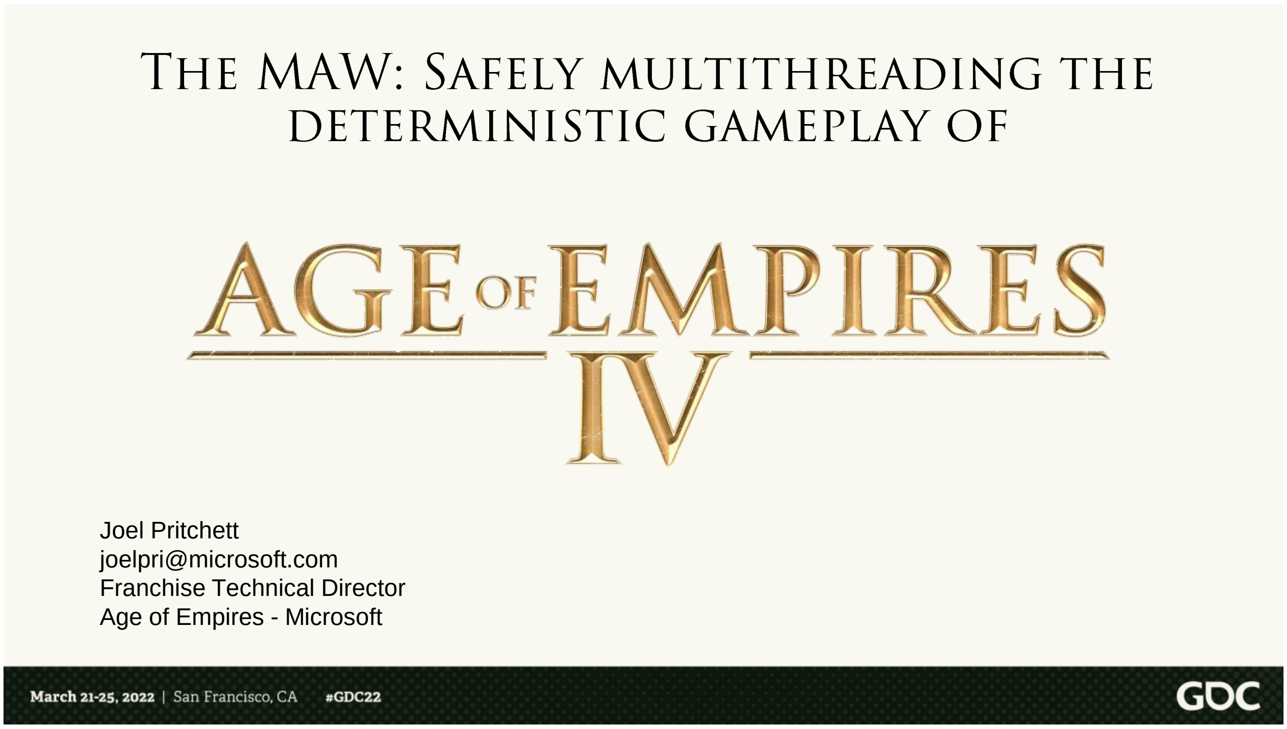
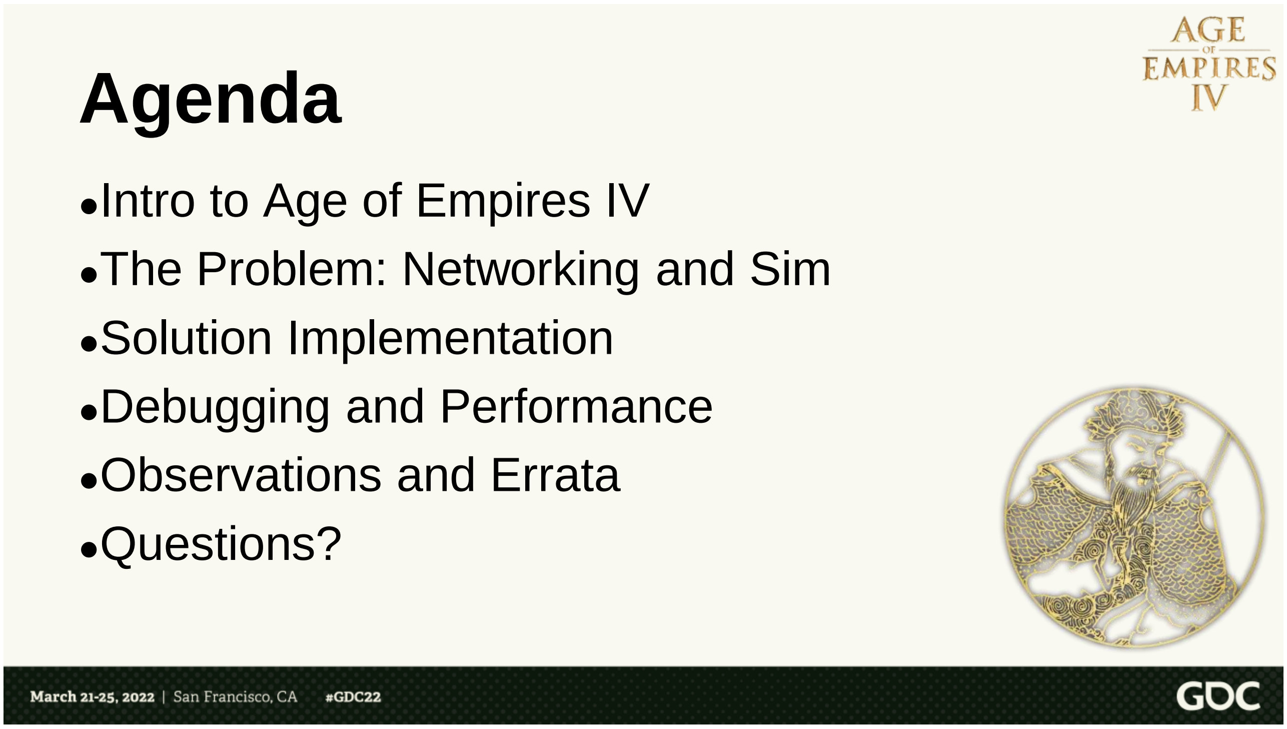


THE MAW: SAFELY MULTITHREADING THE DETERMINISTIC GAMEPLAY OF AGE OF EMPIRES IV

Joel Pritchett
joelpri@microsoft.com
Franchise Technical Director
Age of Empires - Microsoft

Hello everyone, and good afternoon. Before we get started I'd like to remind everyone to please silence your cell phones

My name is Joel Pritchett, I am the tech director for the Age of Empires franchise at Microsoft and today I'm going to talk about tools we built to aid in our engine parallelization efforts for Age IV.



Agenda

- Intro to Age of Empires IV
- The Problem: Networking and Sim
- Solution Implementation
- Debugging and Performance
- Observations and Errata
- Questions?

March 21-25, 2022 | San Francisco, CA #GDC22

In the next 25 minutes I'm going to blitz through a quick intro to what Age of Empires is, incase you hadn't played it.

Then I'll talk about how our simulation can benefit from parallelization, the complications that this causes, specifically with our networking model.

Most importantly I'll cover the tools we built to catch these problems.

I will also share some observations

and learnings from our implementation that you should find helpful if you chose to do something similar.

And we'll have a little time at the end for any questions anyone might have

What is Age of Empires IV?

Every game in the Age of empires franchise has been a historical RTS in which up to 8 players and AI battle it out with hundreds, sometimes over a thousand units. Why parallelize at all? What happens if we didn’t? The most obvious reason, if it wasn’t clear from this video, is that we have a lot of units in our games. Most PCs these days have a lot of cores, and the ability to run our simulation across many of them lets us push more units and give players bigger battles with more spectacle.

The Game

AGE OF EMPIRES IV

March 21-25, 2022 | San Francisco, CA #GDC22

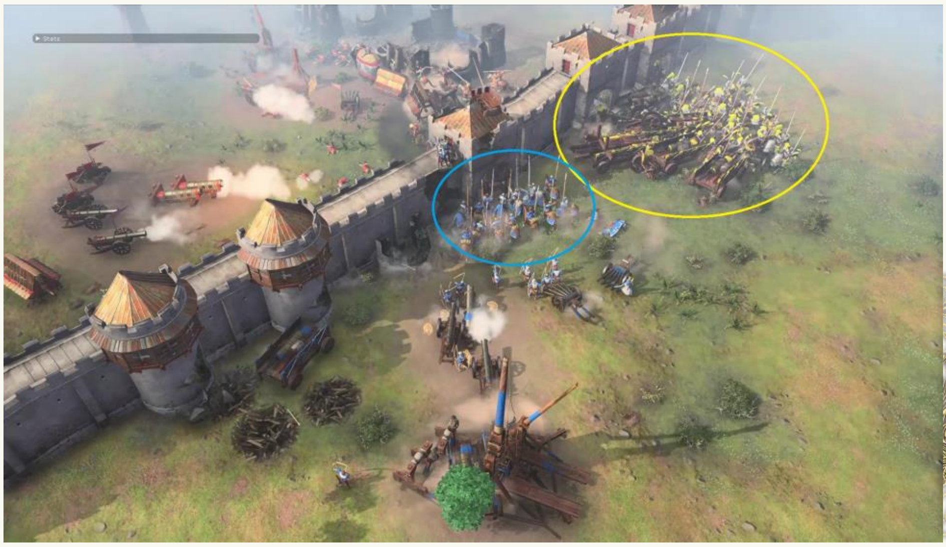
GDC

Our engine goes wide in many places. Some parts of our entity or component updates don’t reach into entities outside themselves, these cases were easy to parallelize. One entity or component per task. Spread the tasks out to all the cores available. Done. We also have some very complicated targeting phases where entities are evaluating all of the entities around them, changing internal state that may impact the decisions of other entities that haven’t updated yet. Maybe its healing, or activating area of effect abilities, or running data driven designer script. Who knows. This phase gets very complicated and was the most expensive part of our update by a long shot. More expensive than pretty much all of the rest of the entity update combined.

