Synchronization

Advanced Embedded Linux Development

with **Dan Walkes**

University of Colorado **Boulder**

Learning objectives:

Understand Race Conditions

Synchronization with PThread

Race Conditions

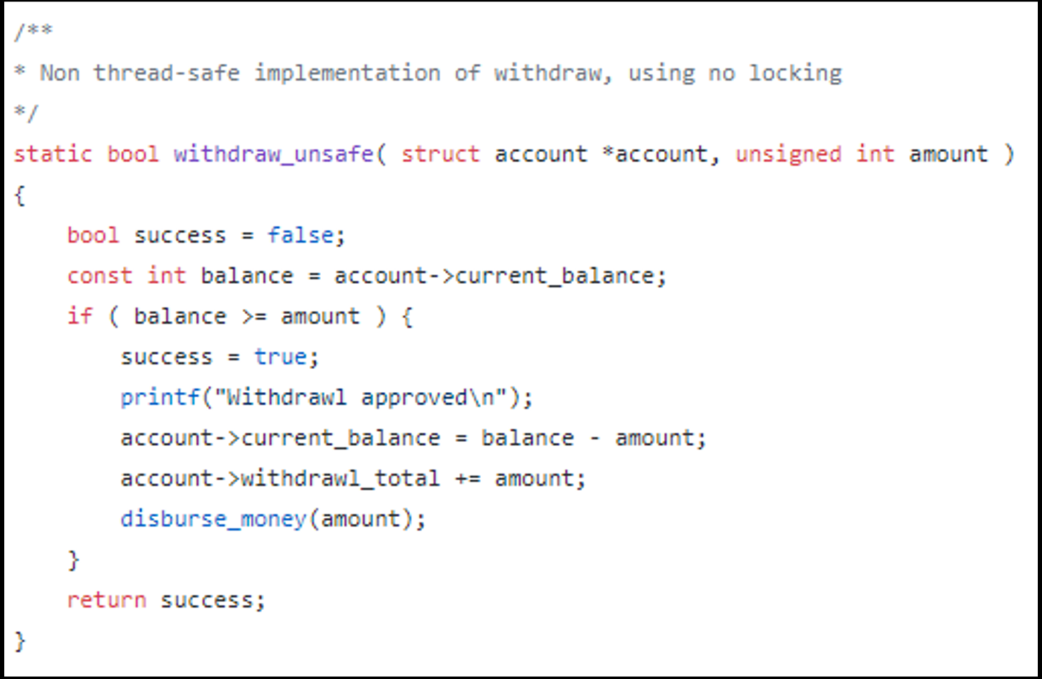
- Different program behavior depending on which thread gets there first
- “Unsynchronized access of a shared resource leads to erroneous program behavior”
    - hardware, kernel resource, memory
    - memory - data race - most common form

Race Conditions

- critical region - region of code which needs synchronization.
- Eliminate races by synchronizing thread access to critical regions.

Example Race Conditions

- When is this code not thread safe?
  - When account is a shared structure in a multi-threaded program

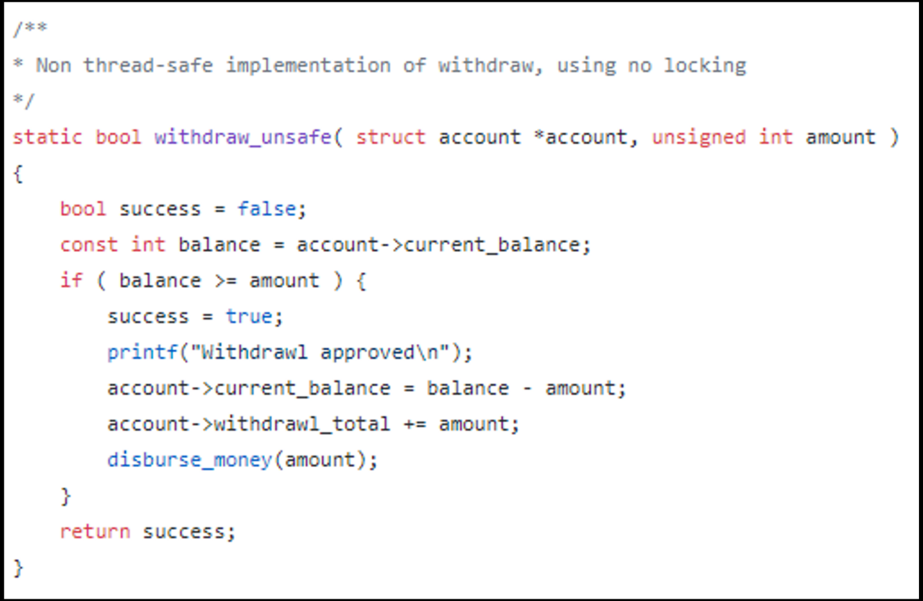


```c
/**
* Non thread-safe implementation of withdraw, using no locking
*/
static bool withdraw_unsafe( struct account *account, unsigned int amount )
{
    bool success = false;
    const int balance = account->current_balance;
    if ( balance >= amount ) {
        success = true;
        printf("Withdrawl approved\n");
        account->current_balance = balance - amount;
        account->withdrawl_total += amount;
        disburse_money(amount);
    }
    return success;
}
```

