Ch27

1. First build main-race.c. Examine the code so you can see the (hopefully obvious) data race in the code. Now run helgrind (by typing valgrind --tool=helgrind main-race) to see how it reports the race. Does it point to the right lines of code? What other information does it give to you?

   gcc main-race.c -o main-race -pthread
   valgrind --tool=helgrind ./main-race

   ```
   ----------------------------------------------------------------
   ==189896==
   ==189896== Possible data race during read of size 4 at 0x601048 by thread #1
   ==189896== Locks held: none
   ==189896==    at 0x400672: main (main-race.c:15)
   ==189896==
   ==189896== This conflicts with a previous write of size 4 by thread #2
   ==189896== Locks held: none
   ==189896==    at 0x40061E: worker (main-race.c:8)
   ==189896==    by 0x4C30EDE: mythread_wrapper (hg_intercepts.c:389)
   ==189896==    by 0x4E43EA4: start_thread (in /usr/lib64/libpthread-2.17.so)
   ==189896==    by 0x5156B0C: clone (in /usr/lib64/libc-2.17.so)
   ==189896==  Address 0x601048 is 0 bytes inside data symbol "balance"
   ==189896==
   ==189896== ----------------------------------------------------------------
   ==189896==
   ==189896== Possible data race during write of size 4 at 0x601048 by thread #1
   ==189896== Locks held: none
   ==189896==    at 0x40067B: main (main-race.c:15)
   ==189896==
   ==189896== This conflicts with a previous write of size 4 by thread #2
   ==189896== Locks held: none
   ==189896==    at 0x40061E: worker (main-race.c:8)
   ==189896==    by 0x4C30EDE: mythread_wrapper (hg_intercepts.c:389)
   ==189896==    by 0x4E43EA4: start_thread (in /usr/lib64/libpthread-2.17.so)
   ==189896==    by 0x5156B0C: clone (in /usr/lib64/libc-2.17.so)
   ==189896==  Address 0x601048 is 0 bytes inside data symbol "balance"
   ```

   Yes, it points to the data race address which is line 15 during read and write to that address.

2. What happens when you remove one of the offending lines of code? Now add a lock around one of the updates to the shared variable, and then around both. What does helgrind report in each of these cases?

Delete line 15:
There is no error anymore.
Add one lock:
Possible data race during read of size 4 at 0x601080 by thread #1
Possible data race during write of size 4 at 0x601080 by thread #1
Add two locks:
There is no error anymore.

3. Now let's look at main-deadlock.c. Examine the code. This code has a problem known as deadlock (which we discuss in much more depth in a forthcoming chapter). Can you see what problem it might have?

   The deadlock would happen when each thread executes the first lock in the if/else block. For example, Thread1 obtains a lock on m1 and Thread2 obtains a lock on m2. Neither of them would gain the next lock, since each of them is holding and waiting for the next to release.

4. Now run helgrind on this code. What does helgrind report?

   Thread #3: lock order "0x601080 before 0x6010C0" violated
   Observed (incorrect) order is: acquisition of lock at 0x6010C0

   Helgrind complained that the lock order is incorrect. If both threads obtain the locks in the same order, the deadlock problem can be solved.

5. Now run helgrind on main-deadlock-global.c. Examine the code; does it have the same problem that main-deadlock.c has? Should helgrind be reporting the same error? What does this tell you about tools like helgrind?
   It should be solved.
   It still report incorrection.
   Thread #3: lock order "0x6010C0 before 0x601100" violated
   ==34169==
   ==34169== Observed (incorrect) order is: acquisition of lock at 0x601100

   Helgrind is not perfect.

6. Let's next look at main-signal.c. This code uses a variable (done) to signal that the child is done and that the parent can now continue. Why is this code inefficient? (what does the parent end up spending its time doing, particularly if the child thread takes a long time to complete?)

   The thread 1 was trapped in the loop, using up the CPU cycles, when waiting for threat 2.

7. Now run helgrind on this program. What does it report? Is the code correct?

   Reports:
   Possible data race during write of size 4 at 0x601048 by thread #2
   This conflicts with a previous read of size 4 by thread #1
   Possible data race during read of size 4 at 0x601048 by thread #1
   This conflicts with a previous write of size 4 by thread #2

   When reading(main) and writing(worker) the done variable. The code is thread safe in this case. However, helgrind highlight it since in other case it may not correct.

8. Now look at as lightly modified version of the code, which is found in main-signal-cv.c. This version uses a condition variable to do the signaling (and associated lock). Why is this code preferred to the previous version? Is it correctness, or performance, or both?

   It's correct and has better performance. Because the spin-waiting is avoided, performance is better.

