More Shared Memory Programming

Shared data structures

We want to make data structures that can be shared by threads.

For example, our program to copy a file from one disk to another used a shared FIFO queue and a shared set.

We start with a very simple data structure to keep track of money in a bank account.

[Note: To keep things simple I will ignore the fact that long integers could overflow. In real code, we should look out for this.]

We have

```
typedef struct Account {
    pthread_mutex_t mtx ;
    long balance ; // In 100ths of dollars.
    } Account ;
void initialize( Account *p ) {
    pthread_mutex_init( & p->mtx, NULL ) ;
    p->balance = 0 ; }
/* Precondition: 0 <= amount */
void deposit( Account *p, long amount ) {
    assert( amount >= 0 ) ;
    pthread_mutex_lock( & p->mtx ) ;
        p->balance += amount ;
    pthread_mutex_unlock( & p->mtx ) ; }
```

And we can write a withdraw function that is similar.

Enforcing an object invariant

Let’s suppose we want to prevent the account balance from ever going below zero.

Such a restriction is called an object invariant. We should document it.

The object invariant must be true whenever the object is no locked.

```
typedef struct Account {
    /* Object invariant: balance >= 0 */
    pthread_mutex_t mtx ;
    long balance ;
    } Account ;
```

The following will not work

```
/* Precondition: 0 <= amount && amount <= p->balance
*/
void withdraw( Account *p, long amount ) {
    assert( amount >= 0 ) ;
    pthread_mutex_lock( & p->mtx ) ;
        p->balance += amount ;
    pthread_mutex_unlock( & p->mtx ) ; }
```

The caller can not be sure of what the balance is, since another thread might change it.

We could do the following

```
/* Precondition: 0 <= amount */
int withdraw( Account *p, long amount ) {
    int result ;
    assert( amount >= 0 ) ;
    pthread_mutex_lock( & p->mtx ) ;
        if( amount <= p->balance ) {
            p->balance += amount ;
            result = 1 ; }
        else result = 0 ;
    pthread_mutex_unlock( & p->mtx ) ;
    return result ; }
```

In this case the function returns a boolean indicating whether the withdrawal was successful.

Waiting

Another way to enforce the object invariant is to force threads that withdraw to wait until there are sufficient funds in the account.

We want something like this

```
/* Precondition: 0 <= amount */
void withdraw( Account *p, long amount ) {
    assert( amount >= 0 ) ;
    pthread_mutex_lock( & p->mtx ) ;
        wait until amount <= p->balance
        p->balance -= amount ;
    pthread_mutex_unlock( & p->mtx ) ; }
```

However, if the thread waits while the object is locked, no thread will be able to deposit, so the thread will wait forever.

We must unlock the object while waiting.

We want something like the following

```
/* Precondition: 0 <= amount */
```

```
void withdraw( Account *p, long amount ) {
   assert( amount >= 0 ) ;
   pthread_mutex_lock( & p->mtx ) ;
      while( ! amount <= p->balance ) {
         pthread_mutex_unlock( & p->mtx ) ;
         wait a little while
         pthread_mutex_lock( & p->mtx ) ;
      }
      p->balance -= amount ;
   pthread_mutex_unlock( & p->mtx ) ; }
```

There is a nice way to implement the code in the box with a device called a **condition variable.**

Each condition variable is associated with some mutex and allows threads to wait until some condition happens.

We add a condition variable to the Account type and initialize it

```
typedef struct Account {
   pthread_mutex_t mtx ;
   long balance ;
   pthread_cond_t bigEnoughBalanceCond ;
   } Account ;
void initialize( Account *p ) {
   pthread_mutex_init( & p->mtx, NULL ) ;
   pthread_cond_init( & p->bigEnoughBalanceCond,
   NULL ) ;
   p->balance = 0 ; }
```

Now in withdraw we wait on the condition variable.

```
/* Precondition: 0 <= amount */
```

```
void withdraw( Account *p, long amount ) {
    assert( amount >= 0 ) ;
    pthread_mutex_lock( & p->mtx ) ;
        while( ! (amount <= p->balance) ) {
            pthread_cond_wait( & p->bigEnoughBalanceCond,
            & p->mtx ) ; }
        p->balance -= amount ;
    pthread_mutex_unlock( & p->mtx ) ; }
```

The call

```
pthread_cond_wait( & p->bigEnoughBalanceCond, &
p->mtx ) ;
```

means

```
pthread_mutex_unlock( & p->mtx ) ;
wait until condition p->bigEnoughBalanceCond is
signalled
pthread_mutex_lock( & p->mtx ) ;
```

Since threads doing a withdrawal may wait for a signal we should add code to signal whenever a deposit is made. We add one line to the deposit method

```
/* Precondition: 0 <= amount */
void deposit( Account *p, long amount ) {
    assert( amount >= 0 ) ;
    pthread_mutex_lock( & p->mtx ) ;
        p->balance += amount ;
        pthread_cond_broadcast( & p-
        >bigEnoughBalanceCond ) ;
    pthread_mutex_unlock( & p->mtx ) ; }
```

The call to *pthread_cond_broadcast* will ‘wake up’ all threads waiting on the condition.

