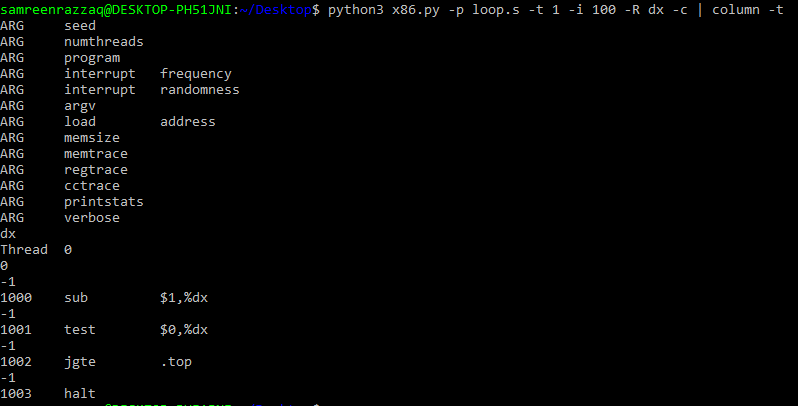


**Lab Task:**

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:**

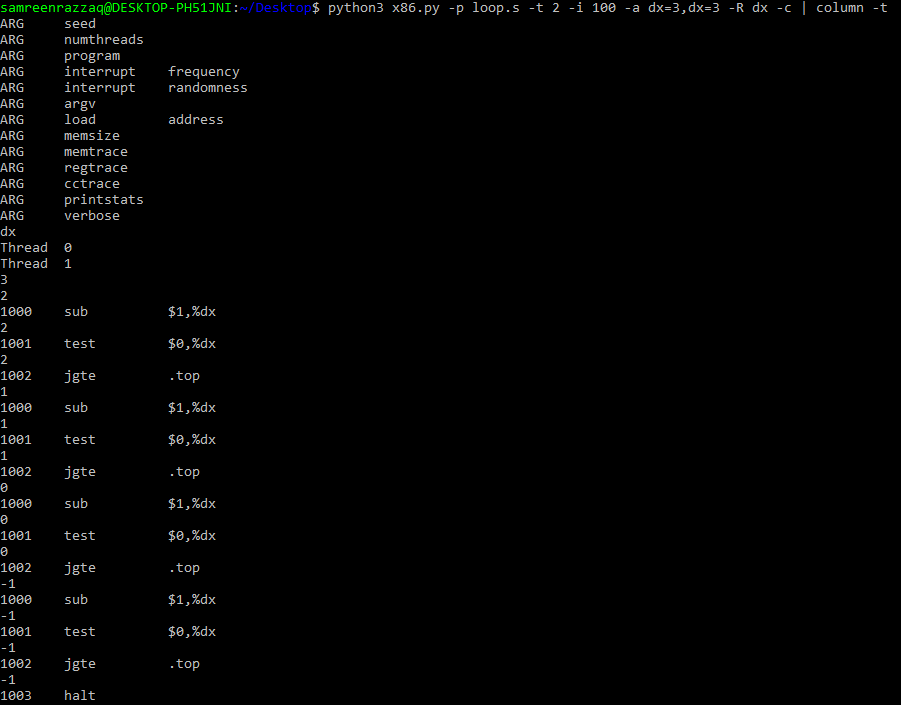
1. Let’s examine a simple program, “loop.s”. First, just read and understand it. Then, run it with these arguments (./x86.py -p loop.s -t 1 -i 100 -R dx) This specifies a single thread, an interrupt every 100 instructions, and tracing of register %dx. What will %dx be during the run? Use the -c flag to check your answers; the answers, on the left, show the value of the register (or memory value) after the instruction on the right has run.

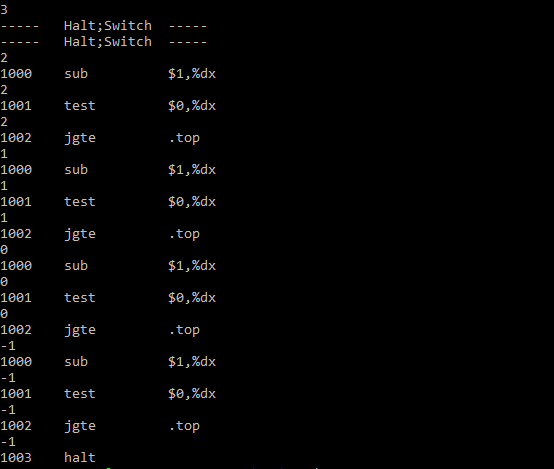
**Solution:**

%dx during the run: The value will be incremented by 1 every 100 instructions.

1. Same code, different 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 to 3. What values will %dx see? Run with -c to check. Does the presence of multiple threads affect your calculations? Is there a race in this code?

