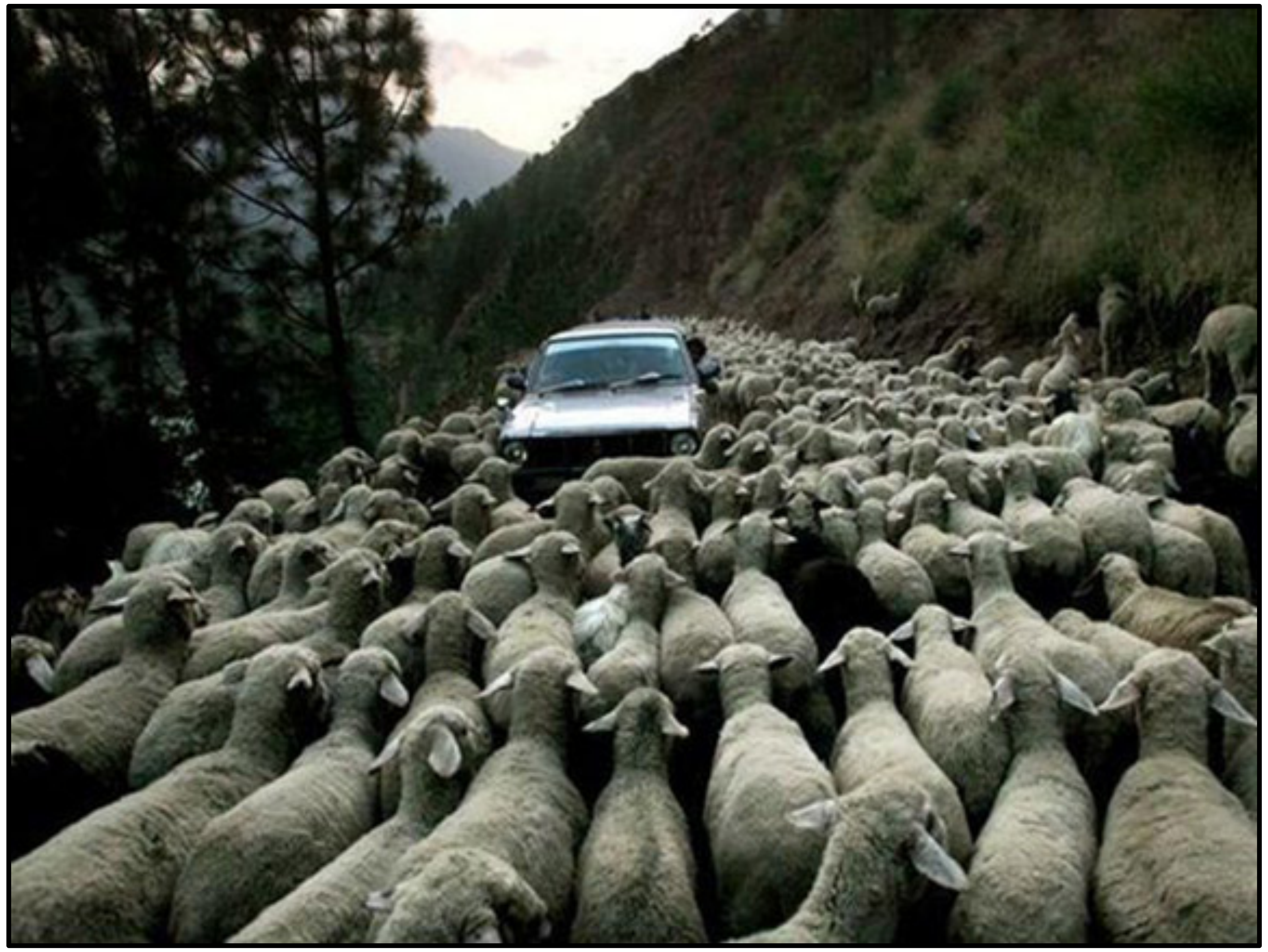
C4GC: Concurrency

DMITRY DANILOV

Why use concurrency?

- Using concurrency for separation of concerns (classic example of GUI and worker threads)
- Using concurrency for performance:
  - Task parallelism – divide tasks into parts and run each in parallel
  - Data parallelism – performing the same operation on multiple sets of data concurrently



CP.1: Assume that your code will run as a part of multi-threaded program

We can’t be sure that concurrency isn’t used now or will be used sometime in the future. Code gets reused. Libraries using threads may be used from other parts of the program.

Example:
The function works perfectly in a single-threaded environment but in a multi-threaded environment the two static variables result in data races!

```cpp
double cached_computation(double x) {
    static double cached_x = 0.0;
    static double cached_result = COMPUTATION_OF_ZERO;
    if (cached_x == x) return cached_result;
    double result = computation(x);
    cached_x = x;
    cached_result = result;
    return result;
}
```

Ways to fix the issue:

- Delegate concurrency concerns upwards to the caller
- Mark the static variables as `thread_local` (which might make caching less effective)
- Implement concurrency control, for example, protecting the two static variables with a static lock (which might reduce performance)
- Refuse to build and/or run in a multi-threaded environment
- Provide two implementations(for example, a template parameter), one to be used in single-threaded environments and the other one for multi-threaded environments' use

CP.22: Never call unknown code while holding a lock(e.g., a callback)

If you don't know what a piece of code does, you are risking deadlock:

Example:

```
void Observable::change() {

    lock_guard<mutex> lock {mutex};
    //do something
    _observer->onUpdate(this);
    //do something
}
```

If you don't know what `onUpdate` does, it may call `Observable::change` (indirectly) and cause a deadlock on `mutex`.

