Lab 11 Design Problem: Tiny OS

Useful links:

- Introduction to BSim (HTML)
- Summary of Beta Instruction Formats (PDF)
- Beta Documentation (PDF)

To complete this design problem, please follow the instructions, which will ask you to modify the TinyOS code to add some new functionality. There is no automated test for the lab. Instead, please see lower section of the main Lab 11 webpage where you'll find an honor-code checkoff that will let you indicate which parts of the lab you've completed.

NOTE: since there is no automated test, you'll always see the green checkmark after you submit. To actually earn points you need to complete the honor-code checkoff on the main Lab 11 webpage.

Instructions

1. The initial configuration of the lab includes a copy the code from tinyOS.uasm in the Lab 11 tab in the BSim window
2. Assemble and run the code; look through the code to figure out how it works
3. Modify your copy of the code as described below and test your changes
4. There is no automated check-in for the lab. When you've completed a step complete the corresponing honor-code checkoff on the main Lab 11 web.page.

Step 0: Run initial Tiny OS code and figure out how it works

TinyOS is a program that implements a simple timesharing system. When you use BSim to load, assemble and then run the TinyOS code the following prompt should appear in the console pane of the Bsim Display Window:

```
Start typing, Bunky.

00000000>
```

As you type, each character is echoed to the console and when you hit return the whole sentence is translated into Pig Latin and written to the console:

```
00000000> 6.004 is a fun course
```

```
6.004AY ISAY AAY UNFAY OURSECAY

00001F61>
```

The hex number written out as part of the prompt is a count of the number of times one of the user-mode processes has been scheduled while you typed in the sentence.

TinyOS implements the following functionality:

- Kernel-mode interrupt routine for handling input from the keyboard. Incoming characters are stored in a kernel buffer for subsequent use by user-mode programs (see the `GetKey()` supervisor call).
- Kernel-mode supervisor call dispatching and a repertoire of call handlers that provide simple I/O services to user-mode programs. Handlers include
  - `Halt()` -- stop a user-mode process
  - `WrMsg()` -- write a null-terminated ASCII string to the console. The string immediately follows the WrMsg() instruction; execution resumes with the instruction following the string. For example:

    ```
    WrMsg()
    .text "This text is sent to the console...\n"
    ...next instruction...
    ```

  - `WrCh()` -- write the ASCII character found in `R0` to the console
  - `GetKey()` -- return the next ASCII character from the keyboard in `R0`; this call does not return to the user until there is a character available.
  - `HexPrt()` -- convert the value passed in `R0` to a hexadecimal string and output it to the console.
  - `Wait()` -- implement atomic WAIT operation on the integer semaphore whose address is passed in `R3`.
  - `Signal()` -- implement atomic SIGNAL operation on the integer semaphore whose address is passed in `R3`.
  - `Yield()` -- immediately schedule the next user-mode program for execution. Execution will resume in the usual fashion when the round-robin scheduler chooses this process again.
- A round-robin scheduler and the necessary kernel data structures to support multiple user-mode programs. The scheduler is invoked by periodic clock interrupts or when a user-mode program makes a `Yield()` supervisor call.

TinyOS also contains code for three programs each of which runs in a separate user-mode process:

- Process 0: prompts the user for new lines of input. Reads lines from the keyboard using the `GetKey()` supervisor call and sends them to Process 1 using the `Send()` procedure. `Send()` implements a bounded buffer synchronized through the use of semaphores.
- Process 1: reads lines of inputs from the bounded buffer (using the `Rcv()` procedure), translates them into Pig Latin and types them out on the console (using the `WrMsg()` and `WrCh()` supervisor calls).
- Process 2: Each time this process is executed, it simply increments a counter and uses the `Yield()` supervisor call to give up the remainder of its quantum. This count is used as part of the prompt displayed by Process 0.

Step 1: Add mouse interrupt handler

When you click the mouse over the console pane, BSim generates an interrupt, forcing the PC to `0x800000010` and saving PC+4 of the interrupted instruction in the XP register. Note that BSim implements a "vectored interrupt" scheme where different types of interrupts force the PC to different addresses (rather than having all interrupts for the PC to `0x80000008`). The following table shows how different exceptions are mapped to PC values:

`0x80000000` reset
`0x80000004` illegal opcode
`0x80000008` clock interrupt*
`0x8000000C` keyboard interrupt**
`0x80000010` mouse interrupt**

* must specify `.options clk` to enable
** must specify `.options tty` to enable

Recall that only user-mode programs can be interrupted. Interrupts signaled while in the Beta is running in kernel-mode (*e.g.*, handling another interrupt or servicing a supervisor call) have no effect until the processor returns to user-mode.

