

Lecture 10

Concurrency & Multithreading

Games are Naturally Multithreaded

- The core game loop is **time constrained**
 - Frame rate sets a budget of how much you can do
 - Exceeding that budget causes frame rate drops
- Sometimes we need an extra thread to ...
 - Offload tasks that *block* drawing (**asset loading**)
 - Offload tasks that *slow* drawing (**pathfinding**)
 - Execute tasks *decoupled* from drawing (**audio**)
- Part of architecture spec: **computation model**

Multithreading in CUGL

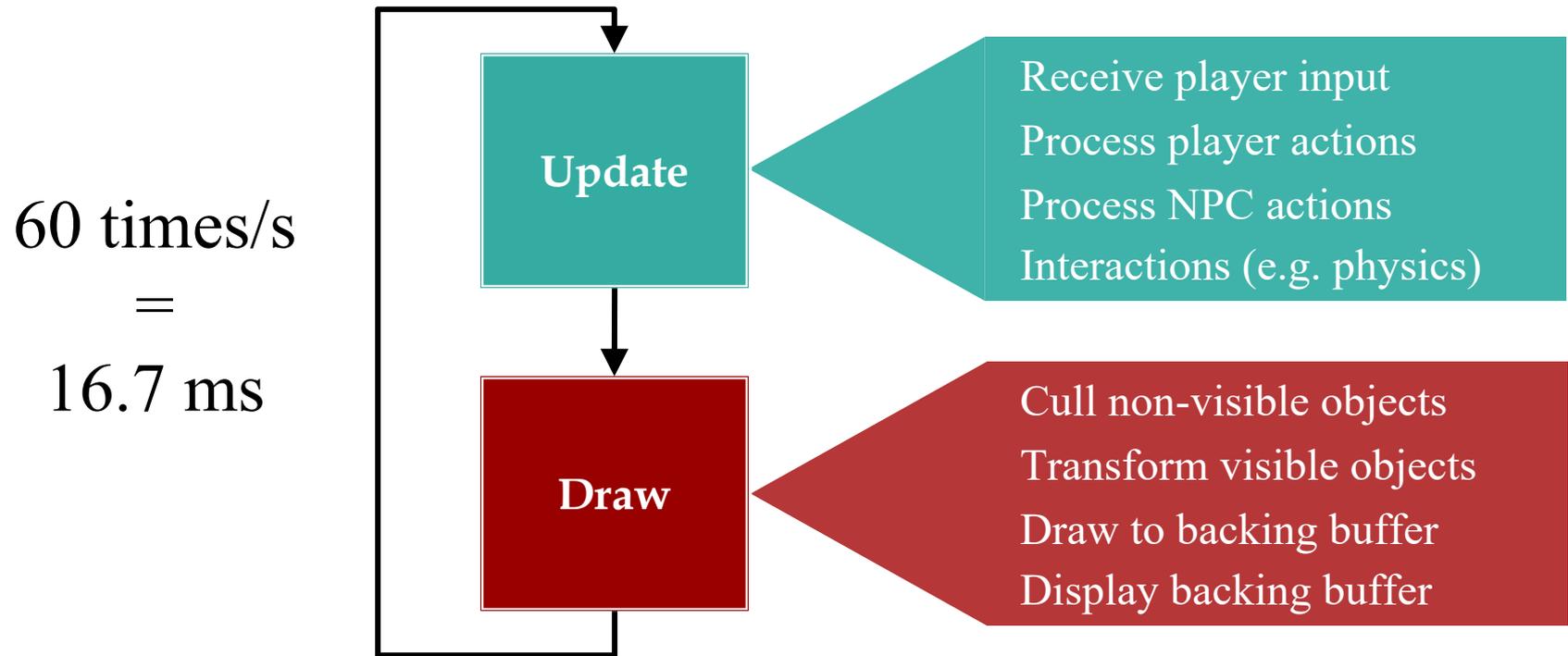
- CUGL has **three** primary threads
 - The `Application`, or main graphics thread
 - The `AssetManager` thread, for loading assets
 - The `AudioEngine` thread, for audio playback
 - Note that only `Application` is required
- Also has tools for making your own threads
 - Most are built on top of C++ and `std::thread`
 - But there are some unique features too

Multithreading in CUGL

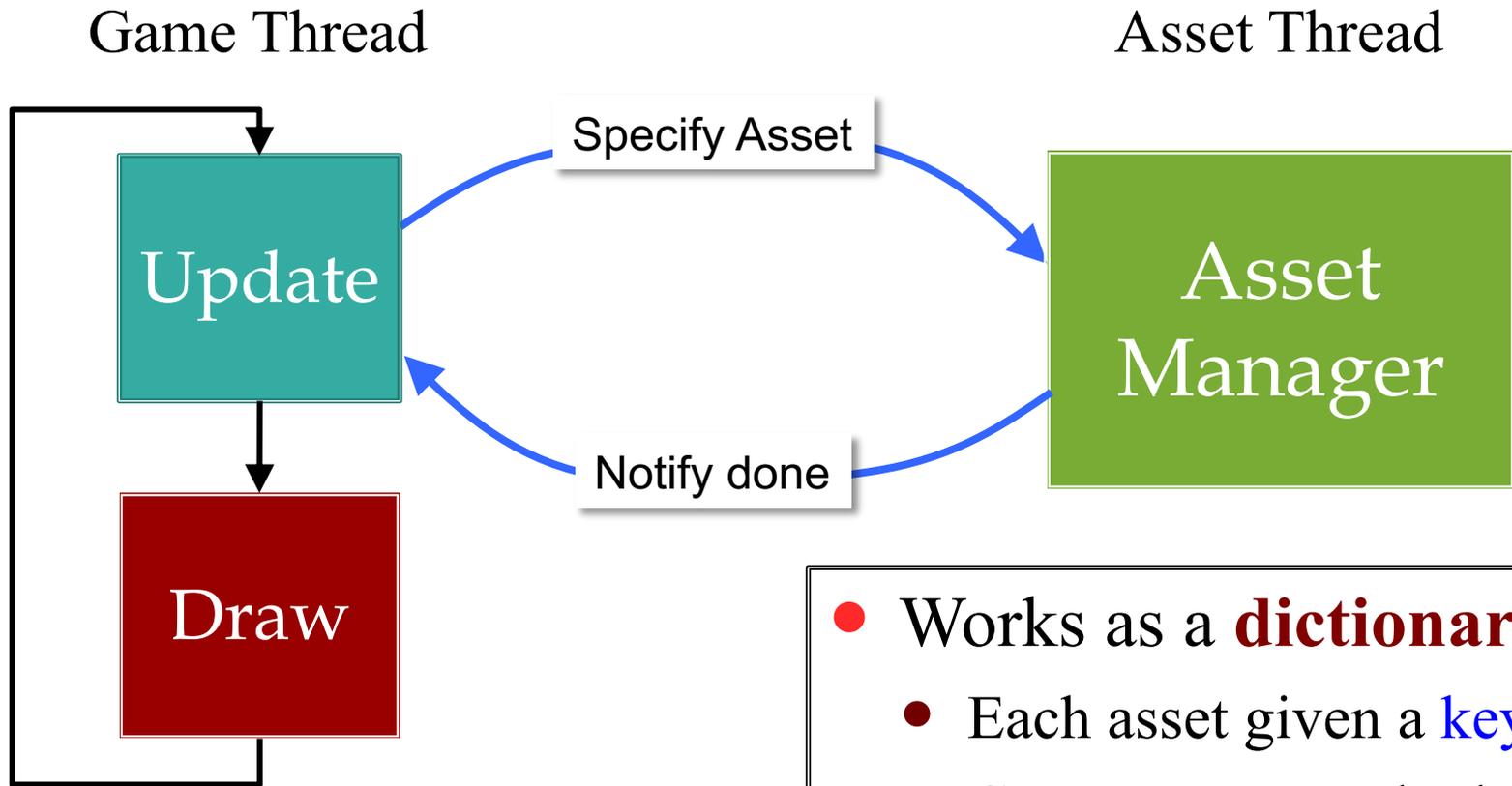
- CUGL has **three** primary threads
 - The [Application](#), or main graphics thread
 - The [AssetManager](#) thread, for loading assets
 - The [AudioManager](#) thread, for loading and playing audio
 - No other threads
- Also has tools for making your own threads
 - Most are built on top of C++ and `std::thread`
 - But there are some unique features too

Understanding the three threads can help us to make our own.

