Condition Variables

CS 241

Prof. Brighten Godfrey

March 14, 2014

University of Illinois

Synchronization primitives

Mutex locks

- Used for exclusive access to a shared resource (critical section)
- Operations: Lock, unlock

Sempahores

- Generalization of mutexes: Count number of available "resources"
- Wait for an available resource (decrement), notify availability (increment)
- Example: wait for free buffer space, signal more buffer space

Condition variables

- Represent an arbitrary event
- Operations: Wait for event, signal occurrence of event
- Tied to a mutex for mutual exclusion

Condition variables

Goal: Wait for a specific event to happen

- Event depends on state shared with multiple threads

Solution: condition variables

- "Names" an event
- Internally, has a queue of threads waiting for the event

Basic operations

- Wait for event
- Signal occurrence of event to one waiting thread
- Signal occurrence of event to all waiting threads

Signaling, not mutual exclusion

- Condition variable is intimately tied to a mutex

cond_wait

Assumption

- Called with mutex locked by calling thread

Action

- Atomically releases mutex, and...
- ...blocks thread until condition is next signaled (past signal not "queued") or maybe only until some interruption occurs

After return

- mutex is already locked again

```
int pthread_cond_wait(pthread_cond_t  * cond,
                      pthread_mutex_t * mutex);
```

cond_signal

Action

- Unblocks at least one blocked thread waiting on signal

Note: "Mesa semantics" described here

- "Hoare semantics" different
- pthreads uses Mesa

```
int pthread_cond_signal(pthread_cond_t  * cond);
```

cond_broadcast

Action

- Unblocks all blocked threads waiting on signal

Note: "Mesa semantics" described here

- "Hoare semantics" different
- pthreads uses Mesa

```
int pthread_cond_broadcast(pthread_cond_t  * cond);
```

Producer-Consumer with Condition Variables

Producer-consumer problem

Chef (Producer)

Waiter (Consumer)

inserts items

removes items

Shared resource:
bounded buffer

Efficient implementation:
circular fixed-size buffer

Designing a solution

Chef (Producer)

Wait for empty slot
Insert item
Signal item arrival

Waiter (Consumer)

Wait for item arrival
Remove item
Signal slot available

What synchronization do we need?

Designing a solution

Chef (Producer)

Waiter (Consumer)

Wait for empty slot
Insert item
Signal item arrival

Mutex
(shared buffer)

Wait for item arrival
Remove item
Signal slot available

What synchronization do we need?

Designing a solution

Chef (Producer)

Waiter (Consumer)

Wait for empty slot
Insert item
Signal item arrival

Condition
slot frees up

Wait for item arrival
Remove item
Signal empty slot available

What synchronization do we need?

Designing a solution

Chef (Producer)

Waiter (Consumer)

Wait for empty slot
Insert item
Signal item arrival

Condition
item arrives

Wait for item arrival
Remove item
Signal empty slot available

What synchronization do we need?

Producer-Consumer with C.V.'s

```
/* Global variables */
pthread_mutex_t m;
pthread_cond_t  item_available;  /* Event: new item inserted */
pthread_cond_t  space_available; /* Event: item removed */
int items_in_buffer;
int max_items;

void init(void) {
    mutex_init(&m, NULL);
    cond_init(&item_available, NULL);
    items_in_buffer = 0;
    max_items = 100;
}
```

(Note: "pthread_" prefix removed from all synchronization calls for compactness)

Producer-Consumer with C.V.'s

```
void consumer(void)
{
    mutex_lock(&m);
    while (items_in_buffer == 0)
        cond_wait(&item_available, &m);
    /* Consume item */
    items_in_buffer--;
    cond_signal(&space_available);
    mutex_unlock(&m);
}
```

Producer-Consumer with C.V.'s

```
void consumer(void)
{
    mutex_lock(&m);
    while (items_in_buffer == 0)
        cond_wait(&item_available, &m);
    /* Consume item */
    items_in_buffer--;
    cond_signal(&space_available);
    mutex_unlock(&m);
}

void producer(void)
{
    mutex_lock(&m);
    while (items_in_buffer == max_items)
        cond_wait(&space_available, &m);
    /* Produce item */
    items_in_buffer++;
    cond_signal(&item_available);
    mutex_unlock(&m);
}
```

Obvious question #1

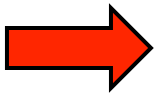
"Why does cond_wait() need to know about my mutex?

