

# watch this for no busy-wait explanation.

is this algorithm correct?

The algorithm shown is an attempt to implement a lock mechanism using a combination of `test&set` and a guard variable to control access without busy-waiting. Here's an analysis of its correctness:

## 1. Busy-Waiting Elimination:

- The use of a guard variable and putting threads to sleep if the lock is busy is intended to avoid busy-waiting, which is beneficial for multiprocessor systems.

## 2. Race Condition in `release`:

- In the `release` function, if there are threads waiting, one is moved to the ready queue and will eventually acquire the lock.
- However, if no one is on the wait queue, it reassigns `*lock = BUSY;` instead of setting it to `FREE`. This seems like a mistake because if no threads are waiting, the lock should be set to `FREE`, allowing future threads to acquire it without waiting.
- Therefore, `*lock = BUSY;` in the `release` function should likely be `*lock = FREE;`.

## 3. Correctness and Potential Deadlock:

- If the `release` function mistakenly keeps the lock in a `BUSY` state even when there are no waiting threads, new threads attempting to acquire the lock will block indefinitely, leading to deadlock.

## Suggested Fix

To make the algorithm correct, update the `release` function so that `*lock = FREE;` if there are no threads on the wait queue.

## Conclusion

This algorithm is almost correct, but it needs a minor modification in `release` to ensure that the lock is set to `FREE` when no threads are waiting.

write the corrected algorithm

Here's the corrected version of the algorithm:

```
c
```

```

#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
}

```

## Explanation of Changes:

- In the `release` function, if there are no threads on the wait queue, the lock is correctly set to `FREE`, ensuring that other threads can acquire it in the future.

This corrected version should now avoid deadlocks and busy-waiting while properly managing the lock state.

[explain the code in details](#)

Here's a detailed breakdown of how this lock implementation works:

## Definitions and Initialization

c

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

- **FREE** and **BUSY** are defined as constants to represent the two possible states of the lock and guard variables.
- **guard** : A variable used to ensure mutual exclusion (similar to a lock) when accessing the lock. The **guard** variable prevents race conditions when multiple threads attempt to acquire or release the lock simultaneously.
- **lock** : The actual lock variable that threads attempt to acquire to enter a critical section.

## Acquire Function (Requesting the Lock)

c

```
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
    }
}
```

### 1. Guard Lock Acquisition:

- `while (test&set(guard));` uses the **test&set** operation to lock the **guard** variable. The **test&set** function atomically sets **guard** to **BUSY** and returns its previous value. If **guard** was **FREE**, it becomes **BUSY**, allowing the thread to enter the critical section.
- This prevents race conditions when multiple threads try to acquire the lock at the same time. Only one thread can access the **lock** variable while **guard** is set to **BUSY**.

## 2. Checking the Lock State:

- After acquiring the `guard`, the thread checks the state of `*lock`.
- If `*lock == BUSY`, it means another thread currently holds the lock, so the current thread cannot enter the critical section.

## 3. Putting the Thread to Sleep:

- If the lock is busy, the thread cannot proceed, so it calls `put_thread_on_wait_queue()` to add itself to a wait queue.
- It then calls `go_to_sleep()`, which makes the thread sleep until it's awakened by the `release` function.
- Finally, the thread sets `guard = FREE`, releasing the `guard` lock so other threads can check the `lock` status.

## 4. Acquiring the Lock:

- If `*lock` is `FREE`, the thread sets `*lock = BUSY`, acquiring the lock and granting itself exclusive access to the critical section.
- The thread then releases the `guard` by setting `guard = FREE`.

## Release Function (Releasing the Lock)

```
c

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
}
```

## 1. Guard Lock Acquisition:

- Similar to the `acquire` function, the `release` function begins by acquiring the `guard` lock using `while (test&set(guard));`. This ensures that only one thread can modify the `lock` state or interact with the wait queue at any given time.

## 2. Checking the Wait Queue:

- Once the `guard` is acquired, the function checks if there are any threads waiting on the wait queue using `any_thread_on_wait_queue()`.
- If there are waiting threads, the function calls `take_thread_off_wait_queue()` to dequeue one thread.
- It then calls `place_thread_on_ready_queue()` to make that thread ready to run. This will allow that thread to eventually wake up and attempt to acquire the lock again.

## 3. Setting the Lock to Free:

- If no threads are waiting on the wait queue, it sets `*lock = FREE`, effectively releasing the lock so other threads can acquire it.
- This ensures that the lock becomes available for other threads only if no threads are already queued to acquire it.

