

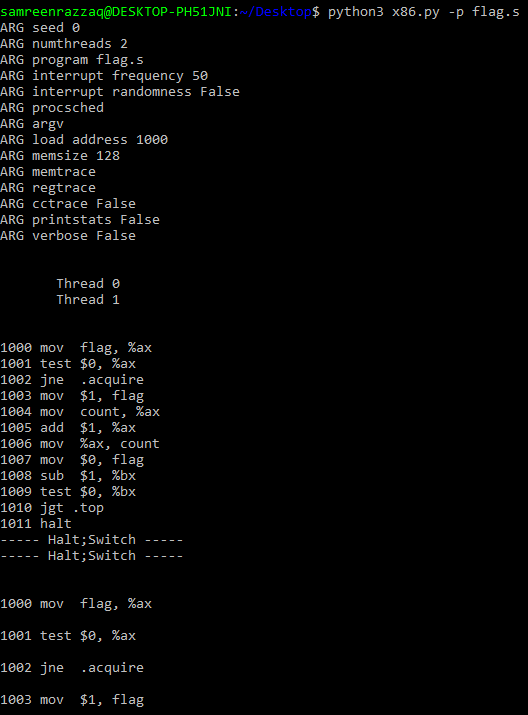
**Lab Task:**

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

**Tasks:**

1. Examine flag.s. This code “implements” locking with a single memory flag. Can you understand the assembly?

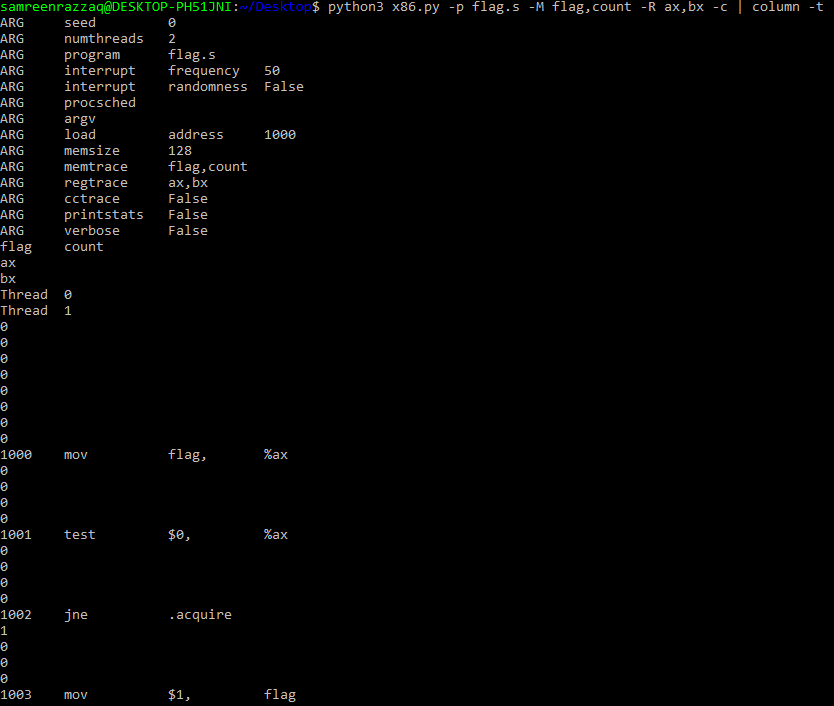
**Solution:**

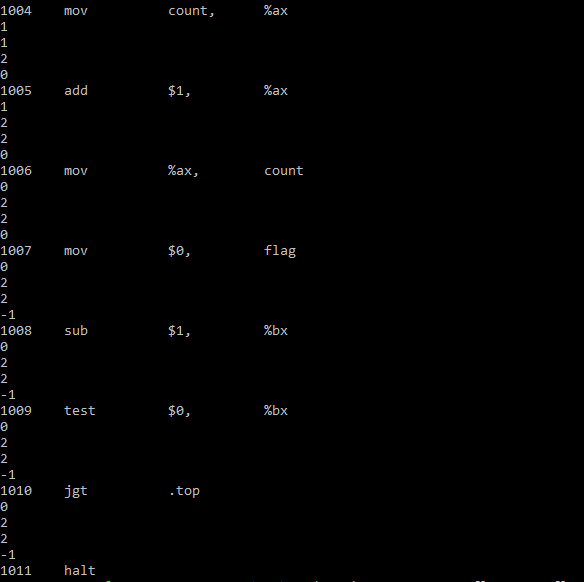


The assembly code in `flag.s` implements a basic locking mechanism using a single memory flag (`flag`). Each thread attempts to acquire the lock, increment a counter (`count`) when the lock is acquired, and then release the lock. The code employs a loop to decrement a value (`%bx`) until it becomes zero. The provided output indicates two threads executing the code concurrently, with occasional switches between them.

