> The purpose of this assignment is to help you understand some of the most
> important issues in low-level shared-memory parallel programming.
>
> The example simulates partitioning jobs and counting these jobs as
> operations. The provided version has several flaws which you are to
> supposed to fix. There is a corresponding chapter in the 'perfbook' (by
> Paul McKenney) and it also relates to the datarace screencast I published.
>
> First, please try to get hold of a desktop, laptop or small server with
> Linux, Unix, or Ubuntu and a modern 'gcc' installed (Ubuntu on Windows is
> also possible).
>
> Then unpack the provided zip file and run
>
> ```
>   ./configure.sh && make && ./bogus 1 9 # optimized code
> ```
>
> which should compile the existing 'bogus' version of this exercise, run
> it, test that it works and print timing information. Then try:
>
> ```
>   ./configure.sh -g && make && ./bogus 1 9 # non-optimized code
> ```
>
> Please, prepare a report answering the following questions. The report
> should be submitted together with the actual source code in a zip file. The
> zip file should also contain the output generated by your programs. If you
> work in a team the quality of the solution is expected to be higher and
> grading will take this into account.
>
> The assignment has 7 questions (percentage achievable points in brackets).
>
>
> [1] Set-Up [10%]
> ----------------
>
> Describe your system including the processor, the number of cores, the
> operating system version and compiler version.

Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz
6 CPU cores, 12 virtual cores

32 GB main memory

Linux sili7 4.15.0-99-generic
#100-Ubuntu SMP Wed Apr 22 20:32:56 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux

gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0

> [2] Analysis of 'bogus' [10%]
> -----------------------------
>
> What running times do you get for both version?
>
> ```
>   ./configure.sh && make && ./bogus 1 9
> ```

```
testing bogus version
initializing 1 workers
executing 1000000000 operations in total
executing 1000000000 operations per worker
SUCCESS: result is 1000000000 as expected
used 0.000 seconds wall-clock and 0.000 process time
utilization 116%, 10330798.0 million operations per second
```

> ```
>   ./configure.sh -g && make && ./bogus 1 9
> ```

```
testing bogus version
initializing 1 workers
executing 1000000000 operations in total
executing 1000000000 operations per worker
SUCCESS: result is 1000000000 as expected
used 2.055 seconds wall-clock and 2.055 process time
utilization 100%, 486.5 million operations per second
```

> Now try the same with two threads:
>
> ```
>   ./configure.sh && make && ./bogus 2 9
> ```

```
testing bogus version
initializing 2 workers
executing 1000000000 operations in total
executing 500000000 operations per worker
SUCCESS: result is 1000000000 as expected
used 0.000 seconds wall-clock and 0.000 process time
utilization 206%, 12336188.2 million operations per second
```

> ```
>   ./configure.sh -g && make && ./bogus 2 9
> ```

```
testing bogus version
initializing 2 workers
executing 1000000000 operations in total
executing 500000000 operations per worker
ERROR: result is 661983624 but expected 1000000000
used 2.568 seconds wall-clock and 4.755 process time
utilization 185%, 389.3 million operations per second
```

> Do the times decrease/increase?

The optimized version did not change (zero seconds for both single and two-threaded version).

For the debugging version we observe a slow-down from single to two-threaded version.

> Do you get correct or incorrect counting?

The result is incorrect for the optimized parallel (two-threaded) version, but correct for the non-optimized version (since the compiler does not move and optimize away accesses even though this needs more investigation).

> What are the three data races in this 'bogus' program.

The first set of data races are accessing 'go', which is read unprotected by worker threads at line 79. In particular the compiler can in principle assume that 'go' is never changed after entering the loop and could thus actually just replace this with 'if (!go) for (;;) ;' and loop forever.

```
   79   while (!go)
   80     ;
```

It is written unprotected at line 164

```
  163
  164   go = true;
  165
```

These (possible) data races might or might not manifest in an actual data race during an exection. Actually in the runs above, it did not happen since the infinite loop above, if really enforced by the compiler, would prevent the program to terminate. This actually happened on my system occasionally for small numbers of the second argument (e.g., './bogus 2 3'). Even though I am not an expert on x86 assembler, but generating assembler with

```
  gcc  -Wall -DNDEBUG -O3 -pthread -o bogus.s bogus.c -S
```

gives 'bogus.s' which contained the following lines for 'run'

```
  run:
  .LFB48:
          .cfi_startproc
          cmpb    $0, go(%rip)
          jne     .L2
  .L3:
          jmp     .L3
          .p2align 4,,10
          .p2align 3
  .L2:
          movq    8(%rdi), %rax
          testq   %rax, %rax
          je      .L4
          addq    %rax, global_result(%rip)
  .L4:
          xorl    %eax, %eax
          ret
```

And there is indeed an infinite loop (.L3: jmp .L3) guarded by checking 'go' against zero. It is unclear why this infinite loop not always triggers.