Linux System Programming Chapter 7
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture7/withdraw_shared.c

Example Race Conditions

- Is this code thread safe when the hardware does not support parallelism?
    - No, concurrency is still an issue.

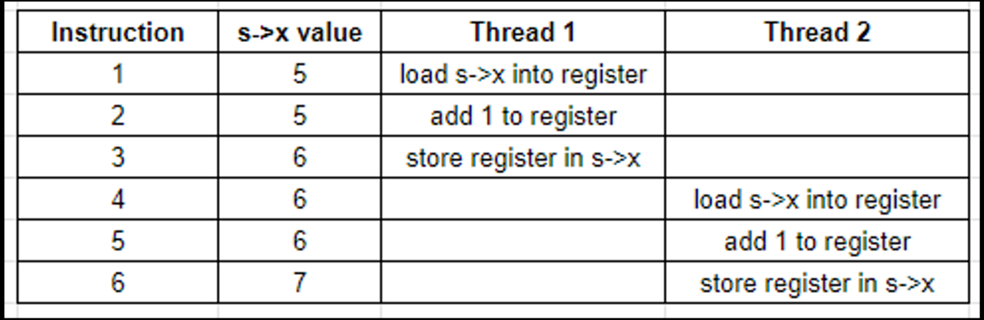


```
/**
* Non thread-safe implementation of withdraw, using no locking
*/
static bool withdraw_unsafe( struct account *account, unsigned int amount )
{
    bool success = false;
    const int balance = account->current_balance;
    if ( balance >= amount ) {
        success = true;
        printf("Withdrawl approved\n");
        account->current_balance = balance - amount;
        account->withdrawl_total += amount;
        disburse_money(amount);
    }
    return success;
}
```

Linux System Programming Chapter 7
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture7/withdraw_shared.c

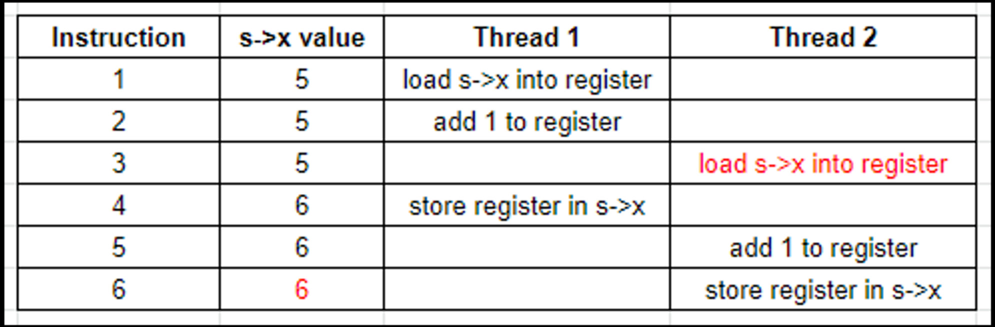
Example Race Conditions

```
struct simple
{
                int x;
}
void somefunc(struct simple *s)
{
        s->x++;
}
```