Recall: The Application Thread

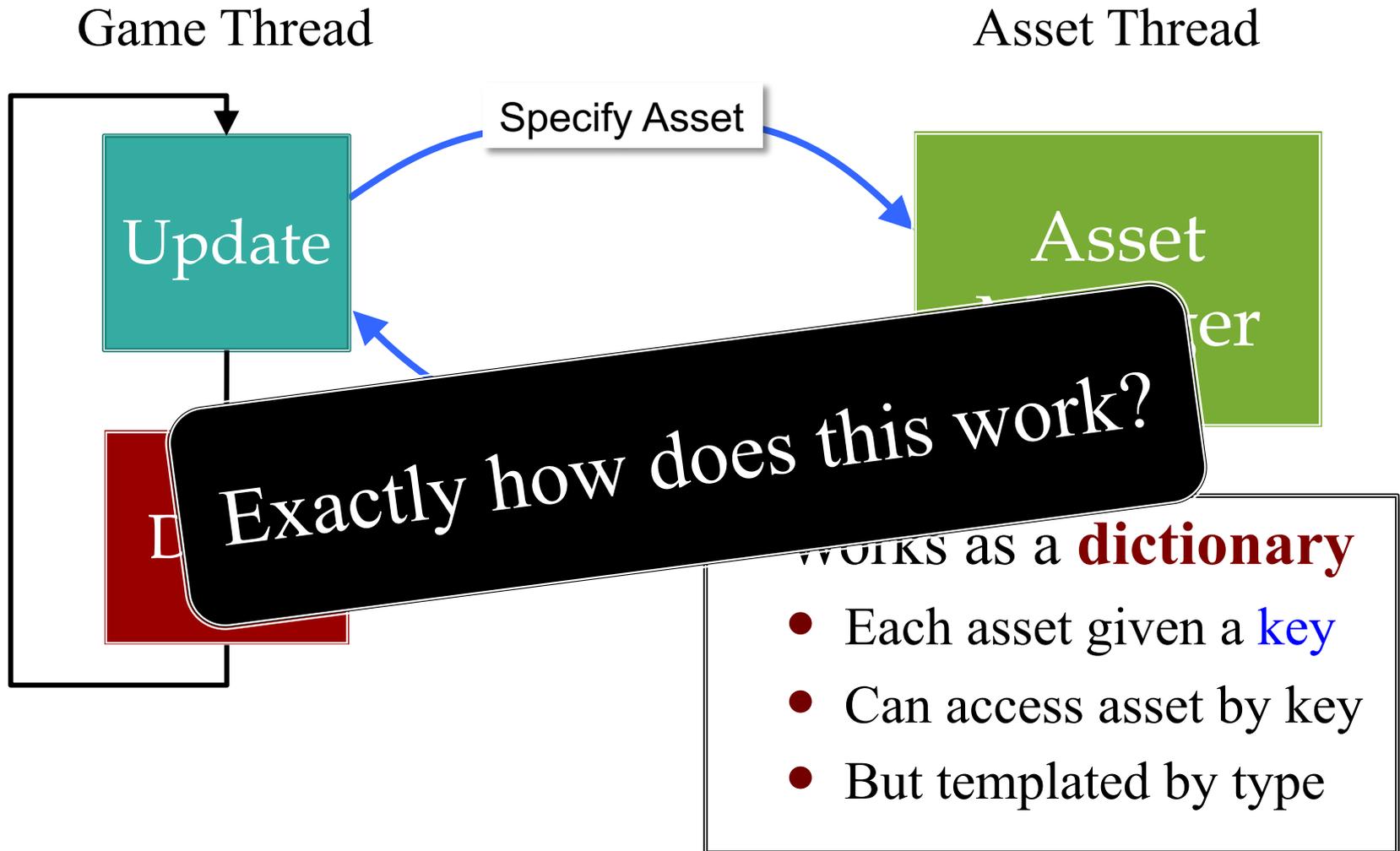


The AssetManager Thread



- Works as a **dictionary**
 - Each asset given a **key**
 - Can access asset by key
 - But templated by type

The AssetManager Thread



Asset Loading Consists of Tasks

Task 1

Load Font
"Times.ttf"

Task 2

Load Image
"smile.png"

Task 3

Load Sound
"music.ogg"

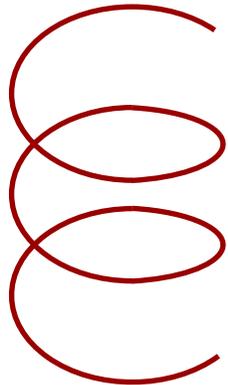
Task 4

Load Widget
"menu.json"

Ideally, Each One is a Thread

Task 1

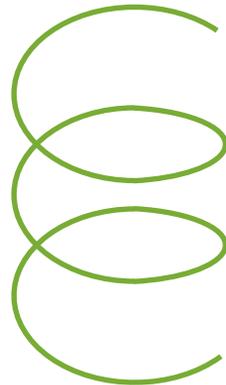
Load Font
"Times.ttf"



Thread 1

Task 2

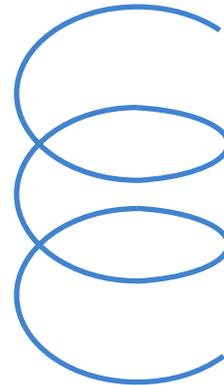
Load Image
"smile.png"



Thread 2

Task 3

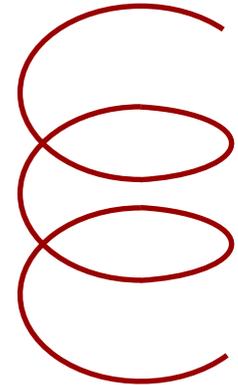
Load Sound
"music.ogg"



Thread 3

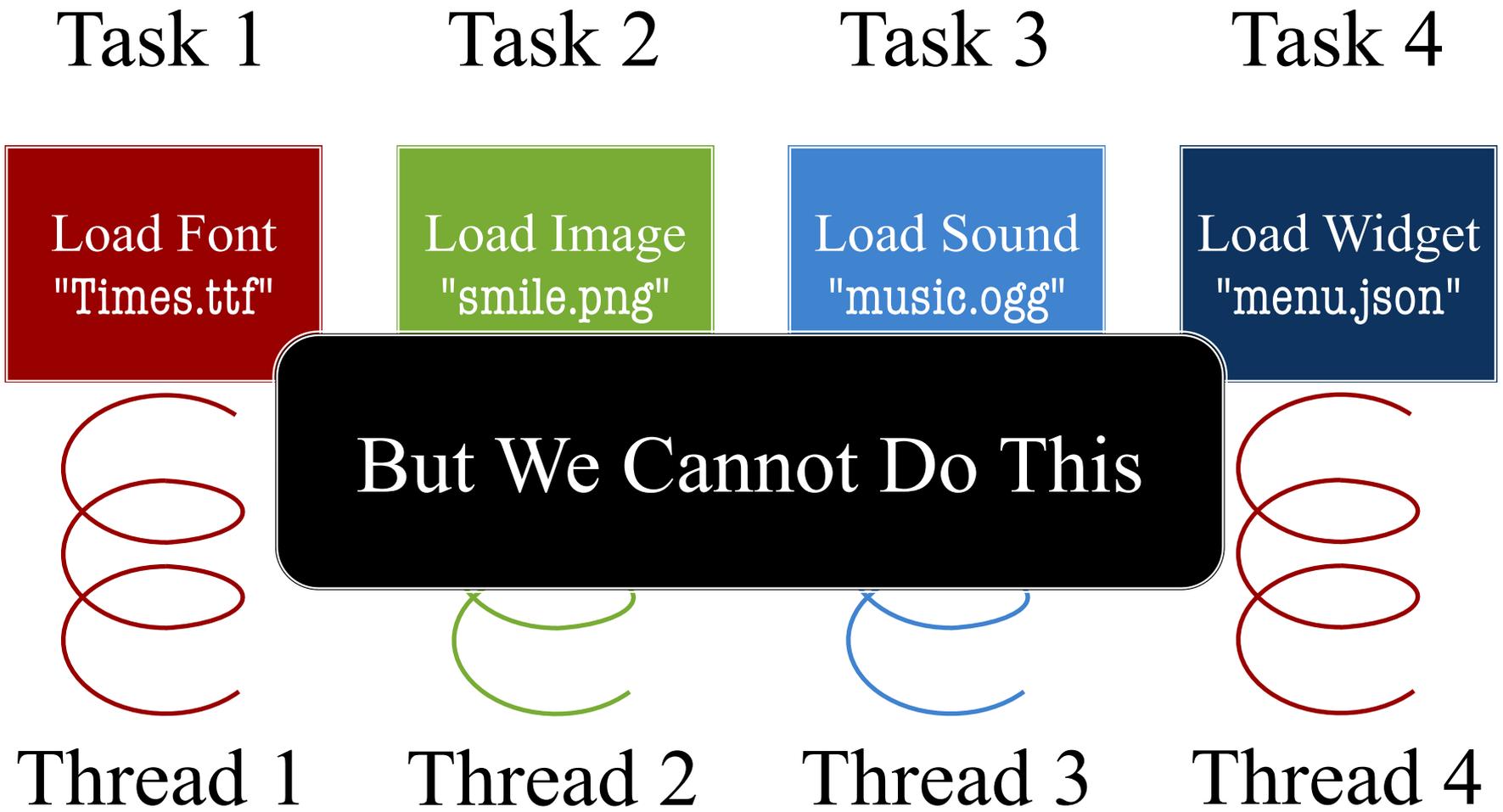
Task 4

Load Widget
"menu.json"



Thread 4

Ideally, Each One is a Thread



What is the Problem?

- Some tasks have **shared resources**
 - **Example:** Fonts all use same engine to make atlases
 - Cannot execute without protecting critical section
 - Typically easier to just **not** do them concurrently
- Some tasks have **dependencies**
 - **Example:** Widgets must come after images, fonts
 - Forces an order on the asset loading
- What we want is a task **service manager**
 - Executes given tasks in a *partial order*

Solution: Thread Pool

Task 4

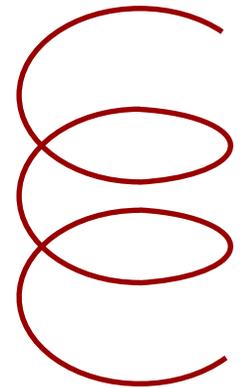
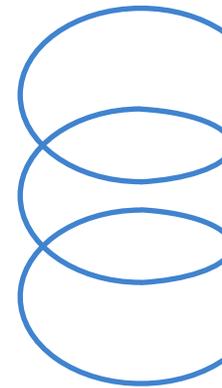
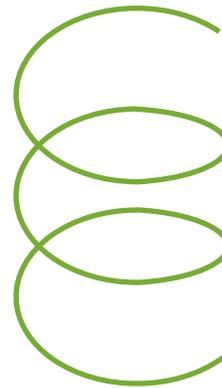
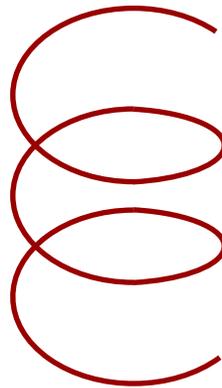
Task 3