1. When you run with the defaults, does flag.s work? 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?

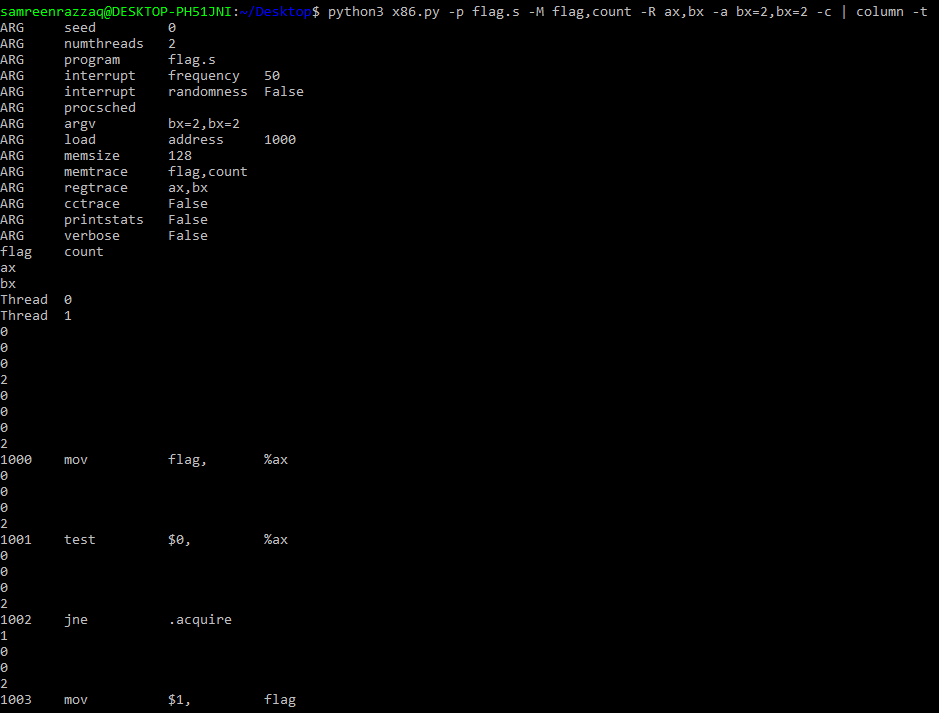
**Solution:**

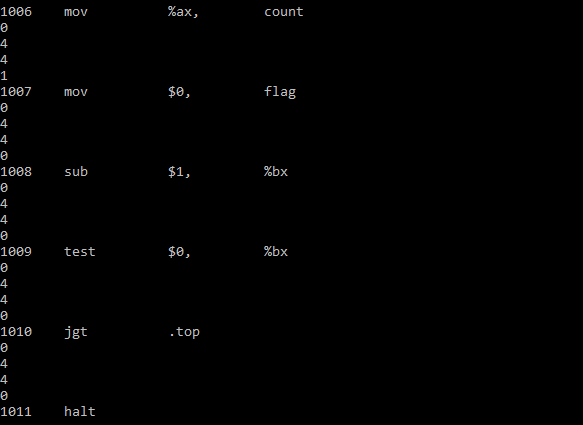




To predict the final value of flag, consider the locking mechanism implemented in the code. The program seems to acquire and release the lock using the flag variable. The final value of flag may be 0, indicating that the lock has been successfully released.