The second set of data races are on 'global_result' which is read and written by worker threads unprotected at line 84 (through the pointer 'p'):

```
   82   uint64_t *p = &global_result;
   83   for (uint64_t i = 0; i < operations; i++)
   84     *p = *p + 1;
```

In the assembler code above the compiler actually eliminated the whole loop and turned it into 'global_result += operations'.

But even if we protect the access to 'global_result' through 'volatile' as in the answer to the second question, there will be our well-know data-race between two-threads incrementing a common global variable (see 'datarace.mp4'). This data race remains and as such could be considered as third now really conceptual data race.

Reading 'global_result' at line 172 is not a data race since it happens after both worker threads were joined.

Similarly the initialization of 'go' and 'global_result' in the data segment after loading might be considered as a potential data race, but it is not, because initialization happens before worker threads are created.

Thread creation and joining are in this sense synchronizations.

> [3] Fix two of the three data-races by using volatile [10%]
> \---
>
> Two of the data races can be fixed by telling the compiler not to assume
> anything about accessing memory and in particular not optimizing away
> the access (reading and writing a variable or pointer). Use the 'volatile'
> keyword to achieve this

Ok this is simple. We replace

```
   62 static bool go;
   63 static uint64_t global_result;
```

by

```
   62 static volatile bool go;
   63 static volatile uint64_t global_result;
```

and then (otherwise there will be a compiler warning) adapt the type of 'p' on line 82 too

```
    82   volatile uint64_t *p = &global_result;
```

> or an 'ACCESS' macro (as in the 'perfbook').
>
> ```
>   #define ACCESS(P) (* (volatile typeof (P) *) &(P))
> ```

This requires more code, but is considered better, since it allows to mark only problematic accesses to these variable, which the compiler should not reorder nor optimize, and keep the other free.

In this version we keep the original lines 62 and 63 but change line 81 to

```
   81   while (!ACCESS (go))
   82     ;
```

and line 166 accordingly to

```
  166   ACCESS (go) = true;
```

This would have of course exactly the same effect as adding making 'go' volatile but is more explicit, since there are no other accesses to it.

We further have to change line line 86 to

```
   84   uint64_t * p = &global_result;
   85   for (uint64_t i = 0; i < operations; i++)
   86     ACCESS (*p) = ACCESS(*p) + 1;
```

but can keep the read access to 'global_result' in the main thread unprotected (no ACCESS). If you generate assembler code for both versions (I have called them 'volatile.c' for the 'volatile' version and as requested 'incorrect.c' for the ACCESS version) and compare it, you will see that the 'volatile' version has more accesses to 'global_result' (one more in the worker loop and then another additional one for the 'msg' call in 'main').

Running optimized versions ('./configure.sh' and arguments '2 9') gave for 'incorrect' and 'volalite' only very small differences execution times though: taking the average among 10 runs gave differences in favor of 'incorrect' in the range of 10ms (1%) of the running time, which in turn is also similar to the variation of average running times, if we run this experiments (10 runs) repeatedly.

> The resulting program should be called 'incorrect.c' (since it still has
> a data race). Just place it in the same directory. The makefile should
> automatically compile it too. Compare running time with those from '2' with:
>
> ```
>   ./configure.sh && make && ./incorrect 2 9
> ```

```
testing incorrect version
initializing 2 workers
executing 1000000000 operations in total
executing 500000000 operations per worker
ERROR: result is 500900368 but expected 1000000000
```

```
used 1.158 seconds wall-clock and 2.305 process time
utilization 199%, 863.5 million operations per second
```

> `./configure.sh && make && ./incorrect 1 9`

```
testing incorrect version
initializing 1 workers
executing 1000000000 operations in total
executing 1000000000 operations per worker
SUCCESS: result is 1000000000 as expected
used 2.344 seconds wall-clock and 2.344 process time
utilization 100%, 426.6 million operations per second
```

> Do you get speed-up?

