# Concurrent Programming Issue

my goal is to implement concurrent programming code in which the Final result should be 200000, instead i got around 199990. where did i get wrong?

```
/* futex_demo.c

    Usage: futex_demo [nloops]
                    (Default: 5)

    Demonstrate the use of futexes in a program where parent and child
    use a pair of futexes located inside a shared anonymous mapping to
    synchronize access to a shared resource: the terminal. The two
    processes each write 'num-loops' messages to the terminal and employ
    a synchronization protocol that ensures that they alternate in
    writing messages.
*/
#define _GNU_SOURCE
#include <err.h>
#include <errno.h>
#include <linux/futex.h>
#include <stdatomic.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/syscall.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <unistd.h>
#include <pthread.h>
#include <stdatomic.h>

/* Global lock variable */
#define MAX 100000
```

```
int count = 0;
uint32_t lock = 1; // Initialize to 1 to
represent "unlocked" state

static int
futex(uint32_t *uaddr, int futex_op, uint32_t
val,
        const struct timespec *timeout, uint32_t
*uaddr2, uint32_t val3)
{
    return syscall(SYS_futex, uaddr, futex_op,
val,
                    timeout, uaddr2, val3);
}

/* Acquire the futex pointed to by 'futexp': wait
for its value to
    become 1, and then set the value to 0. */

static void
acquire(uint32_t *futexp)
{
    long            s;
    const uint32_t  one = 1;

    /* atomic_compare_exchange_strong(ptr,
oldval, newval)
        atomically performs the equivalent of:

            if (*ptr == *oldval)
                *ptr = newval;

        It returns true if the test yielded true
and *ptr was updated. */

    while (1) {

        /* Is the futex available? */
        if
(atomic_compare_exchange_strong(futexp, &one, 0))
            break;      /* Yes */
        /* Futex is not available; wait. */

        s = futex(futexp, FUTEX_WAIT, 0, NULL,
NULL, 0);
        if (s == -1 && errno != EAGAIN)
            err(EXIT_FAILURE, "futex-
FUTEX_WAIT");
    }
}

/* Release the futex pointed to by 'futexp': if
the futex currently
    has the value 0, set its value to 1 and then
wake any futex waiters,
    so that if the peer is blocked in fwait(), it
can proceed. */

static void
release(uint32_t *futexp)
{
    long            s;
    const uint32_t  zero = 0;

    /* atomic_compare_exchange_strong() was
described
        in comments above. */

    if (atomic_compare_exchange_strong(futexp,
&zero, 1)) {
        s = futex(futexp, FUTEX_WAKE, 1, NULL,
NULL, 0);
        if (s  == -1)
            err(EXIT_FAILURE, "futex-
FUTEX_WAKE");
    }
}


void* increasing(void* args) {
    for (int i = 0; i < MAX; i++) {
        acquire(&lock);
        count++;
```

```
            release(&lock);
        }
        return NULL;
    }

    int main(int argc, char* args[]) {
        pthread_t even_thread, odd_thread;

        pthread_create(&even_thread, NULL,
    increasing, NULL);
        pthread_create(&odd_thread, NULL, increasing,
    NULL);

        pthread_join(even_thread, NULL);
        pthread_join(odd_thread, NULL);

        printf("Finally, n = %d\n", count);

        return 0;
    }
```

The issue you're encountering stems from a combination of problems in your synchronization code:

1. **Non-atomic lock variable**: You're using a regular `uint32_t` for your lock variable when it should be atomic.
2. **Misuse of `atomic_compare_exchange_strong`**: You're passing `const` variables as the expected value, which is incorrect because this function modifies the expected value.
3. **Not resetting the expected value**: After a failed compare-and-exchange, the expected value is overwritten and needs to be reset for the next iteration.

Here's how you can fix these issues:

1. **Declare the lock as an atomic variable**:

   ```c
   atomic_uint lock = 1; // Initialize to 1 to represent "unlocked" state
   ```

2. **Modify the `acquire` function**:

c

```
static void acquire(atomic_uint *futexp) {
    uint32_t expected;

    while (1) {
        expected = 1;
        if (atomic_compare_exchange_strong(futexp, &expected, 0))
            break; // Acquired the lock

        // Futex is not available; wait
        if (futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0) == -1 && errno != EAGAIN)
            err(EXIT_FAILURE, "futex-FUTEX_WAIT");
    }
}
```

3. **Modify the `release` function:**

c

```
static void release(atomic_uint *futexp) {
    uint32_t expected = 0;

    if (atomic_compare_exchange_strong(futexp, &expected, 1)) {
        // Wake up one waiter
        if (futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0) == -1)
            err(EXIT_FAILURE, "futex-FUTEX_WAKE");
    }
}
```