Task 2

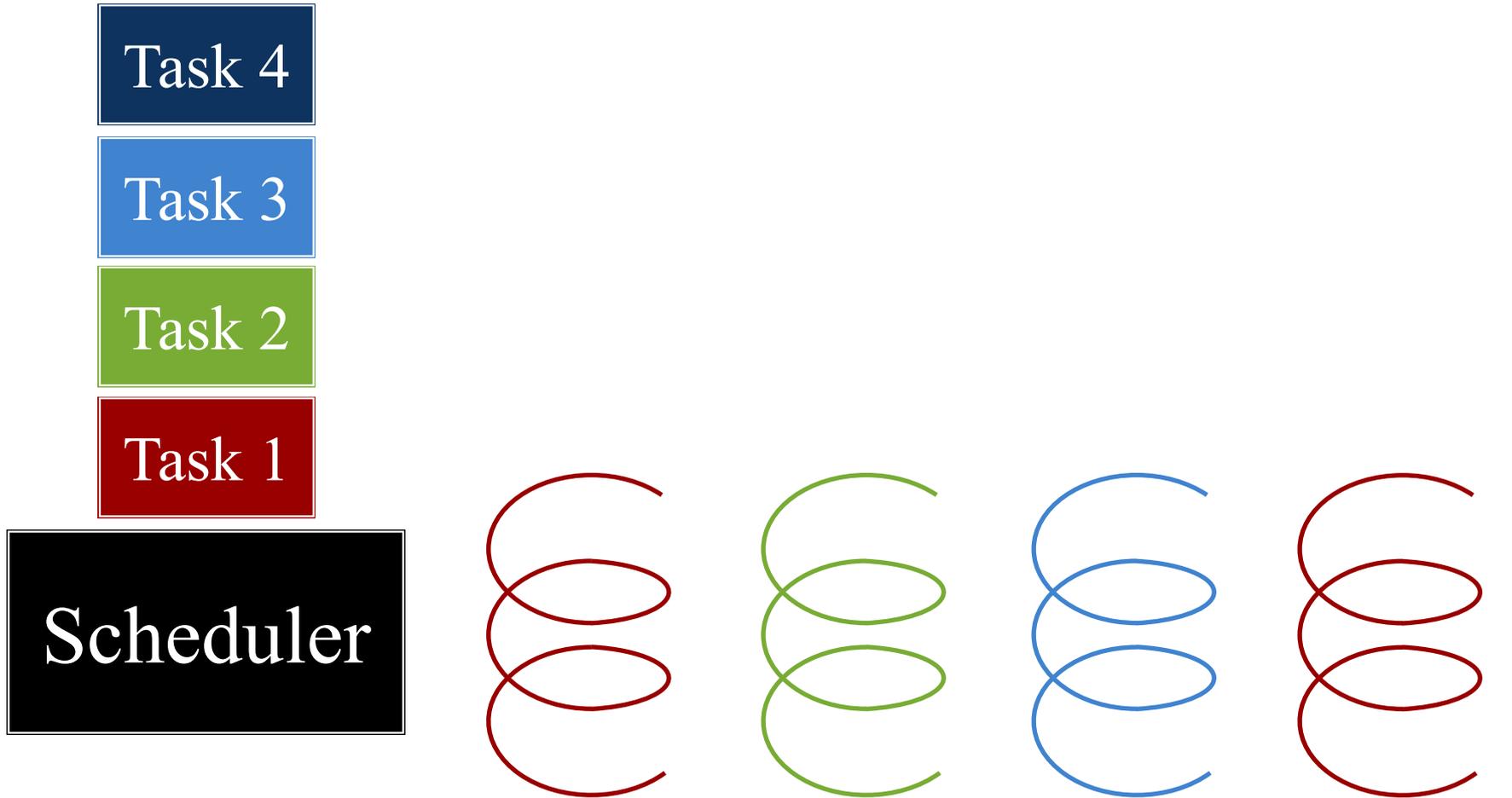
Task 1

Scheduler

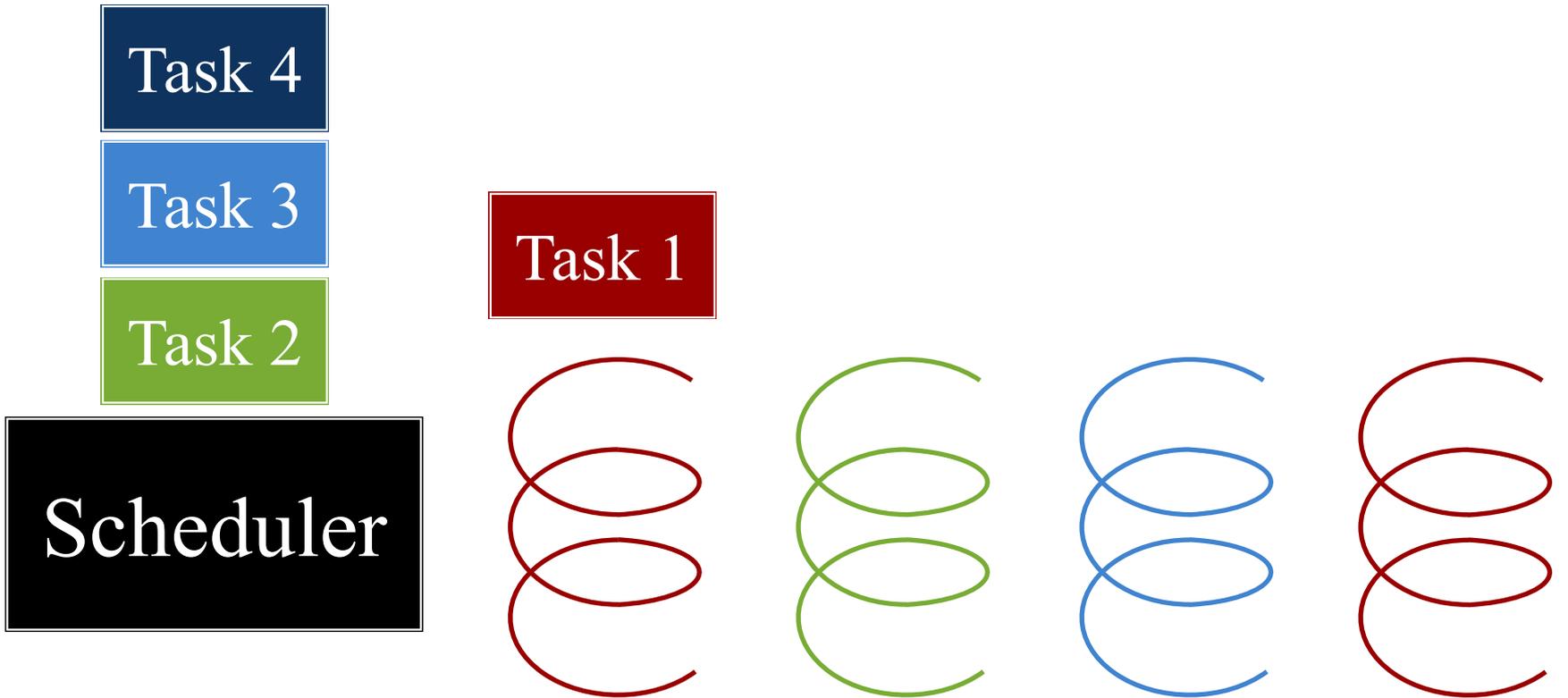
- Threads + scheduler
- Scheduler puts tasks thread
- Uses first available thread
- Holds tasks if all busy



