Indian Institute of Information Technology, Vadodara

Parallel Programming(CS403)

Prof. Reshmi Mitra

Name - Hemant Kumar
ID - 201352026
Collaborator - Dilip Puri(201351014)

Lab 05

Submission Date - October 3, 2016 **Deadline** - Oct 03, 11:59 PM

---

1 In the Helgrind tutorial link, familarize yourself with the use for:

1. Data race condition

   A data race occurs when multiple threads access a shared memory location with undetermined accessing order and at least one access is to write a new data into the shared memory location. In the program data race condition occurs when we want to increment same count with multiple threads.

2. Ordering in POSIX locks

   –track-lockorders=no—yes [default: yes]

   When enabled (the default), Helgrind performs lock order consistency checking. For some buggy programs, the large number of lock order errors reported can become annoying, particularly if you're only interested in race errors. You may therefore find it helpful to disable lock order checking.

3. POSIX misuses detection

   there are many events when threads run on invalid time such as:
   1. unlocking an invalid mutex
   2. unlocking a not-locked mutex
   3. unlocking a mutex held by a different thread
   4. destroying an invalid or a locked mutex
   5. recursively locking a non-recursive mutex

2 - refer to the attached file

1 verify the following

1. the put runs faster with 4 cores than 2 cores, but not 4 times as fast as a single core.

```
hemant@hemant:~$ cd Downloads
hemant@hemant:~/Downloads$ gcc -g -o valgrind_1 valgrind_1.c -pthread
hemant@hemant:~/Downloads$ ./valgrind_1 2
completion time for put phase = 1.835714
0: 22 keys missing
1: 22 keys missing
completion time for get phase = 3.561289
hemant@hemant:~/Downloads$ ./valgrind_1 4
completion time for put phase = 1.407124
1: 26 keys missing
2: 29 keys missing
3: 28 keys missing
0: 29 keys missing
completion time for get phase = 3.762213
hemant@hemant:~/Downloads$
```

Output of fixed code

```
hemant@hemant:~/Desktop/pp/parallel-programming-lab/lab5/code$ ./valgrind_1 4
completion time for put phase = 3.157786
2: 0 keys missing
0: 0 keys missing
3: 0 keys missing
1: 0 keys missing
completion time for get phase = 3.371561
```

as we can see in the figure that is the result when the threads are run for 2 threads and 4 threads, the time taken on 4 threads is less but not the half of the time taken when 4 threads are used. this is happening when there occurs the data race condition as multiple threads tries to get the lock but the threads have to wait for the other thread to unlock the mutex lock. so more the threads more will be the race condition occurred.

2. the get phase ran even slower

   because of the more missing keys it took more time to get the access to the hash table and read the data. Also there were times when the mutex lock could not happen. Because of this mutex lock more conflicts are coming with the threads. The application inserted 22+22 keys in phase 1 that phase 2 couldn't find.

2 Use Valgrind to find race condition. Using mutex lock to fix the race condition. Argue that your changes ensures correctness

```
==7280== This conflicts with a previous write of size 4 by thread #2
==7280== Locks held: none
==7280==    at 0x400CAB: put (valgrind_1.c:65)
==7280==    by 0x400E6F: put_thread (valgrind_1.c:100)
==7280==    by 0x4C30FA6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd
ux.so)
==7280==    by 0x4E45181: start_thread (pthread_create.c:312)
==7280==    by 0x515630C: clone (clone.S:111)
==7280==  Address 0x33c8d28 is 48000008 bytes inside data symbol "table"
==7280==
```

As we can see here that the two threads are conflicting to get the access to the table to put the data to the hash table but because there is is no ordering sequence that is causing no acces to both the threads.

We can solve this problem by putting the lock for access before each thread that gets access to the table to update the data and then unlock the access for the other threads to get access.