[Exercise: Suppose two threads are each waiting to withdraw $100.00. If another thread deposits $200.00, will both threads return from their calls to withdraw? You may assume that there are no other threads. Draw a message sequence diagram (or other kind of picture) illustrating what may happen.]

A Queue data structure

We will define a data type representing FIFO (first-in, first-out) queue.

A thread that tries to take from an empty queue must wait.

A thread that tries to put data into a full queue must wait.

There are two reasons to wait, so we use 2 condition variables.

```
typedef struct Queue {
    pthread_mutex_t mtx ;
    pthread_cond_t notFull ; // When p->size <
    p->capacity
    pthread_cond_t notEmpty ; // When p->size > 0
    // Object invariant: 0 < capacity
    int capacity ;
    // Object invariant: 0 <= size && size <= capacity
    int size ;
    // Object invariant: 0 <= next && next < capacity
    int next ;
    // Object invariant: a points to an array of capacity
    integers
    int *a ;
} Queue ;
```

We have a routine for initializing a Queue. This must be called before the queue is shared.

```
int initialize( Queue *p, int capacity ) {
    pthread_mutex_init( & p->mtx, NULL ) ;
```

```
        pthread_cond_init( & p->notFull, NULL ) ;
        pthread_cond_init( & p->notEmpty, NULL ) ;
        p->capacity = 0 ; p->size = 0 ; p->next = 0 ;
        p-> a = calloc( capacity, sizeof( int ) ) ;
        return p->a != NULL ;
    }
```

To put new data on the queue we must wait until there is space.

We want something like this

```
    void put( Queue *p, int value ) {
        pthread_mutex_lock( & p->mtx ) ;
            wait until the queue is not full
            add a new item
        pthread_mutex_unlock( & p->mtx ) ; }
```

We can implement it like this.

```
    void put( Queue *p, int value ) {
        pthread_mutex_lock( & p->mtx ) ;
            /* Wait until the queue is not full */
            while( p->size == p->capacity ) {
                pthread_cond_wait( & p->notFull, & p->mtx ) ; }
            /* Add a new value */
            p->a[ (p->next + p->size) % p->capacity ] = value ;
            ++ p->size ;
            pthread_cond_broadcast( & p->notEmpty ) ;
        pthread_mutex_unlock( & p->mtx ) ; }
```

The call to broadcast at the end is because after size has been incremented, the queue will not be empty anymore. There may be threads waiting to for this condition.

The code for take is similar to the code for put.

```
int take( Queue *p ) {
    pthread_mutex_lock( & p->mtx ) ;
        /* Wait until the queue is not empty */
        while( p->size == 0 ) {
            pthread_cond_wait( & p->notEmpty, & p->mtx ) ;
            }
        /* Remove one item. */
        int result = p->a[ p->next ] ;
        p->next = (p->next + 1) % p->capacity ;
        -- p->size ;
        pthread_cond_broadcast( & p->notFull ) ;
    pthread_mutex_unlock( & p->mtx ) ;
    return result ; }
```

[Exercise: We also need a shared structure representing a set of integers. Threads can put numbers into the set and they can remove an arbitrary member of the set. However, to take a number, the thread must wait until the set is not empty. Implement this data structure.]

Advice: Only use pthread_cond_wait and pthread_cond_broadcast as illustrated in the examples above. In particular always put calls to pthread_cond_wait inside a while loop and put the loop insize of a lock/unlock pair.

Deadlock

As we have seen the solution to shared memory problems is often waiting.

- Threads must wait to obtain exclusive access
- Threads must wait for conditions to arise.

We wait to ensure safety. Much like walking across a street in China.

A **saftey property** is a property that says that the system will never get into a bad state, e.g. one where object invariants are violated.

However, waiting has a potential hazard of its own.

We may cause threads to wait forever.

Consider the bank example. We will make a new method transfer. We lock both accounts because we want to ensure that there is never a time that that money is missing or extra money is present.

```
/* Precondition: 0 <= amount */
void deposit( Account *p, Account *q, long amount ) {
    assert( amount >= 0 ) ;
    pthread_mutex_lock( & p->mtx ) ;
    pthread_mutex_lock( & q->mtx ) ;
        while( amount > p->balance ) {
            pthread_cond_wait( & p->bigEnoughBalanceCond,
            & p->mtx ) ; }
        p->balance -= amount ;
        q->balance +=amount ;
        pthread_cond_broadcast( & q-
        >bigEnoughBalanceCond ) ;
    pthread_mutex_unlock( & q->mtx ) ;
    pthread_mutex_unlock( & p->mtx ) ; }
```

But what happens if we transfer from account a to account b at the same time?

We could end up with both accounts locked and neither thread able to proceed.

They will remain locked forever.

This is **deadlock**.

It is a violation of a **liveness property**.

Liveness properties state that the system does not get stuck.