Assignment 5 — Traps and Interrupts

Due: Wednesday, May 11th

In a real computer, there are typically far more running processes than there are physical CPU cores. In such situations, rather than having a process waste resources by polling while it waits for user input, another process can be allowed to run during that time[1].

Deliverables:

GitHub Classroom: `https://classroom.github.com/a/prRDtFJH`

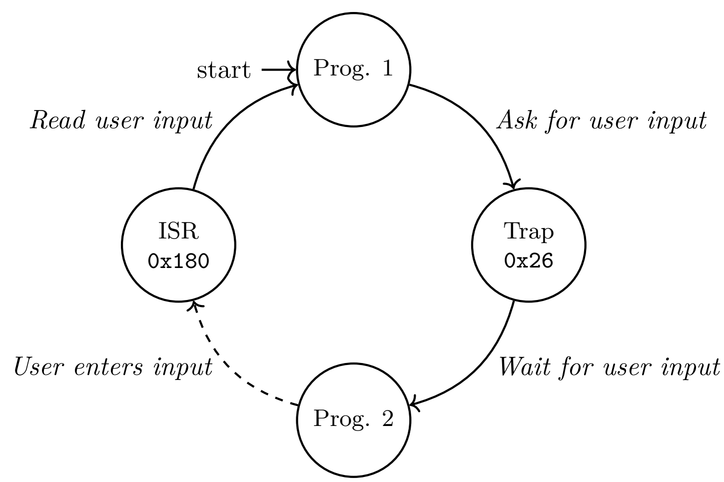
Required Files: `trap26.asm`

Optional Files: *none*

Part 1: Time-Sharing

Implementing interrupt-driven I/O requires a few extra steps. Notably, recall that in order to enable keyboard interrupts, bit 14 must be set to a '`1`' in the KBSR. However, the KBSR is memory-mapped to `0xFE00`, which is only accessible in privileged mode.

In order to properly use interrupts, you will need to implement *both* a trap service routine and an interrupt service routine, which will need to work together to transition to and from the calling program:



1. Execution begins with Program 1, which reads a single character. Rather than using `GETC`, Program 1 instead calls Trap `0x26`.
2. The trap service routine sets up keyboard interrupts, but it does *not* return to Program 1 (which was its caller). Instead, it "returns" to Program 2.
3. Program 2 is allowed to execute while Program 1 is waiting for input.

   Once the user types a character, an interrupt is triggered; ISR `0x180` responds.
4. The interrupt service routine reads the typed character, but it does *not* return to Program 2 (which was interrupted). Instead, it "returns" to Program 1.

From the perspective of Program 1, Trap `0x26` and ISR `0x180` behave as two halves of a single service routine.

[1]Certainly, the mechanism implemented here is a gross oversimplification of how a real operating system implements *time-sharing*. However, within the confines of the LC-3, it will illustrate the fundamentals.

Part 2: Trap `0x26`

Complete the service routine labeled `TRAP26` in `trap26.asm`. This service routine is responsible for setting up keyboard interrupts and switching to execution of Program 2:

- Your implementation must place the address of `ISR180` (which you will implement shortly) into the keyboard's entry in the Interrupt Vector Table.
- It must enable keyboard interrupts by setting bit 14 of the KBSR to '`1`'.
- It must "return" to Program 2. When `TRAP26` is called, the supervisor stack will contain (left):

…the service routine must swap Program 1's PC and PSR with those of Program 2 (right), which will cause an `RTI` to return to the latter. Moreover, `TRAP26` must store all of Program 1's state:

- PC
- PSR
- `R1`
- `R2`
- `R3`
- `R4`
- `R5`
- `R7`
- Keyboard's IVT entry

…the state of Program 2 must then be loaded[2]:

- PC
- PSR
- `R0`
- `R1`
- `R2`
- `R3`
- `R4`
- `R5`
- `R7`

Note that Program 1's `R0` need not be saved; it will eventually be used to return a typed character. You may also assume that `R6` is used exclusively for the supervisor stack, and should not be touched.

Part 3: ISR `0x180`

Complete the service routine labeled `ISR180` in `trap26.asm`. This service routine is responsible for responding to keyboard interrupts and switching back to execution of Program 1:

- Your implementation must read the typed character into `R0`. It may *not* call `GETC` or use polling.
- It must disable keyboard interrupts by setting bit 14 of the KBSR to '`0`'.
- It must "return" to Program 1. When `ISR180` is called, the supervisor stack will contain (left):

…the service routine must swap Program 2's PC and PSR with those of Program 1 (right), which will cause an `RTI` to return to the latter. Moreover, `ISR180` must store all of Program 2's state:

- PC
- PSR
- `R0`
- `R1`
- `R2`
- `R3`
- `R4`
- `R5`
- `R7`

…the state of Program 1 must then be loaded:

- PC
- PSR
- `R1`
- `R2`
- `R3`
- `R4`
- `R5`
- `R7`
- Keyboard's IVT entry

That is to say, `ISR180` must restore all of Program 1's state that was saved by `TRAP26`, and it must save all of Program 2's state that will be restored if `TRAP26` is ever called again.

[2] The initial state of Program 2 has been defined in `trap26.asm`, so that the first call to `TRAP26` will begin Program 2.

Part 4: Testing

Consider the code in `program1a.asm` and `program2.asm`. Here, Program 1 uses `TRAP x26` to read a single character using interrupt-driven I/O. Program 2 simply infinitely, periodically increments the contents of `R1`.

Not only must `program1a`, `program2`, and `trap26` be assembled and loaded into the LC-3, but memory location `0x0026` must be manually set to `0x1000` to enable `TRAP x26`.

- Throughout this assignment, you may *not* alter any of the code for Programs 1 or 2.
- Depending on when characters are typed, the state of Program 2 as saved by `ISR180` and restored by `TRAP26` will vary. It need *not* be possible to rerun Program 1 by manually resetting the PC to `0x3000`.

You may assume that Program 2 will neither alter the keyboard's entry in the Interrupt Vector Table nor itself use `TRAP x26`. You may further assume that the user will wait long enough to type a character that a keyboard interrupt will not be triggered while `TRAP26` is running.

For example:

```
[Program 1] Reading...
[Program 2] R1: 0...
[Program 2] R1: 1...
[Program 2] R1: 2...
[Program 1] Read 'a'.
```

...where the number of lines printed will naturally depend on how long the user waits before typing a character.

Your implementation will be tested with other variations of Program 1 and Program 2. For example, consider `program1b.asm`, which invokes `TRAP x26` within an infinite loop:

```
[Program 1] Reading...
[Program 2] R1: 0...
[Program 2] R1: 1...
[Program 1] Read 'f'.
[Program 1] Reading...
[Program 2] R1: 2...
[Program 1] Read 'o'.
[Program 1] Reading...
[Program 2] R1: 3...
[Program 2] R1: 4...
[Program 2] R1: 5...
[Program 1] Read 'o'.
[Program 1] Reading...
[Program 2] R1: 6...
            ⋮
```

Part 5: Submission

The following files are required and must be pushed to your GitHub Classroom repository by the deadline:

- `trap26.asm` — A working assembly trap and ISR for interrupt-driven keyboard input, as specified.

The following files are optional:

- *none*

Any files other than these will be ignored.