Starve-Free Readers Writers Problem

Overview

Starve Free Readers-Writers Problem: All ***readers*** and ***writers*** will be granted *access to the resource in their order of arrival*. If a writer arrives while readers are accessing the resource, it will wait until those readers free the resource, and then modify it. The same goes for readers a writer has the access to the resource.

This repo contains the `C code` of the solution.

Documentation

Global Declarations:

```
typedef volatile struct {
  volatile atomic_int val;
  volatile atomic_flag mut;
} mysem_t; // Semaphore
```

- Use semaphores for mutex.

Mutex

- A mutex (named for "mutual exclusion") is a binary semaphore with an ownership restriction.
- It can only be unlocked (the signal operation) by the same entity which locked it (the wait operation).
- Thus a mutex offers a somewhat stronger protection than an ordinary semaphore.
- We declare a mutex as: `mymut_t mutex`.

```
  mysem_t queueMutex;
  volatile mymut_t readerMutex = ATOMIC_FLAG_INIT; // Initialized
  volatile mymut_t writerMutex = ATOMIC_FLAG_INIT; // Initialized
  int resource = 1; // Resource
  unsigned int readers = 0;     // Number of readers accessing th
```

- `queueMutex` : Semaphore for queue maintaining to materialize order of arrival. Taken by the entity that requests the access to the resource and is released after it gains the access.
- `writerMutex` : Semaphore for locking resource from writers. Requested by a writer before modifying a resource.
- `readers` : Counter for the number of readers accessing the resource.
- `readerMutex` : Protect the counter against conflicting accesses.

```
#define acquire(m) while (atomic_flag_test_and_set(m)) // Mutex L
#define release(m) atomic_flag_clear(m) // Mutex Release/Unlock

int wait(mysem_t * s) {
  acquire(&s->mut);
  while (atomic_load(&s->val) <= 0);
  atomic_fetch_sub(&s->val, 1);
  release(&s->mut);
  return 0;
}

int signal(mysem_t * s) {
  atomic_fetch_add(&s->val, 1);
  return 0;
}

int init(mysem_t * s, int value){
  atomic_init(&s->val, value);
  return 0;
}
```

- `wait()`
  - Decrements (locks) the semaphore pointed to by sem.
  - If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately.

- If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

- `acquire()`
  - The mutex object referenced by mutex shall be locked by a call to acquire().
  - If the mutex is already locked by another thread, the calling thread shall block until the mutex becomes available.

- `signal()`
  - Increments (unlocks) the semaphore pointed to by sem.
  - If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a wait() call will be woken up and proceed to lock the semaphore.

`wait() or signal()` are same as P() or V() which are generally used with semaphores.

Readers Part:

```
void *reader(void *readerIndex)
{
    wait(&queueMutex);

    acquire(&readerMutex);
    if (readers == 0)
        acquire(&writerMutex);
    // increment the number of readers.
    readers++;

    signal(&queueMutex);
    release(&readerMutex);

    acquire(&readerMutex);
    readers--;
    if( readers == 0 )
        release(&writerMutex);
```

```
    release(&readerMutex);

    return readerIndex;
}
```

Writers Part:

```
void *writer(void* writerIndex)
{
    wait(&queueMutex);
    acquire(&writerMutex);
    signal(&queueMutex);

    resource  = pow(2,*((int *)writerIndex));
    printf("Writer %d modifies resource as %d\n",*((int *)writerI

    release(&writerMutex);

    return writerIndex;
}
```

Running the Code

```
gcc starveFree.c -lpthread -lm -o starvefree && ./starvefree && r
```

The output for a system having `10 readers and 10 writers` is:

```
Reader 1 reads resource as 1
Writer 1 modifies resource as 2
Reader 2 reads resource as 2
Writer 2 modifies resource as 4
Reader 3 reads resource as 4
Writer 3 modifies resource as 8
Reader 4 reads resource as 8
Writer 4 modifies resource as 16
Reader 5 reads resource as 16
Writer 5 modifies resource as 32
```

```
Reader 6 reads resource as 32
Writer 6 modifies resource as 64
Reader 7 reads resource as 64
Writer 7 modifies resource as 128
Reader 8 reads resource as 128
Writer 8 modifies resource as 256
Reader 9 reads resource as 256
Writer 9 modifies resource as 512
Reader 10 reads resource as 512
Writer 10 modifies resource as 1024
```