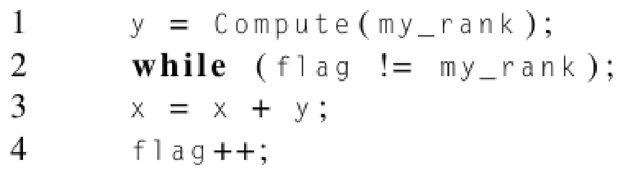
Busy-waiting

To avoid race conditions, threads need exclusive access to shared memory regions.

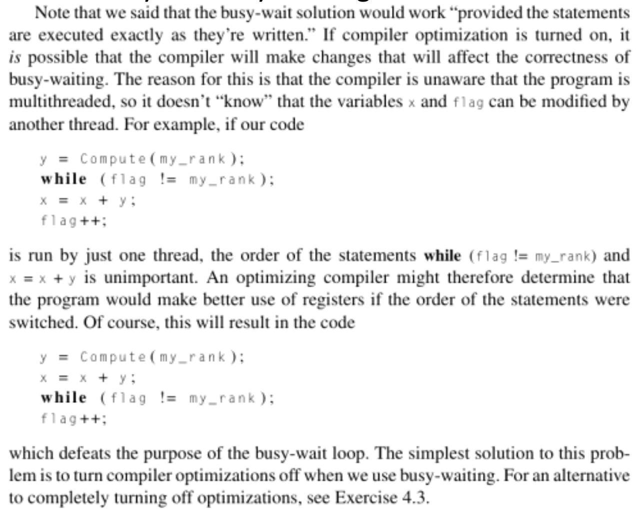
- When say thread 0 wants to execute the statement x = x+y it needs to make sure that thread 1 is not already executing the statement. Once thread 0 makes sure of this it needs to provide some way for thread 1 to know that it is currently executing the statement. Finally thread 0 needs to let it be known that it is done executing the statement so that thread 1 can execute the statement.
- One way to solve this problem is using a simple flag variable.
  - Lets suppose flag is a shared int variable set to 0 in our main thread.
  - We add the following code to our example:



```
1    y = Compute(my_rank);
2    while (flag != my_rank);
3    x = x + y;
4    flag++;
```

Code added:
while(flag!=my_rank);
flag++

    - Note: The while loop has an empty body. This means that in our program as long as flag isnt equal to my_rank we will continue to re-exectute the condition until it is false. We are essentially stuck here until given the go ahead to continue.
  - Since we’re assuming that the main thread has initialized flag to 0, thread 1 won’t proceed to the critical section in Line 3 until thread 0 executes the statement flag++.
  - The key here is that thread 1 cannot enter the critical section until thread 0 has completed the execution of flag++.
  - The while loop is an example of **busy-waiting**.
    - In busy-waiting, a thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value (false in our example).
  - Sometimes with busy-waiting solution we can have situations where compiler optimizations will change our code and destroy our busy-waiting solution.



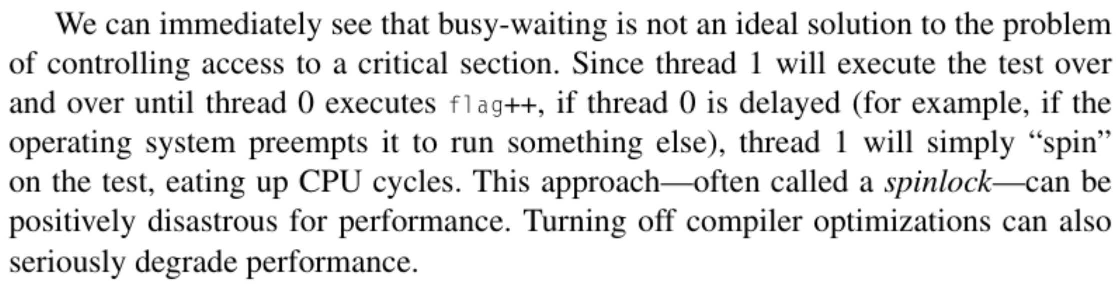
Note that we said that the busy-wait solution would work “provided the statements are executed exactly as they’re written.” If compiler optimization is turned on, it *is* possible that the compiler will make changes that will affect the correctness of busy-waiting. The reason for this is that the compiler is unaware that the program is multithreaded, so it doesn’t “know” that the variables x and flag can be modified by another thread. For example, if our code

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

■ is run by just one thread, the order of the statements **while** (flag != my_rank) and x = x + y is unimportant. An optimizing compiler might therefore determine that the program would make better use of registers if the order of the statements were switched. Of course, this will result in the code

