REPORT

Changkun Jiang (1008008344)
Wu Hung Mao (1008134207)

6 OCT 2025

Valgrind Helgrind Report

```
==113804== Possible data race during write of size 4 at 0x10D244 by thread #3
==113804== Locks held: none
==113804==    at 0x10A01F: admit_jobs (jobs.c:314)
==113804==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==113804==    by 0x4914AC2: start_thread (pthread_create.c:442)
==113804==    by 0x49A5A73: clone (clone.S:100)

==113804== This conflicts with a previous read of size 4 by thread #2
==113804== Locks held: 1, at address 0x10D1A0
==113804==    at 0x10A0D1: execute_jobs (jobs.c:356)
==113804==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==113804==    by 0x4914AC2: start_thread (pthread_create.c:442)
==113804==    by 0x49A5A73: clone (clone.S:100)

==113804==  Address 0x10d244 is 516 bytes inside data symbol "tassadar"
```

Issue: Data race can be a problem when shared resource within a critical session is unprotected. Imagine two threads need to both write to the same global variable, the outcome of the global variable is non-deterministic and depends on the order of thread execution.

```
==114130== Thread #3: Attempt to re-lock a non-recursive lock I already hold
==114130==    at 0x4850C04: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==114130==    by 0x109F6A: admit_jobs (jobs.c:291)
==114130==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==114130==    by 0x4914AC2: start_thread (pthread_create.c:442)
==114130==    by 0x49A5A73: clone (clone.S:100)
==114130==  Lock was previously acquired
==114130==    at 0x4850CCF: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==114130==    by 0x109F6A: admit_jobs (jobs.c:291)
==114130==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==114130==    by 0x4914AC2: start_thread (pthread_create.c:442)
==114130==    by 0x49A5A73: clone (clone.S:100)
```

Issue: A missing unlock would cause the thread to hold the lock indefinitely. Other threads would be unable to obtain the lock, and unable to make any progress.

```
==114267== Thread #3 unlocked a not-locked lock at 0x10D1A0
==114267==    at 0x48511E6: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==114267==    by 0x109F6A: admit_jobs (jobs.c:290)
==114267==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==114267==    by 0x4914AC2: start_thread (pthread_create.c:442)
==114267==    by 0x49A5A73: clone (clone.S:100)
==114267==  Lock at 0x10D1A0 was first observed
==114267==    at 0x4854BFE: pthread_mutex_init (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==114267==    by 0x1098DF: init_executor (jobs.c:110)
==114267==    by 0x10949D: main (executor.c:62)
==114267==  Address 0x10d1a0 is 352 bytes inside data symbol "tassadar"
```

Issue: In my implementation, the program still runs to completion even though I attempt to unlock a non-locking lock. However, I suspect unlocking a non-locking lock could pollute internal state of a lock. This would lead to undefined behaviour when another thread try to acquire it.

```
==115595== Thread #5 unlocked lock at 0x10D250 currently held by thread #4
==115595==    at 0x48511E6: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==115595==    by 0x109FCE: admit_jobs (jobs.c:306)
==115595==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==115595==    by 0x4914AC2: start_thread (pthread_create.c:442)
```

```
==115595==    by 0x49A5A73: clone (clone.S:100)
==115595==  Lock at 0x10D250 was first observed
==115595==    at 0x4854BFE: pthread_mutex_init (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==115595==    by 0x1098DF: init_executor (jobs.c:110)
==115595==    by 0x10949D: main (executor.c:62)
==115595==  Address 0x10d250 is 528 bytes inside data symbol "tassadar"
```

Issue: Thread 5 tries to unlock admission lock that is currently held by thread 4. My program did not run to completion. Similar to above, thread 5 could have polluted the internal state of admission lock by unlocking it. When thread 4 wants to unlock it or other threads try to lock it, the lock might not be working properly anymore.

```
==115904== Thread #8: pthread_cond_{timed}wait called with un-held mutex
==115904==    at 0x4853A15: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==115904==    by 0x10A093: execute_jobs (jobs.c:347)
==115904==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==115904==    by 0x4914AC2: start_thread (pthread_create.c:442)
==115904==    by 0x49A5A73: clone (clone.S:100)
```

Issue: The execution thread tries to wait on a lock that is not being held by any other thread. My program hangs after the execution thread reaches `pthread_cond_wait(&q->admission_cv, &q->lock)`. I suspect that the execution thread is put to sleep and the admission thread is unable to admit a job into the queue and wake up the execution thread. Since wait attempts to unlock a lock, this could cause corruption in the internal state of a lock.

In summary:

1. **Data race:** Possible data race during write ...
2. **Missing unlock:** Attempt to re-lock a non-recursive lock that the thread already holds....
3. **Unlocking a non-locked mutex:** “unlocked a not-locked lock ...
4. **Unlocking a mutex held by a different thread:** “unlocked lock at ...
5. **Calling `pthread_cond_wait` with an unheld mutex:** “pthread_cond_{timed}wait called with un-held mutex...

In addition to these clear messages, developers should inspect the reported memory addresses, call stacks, and thread identifiers to better understand the behavior. The developer should then use this information in GDB to pinpoint synchronization issues in the program. Useful GDB commands include `info threads`, `thread <id>`, `bt`, and `print`, which help reveal the root of synchronization issues. Based on my experience, a synchronization bug reported by a test might not be reproducible under GDB because the debugger disturb thread scheduling and can prevent the specific context switch that triggers the bug.