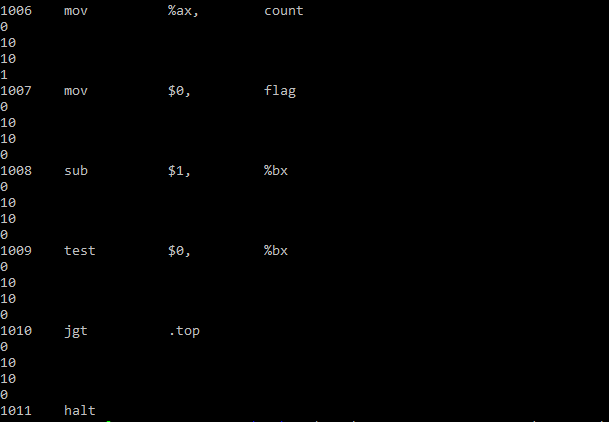
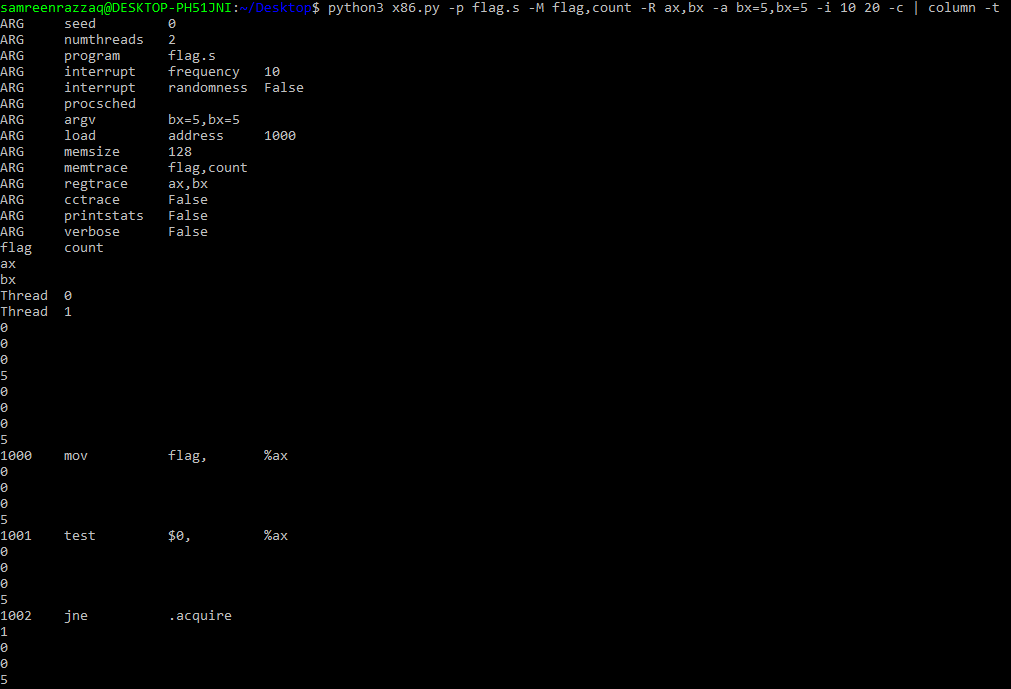
1. Change the value of the register %bx with the -a flag (e.g., -a bx=2, bx=2 if you are running just two threads). What does the code do? How does it change your answer for the question above?

**Solution:**



The instruction `-a bx=2,bx=2` appears to be part of the command line arguments provided when running the Python script `x86.py`. This argument likely signifies that the initial value of the `%bx` register for both threads should be set to 2. In the context of the provided x86 assembly code trace, initializing `%bx` to 2 for both threads establishes an initial condition for the simulation. The x86 assembly program, which involves manipulating variables like `flag` and `count` through conditional jumps and arithmetic operations, would then operate with this starting point. The specific effect on the program's behavior would depend on how the program utilizes the initial value of `%bx` in its execution. Changing the initial value of `%bx` may influence the program's control flow and final outcomes, and it's essential to consider this initialization when interpreting the subsequent register states and the overall behavior of the simulated program.

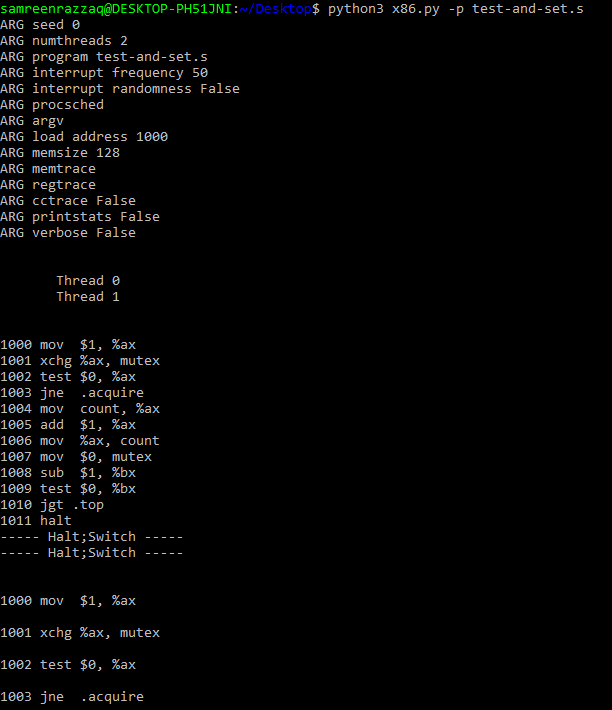
1. Set bx to a high value for each thread, and then use the -i flag to generate different interrupt frequencies; what values lead to a bad outcomes? Which lead to good outcomes?