I'll just unlock the mutex separately."

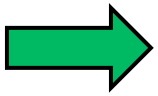
Condition variable without mutex

```
void consumer(void)
{
    mutex_lock(&m);
    while (items_in_buffer == 0) {
        mutex_unlock(&m);
        cond_wait(&item_available);
        mutex_lock(&m);
    }
    /* Consume item */
    items_in_buffer--;
    cond_signal(&space_available);
    mutex_unlock(&m);
}

void producer(void)
{
    mutex_lock(&m);
    ...
    items_in_buffer++;
    cond_signal(&item_available);
    mutex_unlock(&m);
}
```

A game of catch



```
void consumer(void)
{
    mutex_lock(&m);
    while (items_in_buffer == 0) {
        mutex_unlock(&m);
        cond_wait(&item_available);
        mutex_lock(&m);
    }
    /* Consume item */
    items_in_buffer--;
    cond_signal(&space_available);
    mutex_unlock(&m);
}
```



```
void producer(void)
{
    mutex_lock(&m);
    ...
    items_in_buffer++;
    cond_signal(&item_available);
    mutex_unlock(&m);
}
```

A game of catch

```
void consumer(void)
{
    mutex_lock(&m);
    while (items_in_buffer == 0) {
        mutex_unlock(&m);
        cond_wait(&item_available);
        mutex_lock(&m);
    }
    /* Consume item */
    items_in_buffer--;
    cond_signal(&space_available);
    mutex_unlock(&m);
}

void producer(void)
{
    mutex_lock(&m);
    ...
    items_in_buffer++;
    cond_signal(&item_available);
    mutex_unlock(&m);
}
```

} Problem: Not atomic

After unlock, producer acquires lock, creates condition event, sends signal all before wait() gets called!

Signal is lost

A game of catch

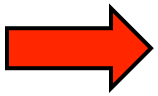
```
void consumer(void)
{
    mutex_lock(&m);
    while (items_in_buffer == 0) {
        cond_wait(&item_available, &m);
    }
    /* Consume item */
    items_in_buffer--;
    cond_signal(&space_available);
    mutex_unlock(&m);
}

void producer(void)
{
    mutex_lock(&m);
    ...
    items_in_buffer++;
    cond_signal(&item_available);
    mutex_unlock(&m);
}
```

Solution: atomic

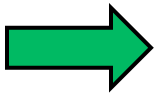
OS guarantees that calling thread will not miss signal

Ties together two actions: Checking if we should wait and Waiting happen while holding the mutex lock.

A successful game of catch



```
void consumer(void)
{
    mutex_lock(&m);
    while (items_in_buffer == 0)
        cond_wait(&item_available, &m);
    /* Consume item */
    items_in_buffer--;
    cond_signal(&space_available);
    mutex_unlock(&m);
}
```



```
void producer(void)
{
    mutex_lock(&m);
    ...
    items_in_buffer++;
    cond_signal(&item_available);
    mutex_unlock(&m);
}
```

Obvious question #2

"Why the while loop?