The original TinyOS code prints out "Illegal interrupt" and then halts if a mouse interrupt is received. Change this behavior in your copy of the code by adding an interrupt handler that stores the click information in a new kernel memory location and then returns to the interrupted process.

You might find the keyboard interrupt handler a good model to follow.

We've added a new Beta instruction your interrupt handler can use to retrieve information about the last mouse click:

```
CLICK()
```

This instruction can only be executed when in kernel mode (*e.g.*, from inside an interrupt handler). It returns a value in `R0`: -1 if there hasn't been a mouse click since the last time `CLICK()` was executed, or a 32-bit integer with the X coordinate of the click in the high-order 16 bits of the word, and the Y coordinate of the click in the low-order 16 bits. The coordinates are non-negative and relative to the upper left hand corner of the console pane. In our scenario, `CLICK()` is only called after a mouse click, so we should never see -1 as a return value.

Testing your implementation: insert a `.breakpoint` before the `JMP(XP)` at the end of your interrupt handler, run the program and click the mouse over the console pane. If things are working correctly the simulation should stop at the breakpoint and you can examine the kernel memory location where the mouse info was stored to verify that it's correct. Continuing execution (click the "Run" button in the toolbar at the top of the window) should return to the interrupted program. When you're done remember to remove the breakpoint.

Step 2: Add Mouse() supervisor call

Implement a `Mouse()` supervisor call that returns the coordinate information from the most recent mouse click (i.e., the information stored by the mouse interrupt handler). Like the `GetKey()` supervisor call, a user-mode call to `Mouse()` should consume the available click information. If no mouse click has occurred since the previous call to `Mouse()`, the supervisor call should "hang" until new click information is available. "Hang" means that the supervisor call should back up the saved PC so that the next user-mode instruction to be executed is the `Mouse()` call and then branch to the scheduler to run some other user-mode program. Thus when the calling program is rescheduled for execution at some later point, the `Mouse()` call is re-executed and the whole process repeated. From the user's point of view, the `Mouse()` call completes execution only when there is new click information to be returned. The `GetKey()` supervisor call is a good model to follow.

Notes: Supervisor calls are actually unimplemented instructions that cause the expected trap when executed. The illegal instruction trap handler looks for illegal instructions it knows to be supervisor calls and

calls the appropriate handler -- look at `SVC_UU0` for details. To define a new supervisor call, add the following definition just after the definition for `Yield()`:

```
.macro Mouse()   SVC(8)
```

This is the ninth supervisor call and the current code at `SVC_UU0` was tailored for processing exactly eight supervisor calls, so you'll need to make the appropriate modifications.

Testing your implementation: Once your `Mouse()` implementation is complete, add a `Mouse()` instruction just after `P2Start`. If things are working correctly, this user-mode process should now hang and `Count3` should not be incremented even if you type in several sentences (*i.e.*, the prompt should always be `00000000>`). Now click the mouse once over the console pane and then type more sentences. The prompt should read `00000001>`. When you're done, remember to remove the `Mouse()` instruction you added.

Step 3: Add fourth user-mode process that reports mouse clicks

Modify the kernel to add support for a fourth user-mode process. Add user-mode code for the new process that calls `Mouse()` and then prints out a message of the form:

```
Click at x=000000EE, y=00000041
```

Each click message should appear on its own line (*i.e.*, it should be preceded and followed by a newline character). You can use `WrMsg()` and `HexPrt()` to send the message; see the code for Process 0 for an example of how this is done.

Testing your implementation: If all three parts are working correctly the appropriate message should be printed out whenever you click the mouse over the console pane. You may find it necessary to use `.breakpoint` commands to debug your user-mode code.

Step 4: Synchronize mouse reporting with other I/O

Using semaphores, coordinate the operation of the user-mode processes so that click messages only appear after the prompt has been output but before you've started typing in a sentence to be translated. In other words, once you start typing in a sentence, click messages should be delayed until after the next prompt. If the user clicks multiple times after they have started typing, the expected behavior is two click messages: one for the first click and one for the last click.

If the user hasn't started typing and they click, then they should see the coordinate printout right away. In other words, the coordinate printout should happen immediately unless it's delayed because the user has started typing in a phrase to be translated.

Testing your implementation: Start typing in a sentence, then click the mouse. The click message should be printed after the translation and the following prompt have been printed.