**Solution:**



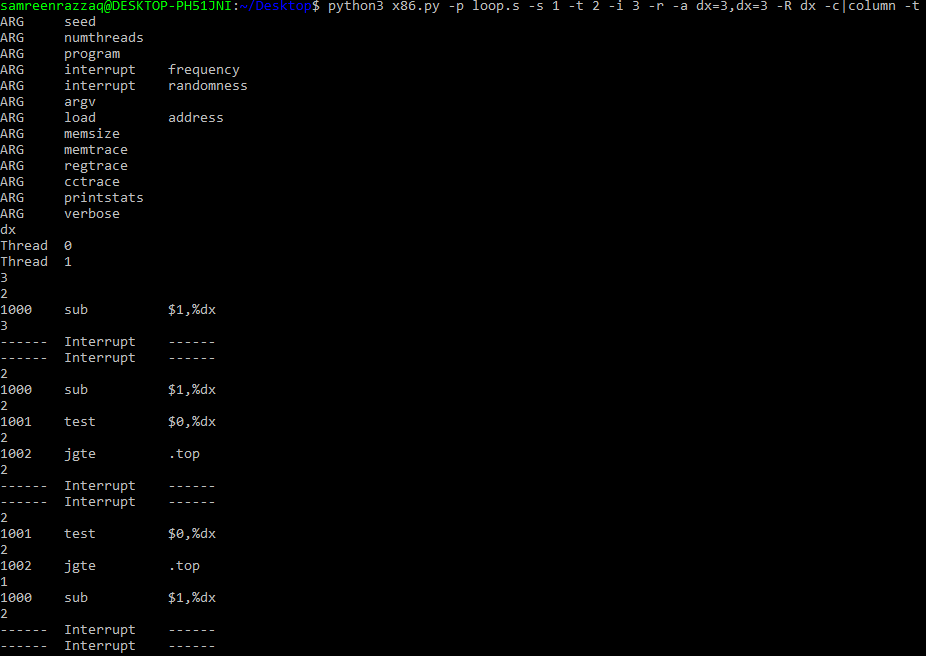


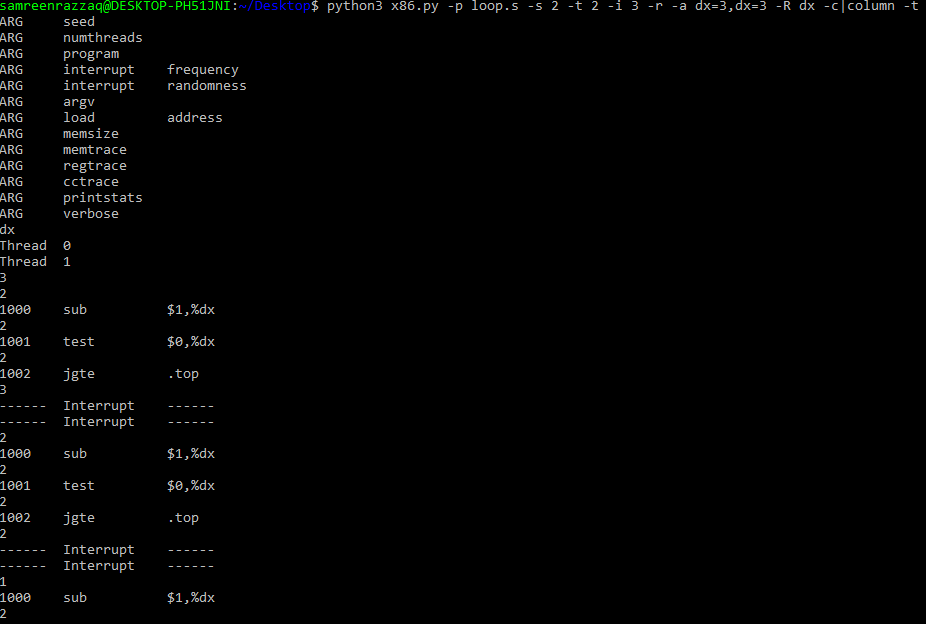
%dx values: Both threads will have their own %dx, initialized to 3. The presence of multiple threads does affect the calculations. Each thread has its own set of registers, and in this case, %dx is initialized independently for each thread. There is no race condition in this code because there is no shared data.

1. Run this: ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx This makes the interrupt interval small/random; use different seeds (-s) to see different interleavings. Does the interrupt frequency change anything?