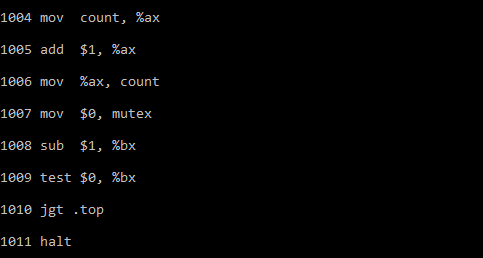
**Solution:**

It introduces varied interrupt frequencies, with Thread 0 experiencing interrupts every 10 cycles and Thread 1 every 20 cycles. The choice of a higher initial `%bx` value and different interrupt frequencies influences the simulated program's behavior, potentially affecting synchronization, thread interleaving, and the overall outcomes. The command then uses `-c` to display the dynamic states of registers, facilitating the observation and analysis of the program's execution, and `column -t` formats the output for clearer readability. The determination of "good" or "bad" outcomes depends on the specific program's objectives and desired results in variables such as `flag` and `count`.

1. 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?

**Solution:**

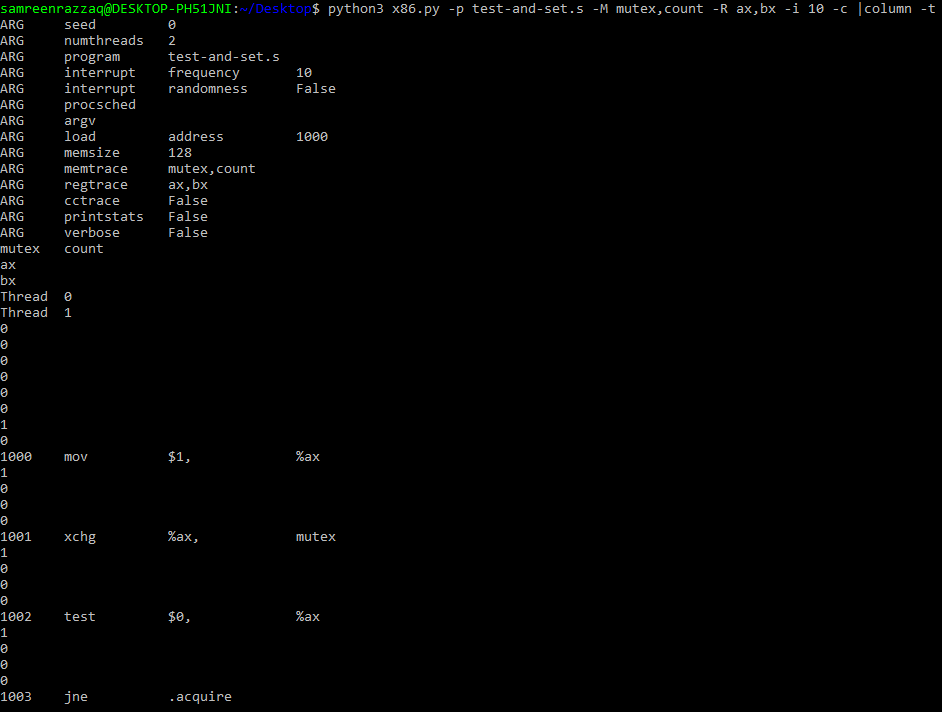


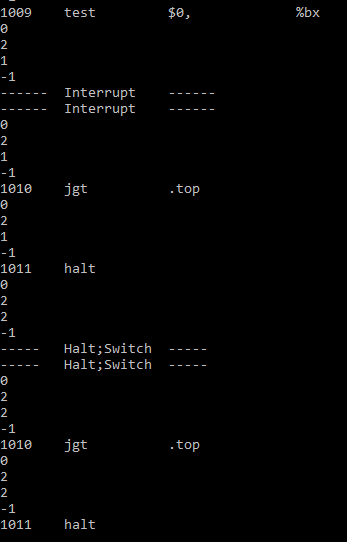


The command `python3 x86.py -p test-and-set.s` executes the x86 assembly program `test-and-set.s`, which employs the `xchg` instruction to construct a basic locking primitive. The program defines a lock acquisition mechanism where Thread 0 sets the value in the `mutex` variable to 1 using `xchg` and checks if the previous value was 0. If it was, the thread proceeds to the critical section; otherwise, it repeats the process. The lock release is achieved by resetting the `mutex` to 0. Thread 1 operates similarly but with a different initial state for the `mutex`.