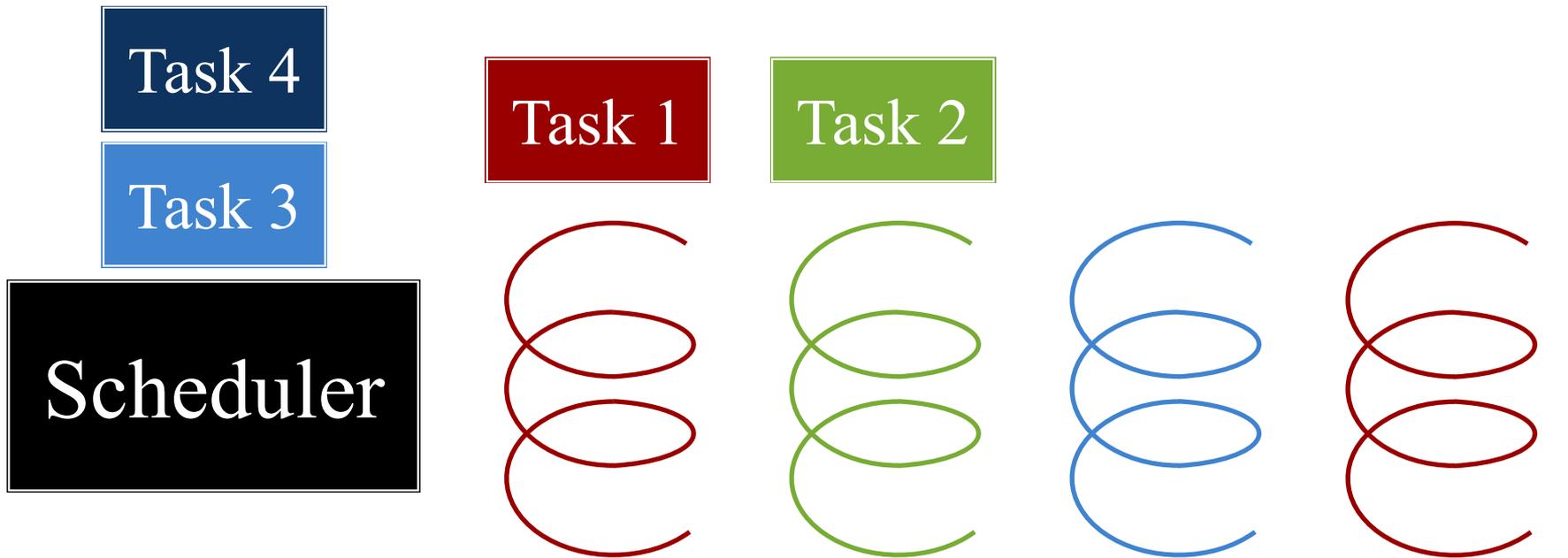
Solution: Thread Pool



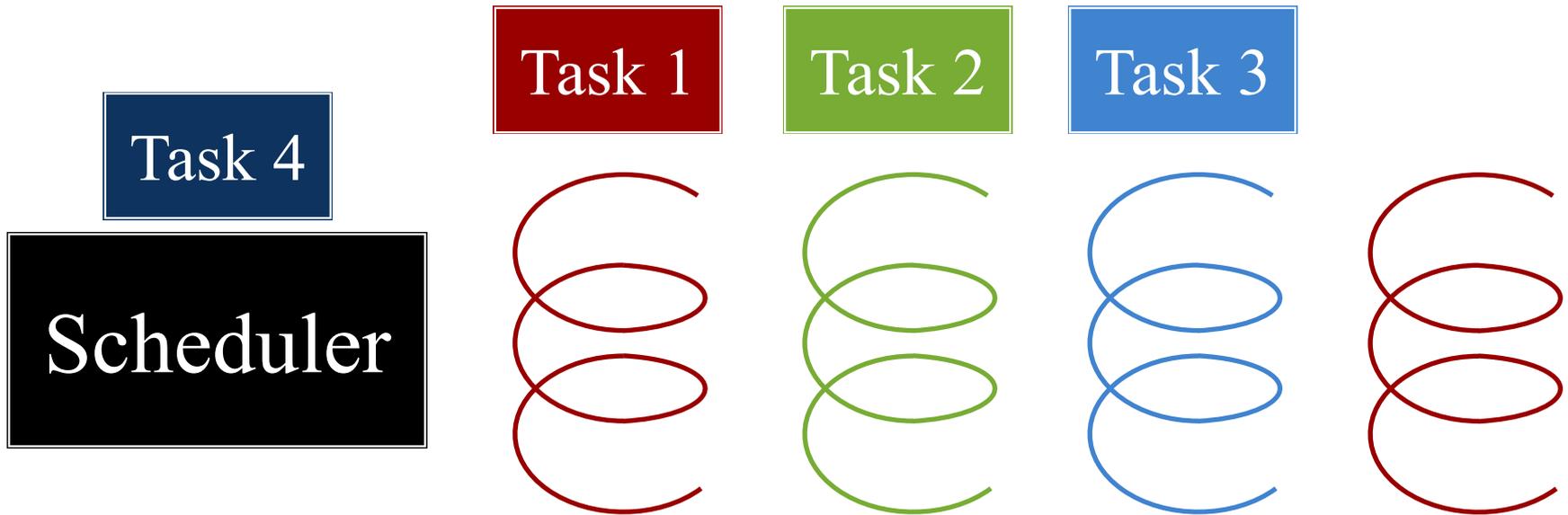
Solution: Thread Pool



Solution: Thread Pool



Solution: Thread Pool



Solution: Thread Pool

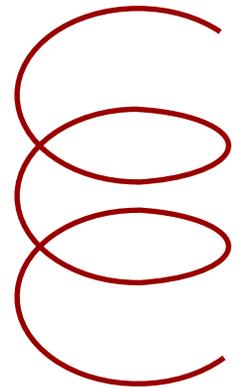
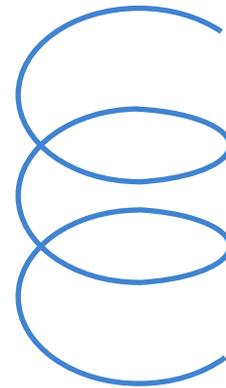
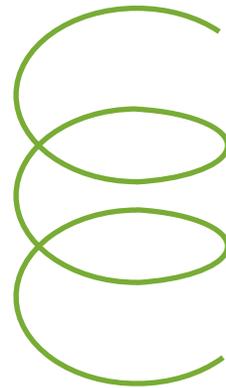
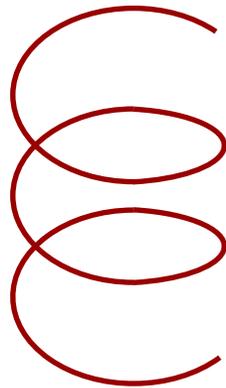
Scheduler

Task 1

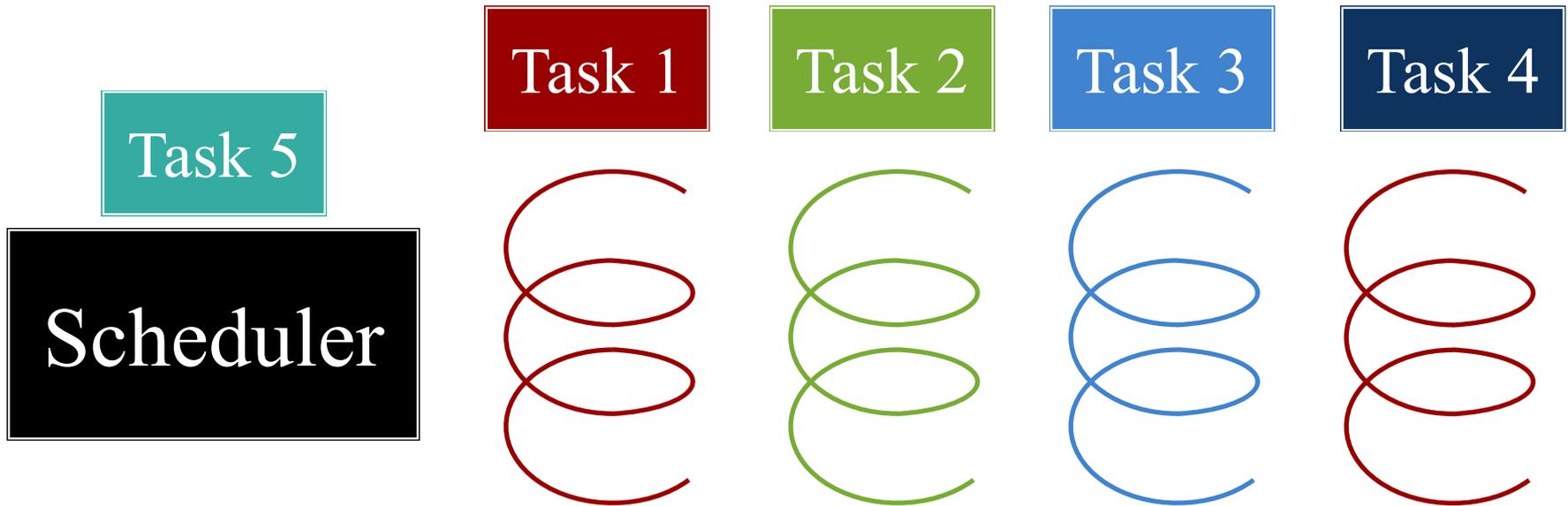
Task 2

Task 3

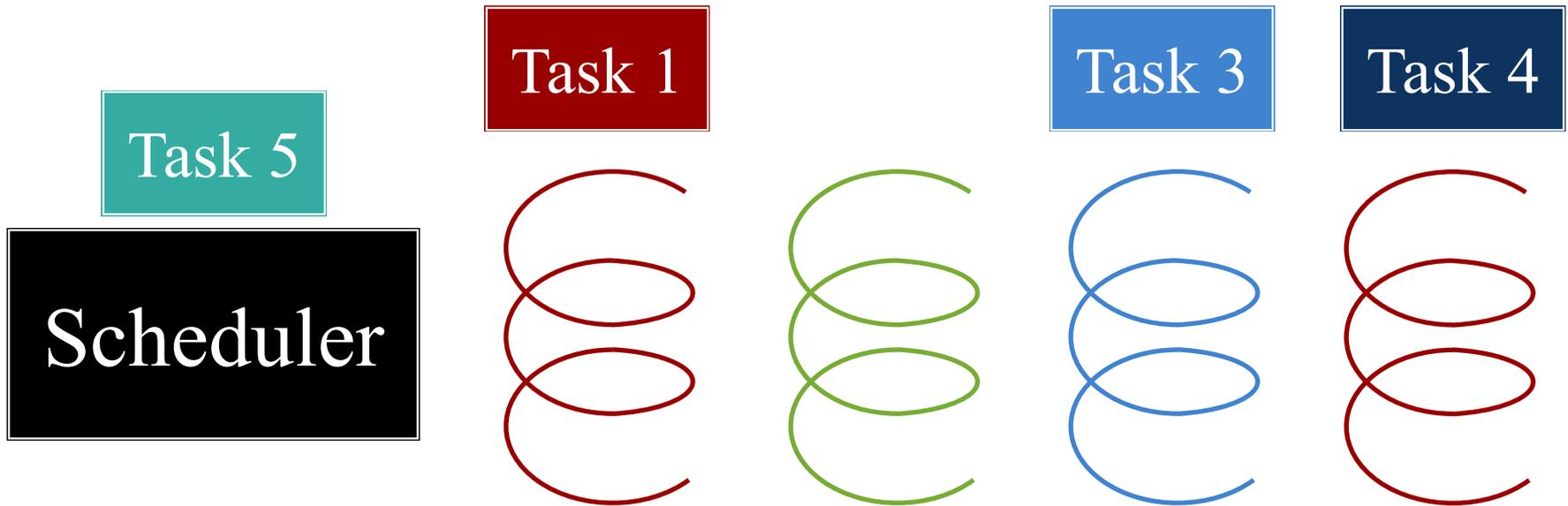
Task 4



Solution: Thread Pool



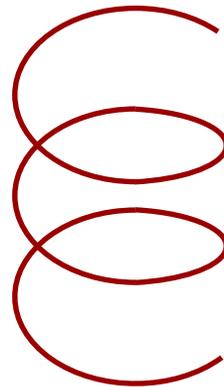
Solution: Thread Pool



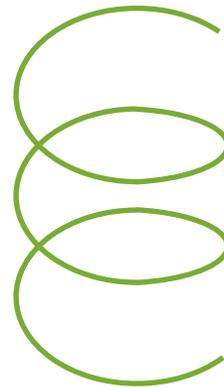
Solution: Thread Pool

Scheduler

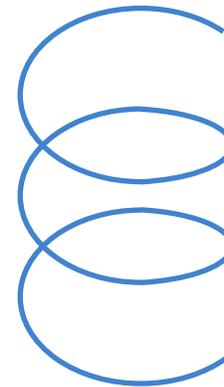
Task 1



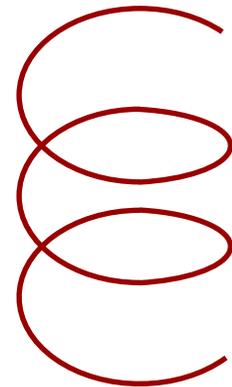
Task 5



Task 3



Task 4



CUGL Support: ThreadPool

- ```
/**
 * Returns a thread pool with the given number of threads.
 *
 * @param threads the number of threads in this pool
 *
 * @return a thread pool with the given number of threads.
 */
static std::shared_ptr<ThreadPool> alloc(int threads = 4)
```

- ```
/**  
 * Adds a task to the thread pool.  
 *  
 * @param task  the function to add to the thread pool  
 */  
void addTask(const std::function<void()> &task)
```