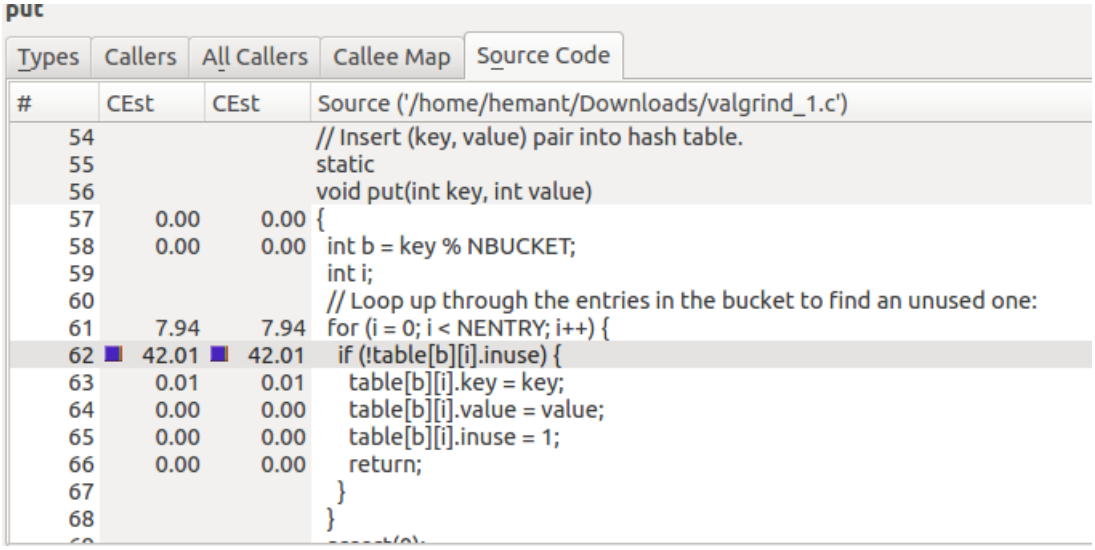
```
put_thread(void *xa)
{
    assert(pthread_mutex_lock(&lock) == 0);
  long n = (long) xa;
  int i;
  int b = NKEYS/nthread;

  for (i = 0; i < b; i++) {
    put(keys[b*n + i], n);
  }
    assert(pthread_mutex_unlock(&lock) == 0);
}
```

Here we have taken mutex lock for the process that i have described above the figure.

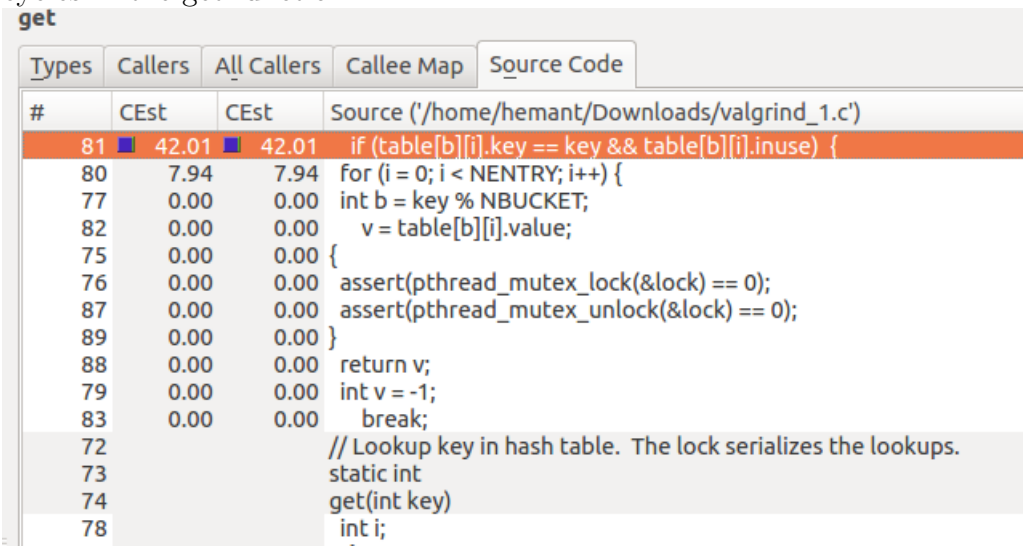
3 Using “cachegrind” and “kcachegrind”, identify the most expensive function calls in terms of cycles spent. Instructions for the tools are available in Lab 4. Attach screen-shots whenever necessary.

1. cycles in the put function



This is the screen shot showing that the if statement after the for loop is taking the maximum cycles as it has to check on every thread call whether the particular location is available or not to read the data.

2. cycles in the get function



In put function also the if condition which is running after the for loop is taking the maximum cycles of the other functions.

But both the places are taking estimated cycles and are equal. So we can not distinguish between these positions.

Thank You