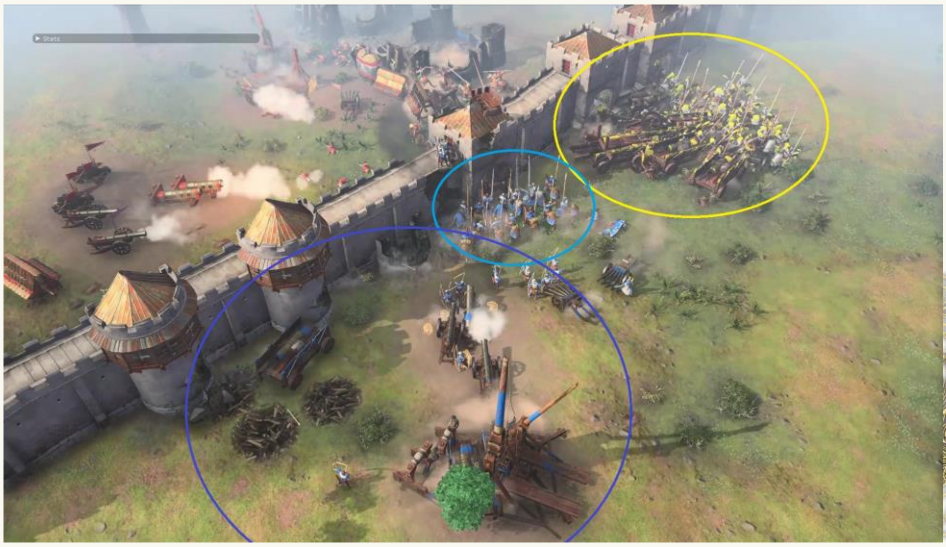
It was a big target for parallelization. The biggest target we had. In a nutshell, we approached the problem by finding ‘islands’, or groups entities who are only looking amongst themselves that we can update in a single task. The next few slides show 5 such islands.

The Game

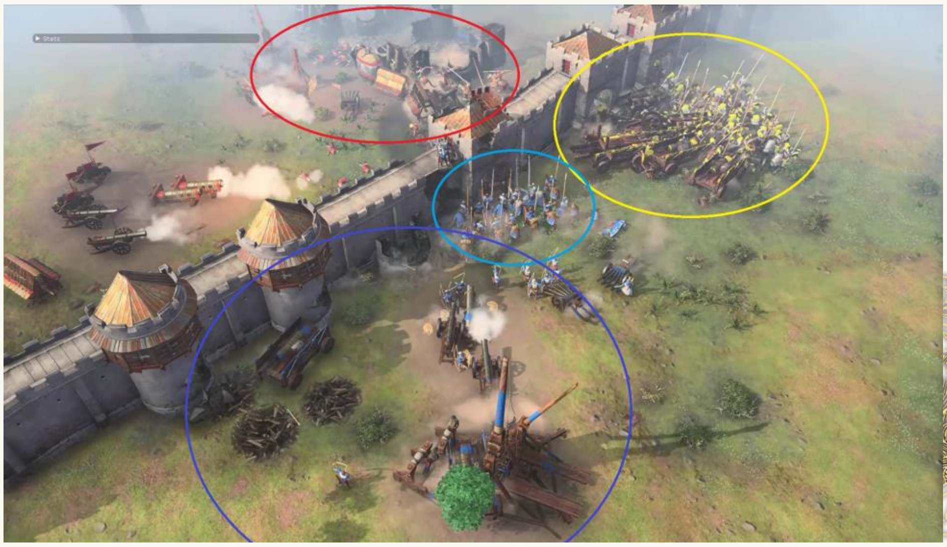
The Game

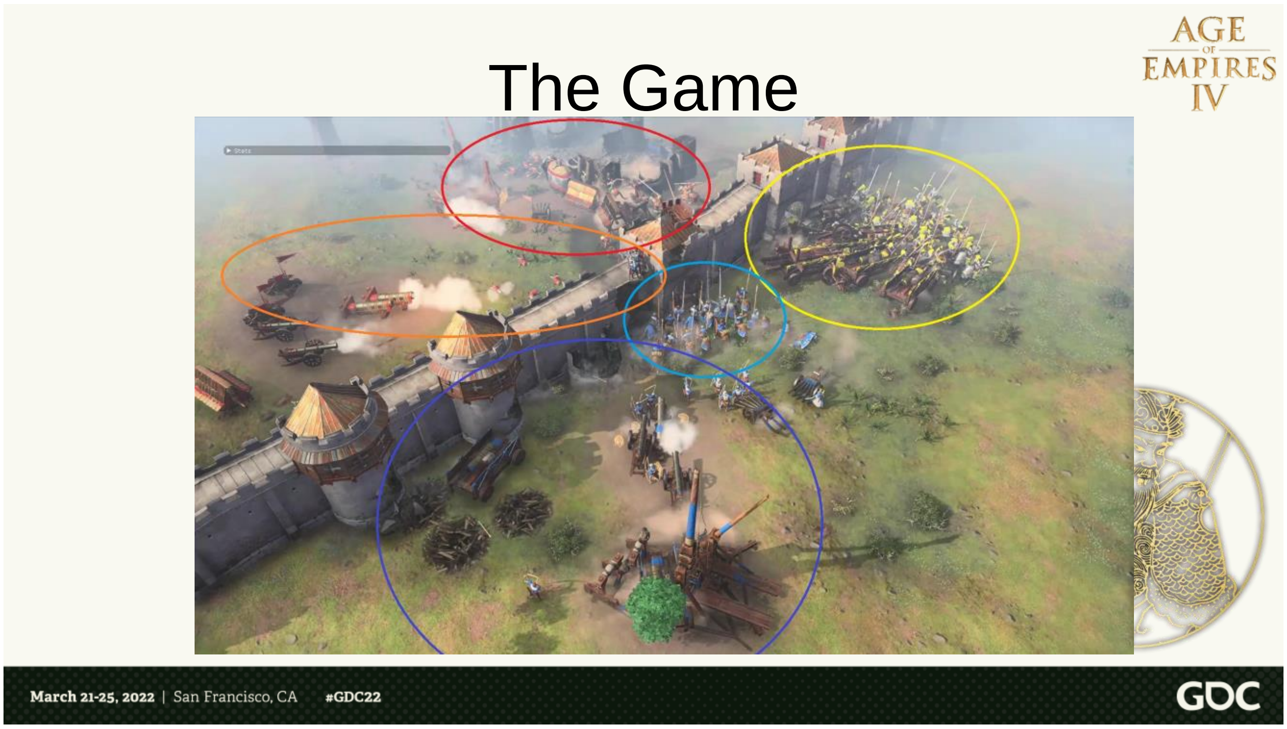


The Game



The Game



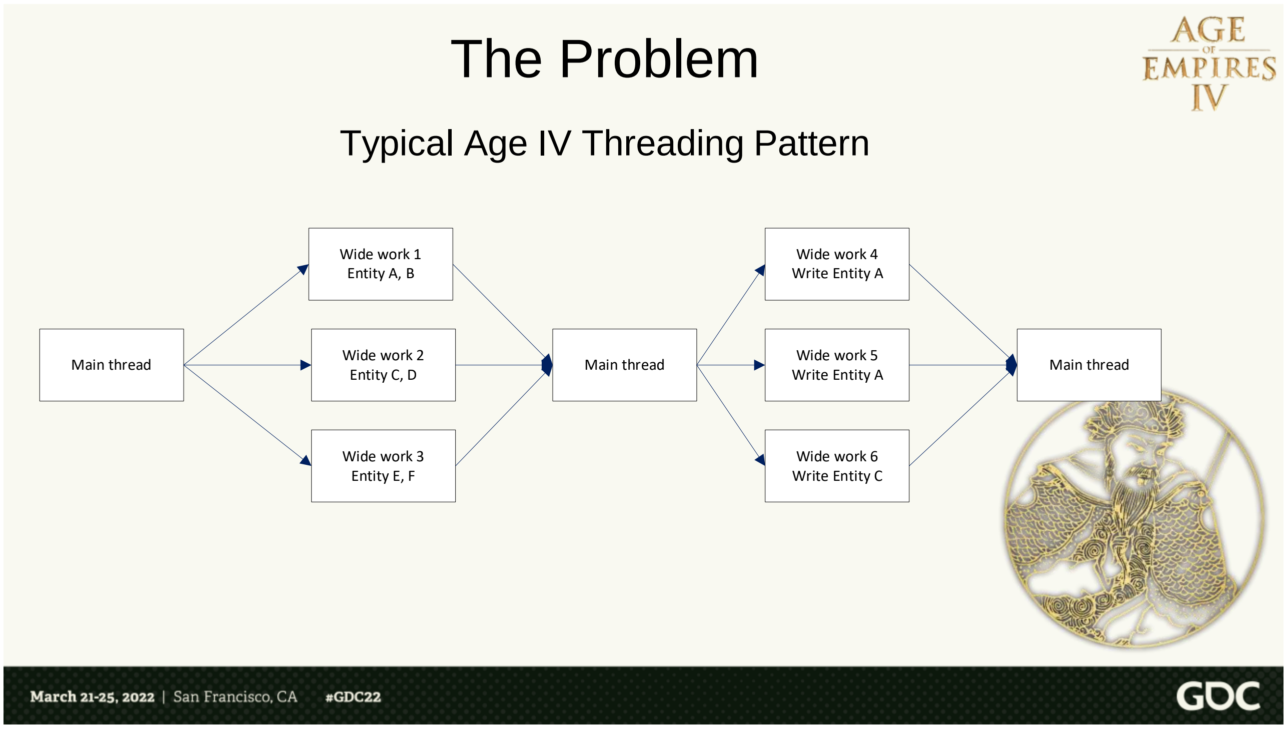


The Game

March 21-25, 2022 | San Francisco, CA #GDC22

Each circle here is a simulation island, an independent group of units where each individual unit, in theory, only is looking at other units in the same island during its the update phases that require modifying unit state. Across the rest of the map, there are probably dozens, if not hundreds, more. Plenty enough islands to spread across the cores of most PCs these days.

Solving selectable set of units by a prepass was also discussed, some extra code each designer/programmer would implement to get actual set of units needed for the island, but this wasn’t a general purpose solution. This is kind of what we did, but we only did the pass from one compoenent. There were dozens other updating during the island update. And the impact of having to implement this pass for all those components was unknown, data driven script behaviors made it even more complicated. On the other hand, MAW is a totally general solution and took ~2 weeks to get up and running plus probably 2 weeks to iterate on the output. It was easier to implement and more reusable.



