Advanced Threads & Monitor-Style Programming

First: Much of What You Know About Threads Is Wrong!

```
// Initially x == 0 and y == 0

// Thread 1         Thread 2
x = 1;              if (y == 1 && x == 0) exit();
y = 1;
```

- Can the above *exit* be called? How?

Threads Semantics

- You should stop thinking of threads as just executing interleaved
  - The interleaving model is called *sequential consistency*. It is not supported in practice.
- Instructions can be reordered!
- By the compiler, by the processor, by the memory subsystem
- Important to always use synchronization (mutexes) to get predictable behavior

Spinning in High-Level Code Is (Almost) Always Wrong!

```
while (!ready) /* do nothing */ ;
```

- The compiler (or hardware) is free to completely ignore this code
- If another thread does *ready* $=$ *true*, this thread may never see it
- Use of mutexes and condition variables inserts the right instructions to push data to main memory

Monitor-Style Programming

- Mutexes and condition variables are the basis of a concurrent programming model called *monitor-style* programming
- With these two constructs, we can implement *any* kind of critical section
- Critical section: code with controlled concurrent access
  - some logic for *concurrency* (which threads can run)
  - some logic for *exclusion* (which threads cannot run)
- Consider abstract operations *lock, unlock, signal, broadcast, wait*
  - map to *pthread_mutex_lock*, *pthread_mutex_unlock*, *pthread_cond_signal*, etc.
- We otherwise ignore thread creation, initialization boilerplate

Monitor-Style Programming Example: Readers/Writers

- Build a critical section that any number of *reader* threads or a single *writer* thread can enter, as long as there is no *writer* thread in it.
- Concurrency logic: multiple reader threads can enter
- Exclusion logic: any writer thread excludes all other threads

Monitor-Style Programming Example: Readers/Writers

```
Mutex mutex;
Condition read_cond, write_cond;
int readers = 0;
bool writer = false;

// READER:
lock(mutex);
while (writer)
  wait(read_cond, mutex);
readers++;
unlock(mutex);
... // read data
lock(mutex);
readers--;
if (readers == 0)
  signal(read_cond);
unlock(mutex);

// WRITER:
lock(mutex);
while (readers>0 ||writer)
  wait(write_cond, mutex);
writer = true;
unlock(mutex);
... // write data
lock(mutex);
writer = false;
broadcast(read_cond);
signal(write_cond);
unlock(mutex);
```

Monitor-Style Programming Example: Recursive Lock

```
Mutex mutex;
Condition held;
int count = 0;
thread_id holder = NULL;

acquire() {
  lock(mutex);
  while (count > 0 && holder != self())
    wait(held, mutex);
  count++;
  holder = self();
  unlock(mutex);
}
release() {
  lock(mutex);
  count--;
  if (count == 0)
    signal(held);
  unlock(mutex);
}
```

General Pattern: *Any* Critical Section

- Usage: *CS_enter(); ... [critical section] ... CS_exit();*

```
[shared data, including Mutex m, Condition c]
CS_enter() {
  lock(m);
  while (![condition])
    wait(c, m);
  [change shared data to reflect in_CS]
  [broadcast/signal as needed]
  unlock(m);
}

CS_exit() {
  lock(m);
  [change shared data to reflect out_of_CS]
  [broadcast/signal as needed]
  unlock(m);
}
```

Why Signal/Broadcast on CS_enter()?

- Any change to shared data may make a condition (on which some thread waits) false
- Example: critical section with red and green threads, up to 3 can enter, red have priority
  - red have priority = no green can enter, if red is waiting

Red+Green, Up to 3, Red Have Priority

```
Mutex mutex;
Condition red_cond, green_cond;
int red_waiting = 0, green = 0, red = 0;

green_acquire() {
  lock(mutex);
  while (green+red == 3 || red_waiting != 0)
    wait(green_cond, mutex);
  green++;
  unlock(mutex);
}
green_release() {
  lock(mutex);
  green--;
  signal(green_cond);
  signal(red_cond);
  unlock(mutex);
}
```

Red+Green, Up to 3, Red Have Priority

```
red_acquire() {
  lock(mutex);
  red_waiting++;
  while (green+red == 3)
    wait(red_cond, mutex);
  red_waiting--;
  red++;
  broadcast(green_cond);
  unlock(mutex);
}

red_release() {
  lock(mutex);
  red--;
  signal(green_cond);
  signal(red_cond);
  unlock(mutex);
}
```

Why Use *while* Around *wait*?