**Solution:**

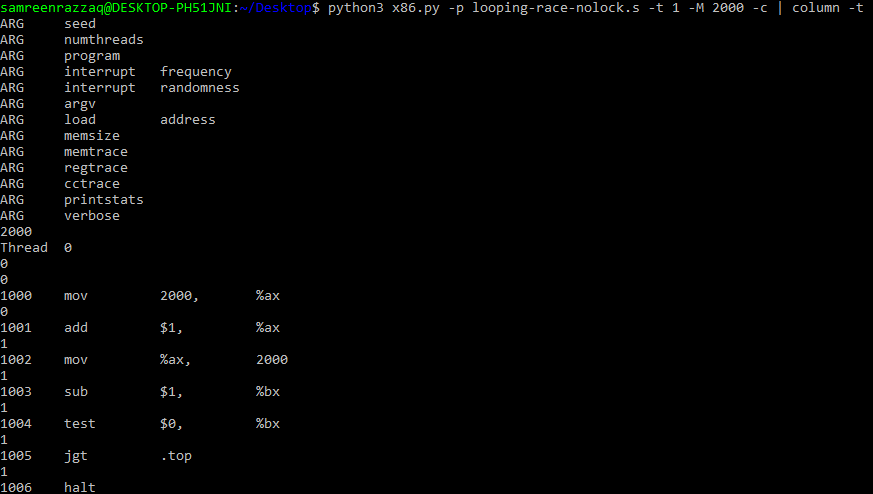
I use different seeds -s 1 and -s 2 to see different interleavings. Interrupt frequency might affect the trace. You can see here:

****

****

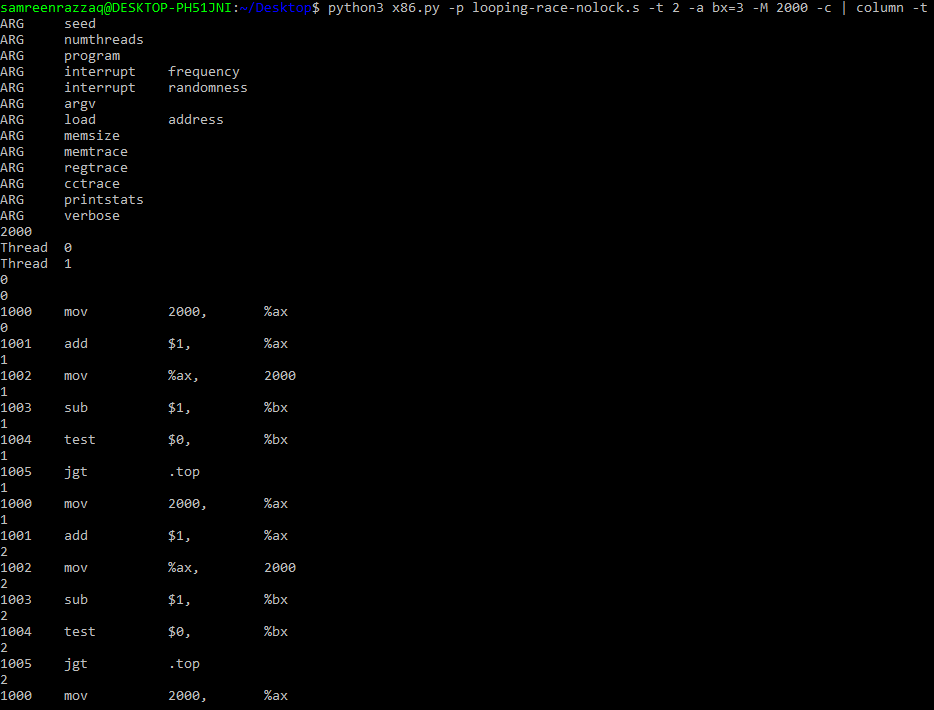
1. Now, a different program, looping-race-nolock.s, which accesses a shared variable located at address 2000; we’ll call this variable value. Run it with a single thread to confirm your understanding: ./x86.py -p looping-race-nolock.s -t 1 - M 2000 What is value (i.e., at memory address 2000) throughout the run? Use -c to check.

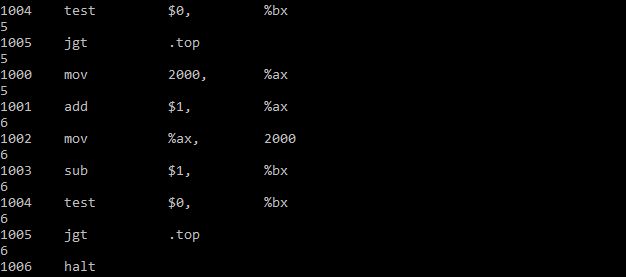
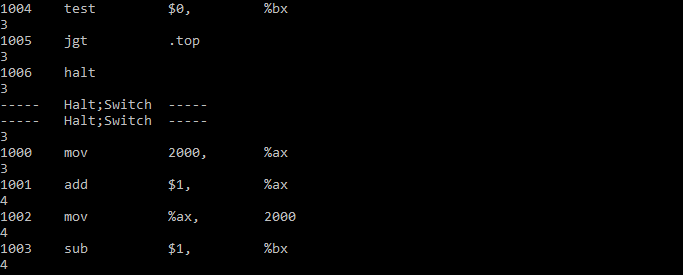
**Solution:**

****Initial value: The value will be incremented by 1 in each iteration. You can see here:

1. Run with multiple iterations/threads: ./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000 Why does each thread loop three times? What is final value of value?

