Homework 5
Operating Systems, V22.0202
Fall 2007, Professor Yap

Due: Thu Nov 15, 2007.

Synchronization in the Toy OS

This homework is about synchronization of processes, and is in two parts. Part A requires some STML programming, and Part B requires modification to our Toy OS.

In the previous hw, you extended the Toy OS to a multi-programming OS. In order to ensure that all of us starts from the same base, I will provide you with a solution to this homework. You can download this as the module `class/os/2007/stm/stm1.2`. The original OS is called `tos`, and the multiprogramming version is called `tos2`. You can run `tos2` by typing `make t11`. Please do this first step right away.

You will also continue to work as a team, and use the same CVS repository for storing your project. In particular, all your work for this homework should be stored in a directory called `hw5` under your TEAM-CODE directory.

1 What is new in TOS2

After you download `stm1.2`, you can explore the various sample files and targets `t1, t2, t3, t4`, etc. Of particular interest is the target called `hello`. It runs the STML program called `hello.stm`, which simply prints the string "`Hello World!`".

This is new! In other words, we have enhanced TRAP 1 and TRAP 2 in THE Machine so that they can do string input/output.

The specs for the new TRAPs 1 and 2 are as follows: recall that previously we used R15, R14, R13 for these traps. We used R14 for the argument for input/output while R13 is used for return or error code. In the enhanced version, we also use R12. If R12 = 0, then the TRAPs behave as before. If R12 > 0, then we have string input/output. A string is just an array of ASCII characters that is terminated by a NULL character (= \0).

If the value of R12 is $n > 0$, then we want to input or output a string of length at most n. R14 stores the address of a buffer area in memory. The buffer area has at least $n + 1$ locations (the +1 is needed for a NULL character). In case of string input (R15=1), the TRAP routine will read a string from standard input and store the first n characters in the buffer area, terminated by the NULL character. In case of string output (R15=2), the TRAP routine will output the string stored in the buffer area (it will stop at the nth character or the first NULL character, whichever comes first). Please study the `hello.stm` program to make sure you understand the new trap.

2 Synchronization with Semaphores

You already know from hw4 that when we run several processes, their outputs gets intermixed and this is confusing. One trick is to set the quantum very high so that each process runs to completion before another process is started! Of course, this defeats the idea of fairness among processes. With our enhanced I/O, we can now write STML programs that print helpful strings (e.g., print the program's name before printing any output integer). But we need a bit more.

To address these issues, we introduce semaphores. Semaphores are basically kernel-level variables that are shared by processes. They can be used to synchronize processes to cooperate among themselves, e.g., to solve the mutual exclusion problem. In our example here, we want processes to synchronize their I/O.

Here is the semaphore system we want you to implement. There are 8 semaphores, indexed 0-7. Each semaphore stores a non-negative integer value, initialized to 1. There is a FIFO queue of processes associated with each semaphore. The operations on the semaphores are defined as follows.

```
DOWN(s):
        if (s > 0) s--;
        else block the current process;


UP(s):
        QQ = the queue of processes waiting for s;
        if (QQ is empty) then s++;
        else
            P = pop the first process off QQ;
            push P onto the end of the ready queue.
```

Semaphore operations are invoked by STML code by invoking traps. Specifically, to execute DOWN(s), load 3 into register R15, load the index of semaphore s into R14, and execute the TRAP instruction. Executing UP(s) is similar, except that we load 4 into R15.

Context Switches. A call to DOWN causes a context switch only if the current process becomes blocked. Otherwise, the rules for context switches is the same as for multiprogramming (i.e., quantum expiration or I/O Traps causes context switches).

Debugging output. If the debugging flag is 1 or 2, then TOS should generate a trace output each time an UP or DOWN occurs. If the operation causes a process to block or to unblock, this should also cause a trace output.

3 PART A: STML Programming

We have several sample STML programs in the `stm1.2` module (they have been upgraded to use the enhanced TRAP functions). We want you to modify two of them, `gcd2.stm` and `sort.stm`, in order to use semaphores. Note that `gcd2.stm` is the version of `gcd.stm` in which all the intermediate sequence of remainders are displayed. Modify the I/O in these programs as follows:

Input Synchronization: When reading inputs, first do a DOWN(0). Next print a short string like "gcd2 in: " or "sort in: ", and proceed to read the inputs. When done with reading, execute an UP(0).

Output Synchronization: When writing outputs, first do a DOWN(1). Next print a short string like "gcd2 out: " or "sort out: ", and proceed to write out the results. When done, execute an UP(1).

Thus, we use semaphore 0 to synchronize inputs, and semaphore 1 to synchronize outputs. But there is no synchronization between input and output.

Your "semaphore versions" should be called `gcd2-s.stm` and `sort-s.stm`. They will be used in Part B.

4 STML Assembler

David Stewart from a previous class has written an **assembler** in Perl script language for STML. You can find a copy of this code `assem.pl` in the module `class/os/2007/stm/assem`. This could speed up your coding in STML. You could download the Perl interpreter from Cygwin.

This software is provided without any warranties. Please use as you see fit, including ignoring it. But please send me any improvements that you might incorporate for the benefit of future students.

NOTE 1: when you submit your STML files, you are responsible for making sure that it human readable (each line should have the usual four parts: hex-code, hex-address, assembly code, comment. E.g.,

```
0x14F0  1    LOA R15 ONE   -- Set trap to input mode
0xF     2    TRP           -- Read A
```

NOTE 2: On some systems, the perl output contains a strange looking character that looks like ”$\hat{”}$ (on my computer, but others report a blank box symbol). This is apparently some encoding of a white space character. Just think of this as a space. The system will treat it as a space. Before you submit your STML files, you need to use an editor to clean up the output.

5 PART B: IMPLEMENTING SEMAPHORES

What you mainly need to do is:

- Implement TRAP 3 and 4.
- Implement data structures for the semaphores and the waiting queues.
- Write procedures for DOWN and UP, and invoke them in `exec_trap()`.
- Fix the context switching procedure to handle the new cases.
- Initialize the semaphores at the beginning of execution.

Note that, unfortunately, the `exec_trap()` routine is in `the.c` and hence you need to modify our ”hardware” to do this problem. But you must not modify `the.h` or any other routines in `the.c`. Your implementations must be encoded in the files called **the3.c, tos3.h, tos3.h**.

Submitting your work:

- Submission and grading are very similar to hw4.
- All files for this hw should be stored under `$TEAMCODE/hw5`. In addition to the files mentioned above, you should include the usual README and Makefile. As usual README must give an overview of your work, and your data structures and key routines must be identified and explained here.
- Please make sure that all code are properly commented (at the high level of overview as well as local comments.
- Your Makefile should be an extension of the Makefile in `stm1.2`, and it should keep the original tos and tos2 targets around. But you should include additional targets:
  (a) A non-default target called `tos3` to compile `tos3`.
  (b) The default target should simply run tos3 using your STML programs `gcd2-s.stm` and `sort-s.stm`.

Grading.

- This homework is worth 150 points, and graded similarly to hw4.
- Following our submission instructions is worth up to 20 points (i.e., Makefile, README, etc).
- PART A is worth 30 points: for the ”fib.stm” code and its Makefile target.
- The correctness of ”tos3” is worth 70 points.
- Good program structure and adequate commenting are worth up to 20 points.
- Your ”tos3” program will be tested on examples other than those provided here. Worth 20 points.
- If ”tos3” does not compile, you get a maximum of 30 points for PART B.