Yes, in essence perfect speed-up, but since we have not fixed the main data races (accessing 'global_result') the 2 thread run produces an incorrect result (still the result '500900368' shows that there is some interleaving going on among the threads in accessing 'global_result' - which is good).

> [4] Fix the main data race through locking [10%]
> --
>
> Use 'pthread_mutex_t' as in the demo for the data-race and call the resulting
> program 'locked.c' and run the tests again.

We start with the 'bogus.c' version but make 'go' volatile. Then we protect 'global_result' by a 'mutex' as follows:

```
static volatile bool go;
static uint64_t global_result;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

...

```
  uint64_t *p = &global_result;
  for (uint64_t i = 0; i < operations; i++)
    {
      if (pthread_mutex_lock (&mutex))
        die ("failed to lock mutex");
      *p = *p + 1;
      if (pthread_mutex_unlock (&mutex))
        die ("failed to unlock mutex");
    }
```

> How much slower are these new versions?

For the rest of this report we always use the optimized versions.

```
testing locked version
initializing 1 workers
executing 1000000000 operations in total
executing 1000000000 operations per worker
SUCCESS: result is 1000000000 as expected
used 23.152 seconds wall-clock and 23.152 process time
utilization 100%, 43.2 million operations per second
```

So 10x slower (single threaded).

> This should give the correct result for two threads.

Yes it does, but run-times are extremeley bad:

The locking slows down the program with two threads by another a factor of two.

```
testing locked version
initializing 2 workers
executing 1000000000 operations in total
executing 500000000 operations per worker
SUCCESS: result is 1000000000 as expected
used 49.701 seconds wall-clock and 91.909 process time
utilization 185%, 20.1 million operations per second
```

(this was the fastest of three runs I tried)

> Run it also for 3 and 4 threads to a getter better picture about speed-up.

```
testing locked version
initializing 3 workers
executing 1000000000 operations in total
executing 333333333 operations per worker
SUCCESS: result is 1000000000 as expected
used 63.634 seconds wall-clock and 189.962 process time
utilization 299%, 15.7 million operations per second
```

(again fastest run out of three)

```
testing locked version
initializing 4 workers
executing 1000000000 operations in total
executing 250000000 operations per worker
SUCCESS: result is 1000000000 as expected
used 67.130 seconds wall-clock and 262.803 process time
utilization 391%, 14.9 million operations per second
```

(ditto fastest out of three)

With three and four threads running times increase further, but not as much as going from one to two threads.

Thus the locked version results in slow-down due to too much locking.

> [5] Instead of locking use atomic increment [10%]
> ---
>
> Call this program 'atomic.c'. You probably want to use the now deprecated
> '__sync...' GCC atomic functions instead of C11 atomics (or even C++11 code).
> What are the running times (again for 1,2,3,4 worker threads)?

Again we start with 'bogus.c' and make 'go' volatile. Then the only additional change is making the update to 'global_result' through 'p' atomic using '__sync_fetch_and_add':

```
  uint64_t *p = &global_result;
  for (uint64_t i = 0; i < operations; i++)
    __sync_fetch_and_add (p, 1);
```

This produces correct results but still slow-downs:

```
testing atomic version
initializing 1 workers
executing 1000000000 operations in total
executing 1000000000 operations per worker
SUCCESS: result is 1000000000 as expected
```

```
used 7.204 seconds wall-clock and 7.204 process time
utilization 100%, 138.8 million operations per second

testing atomic version
initializing 2 workers
executing 1000000000 operations in total
executing 500000000 operations per worker
SUCCESS: result is 1000000000 as expected
used 16.488 seconds wall-clock and 31.960 process time
utilization 194%, 60.6 million operations per second

testing atomic version
initializing 3 workers
executing 1000000000 operations in total
executing 333333333 operations per worker
SUCCESS: result is 1000000000 as expected
used 16.932 seconds wall-clock and 47.204 process time
utilization 279%, 59.1 million operations per second

testing atomic version
initializing 4 workers
executing 1000000000 operations in total
executing 250000000 operations per worker
SUCCESS: result is 1000000000 as expected
used 20.658 seconds wall-clock and 75.149 process time
utilization 364%, 48.4 million operations per second
```

The slow-down increases with the number of threads:
I report only one run per number of threads but the runs are typical
(some times more threads took even slightly less time than less).

> ## [6] Progress printing [30%]
>
> Start an additional thread which sleeps for 1/100 of a second and then prints the total accumulated counts in percent (use for instance 'usleep' for sleeping). Use carriage return '\r' instead of '\n' to print the percentage always at the start of the same line (or terminal codes). Then add a 'fflush' call. Devise a method to stop this thread after all workers have finished their operations. You can of course also do a real progress bar if you fancy that. This program is called 'progress.c'. You can base this part on any of the versions before ('bogus.c', 'incorrect.c', 'locked.c', 'atomic.c'). What is the overhead of printing?

We start from 'atomic.c' and add a 'stop' flag similar to 'go' (also volatile), which is set to 'true' after joining the main thread with the worker thread:

```
static volatile bool go;
static volatile bool stop;
static uint64_t global_result;

...

  go = true;

  for (unsigned i = 0; i < num_workers; i++)
    if (pthread_join (workers[i].thread, 0))
      die ("failed to join worker thread %u", i);

  stop = true;
```

In addition we start before a 'show_progress' thread and join it afterwards.

Here is the code of that function.

```
static double
percent (double a, double b)
{
  return b ? 100*a / b : 0;
}

static void *
show_progress (void * dummy)
{
  while (!go)
    ;
  while (!stop)
    {
      printf ("\033[?25l");
      fflush (stdout);
      printf ("\rprogress %6.2f%%",
              percent (global_result, total_operations));
      fflush (stdout);
      printf ("\033[?25h");
      fflush (stdout);
      usleep (10000);
    }
  return dummy;
}
```

After joining with the 'progress' thread we need to print a new-line.

```
  if (pthread_join (progress_thread, 0))
    die ("failed to join progress printing thread");

  printf ("\rprogress %6.2f%%\n",
          percent (global_result, total_operations));
```

Note that this is a statistical 'counter' in the sense of chapter 5 of McKenney's book, since the 'show_progress' functions does not protect the access to 'global_result', which however is not necessary here, since the progress counter does not have to be precise during printing.

Running times are as follows (again just one run and only with 2 threads):

```
testing progress version
initializing 2 workers
executing 1000000000 operations in total
executing 500000000 operations per worker
progress 100.00%
SUCCESS: result is 1000000000 as expected
used 17.996 seconds wall-clock and 35.497 process time
utilization 197%, 55.6 million operations per second
```

(these 'progress 100.00%' line is updated during the run)

The run time is pretty much the same as the run time of the 'atomic' version and also fastest with just one thread.

> [7] Thread local counting [20%]
> ------------------------------
>
> Move to a thread local counting scheme 'local.c' (start with 'incorrect.c'),
> where each worker thread has its own result, which after a worker has
> finished is added to the global result. Adapt the progress printing thread
> to peek into local thread counters and compute a global count result. There

> are many ways to do this. Again see the corresponding chapter 5 in the
> 'perfbook'. You can try all but one is enough (the simplest statistical
> inexact counting version). A simple solution is to add local counters
> to the 'worker' structure. But then be careful with 'false sharing', where
> different threads write to the same cache line. Accordingly experiment with
> padding (to put those counters into different cache lines).

We started with 'incorrect.c' and added (a non-volatile) 'stop' flag.

```
static bool go;
static bool stop;
static uint64_t global_result;
```

The worker struct (without padding) is as follows:

```
struct worker
{
  pthread_t thread;
  uint64_t operations;
  uint64_t local_result;
};

```

The worker now accesses only local result:

```
  uint64_t *p = &worker->local_result;
  for (uint64_t i = 0; i < operations; i++)
    ACCESS (*p) = ACCESS(*p) + 1;
```

We further made sure to use ACCESS macros in 'show_progress' now (alternatively we could have made 'go' and 'stop' volatile) and simply count over the local results of the treads to get an approximate global result count:

```
static void *
show_progress (void * dummy)
{
  while (!ACCESS (go))
    ;
  while (!ACCESS (stop))
    {
      uint64_t progress_result = 0;
      for (unsigned i = 0; i < num_workers; i++)
        progress_result += workers[i].local_result;
      printf ("\033[?25l");
      fflush (stdout);
      printf ("\rprogress %6.2f%%",
              percent (progress_result, total_operations));
```

The 'go' and 'stop' flags are set to true using 'ACCESS' in the main thread and we need another loop to compute the final 'global_result':

```
  ACCESS (go) = true;

  for (unsigned i = 0; i < num_workers; i++)
    if (pthread_join (workers[i].thread, 0))
      die ("failed to join worker thread %u", i);

  ACCESS (stop) = true;

  for (unsigned i = 0; i < num_workers; i++)
    global_result += workers[i].local_result;
```

This finally gives a scalable program.