CUGL Support: ThreadPool

- ```
/**
 * Returns a thread pool with the given number of threads.
 *
 * @param threads the number of threads in this pool
 *
 * Returns a thread pool with the given number of threads.
```

AssetManager is a **one** thread pool

- ```
/**  
 * Adds a task to the thread pool.  
 *  
 * @param task the function to add to the thread pool  
 */  
void addTask(const std::function<void()> &task)
```

CUGL Asset Management

AssetManager

- Map from keys to assets
 - All access is templated
 - `assets->get<Texture>("image")`
 - Keys unique *per asset*
- Requires attached loaders
 - `a->attach<T>(load1->getHook());`
 - `a->attach<F>(load2->getHook());`
- “Hook” is C++ workaround
 - For template subclassing
 - Make custom loaders easier

Loader

- void `read(key, src, cb, async)`
 - Reads asset from file `src`
 - `async` indicates if in sep thread
 - Callback `cb` executed when done
- void `read(json, cb, async)`
 - Values `key` and `src` now in `json`
 - As are other special properties
- void `materialize(key, asset, cb)`
 - Code to “finish” asset
 - Always in the main thread

CUGL Asset Management

AssetManager

- Map from keys to assets
 - All access is **Thread Safe**
 - `assets->get<T>(key)`
 - Keys unique
- Requires attaching
 - `a->attach<T>(src, cb, async)`
 - `a->attach<File>(src, cb, async)`
- “Hook” is `ClassLoader`
 - For templating
 - Make custom

**Main Thread
Only**

Loader

- `void read(key, src, cb, async)`
 - Reads asset from file `src`
 - `async` indicates if in sep thread
 - Callback `cb` executed when done
- `void read(json, cb, async)`
 - Values `key` and `src` now in `json`
 - As are other special properties
- `void materialize(key, asset, cb)`
 - Code to “finish” asset
 - Always in the main thread

CUGL Asset Management

AssetManager

- Map from keys to assets
 - All access is templated

Each loader
is its own task

- Make custom loaders easier

Loader

- void `read(key, src, cb, async)`
 - Reads asset from file `src`
 - `async` indicates if in sep thread
 - Callback `cb` executed when done
- void `read(json, cb, async)`
 - Values `key` and `src` now in `json`
 - As are other special properties
- void `materialize(key, asset, cb)`
 - Code to “finish” asset
 - Always in the main thread

Executing Tasks on the Main Thread

- Any other thread can access the `Application`
 - Use the static method `Application::get()`
 - This class is essentially a singleton
- That object has a `schedule` method
 - Works much like `addTask` in thread pool
 - But executes that task on the main thread
 - Executed just before the call to your `update`
- **Scheduling** this task is thread safe

The Schedule Method

```
/**  
 * Schedules a task function on the main thread.  
 *  
 * @param cb      The task callback function  
 * @param ms      The number of milliseconds to delay  
 *  
 * @return a unique identifier for the task  
 */  
Uint32 schedule(std::function<bool()> cb, Uint32 ms)
```

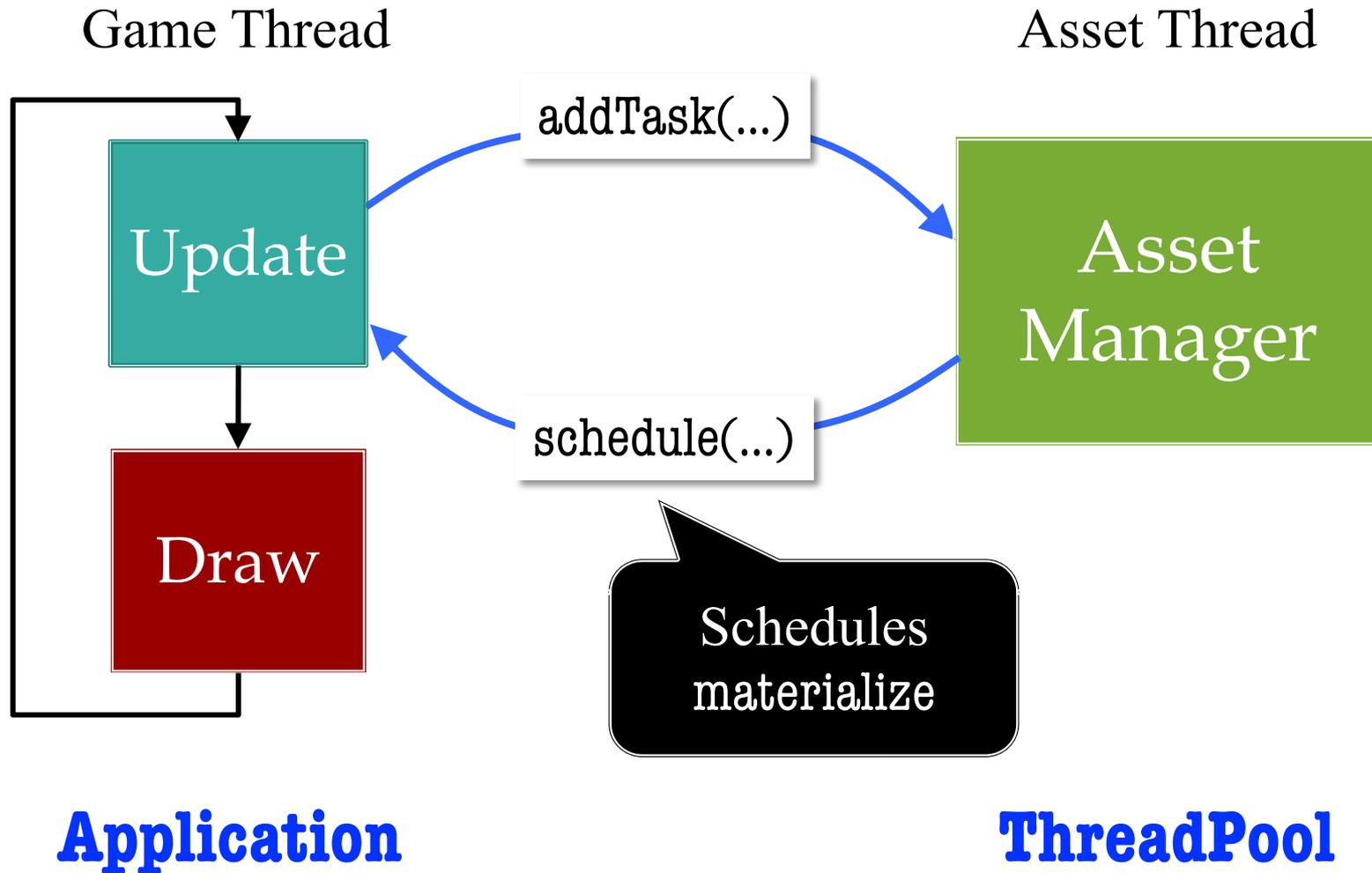
The Schedule Method

```
/**  
 * Schedules a task function on the main thread.  
 *  
 * @param cb The task callback function  
 * @param ms The time in milliseconds  
 *  
 * @return a unique task ID  
 */  
uint32 schedule(std::function<bool()> cb, uint32 ms)
```

Return false to stop execution

Picks first frame after ms millisec

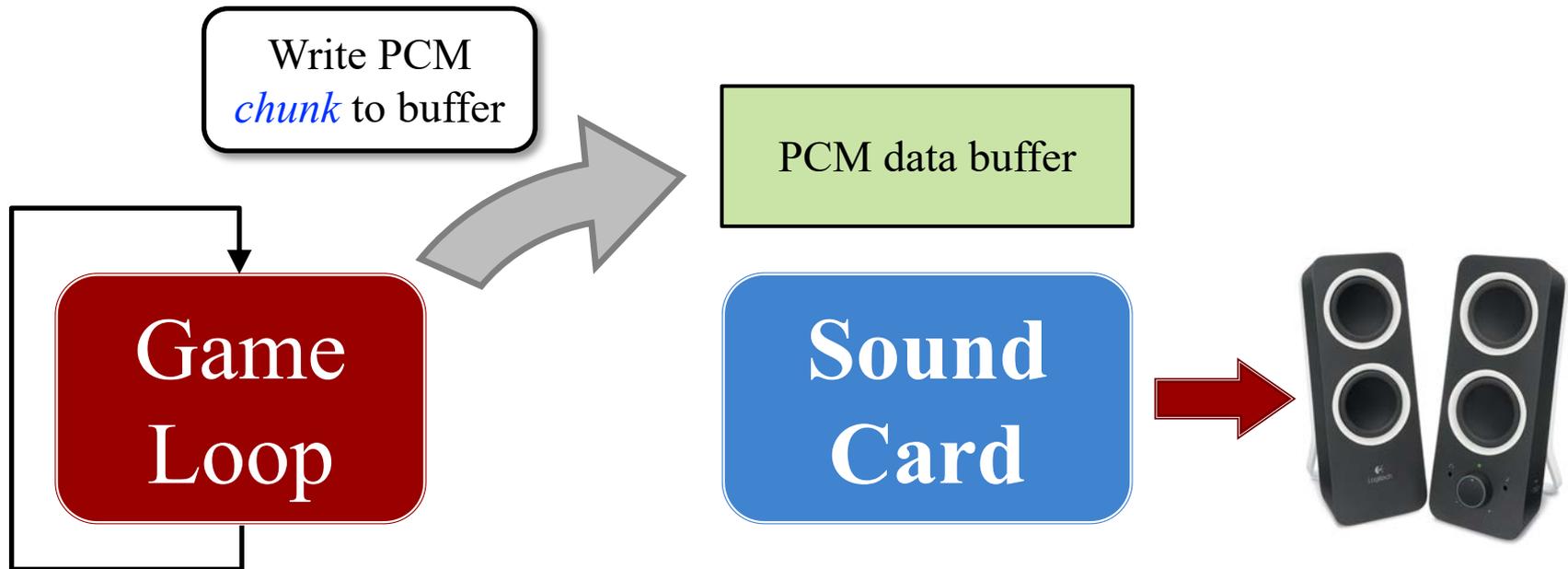
Putting it All Together



Aside: Schedule is Useful in General

- Can specify an event to **run in the future**
 - This is the purpose of the milliseconds
 - May be easier than tracking a timer yourself
- Can specify a task to **run periodically**
 - **Example:** Spawning enemies
 - The task returns true if it wants to run again
 - Same delay is applied as the first time
 - Alternate schedule separates **delay** and **period**