4. **Update your `main` function and variable types accordingly:**

c

```
int main(int argc, char* args[]) {
    pthread_t even_thread, odd_thread;

    pthread_create(&even_thread, NULL, increasing, NULL);
    pthread_create(&odd_thread, NULL, increasing, NULL);

    pthread_join(even_thread, NULL);
    pthread_join(odd_thread, NULL);
    printf("Finally, n = %d\n", count);

    return 0;
}
```

5. **Ensure the `count` variable is atomic if necessary** (since you're protecting it with a lock, this may not be strictly necessary, but it can prevent issues):

   ```c
   int count = 0;
   ```

Here's the corrected code with these changes applied:

```c
#define _GNU_SOURCE
#include <err.h>
#include <errno.h>
#include <linux/futex.h>
#include <stdatomic.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <pthread.h>

#define MAX 100000
int count = 0;
atomic_uint lock = 1; // Initialize to 1 to represent "unlocked" state

static int futex(uint32_t *uaddr, int futex_op, uint32_t val,
                 const struct timespec *timeout, uint32_t *uaddr2, uint32_t val3) {
    return syscall(SYS_futex, uaddr, futex_op, val, timeout, uaddr2, val3);
}

static void acquire(atomic_uint *futexp) {
    uint32_t expected;

    while (1) {
```

```
        expected = 1;
        if (atomic_compare_exchange_strong(futexp, &expected, 0))
            break; // Acquired the lock

        // Futex is not available; wait
        if (futex((uint32_t *)futexp, FUTEX_WAIT, 0, NULL, NULL, 0) == -1 && errno
!= EAGAIN)
            err(EXIT_FAILURE, "futex-FUTEX_WAIT");
    }
}

static void release(atomic_uint *futexp) {
    uint32_t expected = 0;

    if (atomic_compare_exchange_strong(futexp, &expected, 1)) {
        // Wake up one waiter
        if (futex((uint32_t *)futexp, FUTEX_WAKE, 1, NULL, NULL, 0) == -1)
            err(EXIT_FAILURE, "futex-FUTEX_WAKE");
    }
}

void* increasing(void* args) {
    for (int i = 0; i < MAX; i++) {
        acquire(&lock);
        count++;
        release(&lock);
    }
    return NULL;
}

int main(int argc, char* args[]) {
    pthread_t even_thread, odd_thread;

    pthread_create(&even_thread, NULL, increasing, NULL);
    pthread_create(&odd_thread, NULL, increasing, NULL);

    pthread_join(even_thread, NULL);
    pthread_join(odd_thread, NULL);

    printf("Finally, n = %d\n", count);
    return 0;
}
```

**Explanation of the Fixes**:

- **Atomic Lock Variable**: Declaring `lock` as `atomic_uint` ensures that all atomic operations on it are performed correctly and that memory ordering is properly handled.
- **Proper Use of `atomic_compare_exchange_strong`**:
  - The `expected` value passed to `atomic_compare_exchange_strong` should not be `const` because it may be modified by the function.
  - Before each call to `atomic_compare_exchange_strong`, reset `expected` to the value you expect (`1` for `acquire`, `0` for `release`).
- **Resetting `expected`**:
  - If `atomic_compare_exchange_strong` fails, it updates `expected` with the current value of `*futexp`. You need to reset `expected` before the next iteration; otherwise, the function will compare against the wrong value in subsequent iterations.
- **Casting in Futex Calls**:
  - Since `atomic_uint` may have different alignment or representation, cast it to `uint32_t*` when passing to the `futex` system call.

After making these changes, your program should correctly synchronize the threads and the final result should consistently be `200000`.

explain the parallel between the pseudo code and the actual code.