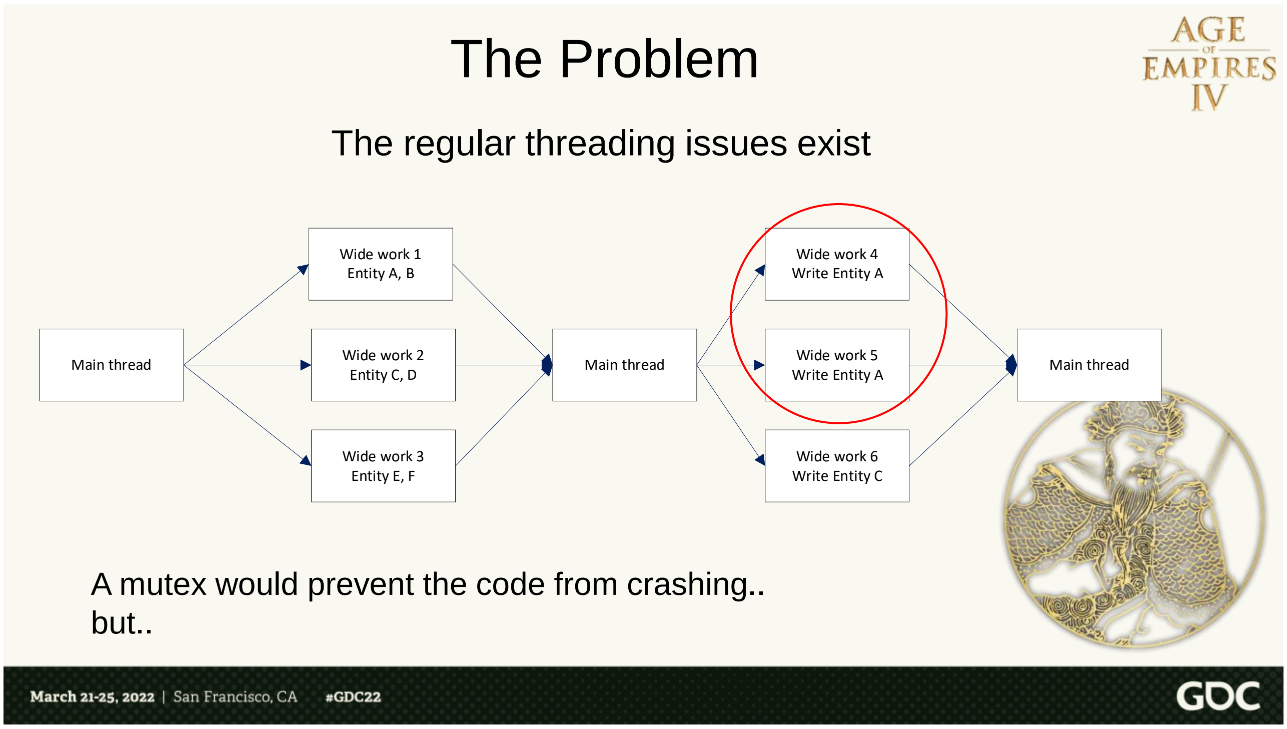
The Problem

Typical Age IV Threading Pattern

This is a simplified view of how our individual update jobs are issued, it is a fork join model. During our simulation update, every time the sim thread got to a parallel phase of its update it would kick off all the tasks at once and go to sleep until they had completed. In the case I just talked about, we'd find a bunch of simulation islands, assign each one to a task and kick it off.

If our island generation was solid and entities only every really looked at entities in the same island we’d be solid. A lot of code across many different components, or even data driven script was about to execute. We couldn’t inspect 100% of the code that was about to run and ensure this was the case in any timely fashion.

we instead looked at what a small set of our components responsible for targeting hoping the entities it cared about would represent a superset of entities the rest of the components cared about. That was not always the case.



The Problem

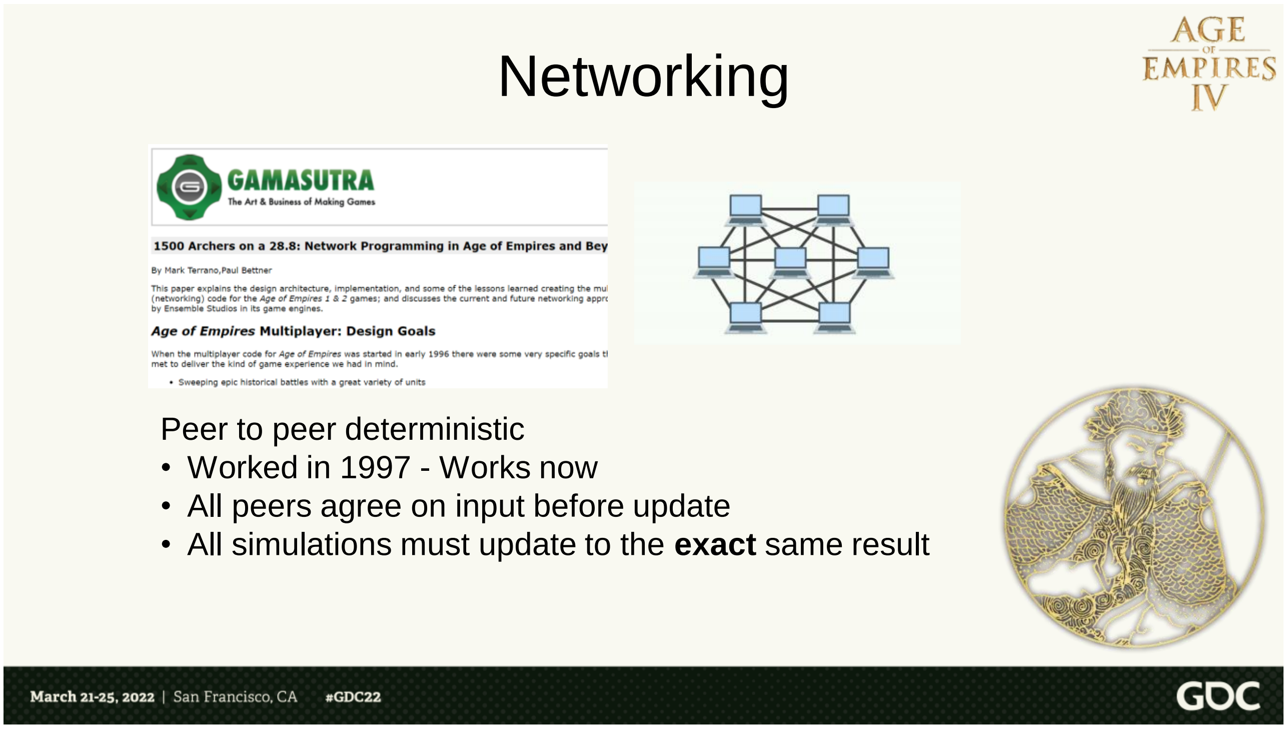
The regular threading issues exist

Main thread
Wide work 1 Entity A, B
Wide work 2 Entity C, D
Wide work 3 Entity E, F
Main thread
Wide work 4 Write Entity A
Wide work 5 Write Entity A
Wide work 6 Write Entity C
Main thread
AGE of EMPIRES IV
March 21-25, 2022 | San Francisco, CA #GDC22
GDC

A mutex would prevent the code from crashing..
but..

If that always happened, we'd never have any issues.

But there is nothing about our engine that inherently prevents any object from reading or modifying any other object and causing well understood issues such as having multiple operations in flight attempting to modify the same object. The team wrote code all the time that broke these rules. Designers built script that broke them. And caused issues like this, just a traditional threading issue where you have two threads trying to modify an object simultaneously. Depending on what that modify operation is doing to the entity, we might cause corrupted data, or maybe even a crash. So we need to fix that.. But we have this other problem we need to deal with



Networking

AGE OF EMPIRES IV

GAMASUTRA
The Art & Business of Making Games

1500 Archers on a 28.8: Network Programming in Age of Empires and Bey

By Mark Terrano,Paul Bettner

This paper explains the design architecture, implementation, and some of the lessons learned creating the mul (networking) code for the *Age of Empires 1 & 2* games; and discusses the current and future networking appro by Ensemble Studios in its game engines.

***Age of Empires* Multiplayer: Design Goals**

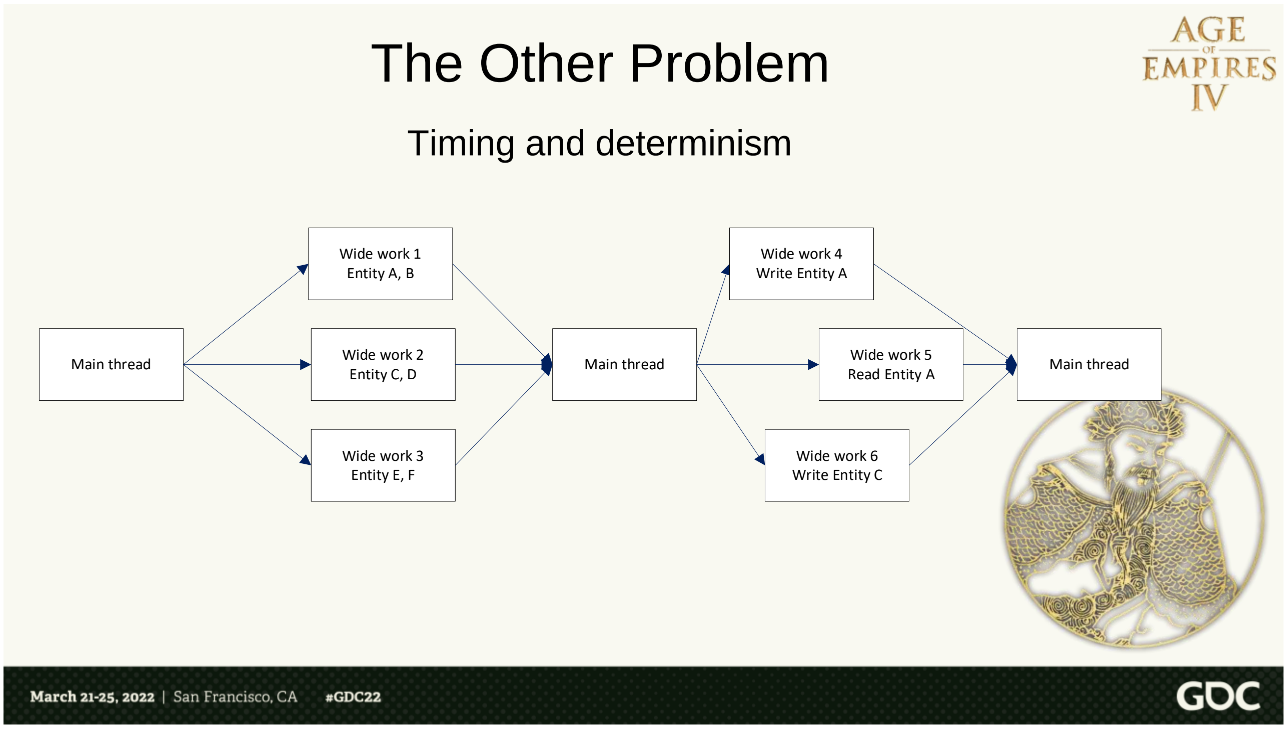
When the multiplayer code for *Age of Empires* was started in early 1996 there were some very specific goals t met to deliver the kind of game experience we had in mind.