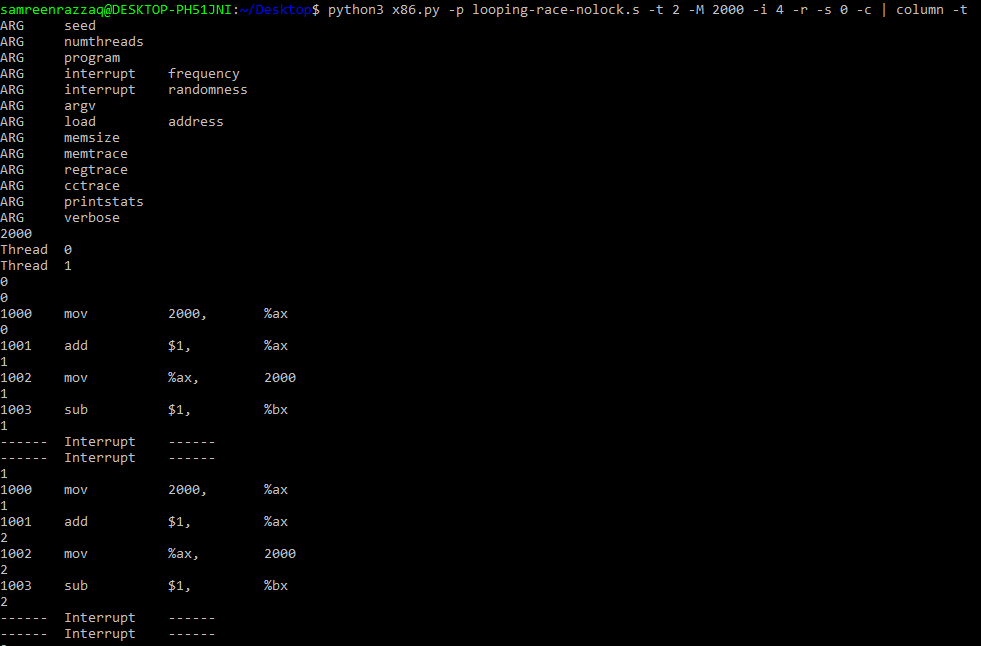
**Solution**:

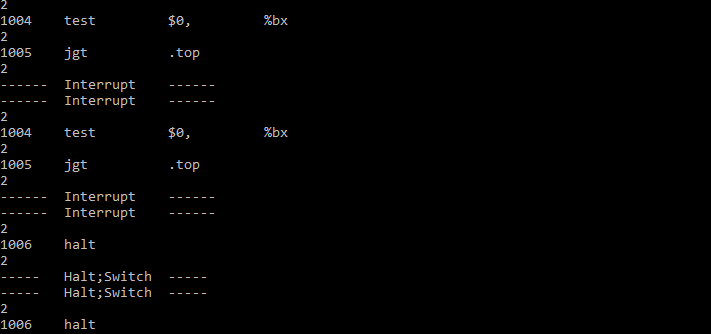
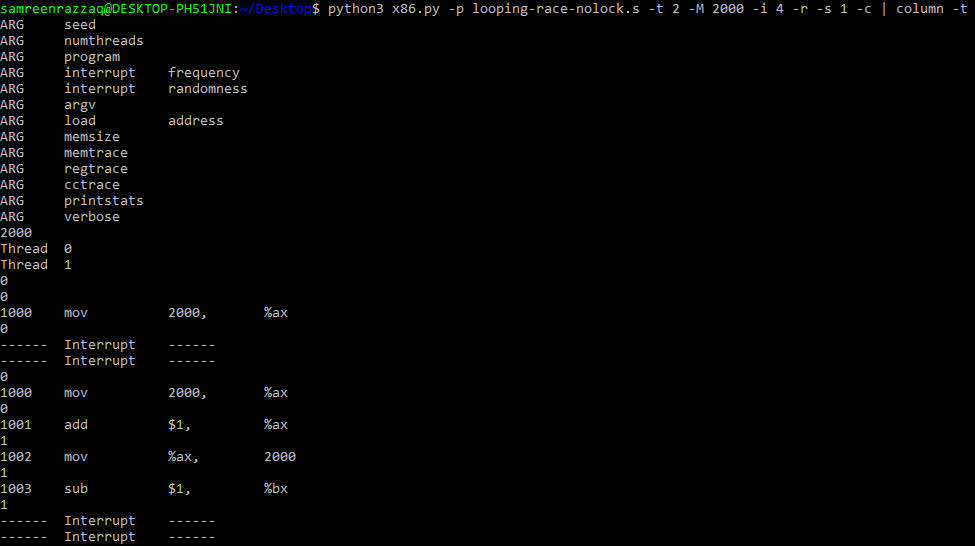
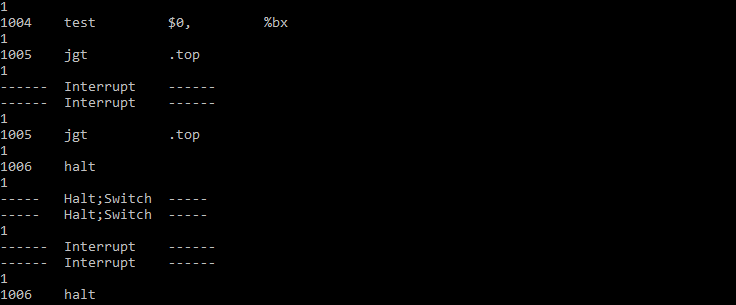
**** Each thread loops three times due to the nature of the code. The final value of the shared variable "value" at memory address 2000 depends on the interleaving of the threads and is subject to race conditions. Therefore, it's difficult to predict the exact final value, and it can vary between runs.