Instruction	s->x value	Thread 1	Thread 2
1	5	load s->x into register	
2	5	add 1 to register	
3	6	store register in s->x	
4	6		load s->x into register
5	6		add 1 to register
6	7		store register in s->x

Concurrency thread switch here is a problem



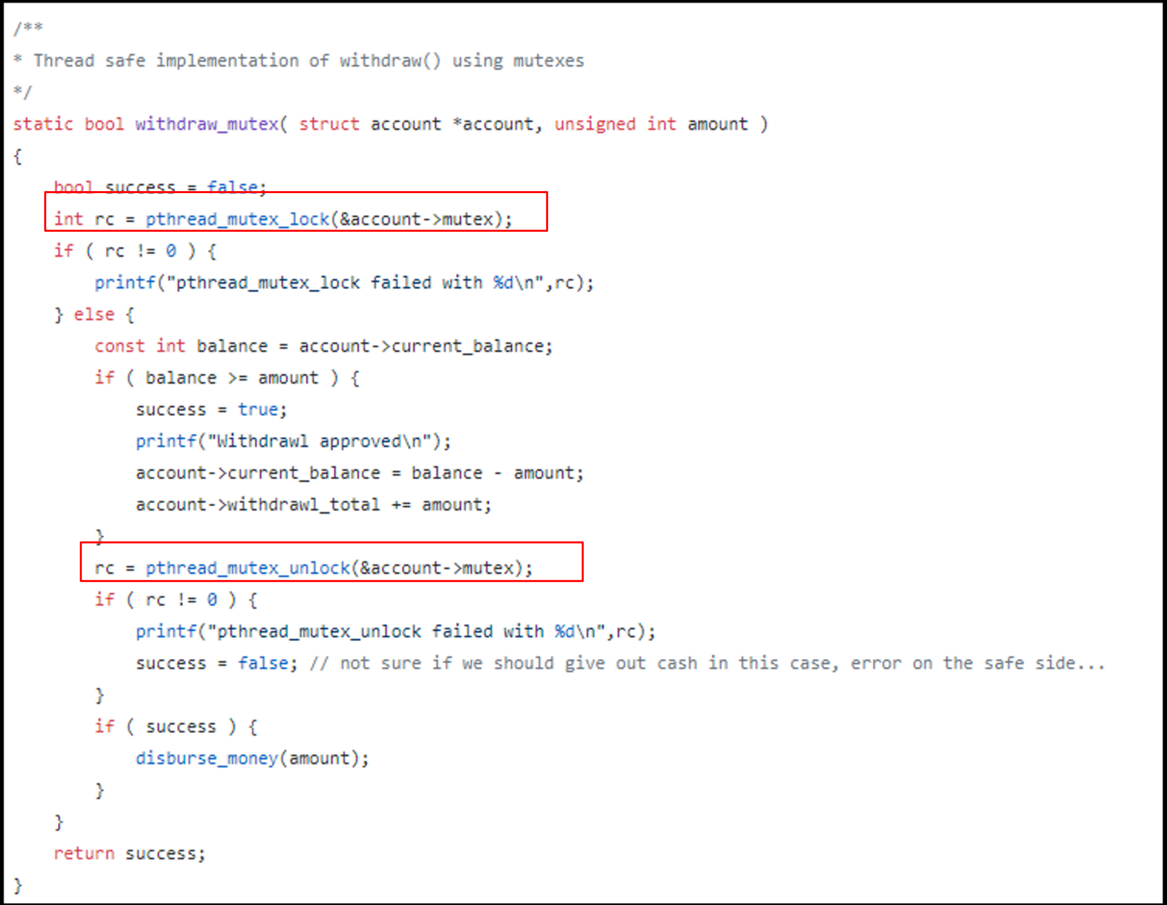
Instruction	s->x value	Thread 1	Thread 2
1	5	load s->x into register	
2	5	add 1 to register	
3	5		load s->x into register
4	6	store register in s->x	
5	6		add 1 to register
6	6		store register in s->x

Example Race Conditions

- An atomic operation is indivisible, unable to be interleaved with other operation, appears instantaneous
- ++ is not atomic
- To make shared memory operations safe we need to make the sequence atomic