pseudo code:

```
#define FREE 0
#define BUSY 1
int guard = FREE;
int lock = FREE;

void acquire(int* lock) {
    while (test&set(guard));          // Acquire guard lock
    if (*lock == BUSY) {              // Check if lock is busy
        put_thread_on_wait_queue();    // Put thread on wait queue
        go_to_sleep();                  // Put thread to sleep
        guard = FREE;                   // Release guard
    } else {
        *lock = BUSY;                   // Acquire lock
        guard = FREE;                   // Release guard
    }
}

void release(int* lock) {
    while (test&set(guard));            // Acquire guard lock
    if (any_thread_on_wait_queue()) {   // Check if there are waiting threads
        take_thread_off_wait_queue();   // Take one thread off wait queue
        place_thread_on_ready_queue();  // Place the thread on the ready queue
    } else {
        *lock = FREE;                   // Set lock to FREE if no waiting threads
    }
    guard = FREE;                       // Release guard
}


actual code:
/* futex_demo.c

    Usage: futex_demo [nloops]
                    (Default: 5)

    Demonstrate the use of futexes in a program where parent and child
    use a pair of futexes located inside a shared anonymous mapping to
    synchronize access to a shared resource: the terminal. The two
    processes each write 'num-loops' messages to the terminal and employ
    a synchronization protocol that ensures that they alternate in
    writing messages.
*/
```

```
#define _GNU_SOURCE
#include <err.h>
#include <errno.h>
#include <linux/futex.h>
#include <stdatomic.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/syscall.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <unistd.h>
#include <pthread.h>
#include <stdatomic.h>

/* Global lock variable */
#define MAX 100000
int count = 0;
uint32_t lock = 1; // Initialize to 1 to represent "unlocked"
state

static int futex(uint32_t *uaddr, int futex_op, uint32_t val,
        const struct timespec *timeout, uint32_t *uaddr2,
uint32_t val3)
{
    return syscall(SYS_futex, uaddr, futex_op, val,
                    timeout, uaddr2, val3);
}

/* Acquire the futex pointed to by 'futexp': wait for its value
to
    become 1, and then set the value to 0. */

static void acquire(uint32_t *futexp)
{
    long            s;
    uint32_t  one; // interestingly, if declare const uint32_t, it
will not work perfectly.

    /* atomic_compare_exchange_strong(ptr, oldval, newval)
        atomically performs the equivalent of:
            if (*ptr == *oldval)
                *ptr = newval;

        It returns true if the test yielded true and *ptr was
updated. */

    while (1) {
        one = 1;
        /* Is the futex available? */
        if (atomic_compare_exchange_strong(futexp, &one, 0))
            break;      /* Yes */

        /* Futex is not available; wait. */

        s = futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
        if (s == -1 && errno != EAGAIN)
            err(EXIT_FAILURE, "futex-FUTEX_WAIT");
    }
}

/* Release the futex pointed to by 'futexp': if the futex
currently
    has the value 0, set its value to 1 and then wake any futex
waiters,
    so that if the peer is blocked in fwait(), it can proceed. */

static void
release(uint32_t *futexp)
{
    long            s;
    const uint32_t  zero = 0;

    /* atomic_compare_exchange_strong() was described
        in comments above. */

    if (atomic_compare_exchange_strong(futexp, &zero, 1)) {
        s = futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
        if (s  == -1)
            err(EXIT_FAILURE, "futex-FUTEX_WAKE");
    }
}
```

```
void* increasing(void* args) {
    for (int i = 0; i < MAX; i++) {
        acquire(&lock);
        count++;
        release(&lock);
    }
    return NULL;
}

int main(int argc, char* args[]) {
    pthread_t even_thread, odd_thread;

    pthread_create(&even_thread, NULL, increasing, NULL);
    pthread_create(&odd_thread, NULL, increasing, NULL);

    pthread_join(even_thread, NULL);
    pthread_join(odd_thread, NULL);

    printf("Finally, n = %d\n", count);

    return 0;
}
```

3/3

### Explanation of the Parallel Between the Pseudo Code and the Actual Code

Your pseudo code and actual code both implement a mutual exclusion (mutex) lock using low-level atomic operations and futexes to synchronize threads. Let's map each part of your pseudo code to the corresponding part of your actual code to understand how they align.

---

## Definitions and Initializations

**Pseudo Code:**

```c
```

```
#define FREE 0
#define BUSY 1
int guard = FREE;
int lock = FREE;
```

- **FREE** and **BUSY**: Constants representing the lock states.
- **guard**: A secondary lock (spinlock) to protect the critical section where the main lock and wait queue are manipulated.
- **lock**: The main mutex lock variable.

**Actual Code:**

```
c

uint32_t lock = 1; // Initialize to 1 to represent "unlocked" state
```

- **lock**: The main mutex lock variable initialized to `1`, representing the unlocked state.

**Mapping:**

- In your actual code, you use `1` to represent the **unlocked** state and `0` for the **locked** state, which is the opposite of the pseudo code where `FREE` is `0` and `BUSY` is `1`.
- There is no explicit `guard` variable in the actual code. Instead, atomic operations and futex system calls handle synchronization without a separate guard lock.