Recall: Playing Sound Directly



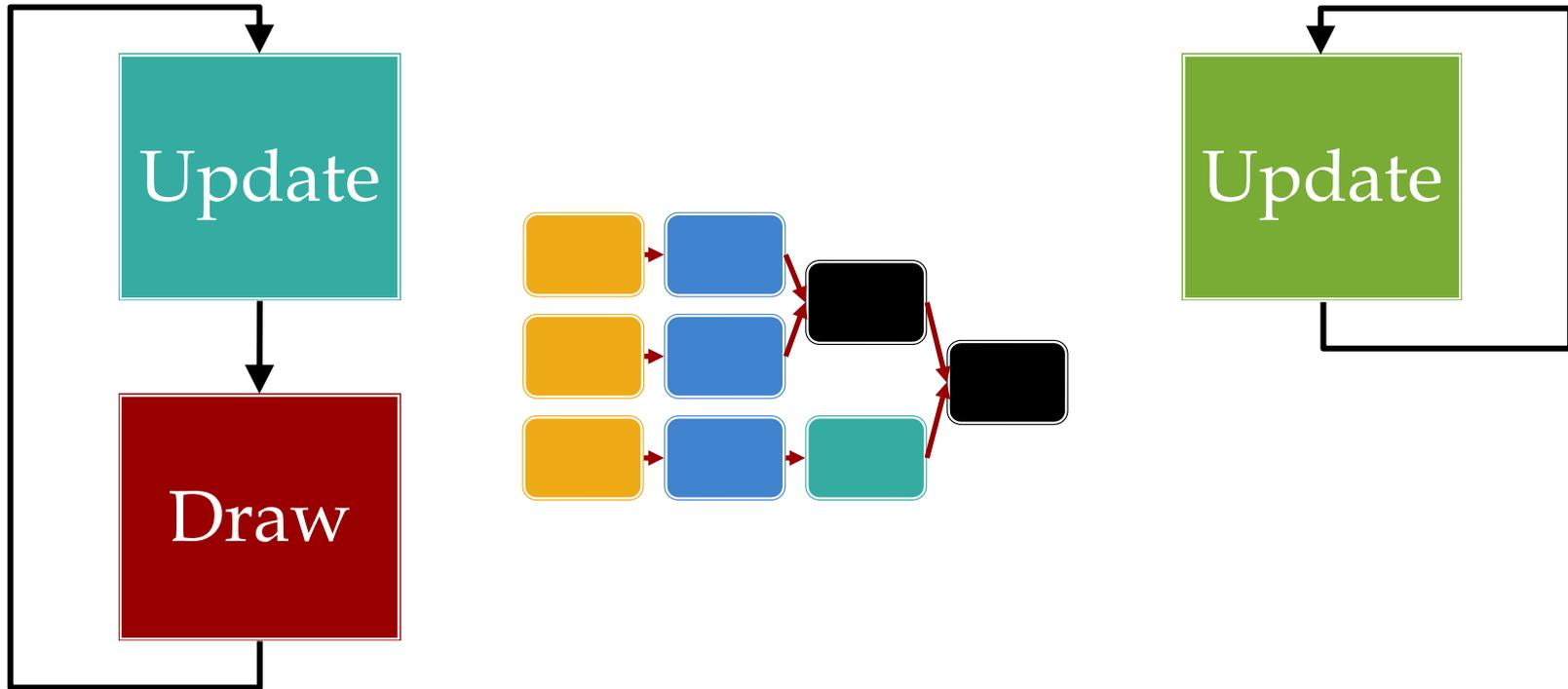
Missing a write causes pops/clicks

The CUGL Approach

Game Thread

DSP Graph

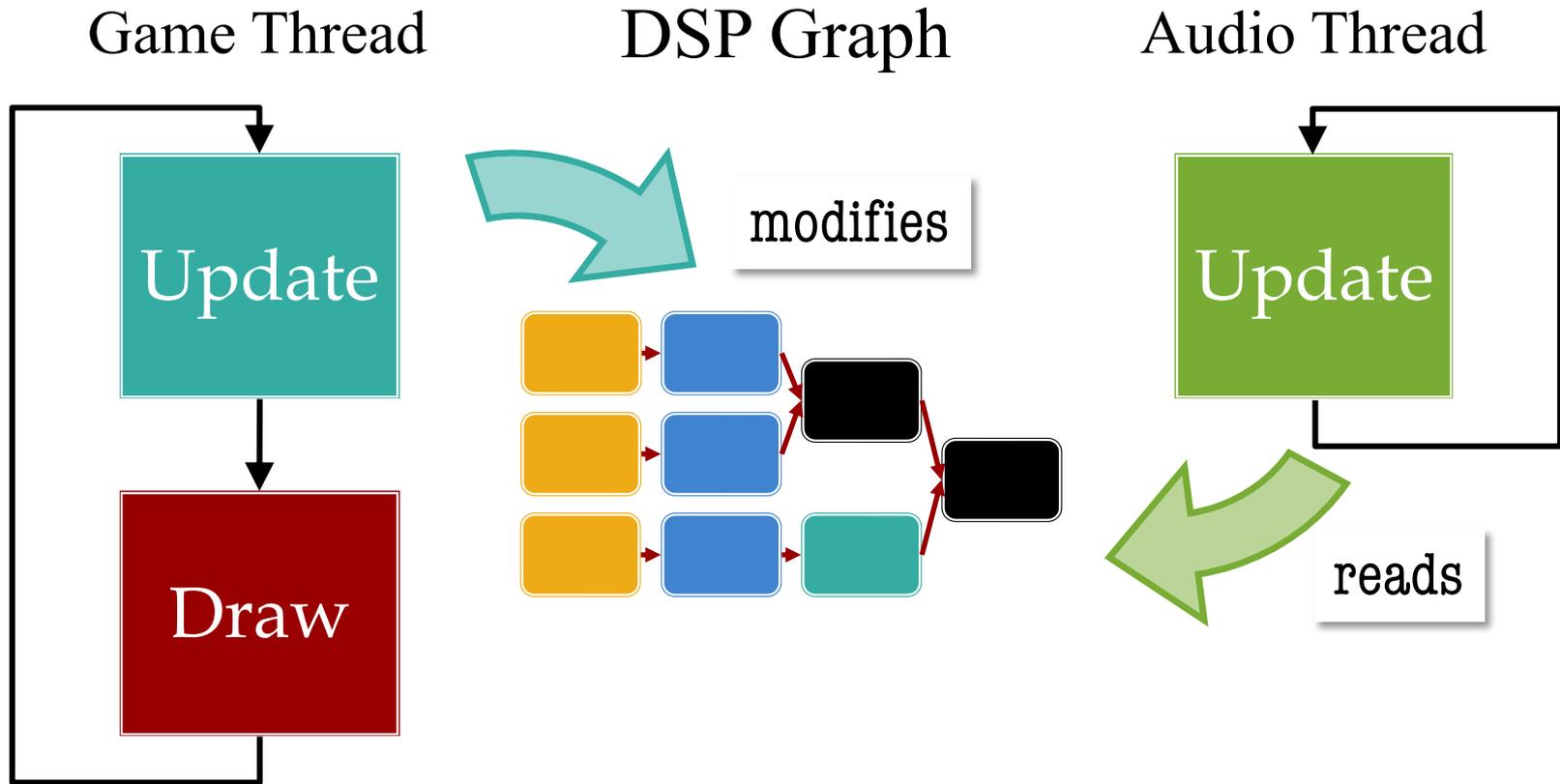
Audio Thread



Application

Thread

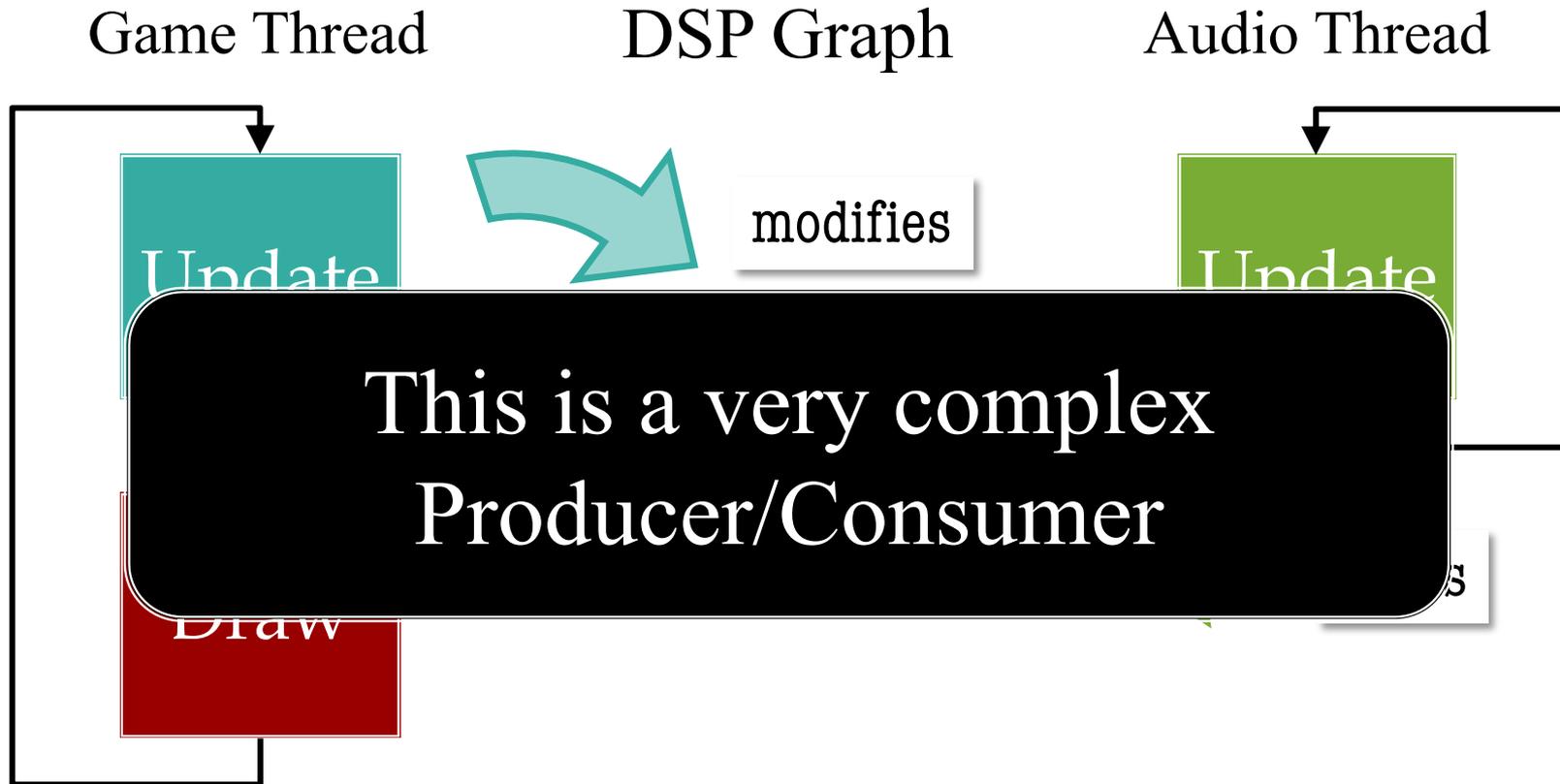
The CUGL Approach



Application

Thread

The CUGL Approach



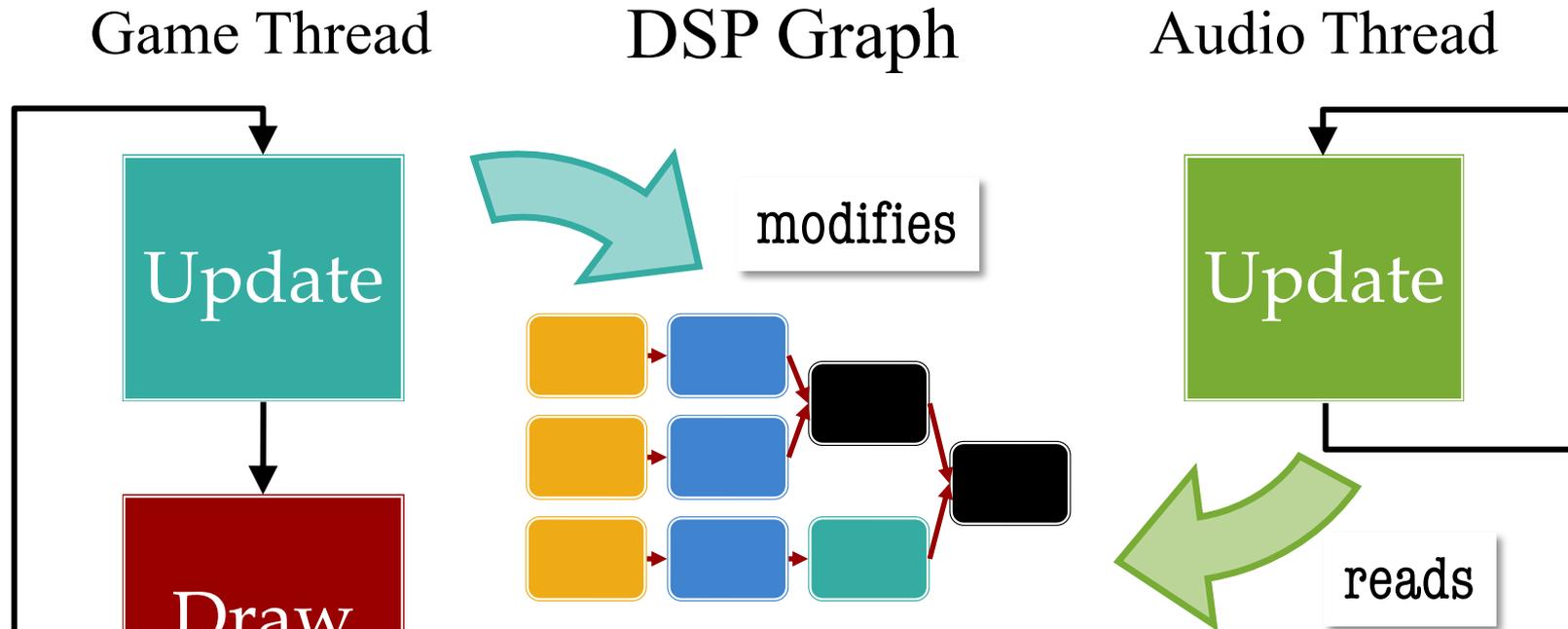
Application

Thread

Aside: Audio is Not a ThreadPool

- Audio is a dedicated `std::thread`
 - Because it needs to run as long as the game does
 - Started when you initialized AudioEngine
- But process is similar to ThreadPool
 - Package your task as a `std::function<void()>`
 - Pass this when you create the thread object
- Difference is that task is in a loop
 - Has an attribute called `running` to manage loop
 - When you set to `false`, the thread is done

The CUGL Approach

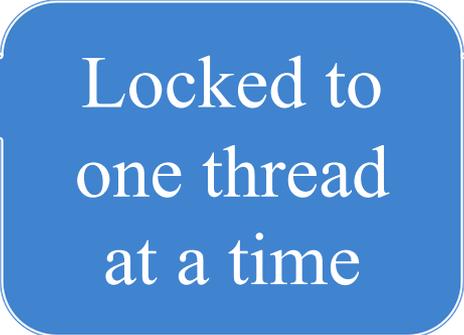


How do we protect the critical section?

AI

The Java Approach: Synchronized

```
public class CriticalSection {  
    synchronized void method1() {...}  
    synchronized void method2() {...}  
    synchronized void method3() {...}  
}
```



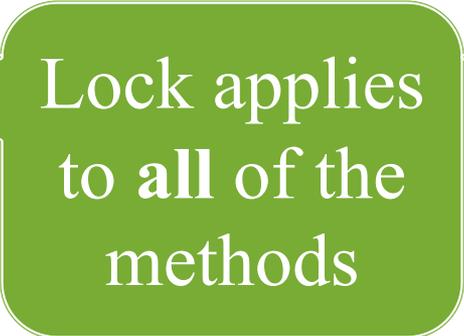
Locked to
one thread
at a time

The Java Approach: Synchronized

```
public class CriticalSection {  
    synchronized void method1() {...}  
    synchronized void method2() {...}  
    synchronized void method3() {...}  
}
```



Locked to
one thread
at a time



Lock applies
to **all** of the
methods

C++ Actually Has Two Tools

`std::mutex`

- Used to protect a **code block**
 - Places lock on code block
 - Only one thread can access
- **Advantages**
 - Can replicate synchronized
 - Relatively easy to use
- **Disadvantages**
 - Locking has some cost
 - Deadlocks easy if careless

`std::atomic`

- Used to protect a **variable**
 - Prevents data races
 - Useful for shared setters
- **Advantages**
 - 10x faster than `std::mutex`
 - *Sometimes* easy to use
- **Disadvantages**
 - Extremely limited in use
 - Advanced use is **advanced**

C++ Actually Has Two Tools

std::mutex