- Defensive programming: if we return from *wait* by mistake (or *spuriously*), we still check
- Other threads may have changed the condition since the time we were signalled
- Recall producer-consumer standard example:

```
// Consumer
lock(mutex);
while (empty(buffer)) wait(empty_cond, mutex);
get_request(buffer);
unlock(mutex);

// Producer
lock(mutex);
put_request(buffer);
broadcast(empty_cond);
unlock(mutex);
```

Monitor-Style Programming Errors

- Most problems with concurrent programming are simple oversights that are easy to introduce *due to partial program knowledge* and near-impossible to debug!
- People forget to access shared variables in locks, forget to signal when a condition changes, etc.

The Golden Rules of Monitor-Style Programming

- Associate (in your mind+comments) every piece of shared data in your program with a mutex that protects it. Use it consistently.
- For every boolean condition (in the program text) use a separate condition variable.
- Every time the boolean condition may have changed, *broadcast* on the condition variable.
- Only call *signal* when you are certain that *any and only one* waiting thread can enter the critical section.
- Globally order locks, acquire in order in all threads.

Example Exercise

- Critical section with red and green threads, up to 3 can enter, not all the same color

Why Multi-Threaded Programming Is Hard

- The most common concurrent programming bug is a *race*
  - Technically, race = unsynchronized accesses to the same shared data by two threads, with either access being a write.
- But that's not the real problem. We can avoid all races automatically:
  - just rewrite the program to have a lock per memory word
  - acquire it before reading/writing
  - release afterwards
- Is this enough?

Race/No-Race Example for Consumer Pattern

```
// Race
lock(mutex);
while (empty(buffer)) wait(empty_cond, mutex);
unlock(mutex);
get_request(buffer);

// No Race
lock(mutex);
while (empty(buffer)) wait(empty_cond, mutex);
unlock(mutex);
lock(mutex);
get_request(buffer);
unlock(mutex);
```

- Equally bad! We turned a race into an *atomicity violation*
- The problem is that some actions need to be consistent/atomic

Other Concurrency Errors

- We already saw *races* and *atomicity violations*
- We also get *ordering violations* and *deadlocks*
- Ordering violation: logical error, where something is read before it is set to the right value
  - much like an atomicity violation
- Deadlock: typically a cycle in the lock holding order
- E.g., thread A locks m1, B locks m2, A tries to lock m2, B tries to lock m1

Why Multi-Threaded Programming Is Hard (II)

- No safe approach:
  - *Coarse-grained* locking: few, central locks (e.g., one per program or per data structure)
    - problem: lack of parallelism, higher chance of deadlock
  - *Fine-grained* locking: locks protecting small amounts of data (e.g., each node of a data structure)
    - problem: higher chance of races, atomicity violations

Why Multi-Threaded Programming Is Hard (III)

- The real problem: holding locks is a global property
  - affects entire program, cannot be hidden behind an abstract interface
  - results in lack of modularity: callers cannot ignore what locks their callees acquire or what locations they access
    - necessary for race avoidance, but also for global ordering to avoid deadlock
    - part of a method's protocol which lock needs to be held when called, which locks it acquires
- Condition variables are also non-local: every time some value changes, we need to know which condition var may depend on it to signal it!
- Everything exacerbated by aliasing (pointers)
  - are two locks the same?
  - are two data locations the same?
- End result: lack of composability, cannot build safe services out of other safe services

Example of Difficulties: Account Library

```
typedef struct account {
  int balance = 0;
  Mutex account_mutex;
} account_type;

void withdraw(account_type *acc, int amount) {...}
void synch_withdraw(account_type *acc, int amount) {
  lock(acc->account_mutex);
  withdraw(acc, amount);
  unlock(acc->account_mutex);
}

void deposit(account_type *acc, int amount) { ... }
void synch_deposit(account_type *acc, int amount) {
  lock(acc->account_mutex);
  deposit(acc, amount);
  unlock(acc->account_mutex);
}
...
```

Example of Difficulties (cont'd)

```
// Client code
void move(account_type *acc1,
          account_type *acc2, int amount) {
  synch_withdraw(acc1, amount);
  synch_deposit(acc2, amount);
}
```

- Problem: atomicity violation
  - state of accounts can be observed *between* withdrawal and deposit
  - how can *move* be made atomic?
  - cannot just use a "move" lock: other code won't respect it

One More Try

- Library can expose unsynchronized functions *withdraw/deposit*

```
// Client
void atomic_move(account_type *acc1,
            account_type *acc2, int amount) {
  lock(acc1->account_mutex);
  lock(acc2->account_mutex);
  withdraw(acc1, amount);
  deposit(acc2, amount);
  unlock(acc2->account_mutex);
  unlock(acc1->account_mutex);
}
```

- Problem: deadlock
  - *move(s,t,...)* parallel with *move(t,s,...)*
  - *move(s,s,...)*: self-deadlock