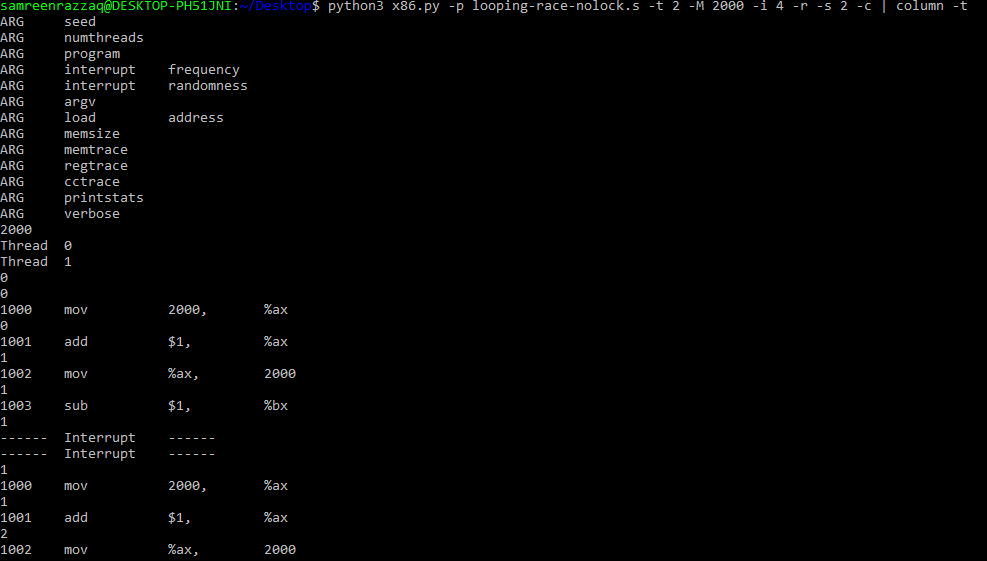
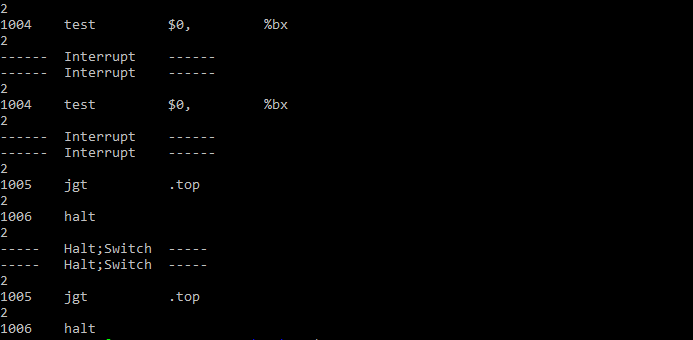


1. Run with random interrupt intervals: ./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0 with different seeds (-s 1, -s 2, etc.) Can you tell by looking at the thread interleaving what the final value of value will be? Does the timing of the interrupt matter? Where can it safely occur? Where not? In other words, where is the critical section exactly?

**Solution:**

Final value: Hard to predict by looking at the interleaving. Timing of the interrupt matters. The critical section in the code is the part where the shared variable "value" is accessed and modified. It is unsafe for interrupts to occur within this critical section because it may lead to race conditions, where multiple threads try to modify the shared variable simultaneously.