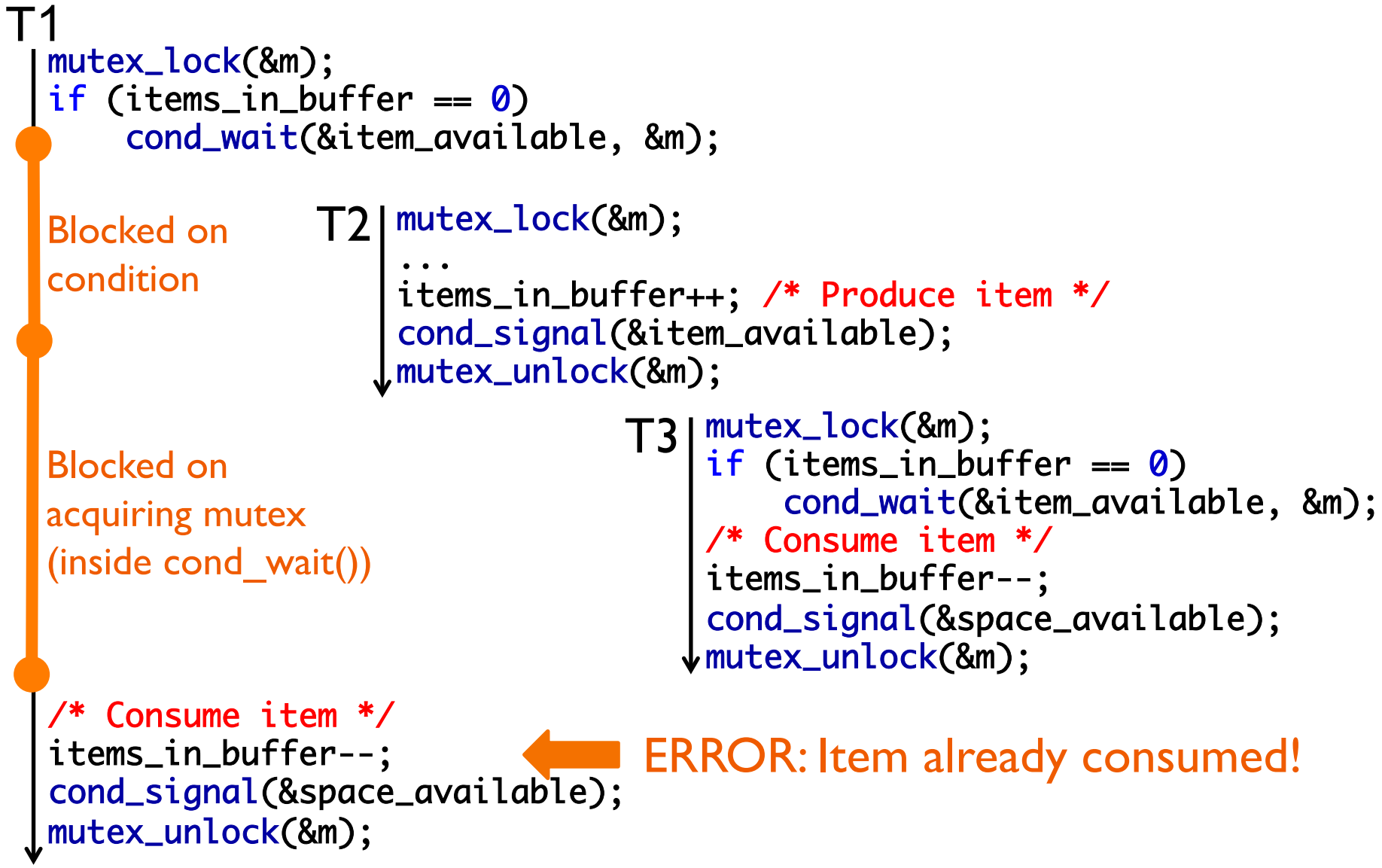
```
...
while (items_in_buffer == 0) {
   cond_wait(&item_available, &m);
...
```

I'll just do an `if` statement."

No while; just an if?

```
void consumer(void)
{
    mutex_lock(&m);
    if (items_in_buffer == 0)
        cond_wait(&item_available, &m);
    /* Consume item */
    items_in_buffer--;
    cond_signal(&space_available);
    mutex_unlock(&m);
}

void producer(void)
{
    mutex_lock(&m);
    ...
    items_in_buffer++;
    cond_signal(&item_available);
    mutex_unlock(&m);
}
```

No while; just an if?



Readers-Writers with Condition Variables

Readers-Writers Problem

Generalization of the mutual exclusion problem

Problem statement:

- *Reader* threads only read the object
- *Writer* threads modify the object
- Writers must have exclusive access to the object
- Unlimited number of readers can access the object

Thread 1 \ Thread 2	Reader	Writer
Reader	OK	No
Writer	No	No

Recall: Semaphore solution

Shared:

```
int readcnt;     /* Initially = 0 */
sem_t mutex, w; /* Both initially = 1 */
```

Writers:

```
void writer(void)
{
    while (1) {
        sem_wait(&w);

        /* Critical section */
        /* Writing here */

        sem_post(&w);
    }
}
```