9. Once again run helgrind on main-signal-cv. Does it report any errors?
   There is no complain.

Ch28

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

2. 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?

   python2 x86.py -p flag.s -M flag,count -R ax,bx -c

   Because the program only executes the loop once on each thread the result is correct. The flag would be 0 (lock is free)then 1 (the thread0 can start lock), then 0 (the thread 0 clear the lock), then 1 (the thread1 can start)then 0 (the thread1 free the lock).
   Flag would be 0 in the end.

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?

python2 x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=2,bx=2

The flag is still 0 in the end.

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?

   When i is 1-10, 12, 13, 14, we have bad outcomes
   python2 x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=10,bx=10 -i 1-10,12,13,14,17

   When i is 11, 15, 16, and multiple of 11, we would have good outcomes
   python2 x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=10,bx=10 -i 11,15,16

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?

   Acquire the lock:
   ```
   mov  $1, %ax
   xchg %ax, mutex     # atomic swap of 1 and mutex
   test $0, %ax        # if we get 0 back: lock is free!
   jne  .acquire       # if not, try again
   ```

   Release lock
   ```
   mov  $0, mutex
   ```

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?

   python2 x86.py -p test-and-set.s -i 10 -R ax,bx -M mutex,count -a bx=5 -c

   It's always correct.
   The program is 11 instructions long. Interrupt intervals of 11 lead to no CPU waste.
   For other situations, it may lead to CPU waste. We can check whether there are loops arround the acquire locks.

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?

   python2 x86.py -p test-and-set.s -M mutex,count -R ax,bx -c -a bx=10,bx=10 -P 0011

Yes, it works fine.I'd want to test a context switch directly after xchg is is executed. Since it's an atomic instruction, there can't be a race condition.

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.

   it 's the simulator of Peterson's algorithm. 1- self is make turn for other thread.

9. 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.
   python2 x86.py -p peterson.s -M count,flag,turn -R ax,cx -a bx=0,bx=1 -c -i 1
   python2 x86.py -p peterson.s -M count,flag,turn -R ax,cx -a bx=0,bx=1 -c -i 2

   Interrupt by different interval

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.

    python2 x86.py -p peterson.s -M turn,count -R bx -a bx=0,bx=1 -P 0000011111 -c
    python2 x86.py -p peterson.s -M turn,count -R bx -a bx=0,bx=1 -P 00000011111 -c

    We can specify exactly that thread0 run repeatedly and then thread1.
    We would find the simulator didn't run as we predicted.

11. 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?

    python2 x86.py -p ticket.s -M count,ticket,turn -R ax,bx,cx -a bx=1000,bx=1000 -c
    Yes. Because the lock is released on each loop interation, each thread only has 1 turn per time slice. Thus the progress is slow, and spinning wastes many time.

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

    python2 x86.py -p ticket.s -M count -t 10 -c -i 5

    Same thing. More threads didn't do much work per time slice.

13. 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?

python2 x86.py -p test-and-set.s -M count,mutex -R ax,bx -a bx=5,bx=5 -c -i 7 | wc -l
returns 216
python2 x86.py -p yield.s -M count,mutex -R ax,bx -a bx=5,bx=5 -c -i 7 | wc -l
returns 210

In 5 iterations, under those conditions, we're looking at 6 less clock cycles to complete with yield.
The savings arise when a context switch occurs while a thread is in a critical section. The newly running thread can't do it's job. It save 1 instruction each cycle.

14. 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?

    In test-and-test-and-set.s, it tests if mutex = 0 before executing xchg instruction. Change mutex to one only if the lock is free. This is like a compare-and-swap behavior. The advantage is that we are only writing to mutex when there is a chance that we will get the lock. This can help avoid unnecessary writing so that reduce bus traffic and better cache coherence.

Ch29

1. We'll start by redoing the measurements within this chapter. Use the call gettimeofday() to measure time within your program. How accurate is this timer? What is the smallest interval it can measure? Gain confidence in its workings, as we will need it in all subsequent questions. You can also look into other timers, such as the cycle counter available on x86 via the rdtsc instruction.

   gettimeofday() can measure microseconds.

   gcc count.c -pthread
   ./a.out 100 5
   5 threads incrementing a locked (monitor) counter 100 times each took 0.000143 seconds.
   Counter:
   Expected: 500
   Actual: 500

2. Now, build a simple concurrent counter and measure how long it takes to increment the counter many times as the number of threads increases. How many CPUs are available on the system you are using? Does this number impact your measurements at all?