Ways to fix the issue:

We can avoid dead locks by using `recursive_mutex` in `Observable::change`.

Example:

```
void Observable::change() {

    lock_guard<recursive_mutex> lock {mutex};
    //do something
    _observer->onUpdate(this);
    //do something
}
```

CP.25: Prefer `gsl::joining_thread` over `std::thread`

Detached threads are hard to monitor. It is harder to ensure absence of errors in detached threads (and potentially detached threads).

Example:

```
void f() { std::cout << "Hello "; }

struct F {
    void operator()() { std::cout << "parallel world "; }
};

int main() {
    std::thread t1{f};      // f() executes in separate thread
    std::thread t2{F()};    // F()() executes in separate thread
}  // spot the bugs
```

Ways to fix the issue:

A `gsl::joining_thread` is a thread that joins automatically at the end of its scope.

Example:

```cpp
void f() { std::cout << "Hello "; }

struct F {
    void operator()() { std::cout << "parallel world "; }
};

int main() {
    gsl::joining_thread t1{f};      // f() executes in separate thread
    gsl::joining_thread t2{F()};    // F()() executes in separate thread
}  // both threads are joined on the end of the scope
```

CP.26: Don't `detach()` a thread

A typical use of `detach()`:

Example:

```
void heartbeat();

void use() {
    std::thread t(heartbeat); // don't join;
    // heartbeat is meant to run forever
    t.detach();
}
```

How do we monitor the detached thread to see if it is alive?

Something might go wrong with the heartbeat, and losing a heartbeat can be very serious in a system for which it is needed.

CP.26: Don’t `detach()` a thread

An alternative, and usually superior solution is to control its lifetime by placing it in a scope outside its point of creation (or activation). For example:

```
void heartbeat();

gsl::joining_thread t(heartbeat); // heartbeat is meant to run "forever"
```

Sometimes, we need to separate the point of creation from the point of ownership:

```
void heartbeat();
unique_ptr<gsl::joining_thread> tick_tock {nullptr};

void use() {
    // heartbeat is meant to run as long as tick_tock lives
    tick_tock = make_unique<gsl::joining_thread>(heartbeat);
}
```

Why use concurrency??? .. Alternatives

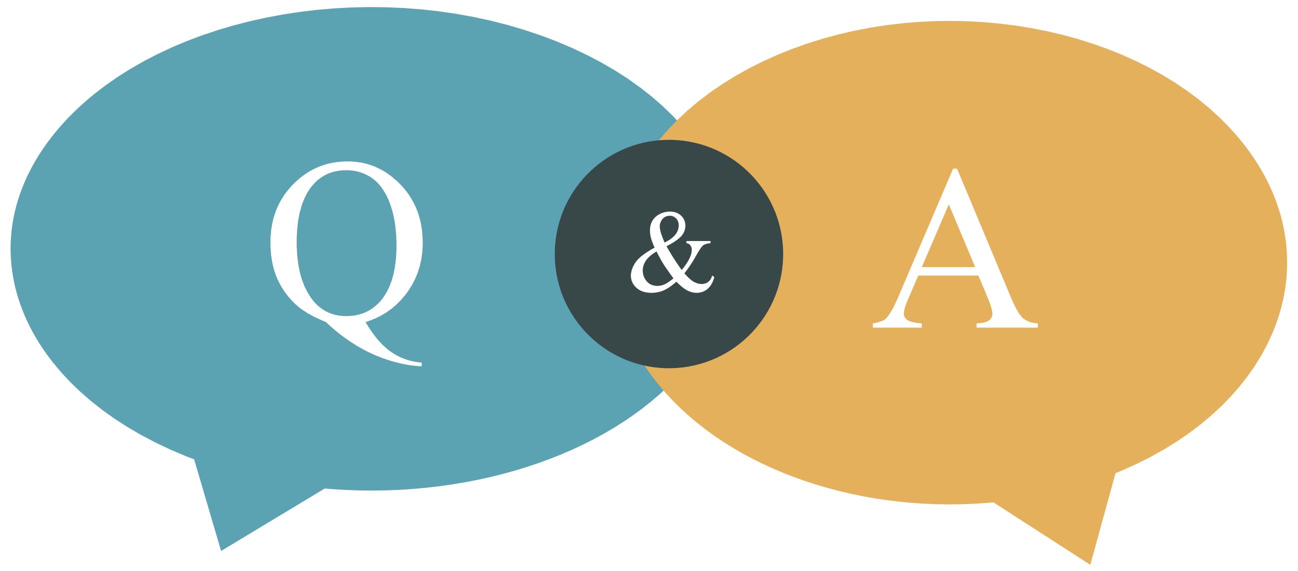
- Think of libraries/frameworks providing higher level of abstraction:
  - Intel® Thread Building Blocks
  - Microsoft® Parallel Patterns Library
  - OpenMP

- Think of threads (or even separate processes) communicating via IPC(message queues), file system events, sockets, etc.

- Go-style concurrency in C(libdill && libmill)

Q & A SESSION

THANKS FOR LISTENING



LINKEDIN/DDANILOV

DMITRY.B.DANILOV@GMAIL.COM