- Sweeping epic historical battles with a great variety of units

Peer to peer deterministic
- Worked in 1997 - Works now
- All peers agree on input before update
- All simulations must update to the **exact** same result

March 21-25, 2022 | San Francisco, CA #GDC22 GDC

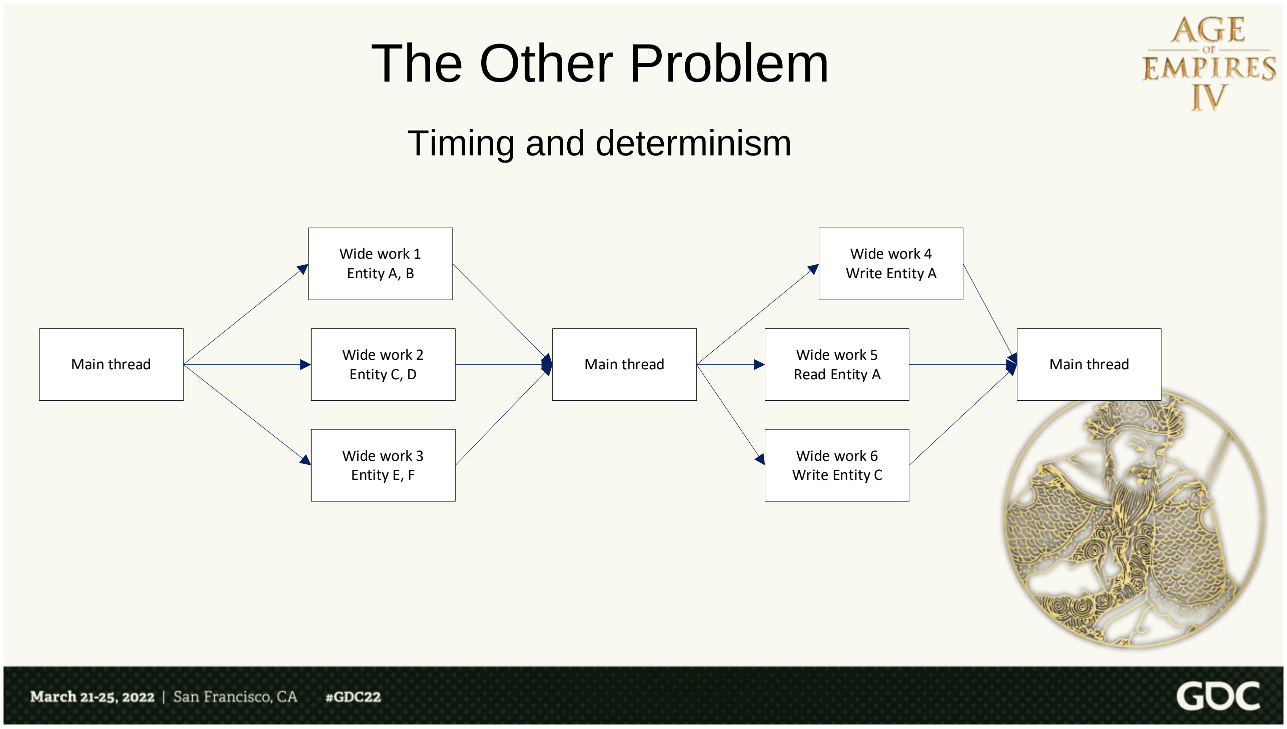
Since the beginning of the Age franchise, and well RTSs probably in general, most RTS titles have relied on peer to peer deterministic networking. What does that mean? It means that each peer starts each simulation tick in the exact same state. All our networking does is share the input from each player before starting to update the simulation. P2P was chosen because it requires minimal bandwidth and the number of entities in the simulation doesn’t matter. Parallelism introduces some interesting problems though.



The Other Problem

Timing and determinism

where parallelization creates an additional problem for our sim in that timing of jobs executing across multiple cores is variable as hypothetically show in the second fork here. Tasks 4, 5, and 6 are kicked off at the same time as part of a group of work, but for whatever reason 4 might run well before 5 on some machines.

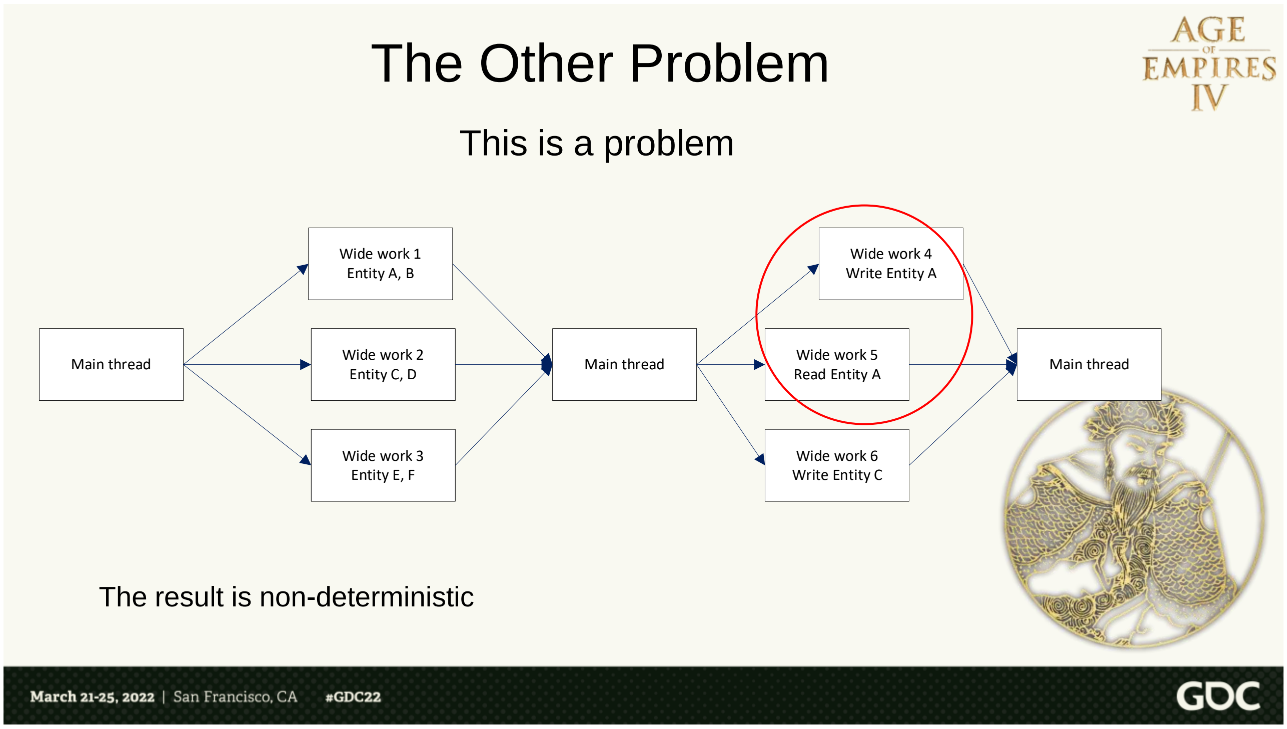


The Other Problem

Timing and determinism

March 21-25, 2022 | San Francisco, CA #GDC22

And due to random uncontrollable circumstances, Task 5 executes before Task 4 on other machines and in this case, 5 will get a value from entity A before 4 modifies it. (don’t mention mutex. Next slide)

