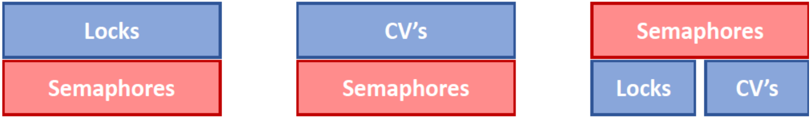
Operating Systems

Autumn 2024

Equivalence

Equivalence

- Claim: Semaphores are equally powerful as lock+CVs
- This means we can build each out of the other



Locks

Semaphores

CV's

Semaphores

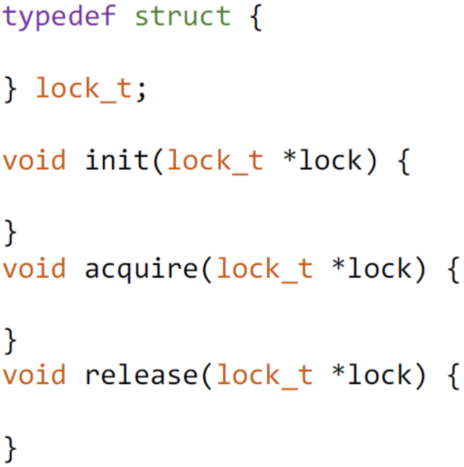
Semaphores

Locks

CV's

Lock implementation using semaphores

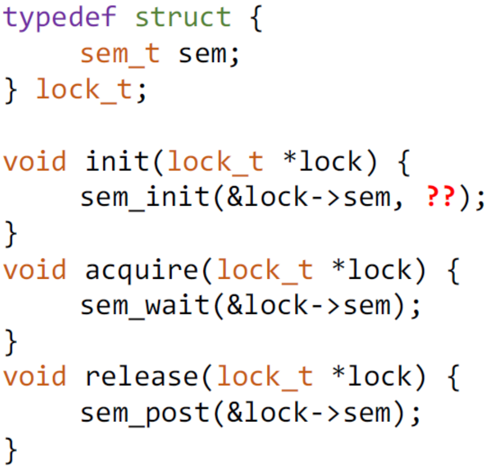
- Finish this implementation



```
typedef struct {

} lock_t;

void init(lock_t *lock) {

}
void acquire(lock_t *lock) {

}
void release(lock_t *lock) {

}
```

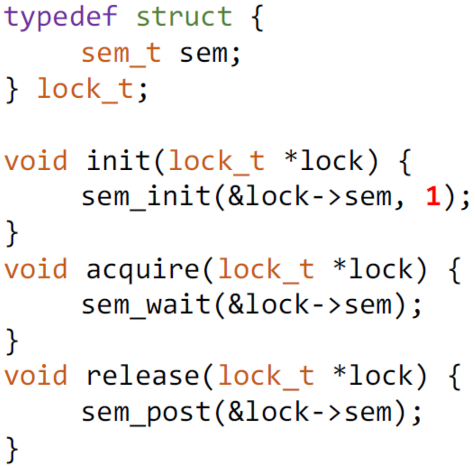
Lock implementation using semaphores

- Semaphore is initialized to ____



```
typedef struct {
     sem_t sem;
} lock_t;

void init(lock_t *lock) {
     sem_init(&lock->sem, ??);
}
void acquire(lock_t *lock) {
     sem_wait(&lock->sem);
}
void release(lock_t *lock) {
     sem_post(&lock->sem);
}
```

Lock implementation using semaphores



```
typedef struct {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {
    sem_init(&lock->sem, 1);
}
void acquire(lock_t *lock) {
    sem_wait(&lock->sem);
}
void release(lock_t *lock) {
    sem_post(&lock->sem);
}
```

CV implementation using semaphores (attempt 1)

- Finish this implementation using semaphores and locks

```
typedef struct {
    sem_t sem;               // initially 0
    lock_t lock;
} cond_t;

void cond_wait(cond_t *c) {
    // assumes that lock is held
    ??
}

void cond_signal(cond_t *c) {
    ??
}
```

CV implementation using semaphores (attempt 1)

- You might have tried . . .

```
typedef struct {
   sem_t sem;              // initially 0
   lock_t lock;
} cond_t;

void cond_wait(cond_t *c) { // assumes lock is held
   release(&c->lock);        // release lock and go to sleep
   sem_wait(&c->sem);
   acquire(&c->lock);        // grab lock before returning
}

void cond_signal(cond_t *c) {
   sem_post(&c->sem);        // wake up a sleeping waiter
}
```

- This solution is incorrect (why?)

CV implementation using semaphores (attempt 1)

- You might have tried . . .

```
typedef struct {
    sem_t sem;            // initially 0
    lock_t lock;
} cond_t;

void cond_wait(cond_t *c) { // assumes lock is held
    release(&c->lock);        // release lock and go to sleep
    sem_wait(&c->sem);
    acquire(&c->lock);        // grab lock before returning
}

void cond_signal(cond_t *c) {
    sem_post(&c->sem);        // wake up a sleeping waiter
}
```

- This solution is incorrect (why?)
  - cond_signal wakes up threads in the far future!

CV implementation using semaphores (attempt 2)

- Finish this implementation using semaphores and locks

```
typedef struct {
    sem_t sem;              // initially 0
    lock_t lock;
    lock_t priv_lock;       // initially 1
    int num_waiters;        // initially 0
} cond_t;

void cond_wait(cond_t *c) {
    // assumes that lock is held
    ??
}

void cond_signal(cond_t *c) {
    ??
}
```

CV implementation using semaphores (attempt 2)

- On the whiteboard

```
void cond_wait(cond_t *c, lock_t *lock) {
   // Assumes that the main lock is held
   lock_acquire(&c->priv_lock); // Protect num_waiters with priv_lock
   c->num_waiters++;            // Increment number of waiters
   lock_release(&c->priv_lock); // Release priv_lock after incrementing

   lock_release(lock);         // Release the main lock

   sem_wait(&c->sem);          // Block the thread on the semaphore (waiting)

   lock_acquire(lock);         // Re-acquire the main lock after being signaled
}
```

```
void cond_signal(cond_t *c) {
   lock_acquire(&c->priv_lock);  // Protect num_waiters with priv_lock
   if (c->num_waiters > 0) {     // If there are any waiters
      c->num_waiters--;          // Decrement the number of waiters
      sem_post(&c->sem);         // Signal one waiting thread
   }
   lock_release(&c->priv_lock);  // Release priv_lock after signaling
}
```