sync-locks

some definitions

mutual exclusion: ensuring only one thread does a particular thing at a time

like updating shared balance

critical section: code that exactly one thread can execute at a time

result of mutual exclusion

lock: object only one thread can hold at a time

interface for creating critical sections

lock analogy

agreement: only change account balances while wearing this hat

normally hat kept on table

put on hat when editing balance

hopefully, only one person (= thread) can wear hat a time

need to wait for them to remove hat to put it on

“lock (or acquire) the lock” = get and put on hat

“unlock (or release) the lock” = put hat back on table

the lock primitive

locks: an object with (at least) two operations:

- *acquire* or *lock* — wait until lock is free, then "grab" it
- *release* or *unlock* — let others use lock, wakeup waiters

typical usage: everyone acquires lock before using shared resource

- forget to acquire lock? weird things happen

```
Lock(account_lock);
balance += ...;
Unlock(account_lock);
```

the lock primitive

locks: an object with (at least) two operations:

acquire or *lock* — *wait* until lock is free, then "grab" it

release or *unlock* — let others use lock, wakeup waiters

typical usage: everyone acquires lock before using shared resource

forget to acquire lock? weird things happen

```
Lock(account_lock);
balance += ...;
Unlock(account_lock);
```

waiting for lock?

when waiting — ideally:

not using processor (at least if waiting a while)

OS can context switch to other programs

pthread mutex

```
#include <pthread.h>

pthread_mutex_t account_lock;
pthread_mutex_init(&account_lock, NULL);
    // or: pthread_mutex_t account_lock =
    //               PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&account_lock);
balance += ...;
pthread_mutex_unlock(&account_lock);
...
pthread_mutex_destroy(&account_lock);
```

exercise

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
string one = "init one", two = "init two";
void ThreadA() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadA";  // (A1)
    pthread_mutex_unlock(&lock1);
    pthread_mutex_lock(&lock2);
    two = "two in ThreadA";  // (A2)
    pthread_mutex_unlock(&lock2);
}
void ThreadB() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadB";  // (B1)
    pthread_mutex_lock(&lock2);
    two = "two in ThreadB";  // (B2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
}
```

possible values of one/two after A+B run?

solution

B1+A2

A: L(1) A1 U(1) L

B: L(1) B1 L(2) B2 U(2) U(1)

A: L(2) A2 U(2)

NOT A1+B2

would need to run B1 before A1 before A2 before B2

not possible because Lock1 held for entire B1+B2 operation

so cannot fit A1+A2 in between B1 and B2

exercise (alternate 1)

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
string one = "init one", two = "init two";
void ThreadA() {
    pthread_mutex_lock(&lock2);
    two = "two in ThreadA";  // (A2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_lock(&lock1);
    one = "one in ThreadA";  // (A1)
    pthread_mutex_unlock(&lock1);
}
void ThreadB() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadB";  // (B1)
    pthread_mutex_lock(&lock2);
    two = "two in ThreadB";  // (B2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
}
```

possible values of one/two after A+B run?

exercise (alternate 2)

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
string one = "init one", two = "init two";
void ThreadA() {
    pthread_mutex_lock(&lock2);
    two = "two in ThreadA";  // (A2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_lock(&lock1);
    one = "one in ThreadA";  // (A1)
    pthread_mutex_unlock(&lock1);
}
void ThreadB() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadB";  // (B1)
    pthread_mutex_unlock(&lock1);
    pthread_mutex_lock(&lock2);
    two = "two in ThreadB";  // (B2)
    pthread_mutex_unlock(&lock2);
}
```

possible values of one/two after A+B run?

POSIX mutex restrictions

pthread_mutex rule: *unlock from same thread you lock in*

does this actually matter?

depends on how pthread_mutex is implemented

preview: general sync

lots of coordinating threads beyond locks

will talk about two general tools later:

- monitors/condition variables
- semaphores [if time]

big added feature: wait for arbitrary thing to happen

also some less general tools: barriers

a bad idea

one *bad* idea to wait for an event:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; bool ready = false;
void WaitForReady() {
    pthread_mutex_lock(&lock);
    do {
        pthread_mutex_unlock(&lock);
        /* only time MarkReady() can run */
        pthread_mutex_lock(&lock);
    } while (!ready);
    pthread_mutex_unlock(&lock);
}
void MarkReady() {
    pthread_mutex_lock(&lock);
    ready = true;
    pthread_mutex_unlock(&lock);
}
```

wastes processor time; MarkReady can stall waiting for unlock window

beyond locks

in practice: want more than locks for synchronization

for waiting for arbtirary events (without CPU-hogging-loop):

- monitors
- semaphores

for common synchornization patterns:

- barriers
- reader-writer locks

higher-level interface:

- transactions

Backup slides

backup slides

Java synchronized primitive

```
Object MilkLock = new Object();

/* lock implicity acquired/released on
   entering/leaving this block */
synchronized (MilkLock) {
    if (no milk) {
        buy milk
    }
}
```

C++11 mutexes

```
#include <mutex>

std::mutex MilkLock;
{
    std::lock_guard nameDoesNotMatter(MilkLock);
    /* nameDoesNotMatter's constructor acquires lock */
    if (no milk) {
        buy milk
    }
    /* nameDoesNotMatter's destructor called automatically
       and releases lock
     */
}
```

are locks enough?

do we need more than locks?

example 1: pipes?

pipes: one thread reads while other writes

want write to complete immediately if buffer space

want read operation to *wait* for write operation

not functionality provided by mutexes/barriers

pthread mutexes: addt'l features

mutex attributes (`pthread_mutexattr_t`) allow:

(reference: `man pthread.h`)

error-checking mutexes

locking mutex twice in same thread?

unlocking already unlocked mutex?

...

mutexes shared between processes

otherwise: must be only threads of same process

(unanswered question: where to store mutex?)

...

recall: pthread mutex

```
#include <pthread.h>

pthread_mutex_t some_lock;
pthread_mutex_init(&some_lock, NULL);
// or: pthread_mutex_t some_lock = PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&some_lock);
...
pthread_mutex_unlock(&some_lock);
pthread_mutex_destroy(&some_lock);
```

C++ containers and locking

can you use a vector from multiple threads?

… question: how is it implemented?

- dynamically allocated array
- reallocated on size changes

can access from multiple threads … *as long as not append/erase/etc.*?

assuming it's implemented like we expect…

- but can we really depend on that?
- e.g. could shrink internal array after a while with no expansion save memory?

C++ standard rules for containers

multiple threads can *read anything at the same time*

can only read element *if no other thread is modifying it*

can safely *add/remove elements if no other threads* are accessing container

(sometimes can safely add/remove in extra cases)

exception: vectors of bools — can't safely read and write at same time

might be implemented by putting multiple bools in one int