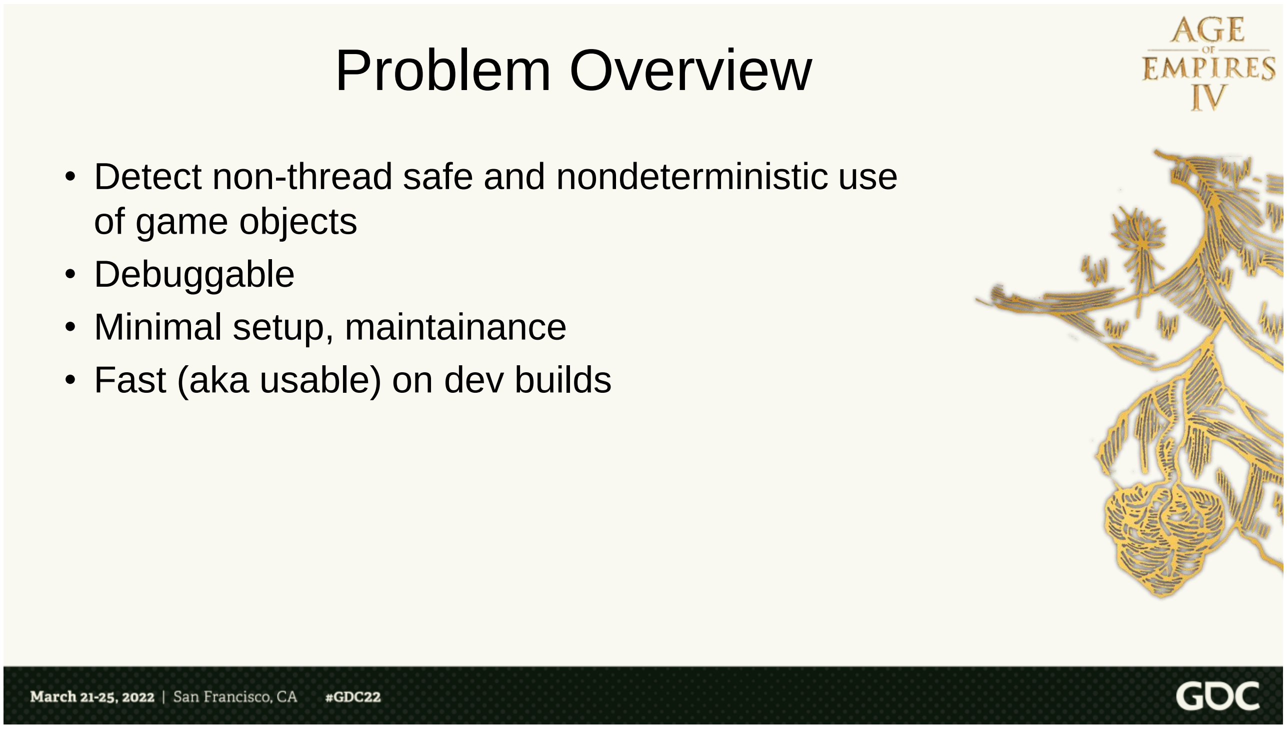


The Other Problem

This is a problem

The result is non-deterministic

this would still cause a non deterministic outcome, a desync. It breaks our networking model. We realized that if we could build a system that could catch the nondeterministic use of an entity shown in these example, it would also guarantee that there won’t be contention for that object across our multiple worker threads. We could get rid of mutexes or locking and solve our determinism at the same time.



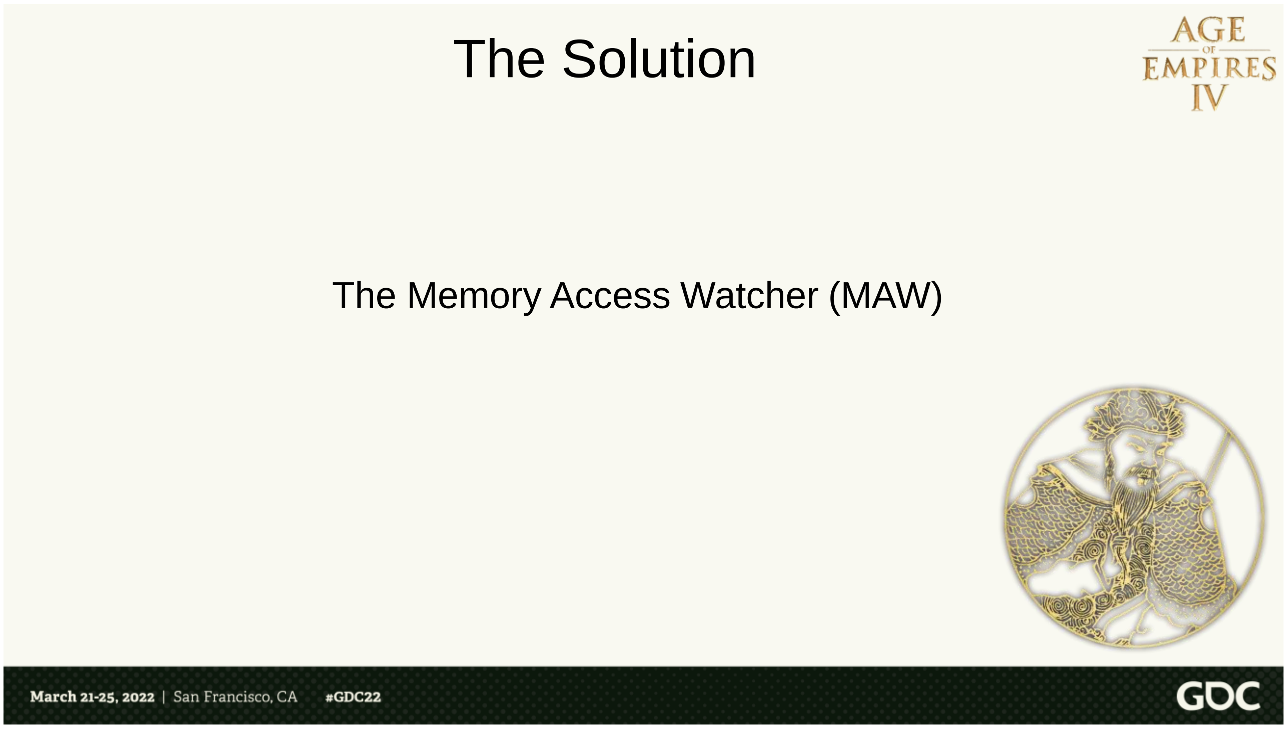
Problem Overview

- Detect non-thread safe and nondeterministic use of game objects
- Debuggable
- Minimal setup, maintainance
- Fast (aka usable) on dev builds

AGE OF EMPIRES IV
March 21-25, 2022 | San Francisco, CA #GDC22
GDC

We set out to design a system that could detect both nonthread safe and nondeterministic use of our game objects.

Our other goals in priority order were obviously to get useful debug information out of any issues the system detected, ease of setup and lastly something fast enough for heavy use by the dev team.

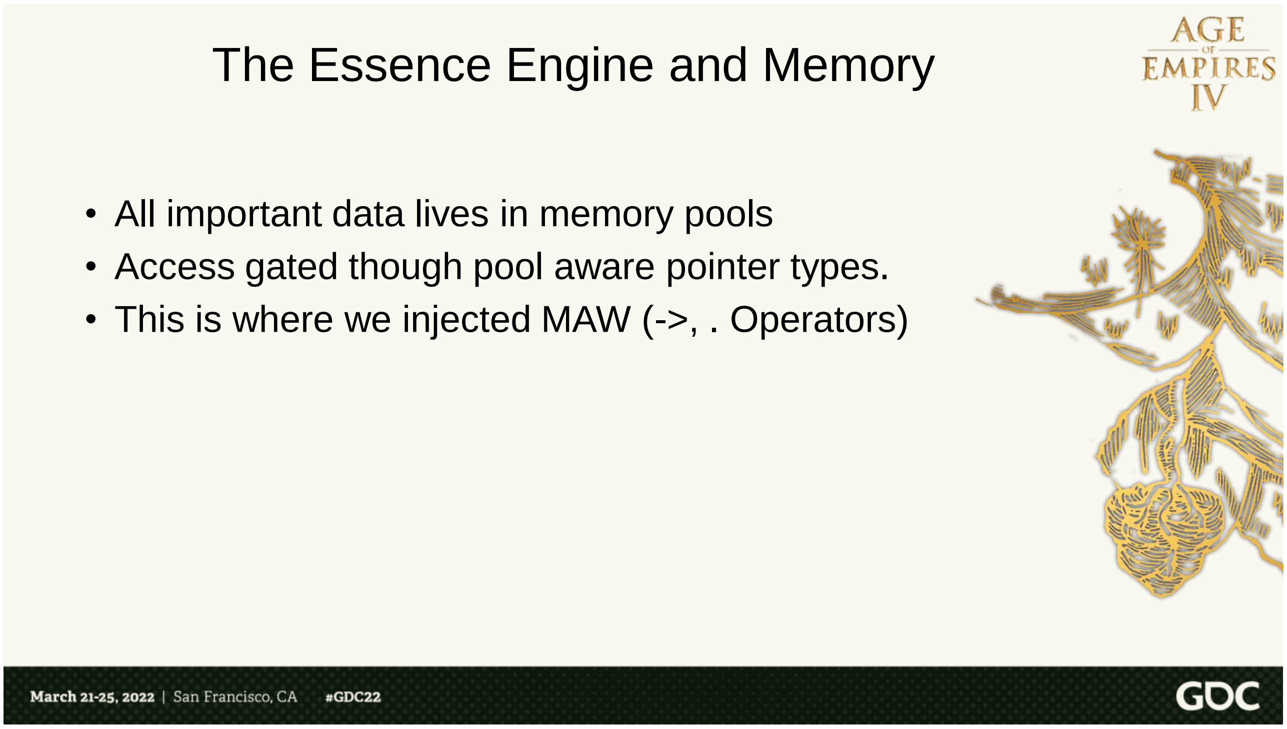


The Solution

The Memory Access Watcher (MAW)

March 21-25, 2022 | San Francisco, CA #GDC22

We called it MAW, or the memory access watcher. Tried real hard here to not have a lame acronym.



The Essence Engine and Memory

- All important data lives in memory pools
- Access gated though pool aware pointer types.
- This is where we injected MAW (->, . Operators)

March 21-25, 2022 | San Francisco, CA #GDC22

AGE OF EMPIRES IV
GDC

To understand how we went about implementing MAW, you need to know a little bit about how our engine works.