## 4. Releasing the Guard:

- Finally, `guard = FREE;` releases the guard lock, allowing other threads to access the lock or the wait queue.

## How It Works Together

- **Guard Usage:** The `guard` variable acts as a meta-lock, protecting access to the actual lock variable and the wait queue. This allows the code to handle scenarios where multiple threads try to acquire or release the lock simultaneously.
- **Avoiding Busy-Waiting:** By using the wait queue and sleep/wake functions, this implementation avoids busy-waiting. Threads that cannot acquire the lock go to sleep instead of continuously checking the lock status, which is beneficial for multiprocessor systems.
- **Atomicity with `test&set`:** The `test&set` operation is atomic, ensuring that race conditions do not occur when multiple threads try to acquire or release the `guard` lock. This atomicity is essential to maintain correctness in concurrent environments.

## Benefits and Correctness

- This implementation provides a non-busy-waiting lock, which is more efficient in multi-threaded, multiprocessor environments.
- The corrected release logic (setting `*lock = FREE` only if there are no waiting threads) prevents deadlocks and ensures that the lock is available to other threads when it's no

longer needed.

In summary, this algorithm carefully manages access to a shared resource using `guard` and `lock` variables, a wait queue, and sleep/wake mechanisms, achieving mutual exclusion without busy-waiting and ensuring fair access to the lock.

explain how two threads could work together using this algorithm

Here's how two threads, say **Thread A** and **Thread B**, could work together using this lock algorithm to synchronize their access to a shared resource.

## Scenario Setup

1. **Shared Resource:** Let's assume there is a shared critical section (some data or a resource) that both threads need to access exclusively.
2. **Goal:** We want to prevent both threads from accessing the critical section at the same time, ensuring mutual exclusion.
3. **Order of Actions:** Assume **Thread A** tries to acquire the lock first, followed by **Thread B**.

## Step-by-Step Execution

### Step 1: Thread A Tries to Acquire the Lock

1. **Guard Acquisition:**
  - **Thread A** enters the `acquire` function and tries to acquire the `guard` lock by calling `test&set(guard);`.
  - Since `guard` is initially `FREE`, `test&set` sets it to `BUSY` and allows **Thread A** to proceed.
2. **Checking Lock Status:**
  - **Thread A** checks the `lock` variable. Since `lock` is initially `FREE`, **Thread A** proceeds to acquire the lock.
3. **Acquiring the Lock:**
  - **Thread A** sets `*lock = BUSY`, indicating that it now holds the lock and has exclusive access to the critical section.
4. **Releasing the Guard:**

- **Thread A** releases the `guard` lock by setting `guard = FREE`, allowing other threads (like **Thread B**) to access the `acquire` function.
- **Thread A** now enters the critical section to perform its task.

## Step 2: Thread B Tries to Acquire the Lock

### 1. Guard Acquisition:

- **Thread B** enters the `acquire` function and tries to acquire the `guard` by calling `test&set(guard);`.
- Since **Thread A** has already set `guard = FREE` after acquiring the lock, **Thread B** can now set `guard = BUSY` and proceed.

### 2. Checking Lock Status:

- **Thread B** checks the `lock` variable and finds that `*lock == BUSY` (since **Thread A** holds the lock).
- This means **Thread B** cannot enter the critical section yet.

### 3. Waiting:

- **Thread B** adds itself to the wait queue by calling `put_thread_on_wait_queue();`.
- **Thread B** then calls `go_to_sleep();`, which puts it to sleep, waiting for **Thread A** to release the lock.
- Before going to sleep, **Thread B** sets `guard = FREE`, allowing other threads to attempt acquiring the lock if they wish.

## Step 3: Thread A Releases the Lock

### 1. Finishing Critical Section:

- **Thread A** completes its task in the critical section and calls the `release` function to release the lock.

### 2. Guard Acquisition:

- **Thread A** acquires the `guard` lock using `while (test&set(guard));`, setting `guard = BUSY`.

### 3. Checking the Wait Queue:

- **Thread A** checks if any threads are waiting on the wait queue using `any_thread_on_wait_queue();`.

- Since **Thread B** is in the wait queue, **Thread A** calls `take_thread_off_wait_queue();` to dequeue **Thread B**.
- It then calls `place_thread_on_ready_queue();`, making **Thread B** ready to run (waking it up).

#### 4. Releasing the Guard:

- **Thread A** sets `guard = FREE`, allowing other threads to acquire the guard and potentially check the lock status.