Synchronizing

- Mutually exclusive lock or *mutex*

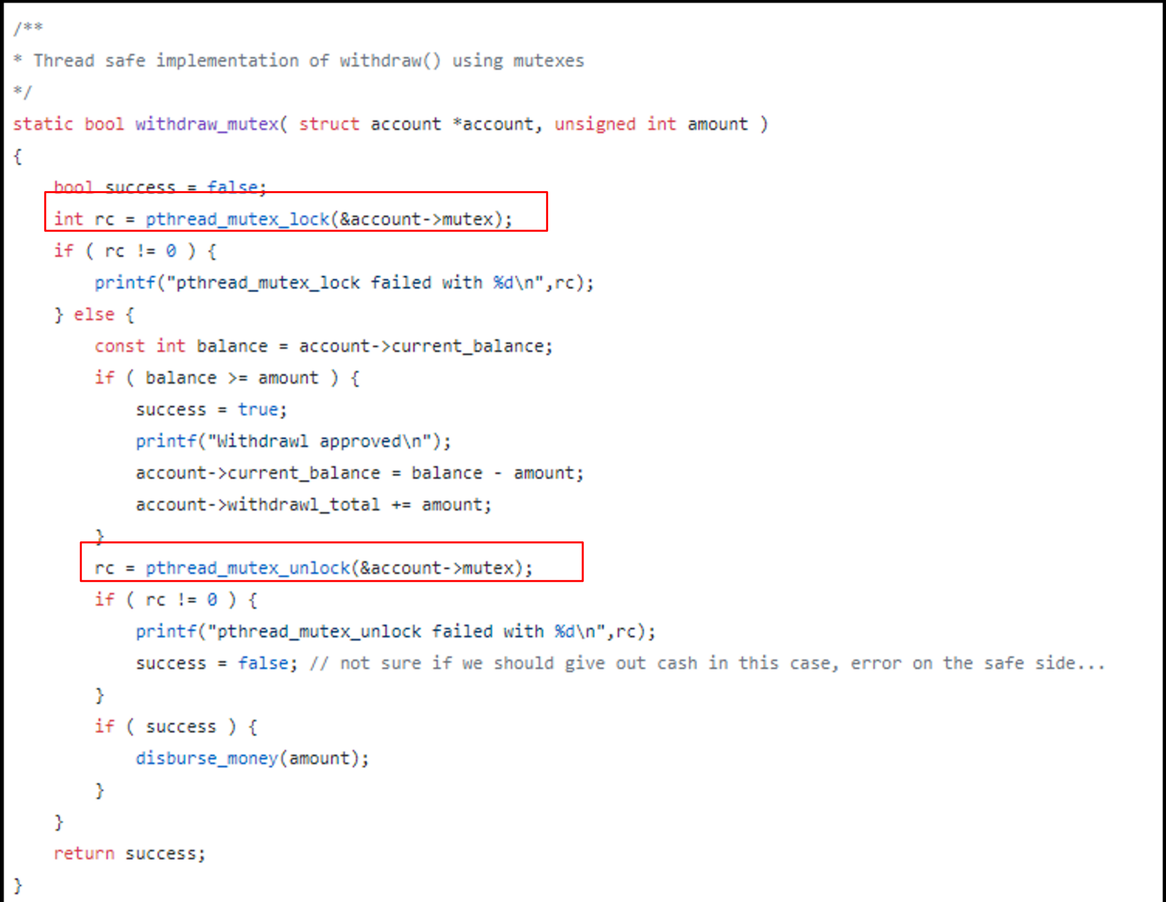


```
/**
* Thread safe implementation of withdraw() using mutexes
*/
static bool withdraw_mutex( struct account *account, unsigned int amount )
{
    bool success = false;
    int rc = pthread_mutex_lock(&account->mutex);
    if ( rc != 0 ) {
        printf("pthread_mutex_lock failed with %d\n",rc);
    } else {
        const int balance = account->current_balance;
        if ( balance >= amount ) {
            success = true;
            printf("Withdrawl approved\n");
            account->current_balance = balance - amount;
            account->withdrawl_total += amount;
        }
        rc = pthread_mutex_unlock(&account->mutex);
        if ( rc != 0 ) {
            printf("pthread_mutex_unlock failed with %d\n",rc);
            success = false; // not sure if we should give out cash in this case, error on the safe side...
        }
        if ( success ) {
            disburse_money(amount);
        }
    }
    return success;
}
```

https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture7/withdraw_shared.c