- Used to protect a **code block**
 - Places lock on code block
 - Only one thread can access
- **Advised** Audio thread uses only when it must do so
 - **Advised** Audio thread uses only when it must do so
- **Disadvantages**
 - Locking has some cost
 - Deadlocks easy if careless

std::atomic

- Used to protect a **variable**
 - Prevents data races
 - Useful for shared setters
- **Advised** Audio thread uses whenever it is possible
 - **Advised** Audio thread uses whenever it is possible
- **Disadvantages**
 - Extremely limited in use
 - Advanced use is **advanced**

Replicating Synchronized

```
class CriticalSection {
private:
    /** Mutex to synchronize methods */
    std::mutex _mutex;

public:
    void method() {
        _mutex.lock();    // Lock method code
        ...
        _mutex.unlock(); // Release when done
    }
}
```

Observations About `std::mutex`

- It is **not** a **reentrant lock** (unlike `synchronized`)
 - Locking it again inside same class will deadlock
 - This matters when you have locks on helpers
 - Must use `std::recursive_mutex` for reentrant lock
- Manual lock/unlock calls are **frowned upon**
 - Too easy to forget to unlock and deadlock
 - Preferred way is to attach a **locking object**
 - When locking object is deleted, so is lock

Using a Locking Object

```
class CriticalSection {
private:
    /** Mutex to synchronize methods */
    std::mutex _mutex;

public:
    void method() {
        std::lock_guard<std::mutex> lock(_mutex);
        ...
        // Mutex unlocked once lock variable deleted
    }
}
```

What If Critical Section is a Variable?

- **Example:** `running` attribute controlling thread
 - Audio thread loops so long as it is `true`
 - Setting it to `false` stops the audio
- Mutexes exist to prevent **inconsistent states**
 - Either all code is executed, or none is
 - Cannot happen to variable assignment, right?
- C++ is not **assembly code!**
 - A single assignment is multiple lines of assembly
 - This is not thread safe (*especially* on Windows)

What If Critical Section is a Variable?