Recall: Semaphore solution

Readers:

```
void reader(void)
{
    while (1) {
        sem_wait(&mutex);
        readcnt++;
        if (readcnt == 1) /* First reader in */
            sem_wait(&w); /* Lock out writers */
        sem_post(&mutex);

        /* Main critical section */
        /* Reading would happen here */

        sem_wait(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            sem_post(&w); /* Let in writers */
        sem_post(&mutex);
    }
}
```

(full code online)

Condition variable solution

Idea:

- If it's safe, just go ahead and read or write
- Otherwise, wait for my "turn"

Initialization:

```
/* Global variables */
pthread_mutex_t m;
pthread_cond_t  turn; /* Event: it's our turn */
int writing;
int reading;

void init(void) {
    pthread_mutex_init(&m, NULL);
    pthread_cond_init(&turn, NULL);
    reading = 0;
    writing = 0;
}
```

Condition variable solution

```
void reader(void)
{
    mutex_lock(&m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

```
void writer(void)
{
    mutex_lock(&m);
    while (reading || writing)
        cond_wait(&turn, &m);
    writing++;
    mutex_unlock(&m);

    /* Writing here */

    mutex_lock(&m);
    writing--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

(Note: "pthread_" prefix removed from all synchronization calls for compactness)

Familiar problem: Starvation

```
void reader(void)
{
    mutex_lock(&m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

```
void writer(void)
{
    mutex_lock(&m);
    while (reading || writing)
        cond_wait(&turn, &m);
    writing++;
    mutex_unlock(&m);

    /* Writing here */

    mutex_lock(&m);
    writing--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

(Note: "pthread_" prefix removed from all synchronization calls for compactness)

Idea: take turns

If a writer is waiting, then reader should wait its turn

- Even if it's safe to proceed (only readers are in critical section)

Requires keeping track of waiting writers

```
/* Global variables */
pthread_mutex_t m;
pthread_cond_t  turn; /* Event: someone else's turn */
int reading;
int writing;
int writers;

void init(void) {
    pthread_mutex_init(&m, NULL);
    pthread_cond_init(&turn, NULL);
    reading = 0;
    writing = 0;
    writers = 0;
}
```

Taking turns

```
void reader(void)
{
    mutex_lock(&m);
    if (writers)
        cond_wait(&turn, &m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

```
void writer(void)
{
    mutex_lock(&m);
    writers++;
    while (reading || writing)
        cond_wait(&turn, &m);
    writing++;
    mutex_unlock(&m);

    /* Writing here */

    mutex_lock(&m);
    writing--;
    writers--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

Another problem :-(

```
void reader(void)
{
    mutex_lock(&m);
    if (writers)
        cond_wait(&turn, &m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

```
void writer(void)
{
    mutex_lock(&m);
    writers++;
    while (reading || writing)
        cond_wait(&turn, &m);
    writing++;
    mutex_unlock(&m);

    /* Writing here */

    mutex_lock(&m);
    writing--;
    writers--;
    cond_signal(&turn);
    mutex_unlock(&m);
}
```

Only unblocks one thread at a time;

Inefficient if many readers are waiting

Easy solution: Wake everyone

```
void reader(void)
{
    mutex_lock(&m);
    if (writers)
        cond_wait(&turn, &m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_broadcast(&turn);
    mutex_unlock(&m);
}
```

```
void writer(void)
{
    mutex_lock(&m);
    writers++;
    while (reading || writing)
        cond_wait(&turn, &m);
    writing++;
    mutex_unlock(&m);