clang -Wall -pthread simple_concurrent_counter.c -lm

1 cpus, 1 threads
global count: 1000000
Time (seconds): 0.052269

1 cpus, 2 threads
global count: 2000000
Time (seconds): 0.070798

1 cpus, 3 threads
global count: 3000000
Time (seconds): 0.077075

1 cpus, 4 threads
global count: 4000000
Time (seconds): 0.081359

1 cpus, 5 threads
global count: 5000000
Time (seconds): 0.099526

1 cpus, 6 threads
global count: 6000000
Time (seconds): 0.117749

1 cpus, 7 threads
global count: 7000000
Time (seconds): 0.137884

1 cpus, 8 threads
global count: 8000000
Time (seconds): 0.158413

1 cpus, 9 threads
global count: 9000000
Time (seconds): 0.177902

1 cpus, 10 threads
global count: 10000000
Time (seconds): 0.202377

1 cpus, 11 threads
global count: 11000000

Time (seconds): 0.220885

1 cpus, 12 threads
global count: 12000000
Time (seconds): 0.240121

1 cpus, 13 threads
global count: 13000000
Time (seconds): 0.261100

1 cpus, 14 threads
global count: 14000000
Time (seconds): 0.281148

1 cpus, 15 threads
global count: 15000000
Time (seconds): 0.304846

1 cpus, 16 threads
global count: 16000000
Time (seconds): 0.317192

1 cpus, 17 threads
global count: 17000000
Time (seconds): 0.337920

1 cpus, 18 threads
global count: 18000000
Time (seconds): 0.357599

1 cpus, 19 threads
global count: 19000000
Time (seconds): 0.365657

1 cpus, 20 threads
global count: 20000000
Time (seconds): 0.383199

1 cpus, 21 threads
global count: 21000000
Time (seconds): 0.411127

1 cpus, 22 threads
global count: 22000000

Time (seconds): 0.426123

1 cpus, 23 threads
global count: 23000000
Time (seconds): 0.442331

1 cpus, 24 threads
global count: 24000000
Time (seconds): 0.471751

1 cpus, 25 threads
global count: 25000000
Time (seconds): 0.481840

1 cpus, 26 threads
global count: 26000000
Time (seconds): 0.502082

1 cpus, 27 threads
global count: 27000000
Time (seconds): 0.536959

1 cpus, 28 threads
global count: 28000000
Time (seconds): 0.551972

1 cpus, 29 threads
global count: 29000000
Time (seconds): 0.562553

1 cpus, 30 threads
global count: 30000000
Time (seconds): 0.585943

1 cpus, 31 threads
global count: 31000000
Time (seconds): 0.597150

1 cpus, 32 threads
global count: 32000000
Time (seconds): 0.626360

1 cpus, 33 threads
global count: 33000000

Time (seconds): 0.638499

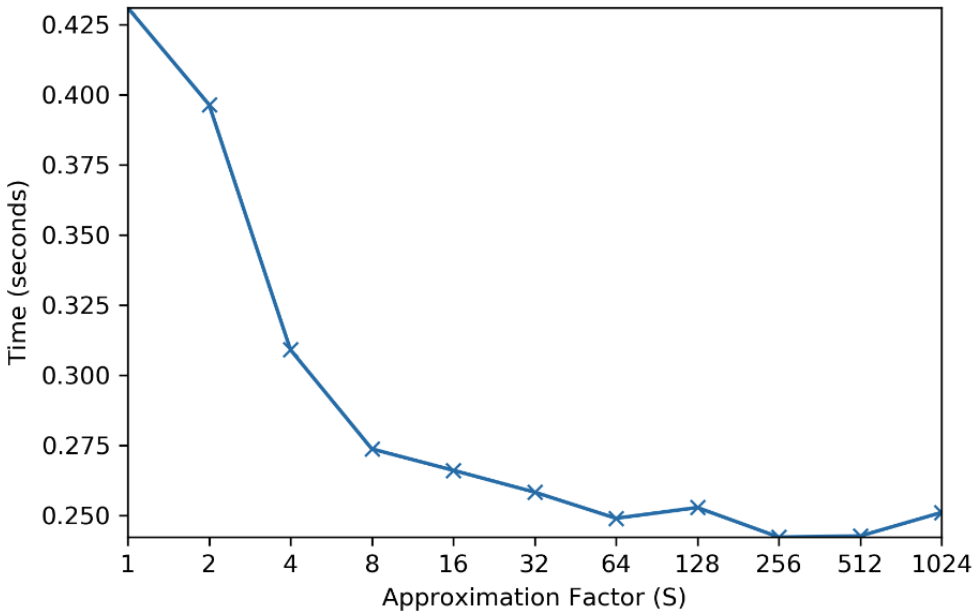
1 cpus, 34 threads
global count: 34000000
Time (seconds): 0.663264

1 cpus, 35 threads
global count: 35000000
Time (seconds): 0.689947

1 cpus, 36 threads
global count: 36000000
Time (seconds): 0.704636

3. Next, build a version of the sloppy counter. Once again, measure its performance as the number of threads varies, as well as the threshold. Do the numbers match what you see in the chapter?