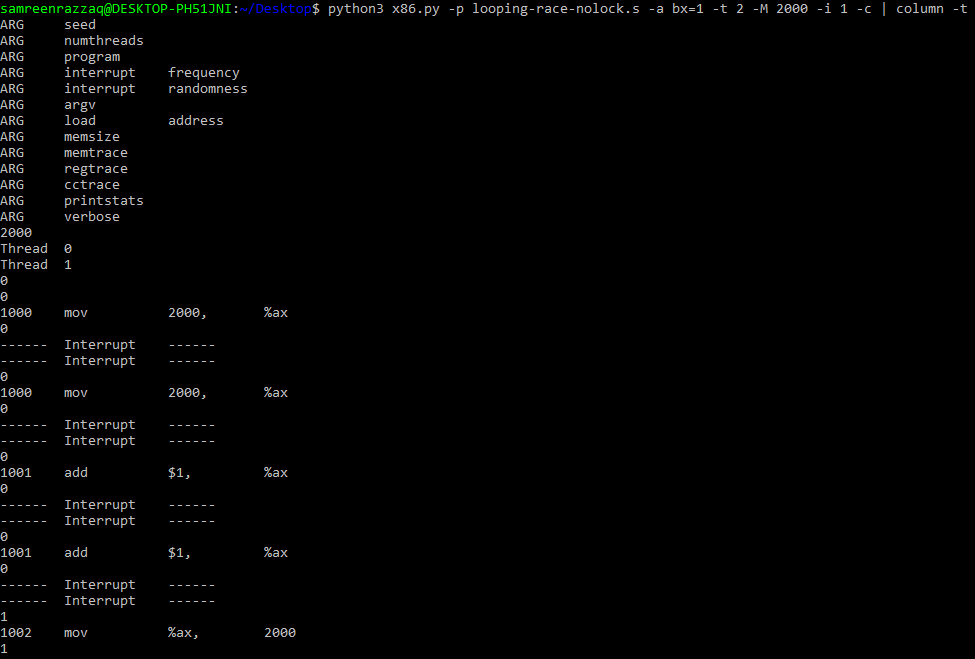


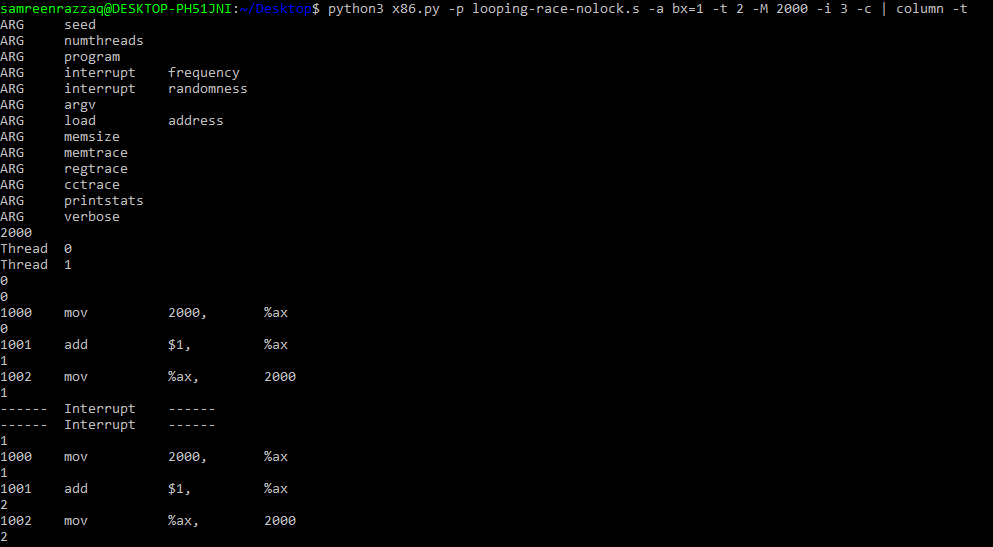
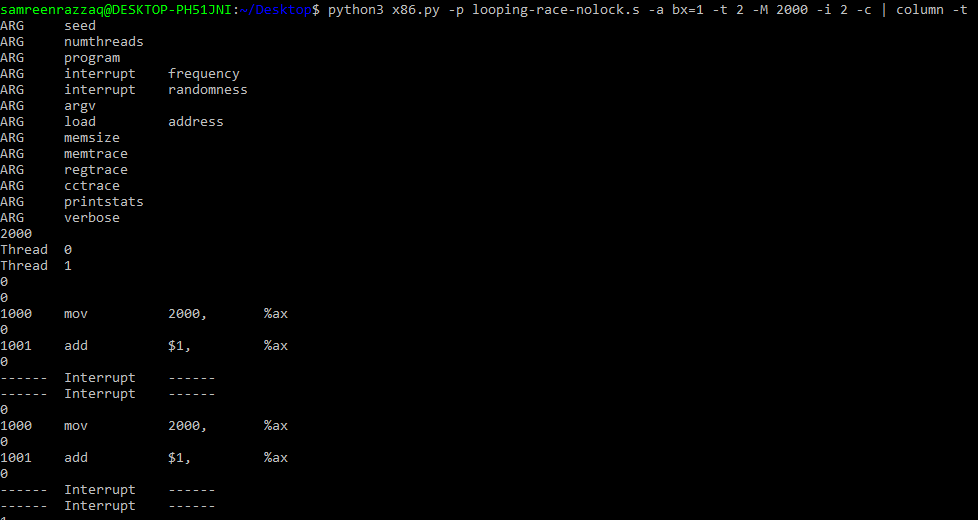


1. Now examine fixed interrupt intervals: ./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1 What will the final value of the shared variable value be? What about when you change -i 2, -i 3, etc.? For which interrupt intervals does the program give the “correct” answer?