Synchronizing

- Why don’t we need a lock around disburse_money?
  - No shared memory reference



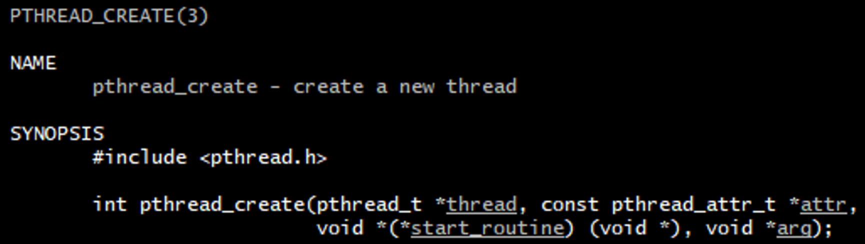
```c
/**
* Thread safe implementation of withdraw() using mutexes
*/
static bool withdraw_mutex( struct account *account, unsigned int amount )
{
    bool success = false;
    int rc = pthread_mutex_lock(&account->mutex);
    if ( rc != 0 ) {
        printf("pthread_mutex_lock failed with %d\n",rc);
    } else {
        const int balance = account->current_balance;
        if ( balance >= amount ) {
            success = true;
            printf("Withdrawl approved\n");
            account->current_balance = balance - amount;
            account->withdrawl_total += amount;
        }
        rc = pthread_mutex_unlock(&account->mutex);
        if ( rc != 0 ) {
            printf("pthread_mutex_unlock failed with %d\n",rc);
            success = false; // not sure if we should give out cash in this case, error on the safe side...
        }
        if ( success ) {
            disburse_money(amount);
        }
    }
    return success;
}
```

Linux System Programming Chapter 7
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture7/withdraw_shared.c

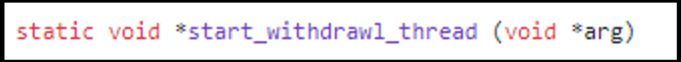
Deadlocks

- Two threads both waiting for each other to finish.
- One thread blocked on a mutex it already holds.
- How to avoid?
  - Lock data not code - are multiple locks really required?
  - Have a specific order for obtaining locks when multiple data locks are required.
  - Release in opposite order obtained.

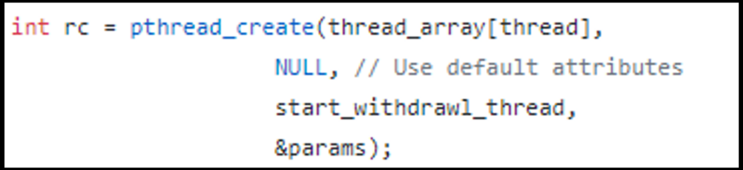
PThread Library - Creating



```
PTHREAD_CREATE(3)

NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);
```



```
static void *start_withdrawl_thread (void *arg)
```



```
int rc = pthread_create(thread_array[thread],
                        NULL, // Use default attributes
                        start_withdrawl_thread,
                        &params);
```

- Thread created/starts execution in start_withdrawl_thread, passed &params
- attr - null or default attributes (stack size, scheduling params, detached state)
- thread - null or location to store thread ID

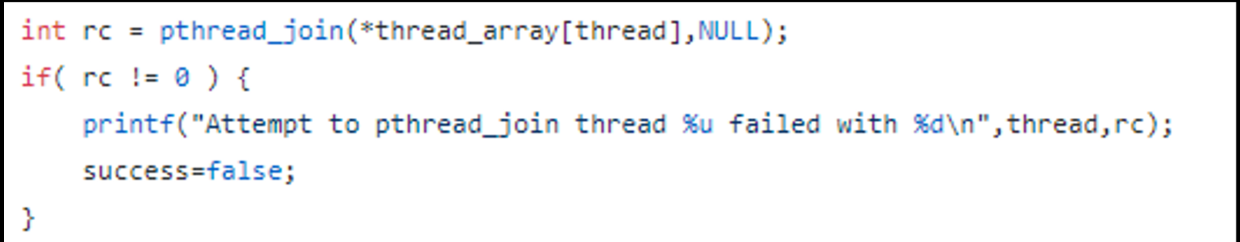
Linux System Programming Chapter 7
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture7/withdraw_threaded.c