1. 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?

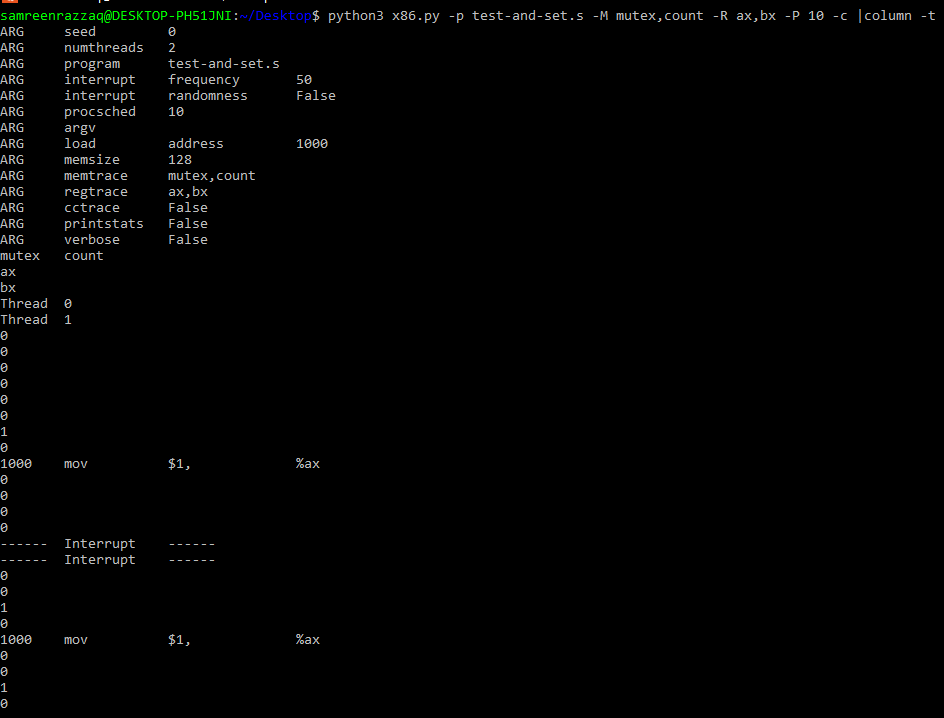
**Solution:**

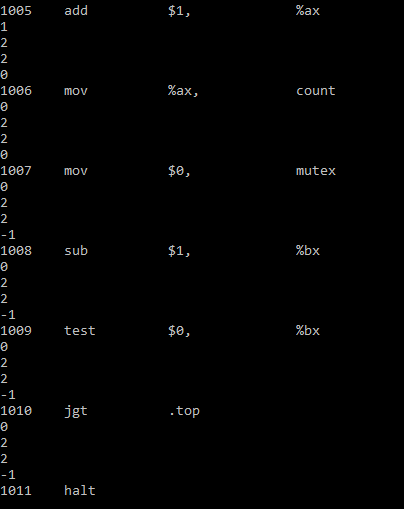




The command `python3 x86.py -p test-and-set.s -M mutex,count -R ax,bx -i 10 -c | column -t` runs the x86 assembly program `test-and-set.s` with a 10-cycle interrupt interval, capturing register and memory states. This modification allows observation of the program's behavior under interruptions. To assess its efficiency and consistency, multiple runs are needed to analyze potential contention for the lock and delays in accessing the critical section due to interrupts. Quantifying inefficiency could involve measuring the time spent waiting for the lock and analyzing interrupt interval distribution to evaluate their impact on program performance.

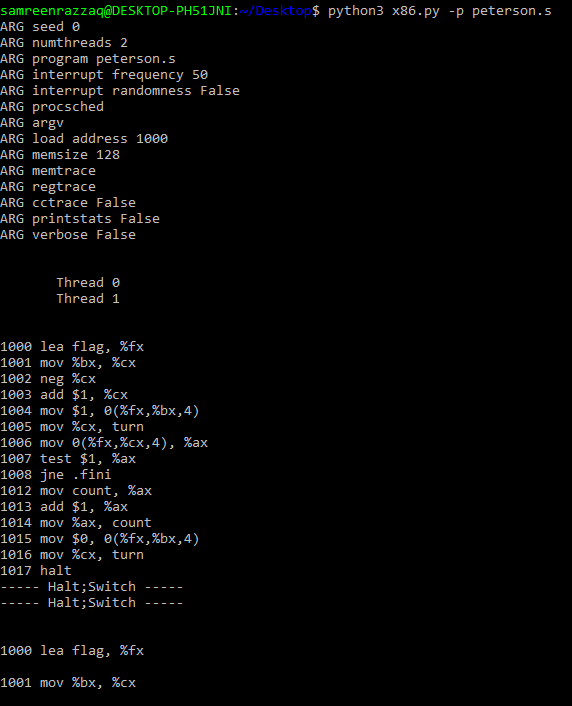
1. Use the -P flag to generate specific tests of the locking code. For example, run a schedule that grabs the lock in the first thread, but then tries to acquire it in the second. Does the right thing happen? What else should you test?