### Step 4: Thread B Acquires the Lock

#### 1. Waking Up:

- After **Thread A** releases the lock and wakes **Thread B**, **Thread B** resumes execution.
- **Thread B** re-enters the `acquire` function and attempts to acquire the `guard` lock again.

#### 2. Guard Acquisition:

- Since `guard` is `FREE`, **Thread B** sets `guard = BUSY` and proceeds.

#### 3. Acquiring the Lock:

- **Thread B** finds that `*lock == FREE` (as **Thread A** released it).
- **Thread B** sets `*lock = BUSY`, acquiring the lock and entering the critical section.

#### 4. Releasing the Guard:

- **Thread B** sets `guard = FREE` and proceeds with its task in the critical section.

## Summary

- **Thread A** and **Thread B** are able to access the shared critical section one at a time because of the `lock` variable that controls exclusive access.
- **Guard Usage:** The `guard` variable ensures that only one thread at a time can access or modify the lock and wait queue, preventing race conditions.
- **Wait Queue and Sleeping:** When **Thread B** finds the lock busy, it goes to sleep, avoiding busy-waiting. When **Thread A** releases the lock, it wakes up **Thread B** if **Thread B** is in the wait queue, allowing **Thread B** to try acquiring the lock again.
- **Efficient and Fair:** This mechanism avoids busy-waiting and provides fair access to the lock, as threads that wait are queued and later given a chance to acquire the lock.



This cooperation mechanism ensures that **Thread A** and **Thread B** can work in tandem without conflicting access to the critical section, while minimizing CPU usage through sleep/wake cycles instead of continuous polling.

How can I implement acquire and release using the following functions:

```
/* 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>
```

```
static uint32_t *futex1, *futex2, *iaddr;
```

```
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
fwait(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
fpost(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");
    }
}

```

To implement `acquire` and `release` functions for a lock using the `fwait` and `fpost` functions defined above, we can think of `fwait` as the function that waits to acquire the lock and `fpost` as the function that releases it. Here's how we can implement the `acquire` and `release` functions for a lock.

## Explanation

### 1. Acquire Function:

- This function waits until the lock is available (value `1`), then it sets it to `0` to indicate that the lock is now held by the current thread.
- It uses the `fwait` function, which performs this operation by checking the value and waiting if the lock is already held.

### 2. Release Function:

- This function releases the lock by setting the lock's value back to `1`.
- It uses the `fpost` function, which sets the lock to `1` if it was `0` and wakes up any other thread waiting on the lock.

## Code Implementation

Here's how you could implement `acquire` and `release` using `fwait` and `fpost`:

```
c

#include <stdint.h>

/* Global lock variable */
uint32_t lock = 1; // Initialize to 1 to represent "unlocked" state

/* Acquire function: Tries to acquire the lock */
void acquire(uint32_t *lock) {
    fwait(lock); // Wait until the lock becomes available
}

/* Release function: Releases the lock */
void release(uint32_t *lock) {
    fpost(lock); // Release the lock and wake up any waiting threads
}
```

## How `acquire` and `release` Work with `fwait` and `fpost`

- **acquire:** When a thread calls `acquire`, it uses `fwait(lock)` to attempt to acquire the lock.
  - `fwait` will check if `lock` is `1` (indicating the lock is free).
  - If the `lock` is `1`, `fwait` will set it to `0`, allowing the calling thread to proceed.
  - If `lock` is `0`, `fwait` will put the thread to sleep until `lock` becomes `1` again, at which point it will try to acquire the lock again.
- **release:** When a thread calls `release`, it uses `fpost(lock)` to release the lock.
  - `fpost` will set the `lock` to `1`, signaling that it is free.
  - `fpost` will also wake up any threads that are waiting for the lock (in `fwait`), so that one of them can proceed and acquire the lock.

## Usage Example

To illustrate how two threads might use this locking mechanism, here's a brief example:

```
c

#include <pthread.h>
#include <stdio.h>
```

```

#include <stdint.h>

uint32_t lock = 1; // Shared lock variable

void* thread_func(void* arg) {
    acquire(&lock);          // Acquire the lock
    printf("Thread %d has the lock\n", *(int*)arg);
    sleep(1);                // Simulate some work
    printf("Thread %d releasing the lock\n", *(int*)arg);
    release(&lock);          // Release the lock
    return NULL;
}

int main() {
    pthread_t t1, t2;
    int id1 = 1, id2 = 2;

    pthread_create(&t1, NULL, thread_func, &id1);
    pthread_create(&t2, NULL, thread_func, &id2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}

```

## Explanation of Example

- **Thread Synchronization:** This example creates two threads that both try to acquire the same `lock`.
- **Acquire and Release:**
  - `acquire` ensures that only one thread can enter the critical section (the code within the lock) at a time.
  - `release` ensures that the lock is available for other threads after the critical section is exited.