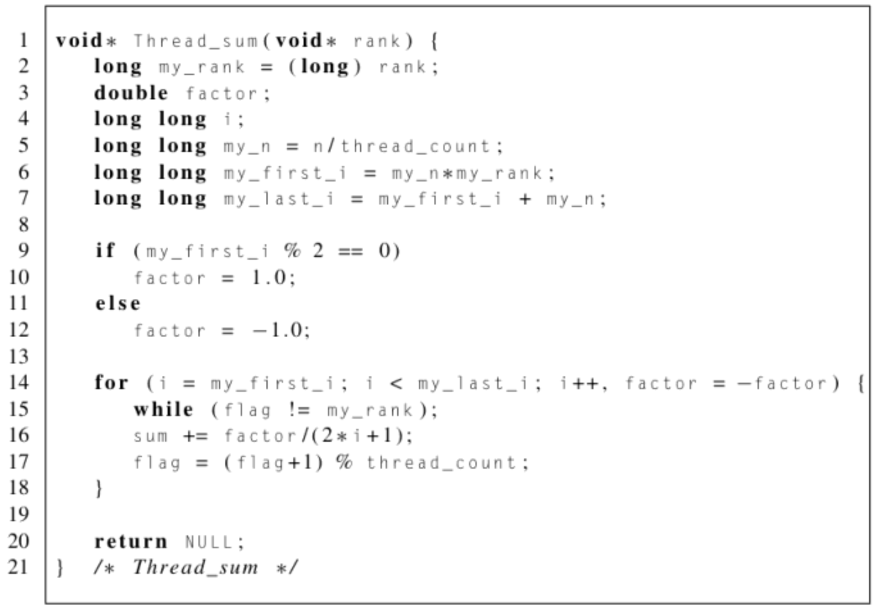
```
y = Compute(my_rank);
x = x + y;
while (flag != my_rank);
flag++;
```

which defeats the purpose of the busy-wait loop. The simplest solution to this problem is to turn compiler optimizations off when we use busy-waiting. For an alternative to completely turning off optimizations, see Exercise 4.3.



- We can immediately see that busy-waiting is not an ideal solution to the problem of controlling access to a critical section. Since thread 1 will execute the test over and over until thread 0 executes flag++, if thread 0 is delayed (for example, if the operating system preempts it to run something else), thread 1 will simply “spin” on the test, eating up CPU cycles. This approach—often called a *spinlock*—can be positively disastrous for performance. Turning off compiler optimizations can also seriously degrade performance.

Example of busy waiting code in a thread function:
Computing Pi



```
 1  void* Thread_sum(void* rank) {
 2     long my_rank = (long) rank;
 3     double factor;
 4     long long i;
 5     long long my_n = n/thread_count;
 6     long long my_first_i = my_n*my_rank;
 7     long long my_last_i = my_first_i + my_n;
 8
 9     if (my_first_i % 2 == 0)
10        factor = 1.0;
11     else
12        factor = -1.0;
13
14     for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15        while (flag != my_rank);
16        sum += factor/(2*i+1);
17        flag = (flag+1) % thread_count;
18     }
19
20     return NULL;
21  }  /* Thread_sum */
```

Program 4.4: Pthreads global sum with busy-waiting.

Note:

- We replaced flag++ with flag = (flag+1) mod thread_count
  - This is because with flag++ eventually we will see that our flag will just advance past the thread_count which will have our busy_waiting condition to never fail since it will never be equal to any rank
    - Why is this the case? It is important to understand what the code is doing. Lets say we have two threads. Each thread takes care of half of the work. In the for loop each thread will go through its iterations one at a time. It is important to understand that both threads in this scenerio will take turns during their iterations, going back and forth between each other caused by the busy-waiting.
    - The reason for flag = (flag+1) mod thread_count is so that we can always go back to the first thread so that it can do its next iteration and so on…
    - The for loop and busy-waiting do this -> thread0 passes busy-waiting condition and does iteration and then sets flag to be +1 and then it waits, thread1 passes busy-waiting condition and does iteration and then sets flag to be +1 but the mod takes it back to thread0 and thread1 waits. Thread0 passes busy-waiting condition… WE REPEAT.
  - This is all incredibly inefficient and with two threads, we actually see serial programs do better.
    - This is because the flag incrementing code on like 17 causes too much delay as well as the continuious swapping of threads each time for each iteration.
    - How to improve code:
      - Therefore, if it's at all possible, we should minimize the number of times we execute critical section code. One way to greatly improve the performance of the sum function is to have each thread use a private variable to store its total contribution to the sum. Then, each thread can add in its contribution to the global sum once, after the for loop. See Program 4.5. When we run this on the dual core system with n = 10^8, the elapsed time is reduced to 1.5 (vs 19.5 seconds which is what it would be with the current code) seconds for two threads, a substantial improvement.