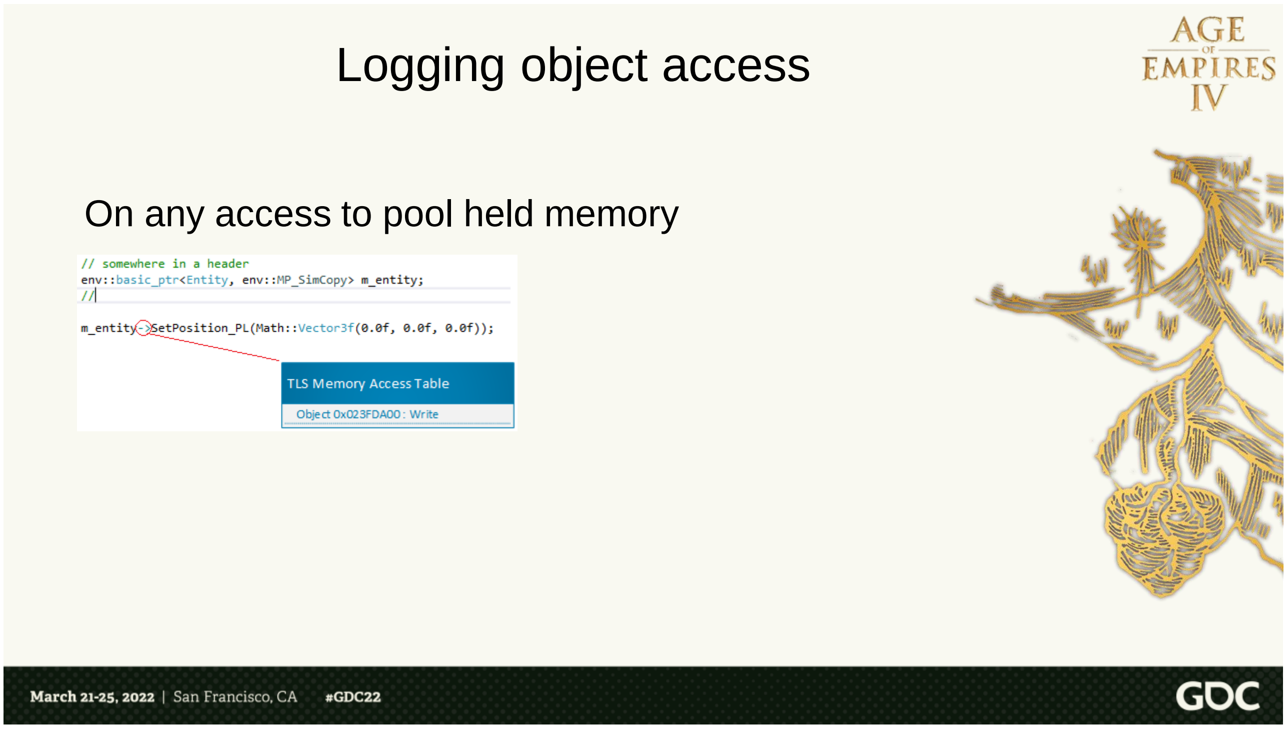
At some point in the past, the team had to separate the simulation from the presentation and put them on their own threads. Since the presentation was using the same objects with the same code and underlying data, the team implemented an offset pool system to avoid having to go through all the data and code being used and figure out exactly which individual bits were needed on the other side. So In this system, when the presentation and simulation sync up, all of the relevant simulation state is copied in one massive go to a second buffer so the presentation has a fixed view of the world to work from while the simulation gets on generating the next simulation frame.

This required using offset pointers. What is an offet pointer? Well.. Its an offset. Instead of a raw pointer under the hood, we store an offset from the beginning of the pool. When dereferenced, the offset pointer looks at a TLS value to figure out whether its in the sim or the pres, finds the pool base pointer and adds its offset to that. In this way the code could be used by the same code on either the simulation or presentation and still find the right data.

It also means we can’t store raw

pointers. The only raw pointer use was only allowed for transient data, operations that will be complete by the next sim tick. The reason for this is because if a raw pointer is stored in an object by the simulation and that object gets copied to the presentation and referenced, the raw pointer will point back to the memory being modified by the simulation update.

I guess the point of all of this explanation is that our offset pointer use was ubiquitous. Gave us an easy point to inject a system like MAW that every object was already using. Many engines I’ve worked in over the years have had similarly ubiquitous smart pointer systems that could be leveraged in the same way.



Logging object access

On any access to pool held memory

```
// somewhere in a header
env::basic_ptr<Entity, env::MP_SimCopy> m_entity;
//

m_entity->SetPosition_PL(Math::Vector3f(0.0f, 0.0f, 0.0f));
```

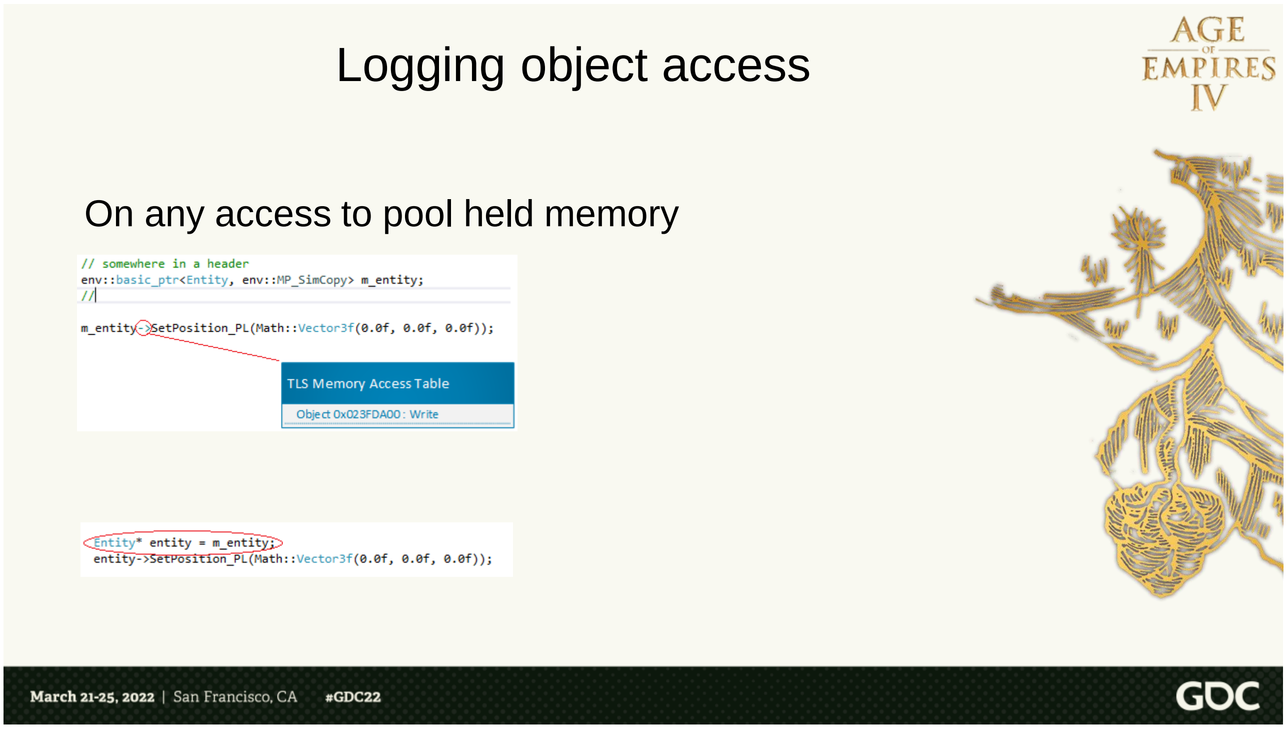
TLS Memory Access Table

Object 0x023FDA00 : Write

March 21-25, 2022 | San Francisco, CA #GDC22

The first time an object is accessed by a task, that generates a MAW access record that we store in a thread local table. We found we only needed keep track of one access per object per task. And it happens to be the most restrictive of all of the accesses to the object, that is, if an entity is accessed many times by a single task, we only bother to store the most restrictive type of access. read only is promoted to write, but write is never demoted. Because the timing is variable, we don’t bother tracking access time and consider an access of any type at any time to hold for the entire execution of the task group during later comparison.

In this example our entity is marked for write access at the instant the dereference operator is invoked. Write is determined by the fact that it’s a non const pointer. The system only ever allows one task in a task group to have write access to an object. If an object is write modified, then no other task in that group is allowed to read or write from that object.



Logging object access

On any access to pool held memory

```
// somewhere in a header
env::basic_ptr<Entity, env::MP_SimCopy> m_entity;
//

m_entity->SetPosition_PL(Math::Vector3f(0.0f, 0.0f, 0.0f));
```

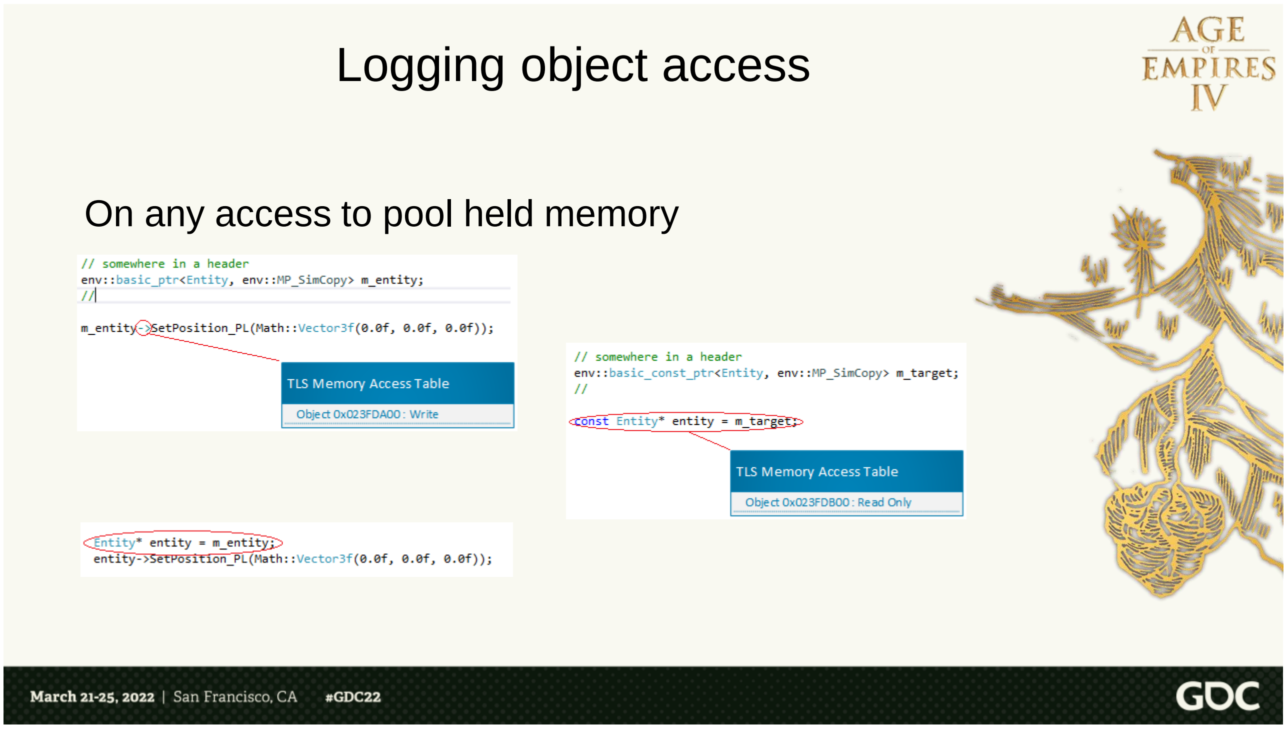
TLS Memory Access Table

Object 0x023FDA00 : Write

```
Entity* entity = m_entity;
entity->SetPosition_PL(Math::Vector3f(0.0f, 0.0f, 0.0f));
```

March 21-25, 2022 | San Francisco, CA #GDC22

Same as the top access, but the access is logged on the top line. We can not track accesses made through raw pointers so when you get one from one of our offset pointers, we assume the worst case based on the constness of our offset handle being accessed.