In this example, only one thread can print its message and perform work at a time, demonstrating the mutual exclusion achieved by this locking mechanism.

is this code correct?

```

/* 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;
    }

```

Your code is mostly correct and should function as expected, but there are a few points to consider for clarity, potential performance improvement, and error handling. Here's a detailed review and explanation of the code:

## Explanation of Code Components

### 1. Lock Mechanism ( `acquire` and `release` ):

- The `acquire` function waits for `lock` to be `1`, then sets it to `0` (indicating the lock is held).
- The `release` function sets `lock` back to `1` and wakes up any thread waiting on it.

### 2. Thread Function ( `increasing` ):

- Each thread increments the global variable `count` `MAX` times.
- The `acquire` function is used to obtain exclusive access before incrementing `count`, ensuring thread safety.
- After incrementing `count`, `release` is called to allow other threads to acquire the lock.

### 3. Main Function:

- Two threads ( `even_thread` and `odd_thread` ) are created, each running the `increasing` function.
- The threads are joined after they complete, and the final value of `count` is printed.

## Code Review and Potential Issues

### 1. Correctness of Synchronization:

- The synchronization logic using `acquire` and `release` is correct. Each increment operation on `count` is protected by a lock, so `count` should be incremented safely even with multiple threads.

## 2. Initialization of `lock` :

- `lock` is initialized to `1`, representing the unlocked state, which is correct for this implementation.

## 3. Atomic Operations:

- The code uses `atomic_compare_exchange_strong` to ensure that changes to `lock` are atomic, making the lock mechanism reliable.

## 4. Busy-Waiting:

- The `acquire` function will call `futex` with `FUTEX_WAIT` if it fails to acquire the lock, which avoids busy-waiting by putting the thread to sleep. This is efficient and prevents high CPU usage.

## 5. Error Handling:

- The `acquire` and `release` functions use `err` to handle `futex` errors. If `futex` calls fail with an error other than `EAGAIN`, the program will exit, which is reasonable here for demonstration.

## 6. Data Race in Output:

- There's no synchronization around `printf("Finally, n = %d\n", count);` in the `main` function, but this should not cause issues since `count` is only accessed after the threads have finished.

# Suggested Improvements

## 1. Remove Unnecessary `#include <stdatomic.h>` :

- You included `<stdatomic.h>` twice; remove the duplicate include for clarity.

## 2. Error Handling in Thread Creation:

- It's often good practice to check the return value of `pthread_create` and `pthread_join` to ensure threads are created and joined successfully.

## 3. Naming:

- The thread names `even_thread` and `odd_thread` might be misleading, as both threads perform the same task (incrementing `count`). Consider renaming them to something more descriptive, like `thread1` and `thread2`.

## Final, Revised Version

Here's a revised version of your code with minor improvements:

c

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

    while (1) {
        if (atomic_compare_exchange_strong(futexp, &one, 0))
            break;

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

    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");
    }
}

/* Thread function to increment 'count' */
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 thread1, thread2;

    /* Create two threads */
    if (pthread_create(&thread1, NULL, increasing, NULL) != 0) {
        err(EXIT_FAILURE, "pthread_create failed for thread1");
    }
    if (pthread_create(&thread2, NULL, increasing, NULL) != 0) {
        err(EXIT_FAILURE, "pthread_create failed for thread2");
    }

    /* Wait for both threads to finish */

```

```

    if (pthread_join(thread1, NULL) != 0) {
        err(EXIT_FAILURE, "pthread_join failed for thread1");
    }
    if (pthread_join(thread2, NULL) != 0) {
        err(EXIT_FAILURE, "pthread_join failed for thread2");
    }

    /* Output the final value of 'count' */
    printf("Finally, count = %d\n", count);

    return 0;
}

```

## Explanation of Changes

- **Error Handling in `pthread_create` and `pthread_join`**: This version checks for errors when creating and joining threads, which provides more robustness.
- **Descriptive Thread Names**: Renamed `even_thread` and `odd_thread` to `thread1` and `thread2` to avoid any confusion.

## Expected Output

With two threads incrementing `count` up to `MAX` (100,000) each, you should expect the final output to be:

```
bash
```

```
Finally, count = 200000
```

This value should consistently be `200000` since the `acquire` and `release` functions ensure that only one thread increments `count` at a time, avoiding race conditions.