- **Example:** `running` attribute controlling thread
 - Audio thread loops so long as it is `true`
 - Setting it to `false` stops the audio
- Mutexes **do not prevent inconsistent states**
 - Either all **This leads to data races!**
 - Cannot lock right?
- C++ is not **assembly code!**
 - A single assignment is multiple lines of assembly
 - This is not thread safe (*especially* on Windows)

std::atomic Protects Assignment

- **Template** around a type: `std::atomic<int>`
 - Supports all primitive C++ types
 - Cannot apply to objects in general, but ...
 - Is possible to make `std::shared_ptr` atomic
- Supported by two methods
 - `load()`: An atomic **getter** for the value
 - `store(value)`: An atomic **setter** for the value
 - Shared pointers are slightly more complicated

std::atomic Protects Assignment

- **Template** around a type: `std::atomic<int>`
 - Supports all primitive C++ types
 - Cannot apply to objects in general, but ...
 - Is possible to make `std::shared_ptr` atomic
- Supported by two methods
 - `load()`: An atomic **getter**
 - `store(value)`: An atomic **setter**
 - Shared pointers are slightly more complicated

Means assignment is atomic, not methods

Only Use If Read/Write Are Separate

```
class WithAtomics {
private:
    std::atomic<int> _xvar; // Atomic integer
public:
    /** Change the value of X */
    void writeX(int val) { _xvar.store(val); }

    /** Use the value of X to compute something */
    void readX() {
        int x = _xvar.load(); // Copy value to local variable
        // Use x in local computation
    }
}
```

Only Use If Read/Write Are Separate

```
class WithAtomics {
private:
    std::atomic<int> _xvar; // Atomic integer
public:
    /** Change the value of X */
    void writeX(int val) { _xvar.store(val); }

    /** Use the value of X to compute something */
    void readX() {
        int x = _xvar.load(); // Copy value to local variable
        // Use x in local code
    }
}
```

**Never store `_xvar`
in same method**

This Is Only Scratching the Surface

- C++ supports **monitors** and **semaphores**
 - These are used for producer/consumer problem
 - Monitor allows consumer to **wait** on producer
- C++ supports **promises**
 - These are threads that return a value
 - Simplify critical section in that case
- Atomics support **memory orders**
 - These are used to **optimize performance**
 - Best avoided unless you know what you are doing

This Is Only Scratching the Surface

- C++ supports **monitors** and **semaphores**
 - These are used for producer/consumer problem
 - Monitor allows consumer to **wait** on producer
- C++ s
 - These
 - Simp
- Atomics support **memory orders**
 - These are used to **optimize performance**
 - Best avoided unless you know what you are doing

See readings if want more

So Why Do We Care?

- All of these threads are made for you!
- But how about making **your own threads**?
 - **Pathfinding** is a classic example
 - **NPC behavior** can also be long-running
- How can **extreme** can we go?
 - What if all **updates** are in separate thread?
 - Then the main thread just **draws**!
 - This can give us potentially very high FPS

This Will Not Quite Work



Frame 1



Frame 2



Frame 3

Without update, redraw same image.
We need animation in the core loop.

Recall: Two Approaches to Animation

Tweening

- Animates **timed actions**
 - Given a duration and a start
 - Interpolates scene over time
- Render thread simply...
 - accesses all active actions
 - moves them forward by dt
- Gameplay **creates** actions
 - Happens less frequently
 - Decoupled from render

Physics

- Animates **physical objects**
 - Bodies with force and mass
 - Also shape for collisions
- Render thread simply...
 - steps simulation forward
 - renders objects at end
- Gameplay **nudges** objects
 - *Might* be less frequent
 - If so, can also decouple

Recall: Two Approaches to Animation

Tweening

- Animates **timed actions**
 - Given a duration and a start
 - Interpolates scene over time

- Render
- a
- r

Like networking, animation uses *dead reckoning* when missing input

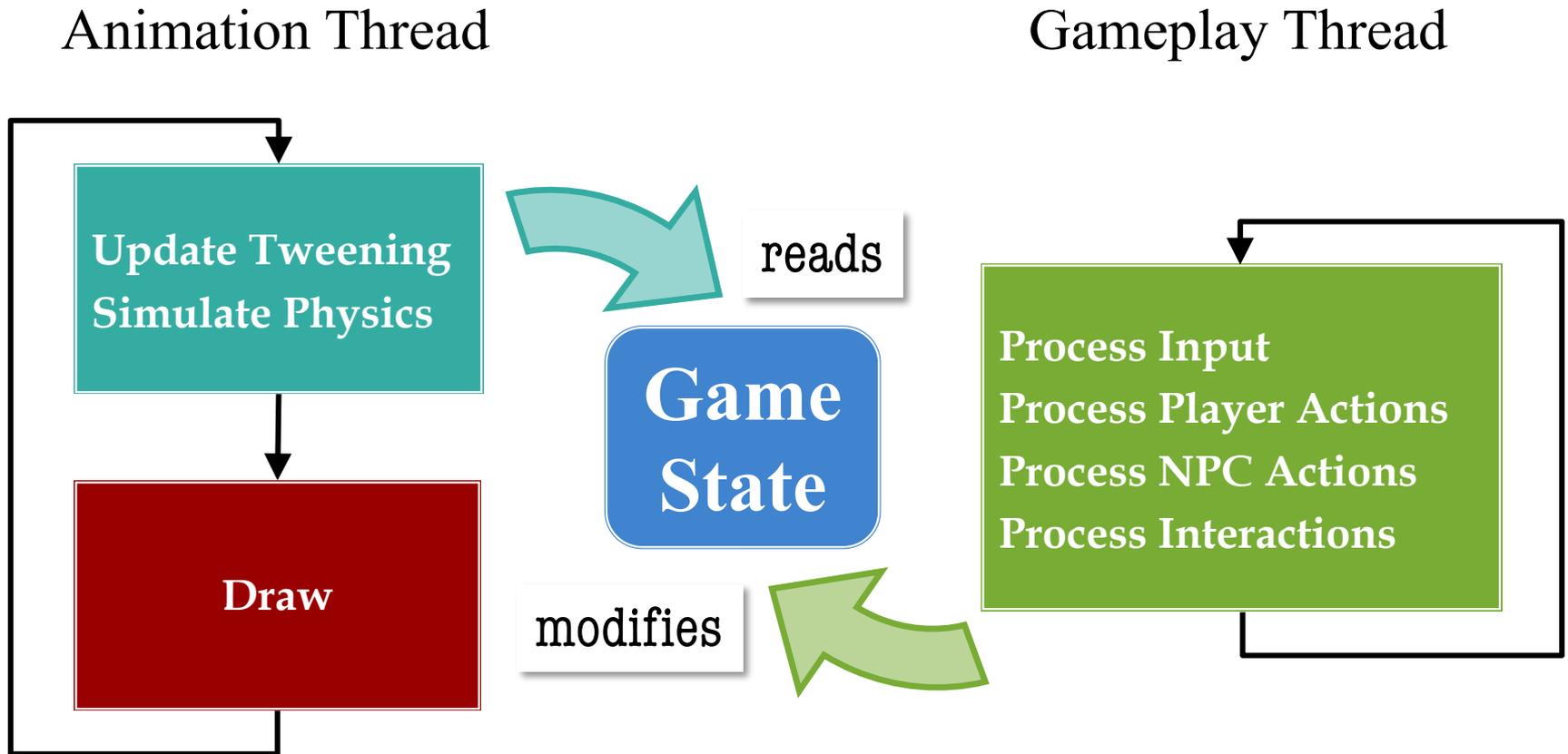
- Gameplay **creates** actions
 - Happens less frequently
 - Decoupled from render

Physics

- Animates **physical objects**
 - Bodies with force and mass
 - Also shape for collisions

- Gameplay **nudges** objects
 - *Might* be less frequent
 - If so, can also decouple

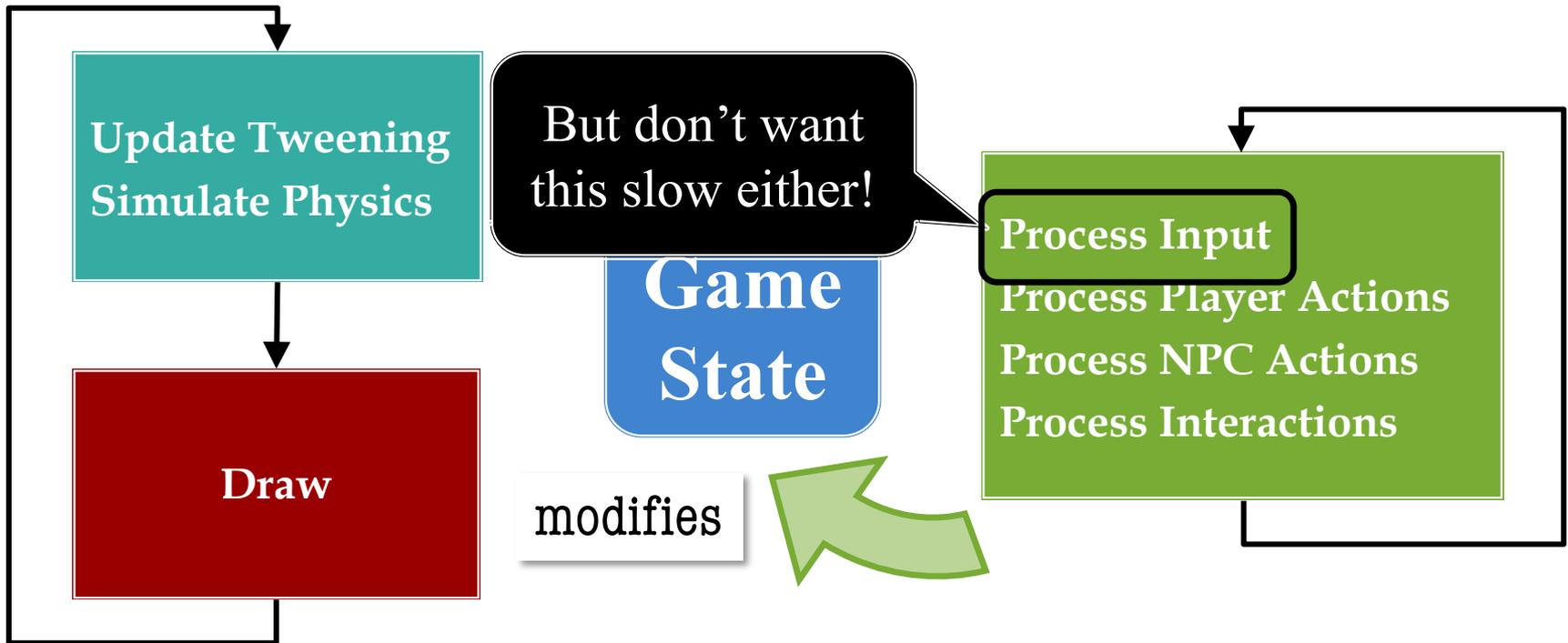
A New Architecture



A New Architecture

Animation Thread

Gameplay Thread



Summary

- Games engines are naturally multithreaded
 - Offload tasks that *block* drawing (**asset loading**)
 - Offload tasks that *slow* drawing (**pathfinding**)
 - Execute tasks *decoupled* from drawing (**audio**)
- CUGL has native **task-based parallelism**
 - `ThreadPool` for tasks off the main thread
 - `Application::schedule` for tasks on main thread
- C++ has general-purpose tools for parallelism
 - `std::thread` class for managing other threads
 - `std::mutex` and `std::atomic` for critical sections