PThread Library

- Ways to terminate a thread:
  - Return from start_thread routine
  - Invoke pthread_exit() (itself)
  - Cancelled by pthread_cancel() (another thread)
- When a process exits, all threads are killed

PThread Library

- Joining a thread
  - Block in one thread while waiting for another to terminate



```
int rc = pthread_join(*thread_array[thread],NULL);
if( rc != 0 ) {
    printf("Attempt to pthread_join thread %u failed with %d\n",thread,rc);
    success=false;
}
```

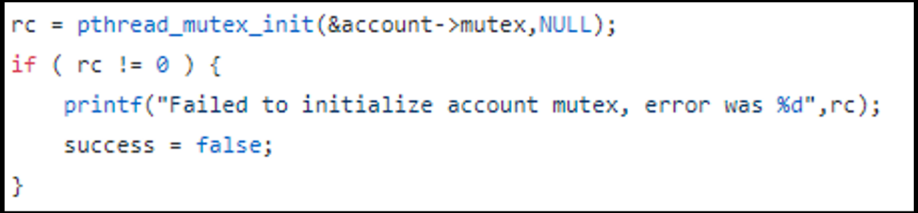
Linux System Programming Chapter 7
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture7/withdraw_threaded.c

PThread Library

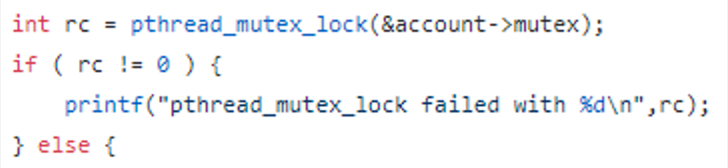
- Detached Threads
  - Threads which aren’t joinable - see pthread_detach()
  - Threads consume system resources until joined
- If attachable threads consume resources until joined, what happens if we don’t call pthread_join() on joinable threads?
  - Memory Leak

Linux System Programming Chapter 7
https://developer.ibm.com/tutorials/l-memory-leaks/

PThread Mutexes

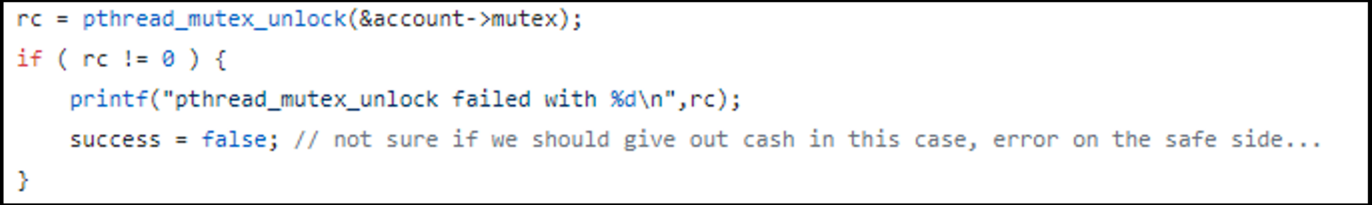


```
rc = pthread_mutex_init(&account->mutex,NULL);
if ( rc != 0 ) {
    printf("Failed to initialize account mutex, error was %d",rc);
    success = false;
}
```



```
int rc = pthread_mutex_lock(&account->mutex);
if ( rc != 0 ) {
    printf("pthread_mutex_lock failed with %d\n",rc);
} else {
```

