DEPARTMENT OF COMPUTER SCIENCE, HANYANG UNIVERSITY

Semaphore and Readers-Writer Lock

Operating Systems: Three Easy Pieces

Programming Project with xv6

Final due date: June 5, 2022

1 OVERVIEW

1. Semaphore

   Semaphores are used to protect critical sections by providing mutual exclusion. The user creates a semaphore with an initial value assigned. The value of the semaphore determines the accessibility to a critical section; if the value is 0, access is not granted, else the value represents the number of threads that can access the critical section concurrently. The semaphore value must be modified accordingly during the entrance and the exit of a critical section.

2. Readers-writer Lock

   One of the most naive method to protect the critical section is by using mutex. Mutex forbids multiple threads from entering the critical section simultaneously, hence preventing race condition. Although correctness is guaranteed, mutex may lack efficiency in certain cases. One well-known example of such case is the readers-writers problem. The problem involves multiple threads performing reads and writes to a shared resource. If the shared resource is not protected by a synchronization mechanism (e.g., mutex), a race condition may occur. The essence of the race condition is the coexistence of readers and writers simultaneously accessing a shared resource. A mutex may be used to protect the shared resource but depending on the number of readers and writers, it may turn out to be a poor solution in terms of performance.

   In the readers-writers problem, readers may access shared resources simultaneously as long as there are no writers. In other words, readers may perform its job without

being blocked by other readers. Mutex does not consider such behavior of an access. Readers are blocked by each other even though there are no writers, resulting in a loss of efficiency. The readers-writer lock considers this issue and allows multiple readers to access a critical section concurrently as long as there are no waiting writers. A read lock is held for a reader entering the critical section, and multiple read locks may be held concurrently. A write lock is held for a writer to enter the critical section, and all write locks are to be held exclusively, blocking the execution of both readers and other writers. In general, the readers-writer lock enhances performance in cases where there are multiple readers and a small number of writers. Since the readers achieve higher concurrency, the performance gain is evident.

2 Implementation Specifications

1. Simplified Semaphore and Readers-writer Lock

   The goal of this project is to create a much-simplified version of a semaphore and a readers-writer lock in the xv6. The semaphore implementation must be able to protect the critical section just like the POSIX semaphore does. The readers-writer lock must be implemented using your semaphore, following the behavior of a readers-writer lock as you learned in the lectures. Both of the implementation must be exposed to the user level, thus allowing users to use the semaphore and the readers-writer lock just like the POSIX thread API. The simplified specification of the two features are listed below.

2. **Design and Implement the Basic Semaphore and the Readers-writer Lock**

   Carefully read the specifications. **Review the lectures related to your project and write a wiki about your understandings of the project**. Please follow the textbook protocol for reader-writer locks and refer to the Linux implementation explained in the offline lecture for semaphores.

   a) **Semaphore / Readers-writer Lock**
      - Briefly explain about the Semaphore.
      - Briefly explain about the Readers-writer lock.

   b) **POSIX semaphore** : Briefly explain the semaphore structure and the APIs related to unnamed-semaphore.
      - `sem_t`
      - `int sem_init(sem_t *sem, int pshared, unsigned int value);`
      - `int sem_wait(sem_t *sem);`
      - `int sem_post(sem_t *sem);`

   c) **POSIX reader-writer lock** : Briefly explain the reader related structure and the related APIs.

- `pthread_rwlock_t`
- `int pthread_rwlock_init(pthread_rwlock_t* lock, const pthread_rwlockattr_t* attr);`
- `int pthread_rwlock_rdlock(pthread_rwlock_t* lock);`
- `int pthread_rwlock_wrlock(pthread_rwlock_t* lock);`
- `int pthread_rwlock_unlock(pthread_rwlock_t* lock);`

d) **Implement the Basic Semaphore**
The structure and operations below should be provided to users through headers or system calls.

i. **xem_t** : You must provide the semaphore structure to the users.
typedef struct your_structure xem_t;

ii. **init** : You must provide a method to initialize the semaphore structure within a user process (initial value should be 1).
`int xem_init(xem_t *semaphore);`
   - semaphore - the xem_t object to be initialized.

iii. **wait** : You must provide a method to restrict the entry to the critical section through a semaphore. If another threads or processes already entered the critical section, threads which call this function should wait until the threads or processes leave the critical section.
`int xem_wait(xem_t *semaphore);`
   - semaphore - the xem_t object to protect the critical section.

iv. **post** : You must provide a method to wake up the threads waiting at the entry point when leaving the critical section.

`int xem_unlock(xem_t *semaphore);`
   - semaphore - the xem_t object to release the protection to the critical section.

e) **Implement the Readers-writer Lock using Semaphores**
The structure and operations below should be provided to users through headers or system calls.

i. **rwlock_t** : You must provide the structure of a readers-writer lock.
typedef struct your_structure rwlock_t;

ii. **init** : You must provide a method to initialize the readers-writer lock structure within a user process.
`int rwlock_init(rwlock_t *rwlock);`
   - rwlock - the rwlock_t object to be initialized.

iii. **acquire_readlock** : You must provide a method to acquire the read lock through a semaphore. If other readers already entered the critical section, writers which call this function should wait until the threads or processes leave the critical section but readers can enter the critical section.
`int rwlock_acquire_readlock(rwlock_t *rwlock);`
   - rwlock - the rwlock_t object to protect the critical section.

iv. **acquire_writelock** : You must provide a method to acquire the write lock through a semaphore. If the other writer already entered the critical section, threads which call this function should wait until the threads or processes leave the critical section.
`int rwlock_acquire_writelock(rwlock_t *rwlock);`
   - rwlock - the rwlock_t object to protect the critical section.

v. **release_readlock** : You must provide a method to release the read lock through a semaphore.
`int rwlock_release_readlock(rwlock_t *rwlock);`
   - rwlock - the rwlock_t object to release the protection to the critical section.

vi. **release_writelock** : You must provide a method to release the write lock through a semaphore.
`int rwlock_release_writelock(rwlock_t *rwlock);`
   - rwlock - the rwlock_t object to release the protection to the critical section.

3 EVALUATION

1. Document

   You must write a detailed document of your work on the gitlab wiki. The project is evaluated based on the document. If you miss a description, it may not be evaluated. Please do a decent job on describing your work. The document may include design, implementation, solved problems (evaluating list below) and considerations for evaluation.

2. How to evaluate and self-test

   The evaluation of this project is based on the requirements of the implementation specifications described above and can be verified through the test-cases provided. The points are given as follows.

Noted items :

1. We recommend you to make various test cases using Pthread on a linux machine and get an intuition for your implementation.

Evaluation items	Points
Documents	20
Basic operations	55
Other corner cases	10
Total Points	85

2. We will run various tests with your implementation of the semaphore and the readers-writer lock. Both simple and complex programs using the semaphores and the readers-writer lock may be used for testing. To prepare for these tests, write a user program in xv6 and test it yourself.

3. **If the current project is not finished, it is impossible to proceed to the final project (i.e., Project-4)**.

4. **Plagiarism issue will not be forgiven under any circumstances as mentioned before. Be careful not to upload any of your code on a public community since both the contributor and the plagiarist will be treated as equal.**