Logging object access

On any access to pool held memory

```
// somewhere in a header
env::basic_ptr<Entity, env::MP_SimCopy> m_entity;
//

m_entity->SetPosition_PL(Math::Vector3f(0.0f, 0.0f, 0.0f));
```

TLS Memory Access Table

Object 0x023FDA00 : Write

```
Entity* entity = m_entity;
entity->SetPosition_PL(Math::Vector3f(0.0f, 0.0f, 0.0f));
```

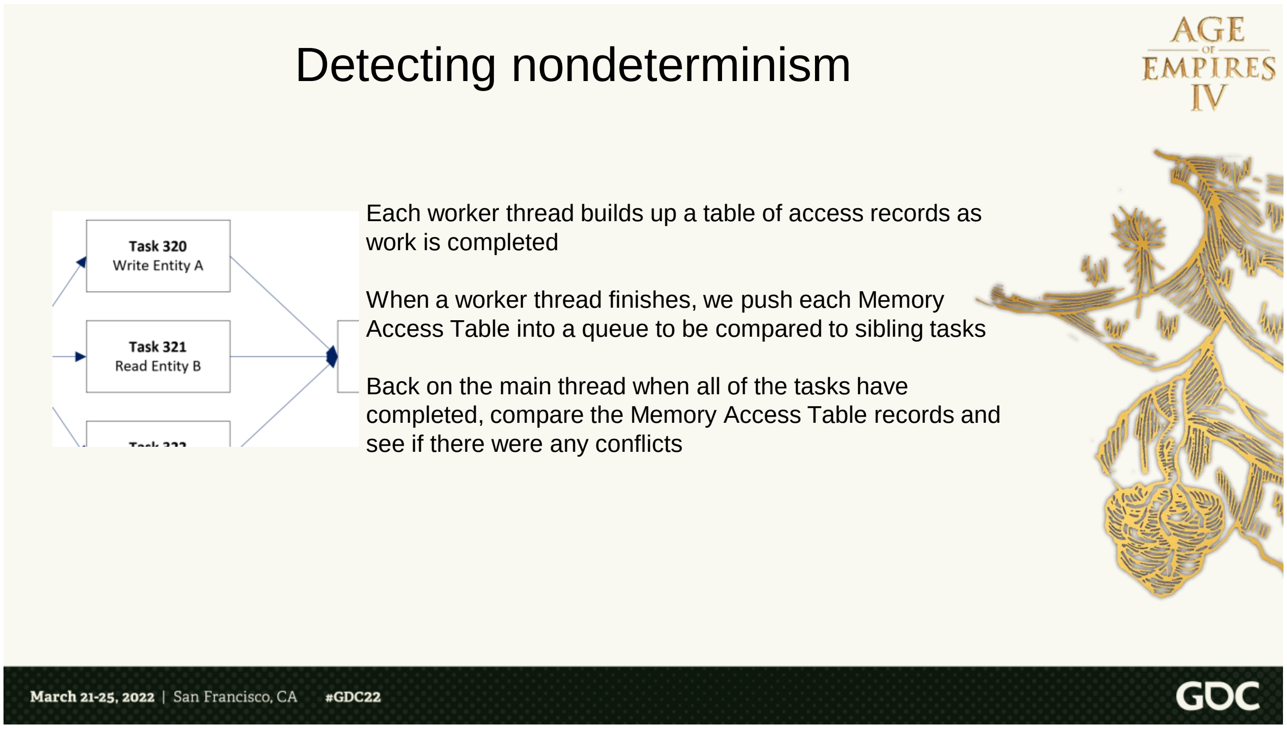
```
// somewhere in a header
env::basic_const_ptr<Entity, env::MP_SimCopy> m_target;
//

const Entity* entity = m_target;
```

TLS Memory Access Table

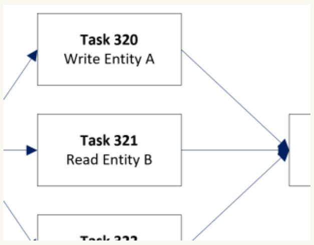
Object 0x023FDB00 : Read Only

March 21-25, 2022 | San Francisco, CA #GDC22

This third example shows one of the several ways you can have a const pointer in our system and is an example of a case that would be logged as a read. Any number of tasks in a group are allowed to read from an entity to long as no other task ever writes to it.



Detecting nondeterminism



Each worker thread builds up a table of access records as work is completed

When a worker thread finishes, we push each Memory Access Table into a queue to be compared to sibling tasks

Back on the main thread when all of the tasks have completed, compare the Memory Access Table records and see if there were any conflicts

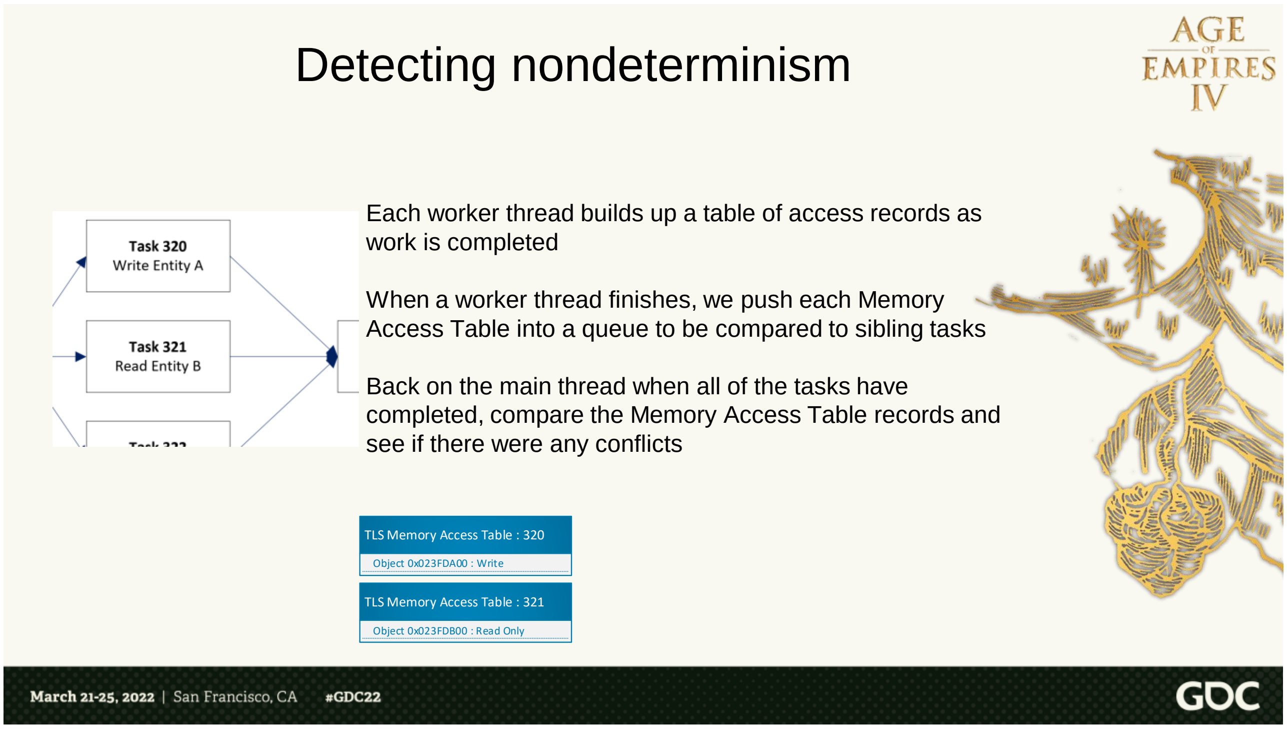
March 21-25, 2022 | San Francisco, CA #GDC22

Each task is generating dozens, if not hundreds of access records into its local table during execution. Each record is just the object accessed and the type of access requested. And that can only be read or write/modify. Time is irrelevant, as it is assumed a task could overlap any other task. What this means is that once an object is accessed by a task that task has ownership of said object from the fork to the following join.

When each task finishes, it pushes its table into a queue for the main thread to process once all the tasks in the group have completed but before the main thread begins work again.

Doesn’t really need to be a strict fork join. This model would work for any thread blocking on another thread or threads.

If you are just tracking read or write access on an object, there are not many possible outcomes for the comparisons that happen. Lets just take a quick look at them.