**Solution:**

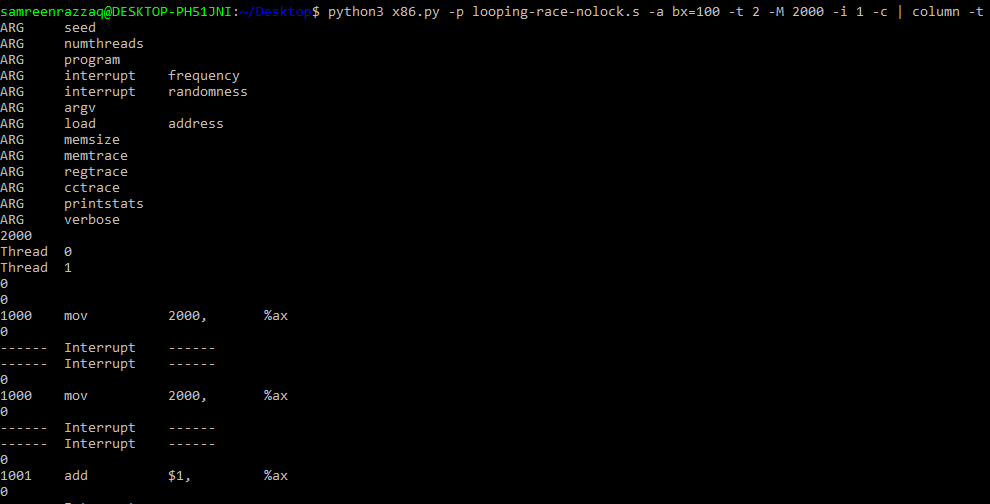
The final value of the shared variable "value" in looping-race-nolock.s can be predicted. However, changing the interval to 2, 3, etc., makes it challenging to determine the final value due to potential race conditions. The program gives the "correct" answer when using smaller interrupt intervals that avoid interference in the critical section where shared data is accessed and modified.

