Homework

This program, `x86.py`, allows you to see how different thread interleavings either cause or avoid race conditions. See the README for details on how the program works and its basic inputs, then answer the questions below.

Questions

1. To start, let's examine a simple program, "loop.s". First, just look at the program, and see if you can understand it: `cat loop.s`. Then, run it with these arguments:

   ```
   ./x86.py -p loop.s -t 1 -i 100 -R dx
   ```

   This specifies a single thread, an interrupt every 100 instructions, and tracing of register `%dx`. Can you figure out what the value of `%dx` will be during the run? Once you have, run the same above and use the `-c` flag to check your answers; note the answers, on the left, show the value of the register (or memory value) *after* the instruction on the right has run.

2. Now run the same code but with these flags:

   ```
   ./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx
   ```

   This specifies two threads, and initializes each %dx register to 3. What values will `%dx` see? Run with the `-c` flag to see the answers. Does the presence of multiple threads affect anything about your calculations? Is there a race condition in this code?

3. Now run the following:

   ```
   ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx
   ```

   This makes the interrupt interval quite small and random; use different seeds with `-s` to see different interleavings. Does the frequency of interruption change anything about this program?

4. Next we'll examine a different program (`looping-race-nolock.s`). This program accesses a shared variable located at memory address 2000; we'll call this variable `x` for simplicity. Run it with a single thread and make sure you understand what it does, like this:

   ```
   ./x86.py -p looping-race-nolock.s -t 1 -M 2000
   ```

   What value is found in `x` (i.e., at memory address 2000) throughout the run? Use `-c` to check your answer.

5. Now run with multiple iterations and threads:

```
./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000
```

   Do you understand why the code in each thread loops three times? What will the final value of `x` be?

6. Now run with random interrupt intervals:

```
./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0
```

   Then change the random seed, setting `-s 1`, then `-s 2`, etc. Can you tell, just by looking at the thread interleaving, what the final value of `x` will be? Does the exact location of the interrupt matter? Where can it safely occur? Where does an interrupt cause trouble? In other words, where is the critical section exactly?

7. Now use a fixed interrupt interval to explore the program further. Run:

```
./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1
```

   See if you can guess what the final value of the shared variable `x` will be. What about when you change `-i 2`, `-i 3`, etc.? For which interrupt intervals does the program give the "correct" final answer?

8. Now run the same code for more loops (e.g., set `-a bx=100`). What interrupt intervals, set with the `-i` flag, lead to a "correct" outcome? Which intervals lead to surprising results?

9. We'll examine one last program in this homework (`wait-for-me.s`). Run the code like this:

```
./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000
```

   This sets the `%ax` register to 1 for thread 0, and 0 for thread 1, and watches the value of `%ax` and memory location 2000 throughout the run. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?

10. Now switch the inputs:

```
./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000
```

   How do the threads behave? What is thread 0 doing? How would changing the interrupt interval (e.g., `-i 1000`, or perhaps to use random intervals) change the trace outcome? Is the program efficiently using the CPU?