CS330: Operating Systems

Read-write locks, classical problems

Recap: Spinlocks and semaphores

- Lock implementation using atomic exchange, compare-and-swap, LL–SC and atomic add
- Ticket spinlocks provide fairness in locking, example implementation with atomic-add (xadd)
- Software-only lock implementation (Peterson's solution)
- Semaphore: counting semaphore, binary semaphore (mutex)

Agenda: Read-write locks, producer consumer problem, concurrency bugs

Reader-writer locks

- Allows *multiple readers* or *a single writer* to enter the CS
- Example: Insert, delete and lookup operations on a search tree

Reader-writer locks

- Allows *multiple readers* or *a single writer* to enter the CS
- Example: Insert, delete and lookup operations on a search tree

```
struct BST{
        struct node *root;
        rwlock_t *lock;
};
```

```
struct node{
        item_t item;
        struct node *left;
        struct node* right;
};
```

```
void insert(BST *t, item_t item);
void lookup(BST *t, item_t item);
```

Reader-writer locks

- Allows *multiple readers* or *a single writer* to enter the CS
- Example: Insert, delete and lookup operations on a search tree

```
struct BST{
        struct node *root;
        rwlock_t *lock;
};
```

```
struct node{
        item_t item;
        struct node *left;
        struct node* right;
};
```

```
void insert(BST *t, item_t item);
void lookup(BST *t, item_t item);
```

- If multiple threads call lookup( ), they may traverse the tree in parallel

Implementation of read-write locks

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
```

```
init_lock(rwlock_t *rL)
{
    init_lock(&rL → read_lock);
    init_lock(&rL → write_lock);
    rL → num_readers = 0;
}
```

Implementation of read-write locks (writers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
```

```
init_lock(rwlock_t *rL)
{
  init_lock(&rL → read_lock);
  init_lock(&rL → write_lock);
  rL → num_readers = 0;
}
```

```
void write_lock(rwlock_t *rL)
{
    lock(&rL → write_lock);
}
```

```
void write_unlock(rwlock_t *rL)
{
    unlock(&rL → write_lock);
}
```

- Write lock behavior is same as the typical lock, only one thread allowed to acquire the lock

Implementation of read-write locks (readers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
```

```
void read_lock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers++;
   if(rL → num_readers == 1)
      lock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

```
void read_unlock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers--;
   if(rL → num_readers == 0)
      unlock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

Implementation of read-write locks (readers)

```
struct rwlock_t{
    Lock read_lock;
    Lock write_lock;
    int num_readers;
}
```

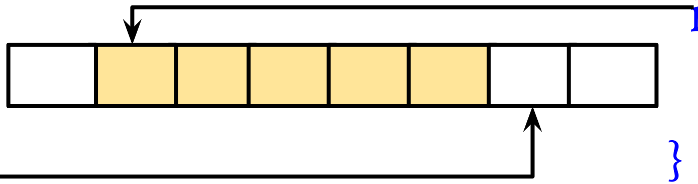
- The first reader acquires the write lock prevents writers to acquire lock
- The last reader releases the write lock to allow writers

```
void read_lock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers++;
   if(rL → num_readers == 1)
       lock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

```
void read_unlock(rwlock_t *rL)
{
   lock(&rL → read_lock);
   rL → num_readers--;
   if(rL → num_readers == 0)
       unlock(&rL → write_lock);
   unlock(&rL → read_lock);
}
```

Producer-consumer problem

```
DoProducerWork( ){
 while(1){
   item_t item = prod_p( );
   produce(item);
 }
}
```



```
DoConsumerWork( ){
 while(1){
  item_t item = consume( );
   cons_p(item);
  }
}
```

- A buffer of size N, one or more producers and consumers
- Producer produces an element into the buffer (after processing)
- Consumer extracts an element from the buffer and processes it
- Example: A multithreaded web server, network protocol layers etc.
- How to solve this problem using semaphores?

Buggy #1

```
item_t A[n], pctr=0, cctr = 0;
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used);
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    sem_post(&empty);
    return item;
}
```

- This solution does not work. What is the issue?

Buggy #1

```
item_t A[n], pctr=0, cctr = 0;
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used);
    item_t item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty);
    return item;
}
```

- This solution does not work. What is the issue?
- The counters (pctr and cctr) are not protected, can cause race conditions

Buggy #2

```
item_t A[n], pctr=0, cctr = 0;  lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    lock(L);   sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);  unlock(L);
}
```

```
item_t consume( ) {
    lock(L);  sem_wait(&used);
    item_t item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty); unlock(L);
    return item;
}
```

- What is the problem?

Buggy #2

```
item_t A[n], pctr=0, cctr = 0;  lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    lock(L);   sem_wait(&empty);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    sem_post(&used);  unlock(L);
}
```

```
item_t consume( ) {
    lock(L);  sem_wait(&used);
    item_t  item = A[cctr];;
    cctr = (cctr + 1) % n;
    sem_post(&empty); unlock(L);
    return item;
}
```

- What is the problem?
- Consider empty = 0 and producer has taken lock before the consumer. This results in a deadlock, consumer waits for L and producer for empty

A working solution

```
item_t A[n], pctr=0, cctr = 0;  lock_t *L = init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty); lock(L);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    unlock(L); sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used); lock(L)
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    unlock(L); sem_post(&empty);
    return item;
}
```

- The solution is deadlock free and ensures correct synchronization, but very much serialized (inside produce and consume)
- What if we use separate locks for producer and consumer?

Solution with separate mutexes

```
item_t A[n], pctr=0, cctr = 0;  lock_t *P = init_lock( ), *C=init_lock( );
sem_t empty = sem_init(n), used = sem_init(0);
```

```
produce(item_t item){
    sem_wait(&empty); lock(P);
    A[pctr] = item;
    pctr = (pctr + 1) % n;
    unlock(P); sem_post(&used);
}
```

```
item_t consume( ) {
    sem_wait(&used); lock(C)
    item_t item = A[cctr];
    cctr = (cctr + 1) % n;
    unlock(C); sem_post(&empty);
    return item;
}
```

- Does this solution work?
- Homework: Assume that item is a large object and copy of item takes long time. How can we perform the copy operation without holding the lock?