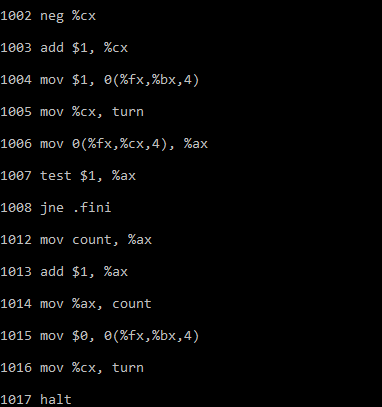
**Solution:**



With `-P 10`, it schedules scenarios where the lock is initially acquired in the first thread and then attempted in the second. The output, formatted for clarity with `column -t`, presents a detailed register and memory trace for analysis. This setup allows testing the correctness of the locking mechanism under different conditions, such as concurrent lock access by multiple threads. Further testing could include variations in initial conditions, interrupt intervals, and diverse scheduling scenarios to ensure the robustness of the locking code.

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.

**Solution:**

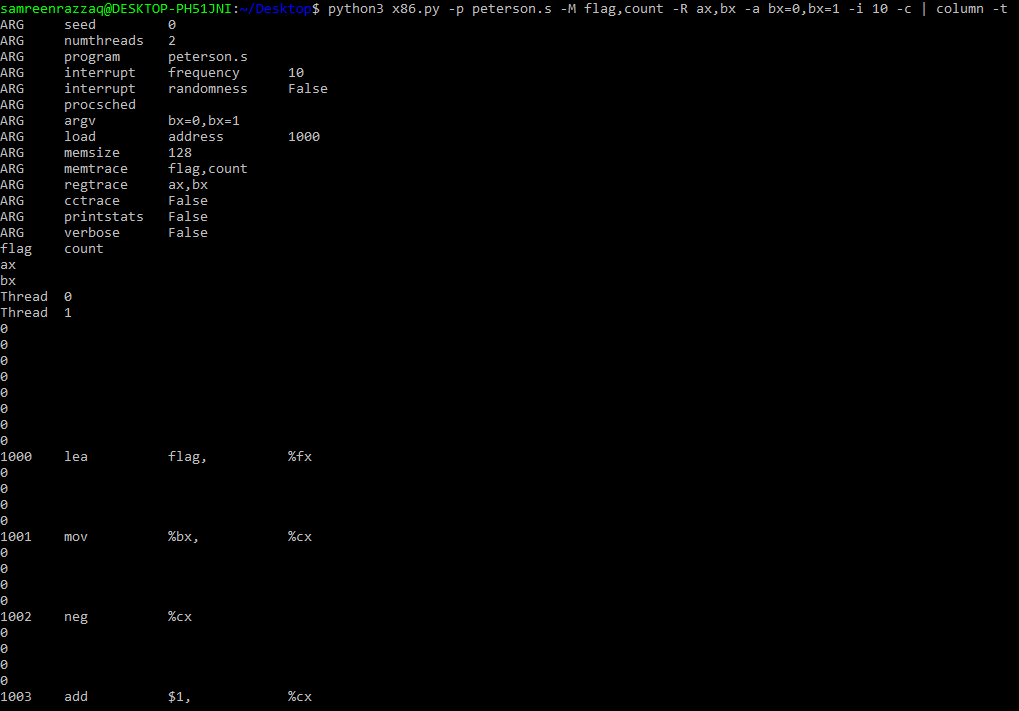


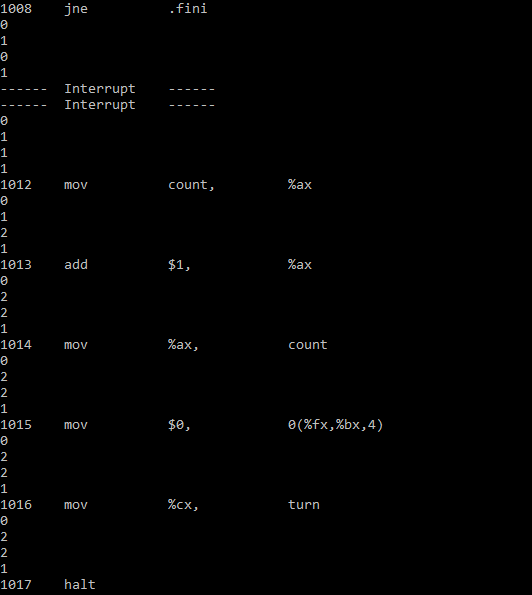
This program implements Peterson's algorithm, as mentioned in a sidebar in the text. Peterson's algorithm is a synchronization algorithm designed for two processes to share a critical section without interference. The code likely involves the use of shared variables and atomic instructions to ensure mutual exclusion. By running this