---

## Acquire Function

**Pseudo Code:**

```
c

void acquire(int* lock) {
    while (test&set(guard));          // (1) Acquire guard lock
    if (*lock == BUSY) {              // (2) Check if lock is busy
        put_thread_on_wait_queue();   // (3) Put thread on wait queue
        go_to_sleep();                // (4) Put thread to sleep
        guard = FREE;                 // (5) Release guard
    } else {
        *lock = BUSY;                  // (6) Acquire lock
        guard = FREE;                  // (7) Release guard
    }
}
```

**Actual Code:**

```c
static void acquire(uint32_t *futexp)
{
    long s;
    uint32_t one;

    while (1) {
        one = 1;
        // (1 & 2) Attempt to acquire lock atomically
        if (atomic_compare_exchange_strong(futexp, &one, 0))
            break; // (6) Lock acquired

        // (3 & 4) Futex is not available; wait
        s = futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
        if (s == -1 && errno != EAGAIN)
            err(EXIT_FAILURE, "futex-FUTEX_WAIT");
    }
}
```

**Mapping Steps:**

1. **Acquiring the Guard Lock:**
    - *Pseudo Code (1):* `while (test&set(guard));` acquires the guard lock to ensure exclusive access to the lock manipulation and wait queue.
    - *Actual Code (1 & 2):* The `atomic_compare_exchange_strong` function atomically checks if `*futexp` is `1` (unlocked) and sets it to `0` (locked). This operation serves both as acquiring the guard and attempting to acquire the lock.
2. **Checking if Lock is Busy:**
    - *Pseudo Code (2):* Checks if the lock is already held (`*lock == BUSY`).

   - *Actual Code (1 & 2):* If `atomic_compare_exchange_strong` fails, it indicates the lock is busy ( `*futexp` was not `1` ).

3. **Putting Thread on Wait Queue:**
   - *Pseudo Code (3 & 4):* The thread is placed on a wait queue and put to sleep.
   - *Actual Code (3 & 4):* The thread calls `futex` with `FUTEX_WAIT` , which puts it to sleep if the lock value is still `0` .
4. **Releasing the Guard Lock:**
   - *Pseudo Code (5):* `guard = FREE;` releases the guard lock.
   - *Actual Code:* Since there's no explicit guard, this step isn't directly mapped. The atomicity of `atomic_compare_exchange_strong` ensures thread safety.
5. **Acquiring the Lock:**
   - *Pseudo Code (6 & 7):* If the lock was free, it sets `*lock = BUSY` and releases the guard.
   - *Actual Code (6):* Successfully setting `*futexp` to `0` means the lock is acquired.

---

## Release Function

### Pseudo Code:

```c
void release(int* lock) {
    while (test&set(guard));            // (1) Acquire guard lock
    if (any_thread_on_wait_queue()) {   // (2) Check if there are waiting threads
        take_thread_off_wait_queue();   // (3) Take one thread off wait queue
        place_thread_on_ready_queue();  // (4) Place the thread on the ready queue
    } else {
        *lock = FREE;                   // (5) Set lock to FREE if no waiting threads
    }
    guard = FREE;                       // (6) Release guard
}
```

### Actual Code:

```c
static void release(uint32_t *futexp)
{
    long s;
    const uint32_t zero = 0;

    // (1 & 2) Attempt to release lock atomically
    if (atomic_compare_exchange_strong(futexp, &zero, 1)) {
        // (3 & 4) Wake up one waiter
        s = futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
        if (s == -1)
            err(EXIT_FAILURE, "futex-FUTEX_WAKE");
    }
}
```

**Mapping Steps:**

1. **Acquiring the Guard Lock:**
    - *Pseudo Code (1):* Acquires the guard lock to ensure exclusive access during lock release.
    - *Actual Code (1 & 2):* The `atomic_compare_exchange_strong` function atomically changes the lock from `0` to `1`, serving both to safely check and modify the lock.
2. **Checking for Waiting Threads:**
    - *Pseudo Code (2):* Checks if any threads are waiting on the lock.
    - *Actual Code (1 & 2):* The success of `atomic_compare_exchange_strong` implies that the lock was held (`*futexp == 0`). We proceed to wake up any waiting threads.
3. **Waking Up Waiting Threads:**
    - *Pseudo Code (3 & 4):* A waiting thread is taken off the wait queue and placed on the ready queue.
    - *Actual Code (3 & 4):* `futex` with `FUTEX_WAKE` wakes up one waiting thread.
4. **Setting Lock to Free:**
    - *Pseudo Code (5):* Sets the lock to `FREE` if no threads are waiting.
    - *Actual Code:* The lock has already been set to `1` (unlocked) by `atomic_compare_exchange_strong`.