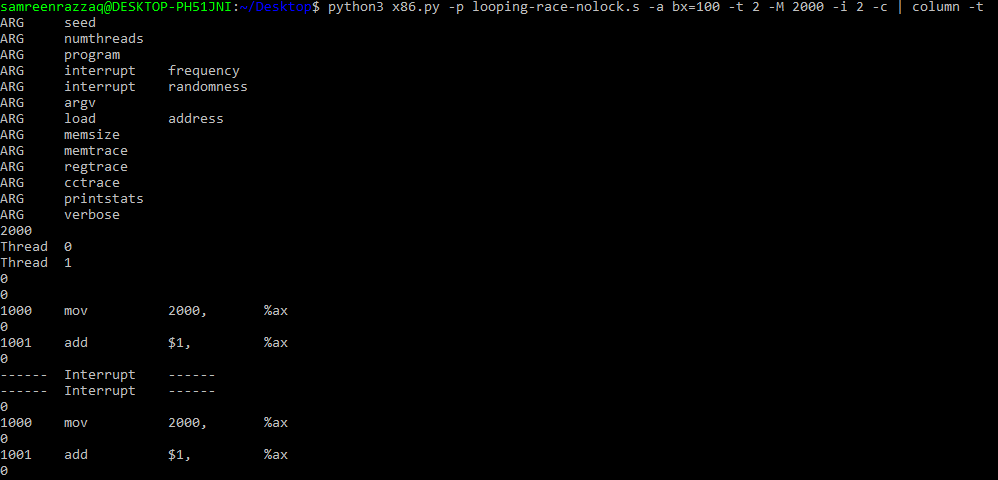


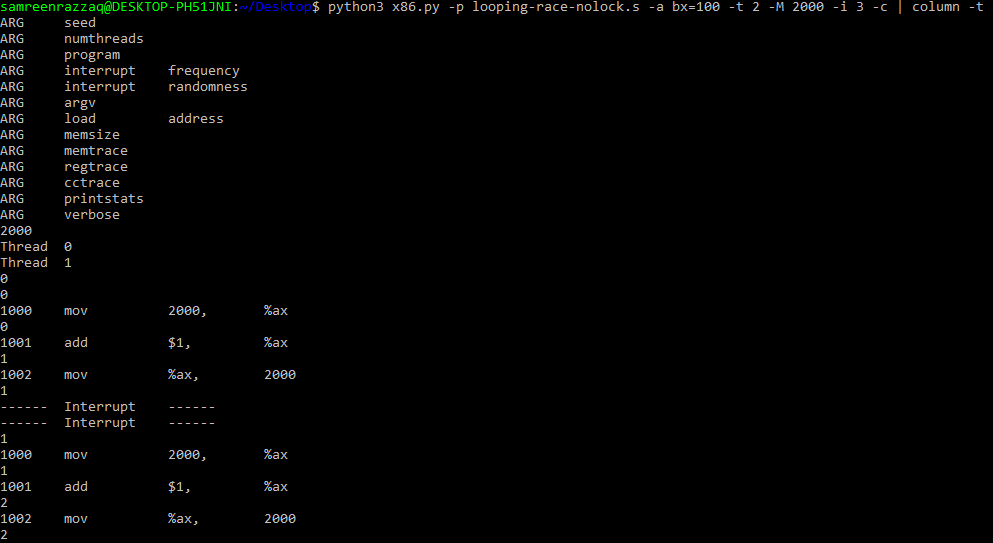


1. Run the same for more loops (e.g., set -a bx=100). What interrupt intervals (-i) lead to a correct outcome? Which intervals are surprising?

**Solution:**

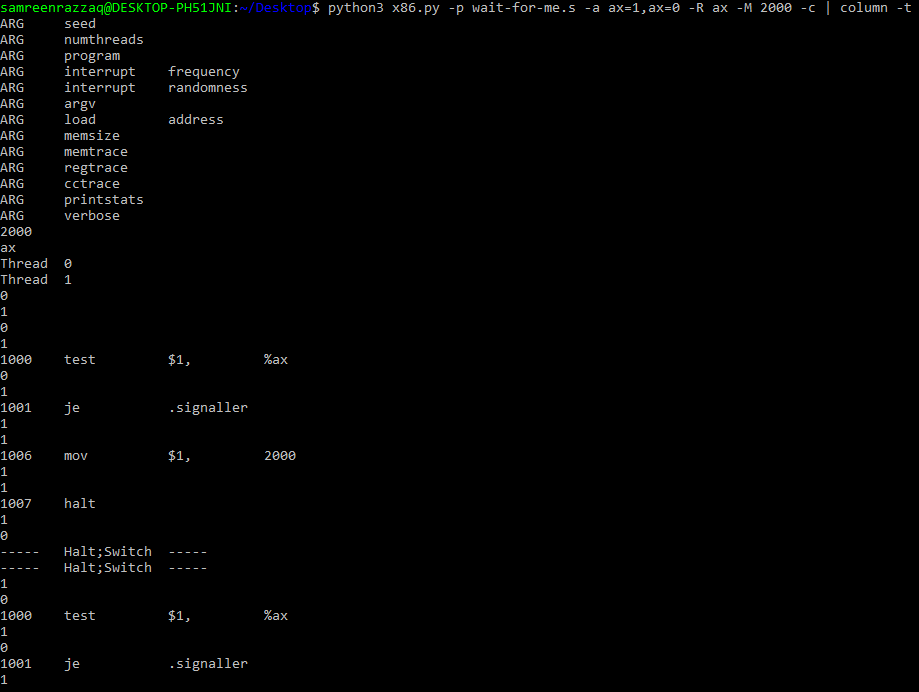
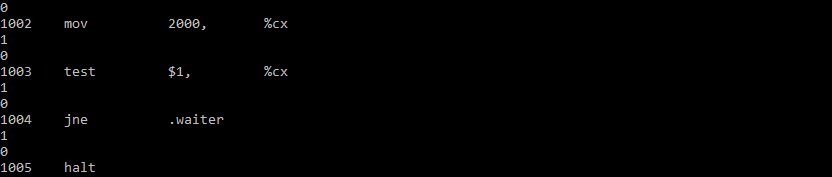
The smaller interrupt intervals (-i) are likely to lead to correct outcomes by minimizing interference in critical sections. Smaller intervals, such as 1 or 2, are often effective. Larger intervals might result in surprising outcomes due to increased chances of race conditions.





1. One last program: wait-for-me.s. Run: ./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 %ax and memory location 2000. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?

**Solution:**

The behavior involves thread synchronization: thread 1 waits for thread 0 to set the memory location. The final value at location 2000 should be 1, indicating successful synchronization between the threads.

1. 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?

**Solution:**

Switching the inputs (`-a ax=0,ax=1`), thread 0 sets %ax to 0, and thread 1 sets %ax to 1. Thread 0 proceeds without waiting, potentially leading to incorrect synchronization. Changing the interrupt interval or using random intervals may impact thread interleaving, affecting the trace outcome. The efficiency of CPU usage depends on how well the program manages thread synchronization and the chosen interrupt strategy.