    /* Writing here */

    mutex_lock(&m);
    writing--;
    writers--;
    cond_broadcast(&turn);
    mutex_unlock(&m);
}
```

Pitfalls

signal() before wait()

- Waiting thread will miss the signal

Fail to lock mutex before calling wait()

- Might return error, or simply not block

`if (!condition) wait();` instead of `while (!condition) wait();`

- condition may still be false when wait returns!
- can lead to arbitrary errors (e.g., following NULL pointer, memory corruption, ...)

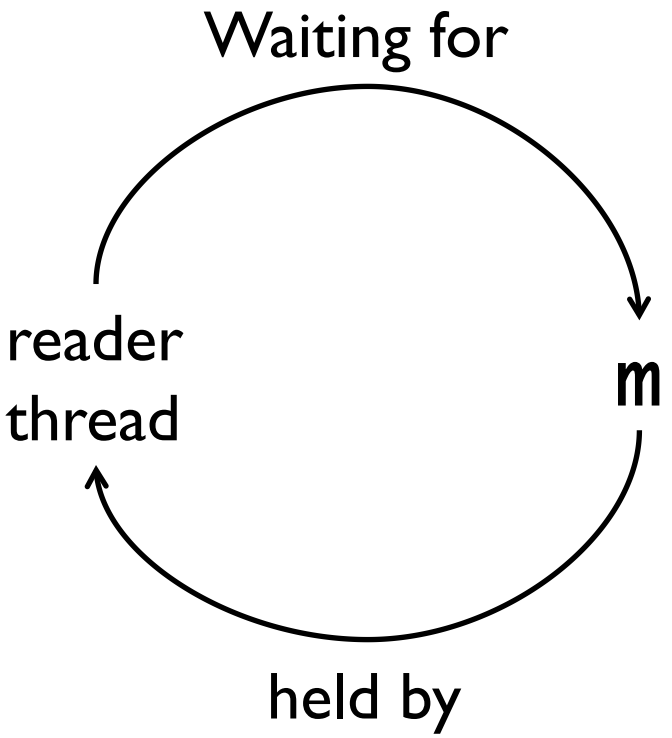
Forget to unlock mutex

- uh oh...

Forgetting to unlock the mutex

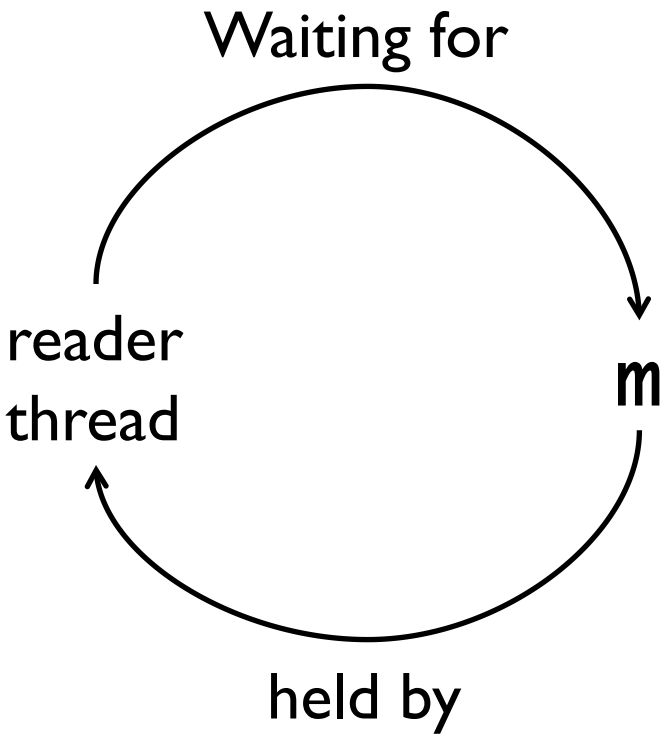
```
void reader(void)
{
    mutex_lock(&m);
    if (writers)
        cond_wait(&turn, &m);
    while (writing)
        cond_wait(&turn, &m);
    reading++;
    mutex_unlock(&m);

    /* Reading here */

    mutex_lock(&m);
    reading--;
    cond_broadcast(&turn);
    mutex_unlock(&m);
}

while (1) { reader() };
```

After running once, next time reader calls `mutex_lock(&m)`:



Forgetting to unlock the mutex

After running once,
next time reader calls
`mutex_lock(&m)`:



DEADLOCK

thread waits forever
for event that will
never happen

Semaphores vs. Condition Variables

Semaphore

- Integer value (≥ 0)
- Wait doesn't always block
- Signal either un-blocks thread or increments counter
- If signal releases thread, both may continue concurrently

Condition Variable

- No value
- Wait always blocks
- Signal either un-blocks thread or is lost
- If signal releases thread, only one continues
  - Need to hold mutex lock to proceed
  - Other thread is released from waiting on condition, but still has to wait to obtain the mutex again

Conclusion

Condition variables

- convenient way of signaling general-purpose events between threads

Common implementation: “monitors”

- An object which does the locking/unlocking for you when its methods are called
- See `synchronized` keyword in Java

Beware pitfalls...

- especially deadlock: our next topic