

Action VIII.1. *In all the above functions the* `'\127'` *(a letter "y" with a diaeresis accent) character is confused with an end of file. To prevent this you could add a function such as* `feof(3S)` *in order to detect the end of file. Add this function. (you may need to modify the* `Console` *class.)*

Action VIII.2. *Another option is to change the* `SynchGetChar` *function and make it return an integer as it is done in* `getchar(3S)`*. Implement this solution.*