Critical section goes here

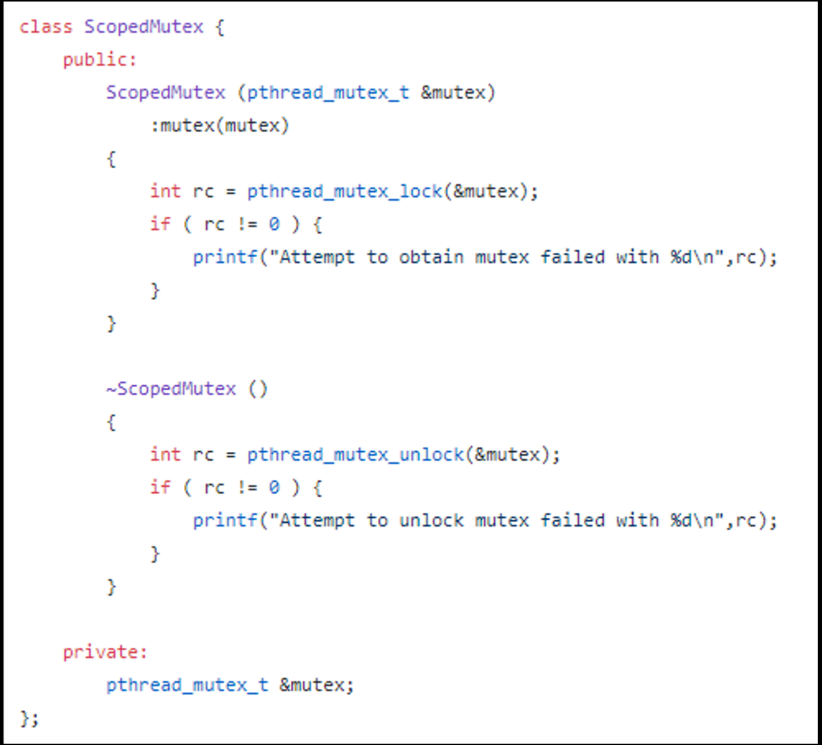


```
rc = pthread_mutex_unlock(&account->mutex);
if ( rc != 0 ) {
    printf("pthread_mutex_unlock failed with %d\n",rc);
    success = false; // not sure if we should give out cash in this case, error on the safe side...
}
```

- Why store the mutex in the account structure?
  - Lock data not code

Linux System Programming Chapter 7
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture7/withdraw_shared.c

Scoped Locks

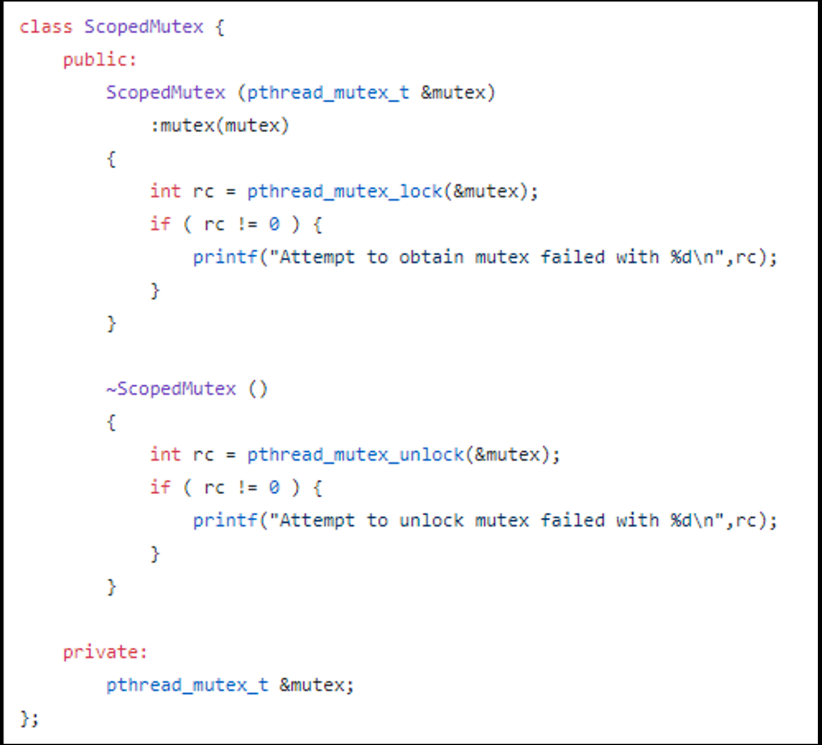


```
class ScopedMutex {
    public:
        ScopedMutex (pthread_mutex_t &mutex)
            :mutex(mutex)
        {
            int rc = pthread_mutex_lock(&mutex);
            if ( rc != 0 ) {
                printf("Attempt to obtain mutex failed with %d\n",rc);
            }
        }

        ~ScopedMutex ()
        {
            int rc = pthread_mutex_unlock(&mutex);
            if ( rc != 0 ) {
                printf("Attempt to unlock mutex failed with %d\n",rc);
            }
        }

    private:
        pthread_mutex_t &mutex;
};
```