command, the x86 assembly program will be simulated, providing a dynamic trace of register and memory states to facilitate the study and comprehension of Peterson's algorithm implementation within the code.

1. Now run the code with different values of -i. What kinds of different behavior do you see? Make sure to set the thread IDs appropriately (using -a bx=0, bx=1 for example) as the code assumes it.

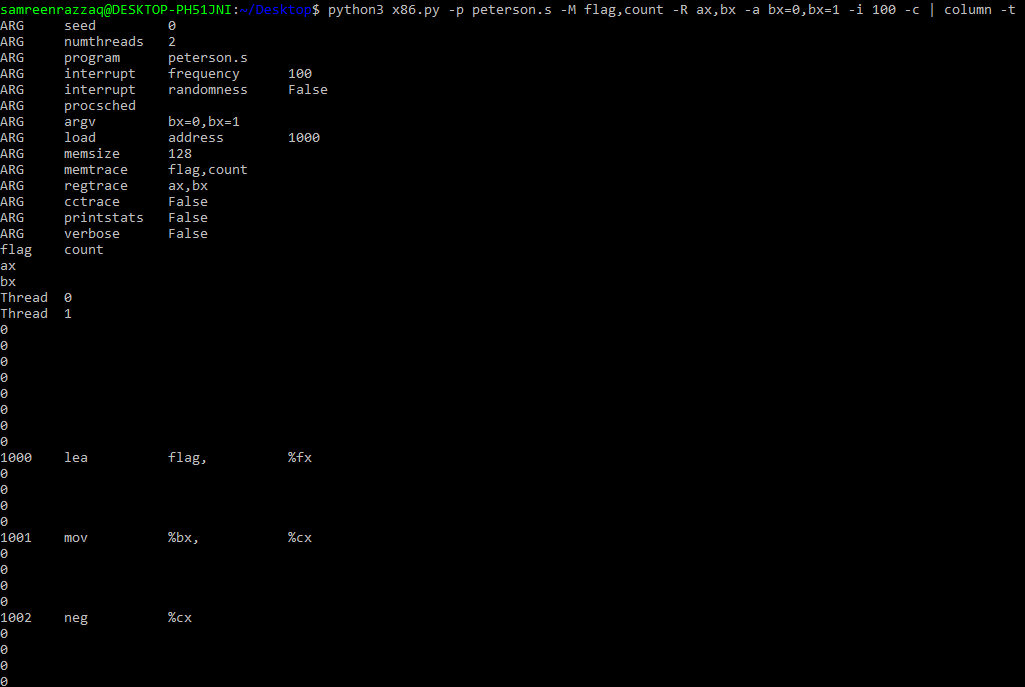
**Solution:**

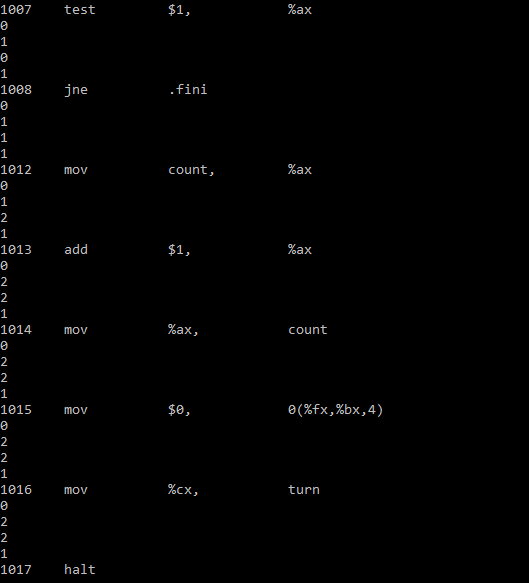
**-i 10: (high interrupt frequency)**





**-i 100: (low interrupt frequency)**





It monitors variables (`flag` and `count`) and registers (`ax` and `bx`), captures detailed traces, and initializes thread IDs with `-a bx=0,bx=1`. The parameter `-i 100` adjusts the interrupt frequency. This setup allows the examination of different behaviors in the context of Peterson's algorithm as influenced by varying interrupt intervals, providing insights into the synchronization and mutual exclusion achieved by the algorithm for different thread IDs. The first query involves “high frequency interrupt”.