5. **Releasing the Guard Lock:**
    - *Pseudo Code (6):* Releases the guard lock.
    - *Actual Code:* No explicit guard lock to release.

---

## Key Differences and Similarities

- **Guard Lock:**
    - *Pseudo Code:* Uses a separate `guard` variable to protect critical sections.
    - *Actual Code:* Relies on the atomicity of `atomic_compare_exchange_strong` and futex system calls, eliminating the need for an explicit guard lock.
- **Lock States:**
    - *Pseudo Code:* `FREE` is `0`, `BUSY` is `1`.
    - *Actual Code:* `1` represents unlocked, `0` represents locked.
- **Atomic Operations:**
    - Both codes use atomic operations to ensure that only one thread can modify the lock at a time.
- **Waiting Mechanism:**
    - *Pseudo Code:* Abstract functions like `put_thread_on_wait_queue()` and `go_to_sleep()`.
    - *Actual Code:* Uses the `futex` system call with `FUTEX_WAIT` to put threads to sleep.
- **Waking Mechanism:**
    - *Pseudo Code:* Abstract functions to manage wait queues and ready queues.
    - *Actual Code:* Uses `futex` with `FUTEX_WAKE` to wake up waiting threads.

---

## Detailed Walkthrough of the Actual Code Functions

### Acquire Function (`acquire`)

1. **Initialize Expected Value:**

```c
one = 1;
```

- Sets the expected current value of the lock to `1` (unlocked).

2. **Attempt to Acquire Lock Atomically:**

```c
if (atomic_compare_exchange_strong(futexp, &one, 0))
    break; // Lock acquired
```

- Tries to change `*futexp` from `1` to `0`. If successful, the lock is acquired.
- If it fails, `one` is updated with the actual value of `*futexp`.

3. **Wait if Lock is Busy:**

```c
s = futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
```

- The thread sleeps until `*futexp` is no longer `0`.
- This corresponds to the thread waiting for the lock to become available.

**Release Function (`release`)**

1. **Initialize Expected Value:**

```c
const uint32_t zero = 0;
```

- Sets the expected current value of the lock to `0` (locked).

2. **Attempt to Release Lock Atomically:**

```c
if (atomic_compare_exchange_strong(futexp, &zero, 1))
```

- Tries to change `*futexp` from `0` to `1`. If successful, the lock is released.

3. **Wake Up Waiting Threads:**

```c
s = futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
```

  - Wakes up one thread waiting on the lock.

---

## Why the Guard Lock is Implicit in the Actual Code

- **Atomic Operations:**
  - The `atomic_compare_exchange_strong` function ensures that the read-modify-write operation on the lock variable is atomic. This means no two threads can successfully modify the lock at the same time.
- **Synchronization Without Explicit Guard:**
  - By using atomic operations and futexes, the need for an explicit guard lock is eliminated. The atomic operations prevent race conditions on the lock variable itself.
- **Efficiency:**
  - Eliminating the guard lock reduces overhead and complexity, making the code more efficient.

---

## Understanding the Futex System Calls

- `FUTEX_WAIT`:
  - The calling thread sleeps if `*futexp` equals the expected value (`0` in this case).
  - The thread remains asleep until it is woken up by a `FUTEX_WAKE` call or a signal.
- `FUTEX_WAKE`:
  - Wakes up a specified number of threads waiting on the futex (`1` in this case).
  - If no threads are waiting, the call has no effect.

---

## Final Notes

- **Atomic Variables:**
    - It's crucial to use atomic types and operations to prevent data races and ensure thread safety.
    - In your actual code, declaring `lock` as `uint32_t` works, but declaring it as `atomic_uint` or similar is safer.
- **Resetting Expected Values:**
    - In `acquire`, the `one` variable must be reset to `1` in each loop iteration because `atomic_compare_exchange_strong` updates it if the comparison fails.
    - Not resetting `one` can lead to incorrect behavior.
- **Correct Usage of `atomic_compare_exchange_strong`:**
    - The function expects a pointer to the expected value (`&one` or `&zero`) because it may modify it.
    - Do not declare the expected value as `const` since it needs to be mutable.

---

**Summary:**

- Both the pseudo code and actual code implement a mutex lock with thread synchronization.
- The actual code uses atomic operations and futex system calls to manage synchronization efficiently without an explicit guard lock.
- Understanding the mapping between the pseudo code and the actual code helps in grasping how low-level synchronization mechanisms work in practice.

By carefully aligning each step of your pseudo code with the corresponding operations in your actual code, we've highlighted how the two implementations achieve the same synchronization goals using different approaches.