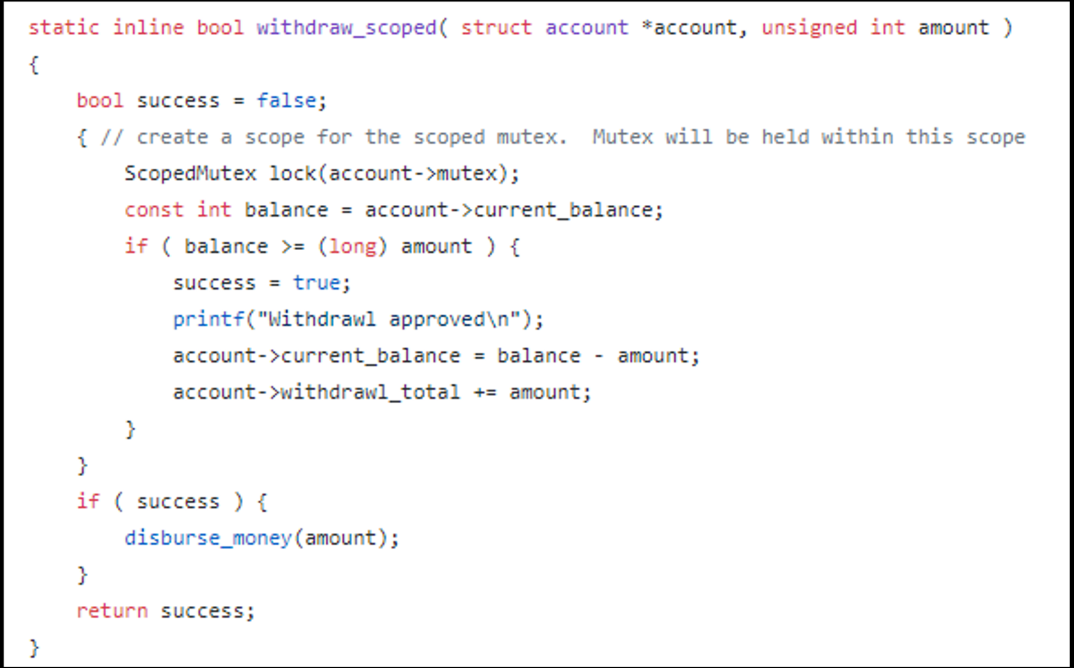
- Uses C++ allocation constructor/destructor to:
  - Acquire mutex on create (constructor)
  - Release mutex when falling out of scope (destructor
  - Known as RAII (Resource Acquisition Is Initialization)

Linux System Programming Chapter 7
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture7/withdraw_scoped.h

Scoped Locks



```
class ScopedMutex {
    public:
        ScopedMutex (pthread_mutex_t &mutex)
            :mutex(mutex)
        {
            int rc = pthread_mutex_lock(&mutex);
            if ( rc != 0 ) {
                printf("Attempt to obtain mutex failed with %d\n",rc);
            }
        }

        ~ScopedMutex ()
        {
            int rc = pthread_mutex_unlock(&mutex);
            if ( rc != 0 ) {
                printf("Attempt to unlock mutex failed with %d\n",rc);
            }
        }

    private:
        pthread_mutex_t &mutex;
};
```



```
static inline bool withdraw_scoped( struct account *account, unsigned int amount )
{
    bool success = false;
    { // create a scope for the scoped mutex.  Mutex will be held within this scope
        ScopedMutex lock(account->mutex);
        const int balance = account->current_balance;
        if ( balance >= (long) amount ) {
            success = true;
            printf("Withdrawl approved\n");
            account->current_balance = balance - amount;
            account->withdrawl_total += amount;
        }
    }
    if ( success ) {
        disburse_money(amount);
    }
    return success;
}
```

Linux System Programming Chapter 7
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture7/withdraw_scoped.h