The `-i 10` flag sets the interrupt frequency to 10 cycles. This setup allows the study of different behaviors in the context of Peterson's algorithm, particularly how varying interrupt frequencies influence synchronization and mutual exclusion. It's important to note that appropriate thread IDs are set using `-a bx=0,bx=1` to align with the assumptions in the code. The second query involves “low frequency interrupt”.

1. Can you control the scheduling (with the -P flag) to “prove” that the code works? What are the differentcases you should show hold? Think about mutual exclusion and deadlock avoidance.

**Solution:**

The command `python3 x86.py -p peterson.s -M flag,count -R ax,bx -c -a bx=0,bx=1 -i 10 -P` runs the x86 assembly program from `peterson.s` with monitoring and specific initializations. It uses the `-P` flag to generate specific tests for the code, allowing control over scheduling scenarios to validate the functionality of Peterson's algorithm. This command enables the demonstration of cases that affirm mutual exclusion, showcasing situations where one thread entering the critical section prevents the other from doing so concurrently. Additionally, the -P flag can be used to test deadlock avoidance, ensuring that both threads don't end up waiting indefinitely for access to the critical section. By carefully selecting and testing different schedules, one can provide evidence that the code effectively achieves mutual exclusion and avoids deadlock in various scenarios.

1. Now study the code for the ticket lock in ticket.s. Does it match the code in the chapter? Then run with the following flags: -a bx=1000, bx=1000 (causing each thread to loop through the critical section 1000 times). Watch what happens; do the threads spend much time spin-waiting for the lock?

**Solution:**

The command `python3 x86.py -p ticket.s -R ax,bx -a bx=1000,bx=1000` executes the x86 assembly program from `ticket.s`, monitoring registers `ax` and `bx`, and initializing both threads to loop through the critical section 1000 times. This setup enables the observation of the ticket lock implementation's efficiency, specifically assessing whether the threads spend significant time spin-waiting for the lock under the specified conditions.

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

**Solution:**

The command `python3 x86.py -p ticket.s -R ax,bx -a bx=1000,bx=1000` executes the x86 assembly program from `ticket.s` with two threads looping through the critical section 1000 times each. To understand the behavior with more threads, the command could be modified to include additional threads by adjusting the `-a bx` parameter accordingly. This setup allows the assessment of how the ticket lock implementation handles increased concurrency and whether its efficiency is impacted as more threads contend for access to the critical section.

1. Now examine yield.s, in which a yield instruction enables one thread to yield control of the CPU (realistically, this would be an OS primitive, but for the simplicity, we assume an instruction 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?

**Solution:**

The examination of the `yield.s` code, which involves a yield instruction allowing one thread to yield CPU control, raises questions about scenarios where `test-and-set.s` may waste cycles spinning while `yield.s` does not. In a situation where multiple threads contend for a lock implemented by `test-and-set.s`, the spinning threads may continuously check the lock status, wasting cycles in the process. In contrast, `yield.s` introduces a more efficient mechanism. When one thread needs to wait for a lock, it yields control, allowing other threads to execute, potentially reducing the overall spinning time. The savings in instructions arise from the fact that the yielding thread doesn't consume CPU cycles needlessly, enabling more productive execution by other threads. The efficiency gains are particularly prominent in scenarios with contention, where threads frequently request the lock. The exact number of instructions saved would depend on the specific implementation and frequency of lock contention in the given scenario.

1. 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?

**Solution:**

The examination of `test-and-test-and-set.s` involves understanding the behavior of this locking mechanism. In contrast to `test-and-set.s`, which directly checks and sets the lock in one atomic operation, `test-and-test-and-set.s` introduces a two-step process. First, it tests the lock to see if it is set, and if not, it performs a test-and-set operation. This introduces a "test-then-set" approach, allowing the thread to potentially avoid the costly atomic operation when the lock is already set, reducing contention overhead.

The savings introduced by `test-and-test-and-set.s` compared to `test-and-set.s` are primarily in scenarios where the lock is frequently uncontested. In such cases, the two-step process can avoid unnecessary atomic operations, leading to potential efficiency gains. The lock's behavior optimizes for situations where threads can quickly determine the lock status without resorting to the atomic test-and-set operation unless absolutely necessary. The exact extent of the savings would depend on the specific workload and contention patterns in the given scenario.