```
initializing 4 workers
executing 1000000000 operations in total
executing 250000000 operations per worker
progress 100.00%
SUCCESS: result is 1000000000 as expected
used 0.672 seconds wall-clock and 2.602 process time
utilization 387%, 1487.5 million operations per second
```

To determine the scaling I used 1e10 operations and

```
  i=1; while [ $i -le 18 ]; do \
    printf "$i workers "; ./local $i 10|grep used; \
    i=`expr $i + 1`; \
  done
```

and got the following:

```
1 workers used 23.459 seconds wall-clock and 23.499 process time
2 workers used 11.820 seconds wall-clock and 23.533 process time
3 workers used 8.937 seconds wall-clock and 26.489 process time
4 workers used 6.751 seconds wall-clock and 26.320 process time
5 workers used 5.885 seconds wall-clock and 28.080 process time
6 workers used 5.079 seconds wall-clock and 29.763 process time
7 workers used 4.457 seconds wall-clock and 28.580 process time
8 workers used 3.924 seconds wall-clock and 28.956 process time
9 workers used 4.811 seconds wall-clock and 32.035 process time
10 workers used 4.404 seconds wall-clock and 32.083 process time
11 workers used 3.926 seconds wall-clock and 33.098 process time
12 workers used 3.891 seconds wall-clock and 36.901 process time
13 workers used 3.156 seconds wall-clock and 32.266 process time
14 workers used 2.927 seconds wall-clock and 30.885 process time
15 workers used 2.922 seconds wall-clock and 31.893 process time
16 workers used 2.880 seconds wall-clock and 30.984 process time
17 workers used 3.038 seconds wall-clock and 31.250 process time
18 workers used 2.596 seconds wall-clock and 28.558 process time
```

So until the real number of cores (6) is used we got linear speed-ups, then the speed-ups fluctuate (sometimes they increase even).

Padding the workers to be cache line size aligned did not really help. Here is what would be needed to change for the 'worker struct':

```
struct worker
{
  pthread_t thread;
  uint64_t operations;
  uint64_t local_result;
} __attribute__ ((aligned (64)));
```

The padding actually can only help if we would need atomic operations on these, which without padding generates many cache flushing events. This is not needed for the statistical counter we used. So in order to test these 'false sharing' issue we copied 'local.c' to 'precise.c' and factored out the 'local_result' in a separate array to cramp all those local counters into as few cachelines as possible (for 8 threads one cache line of 64 bytes for instance). The 'show_progress' thread is still statistical. Then we replace the update to 'local_result' to be atomic (as in 'atomic.c').

Here are the running times we get:

```
    i=1; while [ $i -le 6 ]; do \
      printf "$i workers "; ./precise $i 9|grep used; \
      i=`expr $i + 1`; \
    done
```

(note only 1e9 operations)

```
1 workers used 7.197 seconds wall-clock and 7.210 process time
2 workers used 23.485 seconds wall-clock and 46.726 process time
3 workers used 17.181 seconds wall-clock and 47.055 process time
4 workers used 20.438 seconds wall-clock and 75.098 process time
5 workers used 22.026 seconds wall-clock and 99.850 process time
6 workers used 13.745 seconds wall-clock and 61.133 process time
```

so similar to 'atomic.c' (which was expected).

If we now increase the 'local_result' array by a factor of 8 to match the cache line size of 64 bytes and always index the array by multiplying indices by 8 (alternatively we could have added back the 'local_result' to the worker struct and make the struct 64 bytes aligned) we get

```
    i=1; while [ $i -le 6 ]; do \
      printf "$i workers "; ./padded $i 9|grep used; \
      i=`expr $i + 1`; \
    done
```

(again only 1e9 operations)

```
1 workers used 7.198 seconds wall-clock and 7.210 process time
2 workers used 3.643 seconds wall-clock and 7.253 process time
3 workers used 2.402 seconds wall-clock and 7.207 process time
4 workers used 1.810 seconds wall-clock and 7.232 process time
5 workers used 1.496 seconds wall-clock and 7.308 process time
6 workers used 1.262 seconds wall-clock and 7.323 process time
```

which shows that ignoring false sharing can give an order of magnitude slow-dow. This version is not as good as thread local counting though (four workers too there 0.672 seconds, so by a factor of 3 roughly). Also the progress bar is still not precise and one could try some of the other counters in the 'perfbook' as a continuation of this exercise.