clang -Wall -Wextra -pthread approximate_counter.c -lm



1 threads, 1 threshold
1000000 global counter

Time (seconds): 0.046986

2 threads, 1 threshold
2000000 global counter
Time (seconds): 0.214507

3 threads, 1 threshold
3000000 global counter
Time (seconds): 0.447831

4 threads, 1 threshold
4000000 global counter
Time (seconds): 1.197335

1 threads, 2 threshold
1000000 global counter
Time (seconds): 0.046838

2 threads, 2 threshold
2000000 global counter
Time (seconds): 0.250059

3 threads, 2 threshold
3000000 global counter
Time (seconds): 0.310765

4 threads, 2 threshold
4000000 global counter
Time (seconds): 0.919643

1 threads, 4 threshold
1000000 global counter
Time (seconds): 0.067100

2 threads, 4 threshold
2000000 global counter
Time (seconds): 0.257514

3 threads, 4 threshold
3000000 global counter
Time (seconds): 0.403177

4 threads, 4 threshold
4000000 global counter

Time (seconds): 0.863276

1 threads, 8 threshold
1000000 global counter
Time (seconds): 0.062422

2 threads, 8 threshold
2000000 global counter
Time (seconds): 0.232636

3 threads, 8 threshold
3000000 global counter
Time (seconds): 0.262640

4 threads, 8 threshold
4000000 global counter
Time (seconds): 0.530257

1 threads, 16 threshold
1000000 global counter
Time (seconds): 0.055578

2 threads, 16 threshold
2000000 global counter
Time (seconds): 0.216613

3 threads, 16 threshold
3000000 global counter
Time (seconds): 0.370755

4 threads, 16 threshold
4000000 global counter
Time (seconds): 0.690549

1 threads, 32 threshold
1000000 global counter
Time (seconds): 0.055556

2 threads, 32 threshold
2000000 global counter
Time (seconds): 0.214723

3 threads, 32 threshold
3000000 global counter

Time (seconds): 0.322316

4 threads, 32 threshold
4000000 global counter
Time (seconds): 0.635870

1 threads, 64 threshold
1000000 global counter
Time (seconds): 0.057168

2 threads, 64 threshold
2000000 global counter
Time (seconds): 0.135173

3 threads, 64 threshold
3000000 global counter
Time (seconds): 0.331226

4 threads, 64 threshold
4000000 global counter
Time (seconds): 0.657275

1 threads, 128 threshold
999936 global counter
Time (seconds): 0.021795

2 threads, 128 threshold
2000000 global counter
Time (seconds): 0.201170

3 threads, 128 threshold
2999936 global counter
Time (seconds): 0.316283

4 threads, 128 threshold
4000000 global counter
Time (seconds): 0.602716

1 threads, 256 threshold
999936 global counter
Time (seconds): 0.057886

2 threads, 256 threshold
1999872 global counter

Time (seconds): 0.252846

3 threads, 256 threshold
2999808 global counter
Time (seconds): 0.305332

4 threads, 256 threshold
4000000 global counter
Time (seconds): 0.581310

1 threads, 512 threshold
999936 global counter
Time (seconds): 0.022160

2 threads, 512 threshold
1999872 global counter
Time (seconds): 0.168481

3 threads, 512 threshold
2999808 global counter
Time (seconds): 0.287035

4 threads, 512 threshold
3999744 global counter
Time (seconds): 0.559594

1 threads, 1024 threshold
999424 global counter
Time (seconds): 0.057596

2 threads, 1024 threshold
1999872 global counter
Time (seconds): 0.251326

3 threads, 1024 threshold
2999296 global counter
Time (seconds): 0.292125

4 threads, 1024 threshold
3999744 global counter
Time (seconds): 0.552106

4. Build a version of a linked list that uses hand-over-hand locking[MS04], as cited in the chapter. You should read the paper first to understand how it works, and then implement it. Measure its performance. When does a hand-over-hand list work better than a standard list as shown in the chapter?
   It won't be better than a standard list as shown in the chapter.

5. Pick your favorite interesting data structure, such as a B-tree or other slightly more interested structure. Implement it, and start with a simple locking strategy such as a single lock. Measure its performance as the number of concurrent threads increases.

   clang -Wall -Wextra -pthread btree.c

   1 threads
   Time (seconds): 0.000517

   size: 100

   2 threads
   Time (seconds): 0.000248

   size: 100

   3 threads
   Time (seconds): 0.000199

   size: 99

   4 threads
   Time (seconds): 0.000208

   size: 100

   5 threads
   Time (seconds): 0.000228

   size: 100

   6 threads
   Time (seconds): 0.000997

   size: 96

   7 threads

Time (seconds): 0.000277

size: 98

8 threads
Time (seconds): 0.001061

size: 96

9 threads
Time (seconds): 0.000395

size: 99

10 threads
Time (seconds): 0.001139

size: 100