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. First let's get ready to run `x86.py` with the flag `-p flag.s`. This code "implements" locking with a single memory flag. Can you understand what the assembly code is trying to do?
2. When you run with the defaults, does `flag.s` work as expected? Does it produce the correct result? Use the `-M` and `-R` flags to trace variables and registers (and turn on `-c` to see their values). Can you predict what value will end up in `flag` as the code runs?
3. Change the value of the register `%bx` with the `-a` flag (e.g., `-a bx=2,bx=2` if you are running just two threads). What does the code do? How does it change your answer for the question above?
4. Set `bx` to a high value for each thread, and then use the `-i` flag to generate different interrupt frequencies; what values lead to a bad outcomes? Which lead to good outcomes?
5. Now let's look at the program `test-and-set.s`. First, try to understand the code, which uses the `xchg` instruction to build a simple locking primitive. How is the lock acquire written? How about lock release?
6. Now run the code, changing the value of the interrupt interval (`-i`) again, and making sure to loop for a number of times. Does the code always work as expected? Does it sometimes lead to an inefficient use of the CPU? How could you quantify that?
7. Use the `-P` flag to generate specific tests of the locking code. For example, run a schedule that grabs the lock in the first thread, but then tries to acquire it in the second. Does the right thing happen? What else should you test?
8. Now let's look at the code in `peterson.s`, which implements Peterson's algorithm (mentioned in a sidebar in the text). Study the code and see if you can make sense of it.
9. Now run the code with different values of `-i`. What kinds of different behavior do you see?
10. Can you control the scheduling (with the `-P` flag) to "prove" that the code works? What are the different cases you should show hold? Think about mutual exclusion and deadlock avoidance.

11. Now study the code for the ticket lock in `ticket.s`. Does it match the code in the chapter?

12. Now run the code, with the following flags: `-a bx=1000,bx=1000` (this flag sets each thread to loop through the critical 1000 times). Watch what happens over time; do the threads spend much time spinning waiting for the lock?

13. How does the code behave as you add more threads?

14. Now examine `yield.s`, in which we pretend that a `yield` instruction enables one thread to yield control of the CPU to another (realistically, this would be an OS primitive, but for the simplicity of simulation, we assume there is an instruction that does the task). Find a scenario where `test-and-set.s` wastes cycles spinning, but `yield.s` does not. How many instructions are saved? In what scenarios do these savings arise?

15. Finally, examine `test-and-test-and-set.s`. What does this lock do? What kind of savings does it introduce as compared to `test-and-set.s`?