Detecting nondeterminism

Task 320
Write Entity A
Task 321
Read Entity B

Each worker thread builds up a table of access records as work is completed

When a worker thread finishes, we push each Memory Access Table into a queue to be compared to sibling tasks

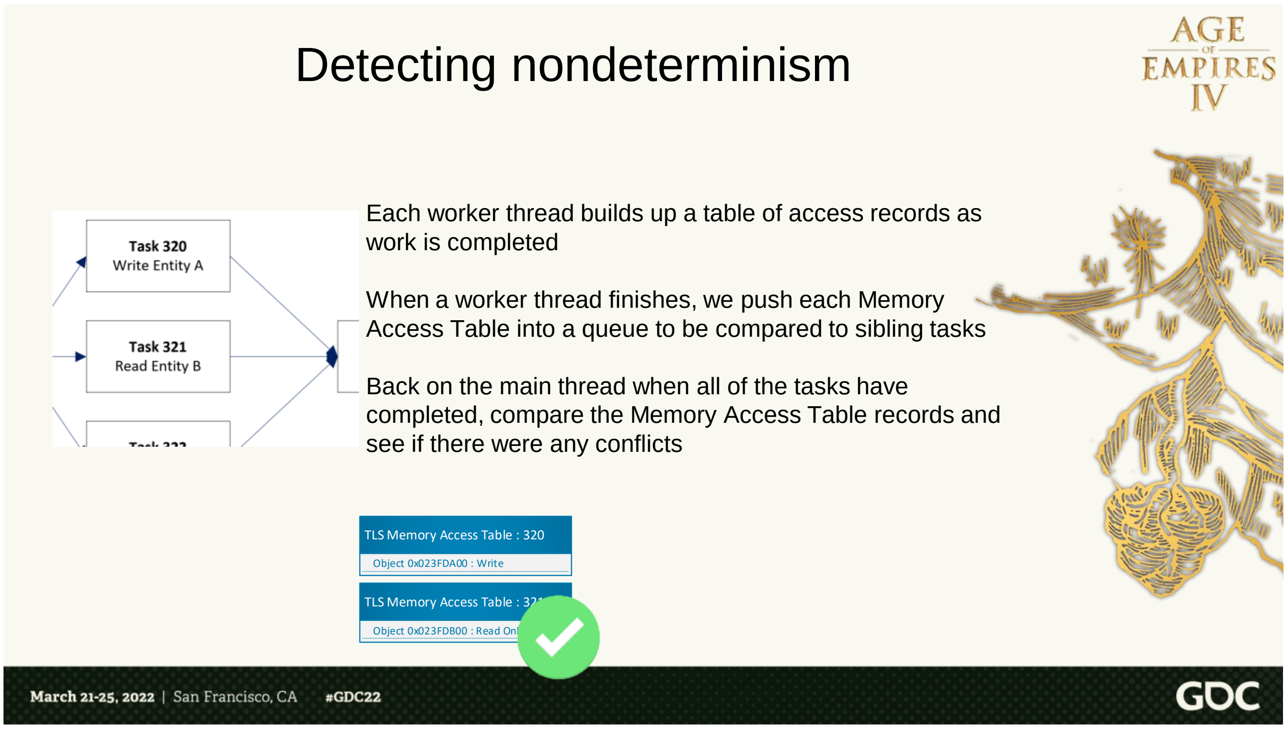
Back on the main thread when all of the tasks have completed, compare the Memory Access Table records and see if there were any conflicts

TLS Memory Access Table : 320
Object 0x023FDA00 : Write

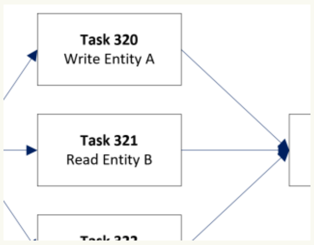
TLS Memory Access Table : 321
Object 0x023FDB00 : Read Only

March 21-25, 2022 | San Francisco, CA #GDC22 GDC

First scenario: the access tables contain only accesses of completely unrelated objects.



Detecting nondeterminism



Each worker thread builds up a table of access records as work is completed

When a worker thread finishes, we push each Memory Access Table into a queue to be compared to sibling tasks

Back on the main thread when all of the tasks have completed, compare the Memory Access Table records and see if there were any conflicts

TLS Memory Access Table : 320

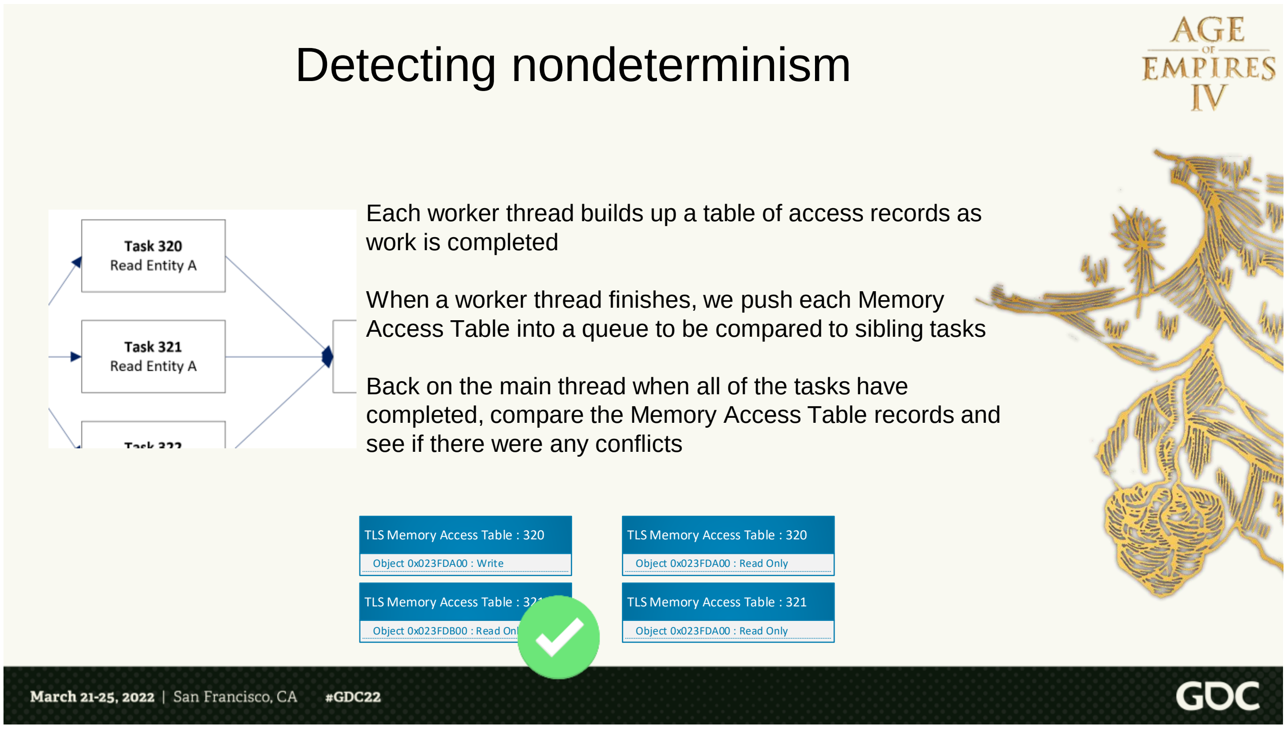
Object 0x023FDA00 : Write

TLS Memory Access Table : 32

Object 0x023FDB00 : Read Onl

March 21-25, 2022 | San Francisco, CA #GDC22

Perfectly OK


Task 320
Read Entity A
Task 321
Read Entity A
Task 322
AGE OF EMPIRES IV
GDC

Detecting nondeterminism

Each worker thread builds up a table of access records as work is completed

When a worker thread finishes, we push each Memory Access Table into a queue to be compared to sibling tasks

Back on the main thread when all of the tasks have completed, compare the Memory Access Table records and see if there were any conflicts

TLS Memory Access Table : 320
Object 0x023FDA00 : Write

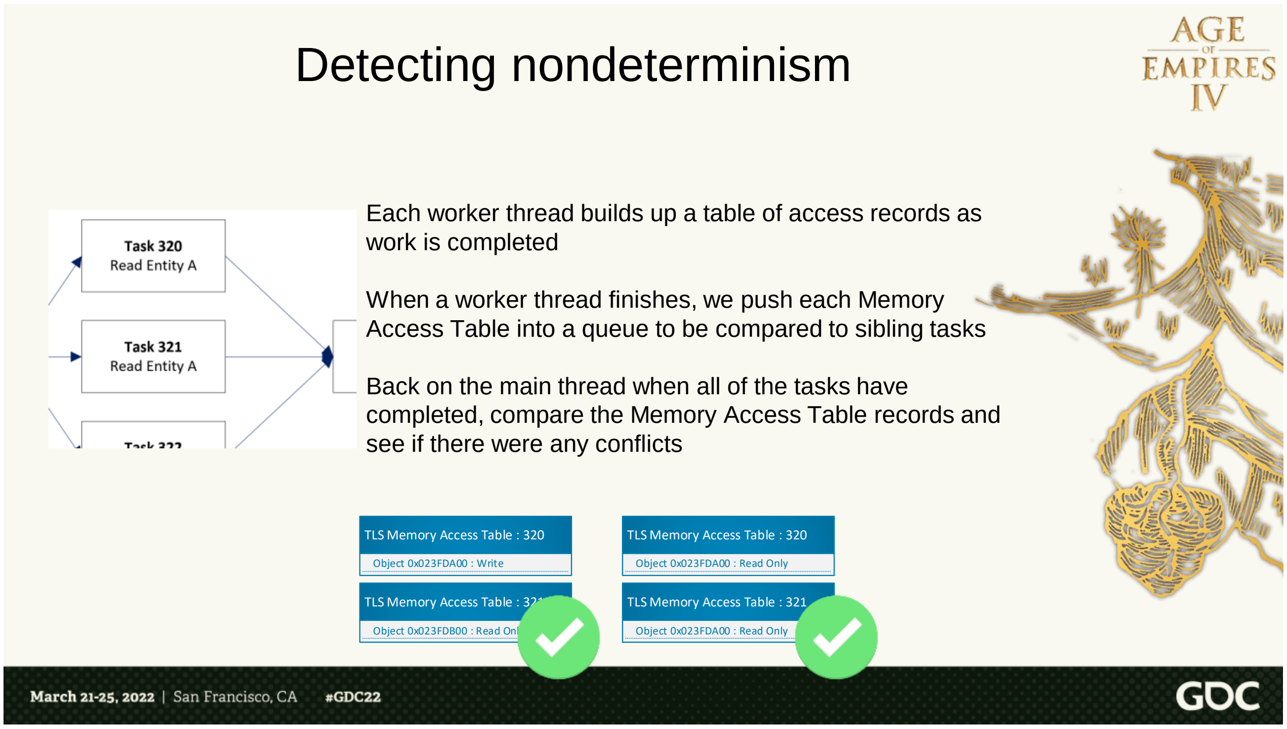
TLS Memory Access Table : 321
Object 0x023FDB00 : Read Only

TLS Memory Access Table : 320
Object 0x023FDA00 : Read Only

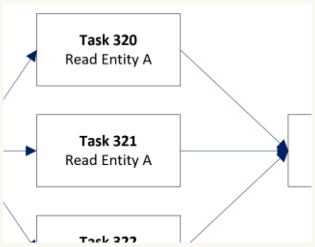
TLS Memory Access Table : 321
Object 0x023FDA00 : Read Only

March 21-25, 2022 | San Francisco, CA #GDC22

Two non competing accesses of the same object?



Detecting nondeterminism



Each worker thread builds up a table of access records as work is completed

When a worker thread finishes, we push each Memory Access Table into a queue to be compared to sibling tasks

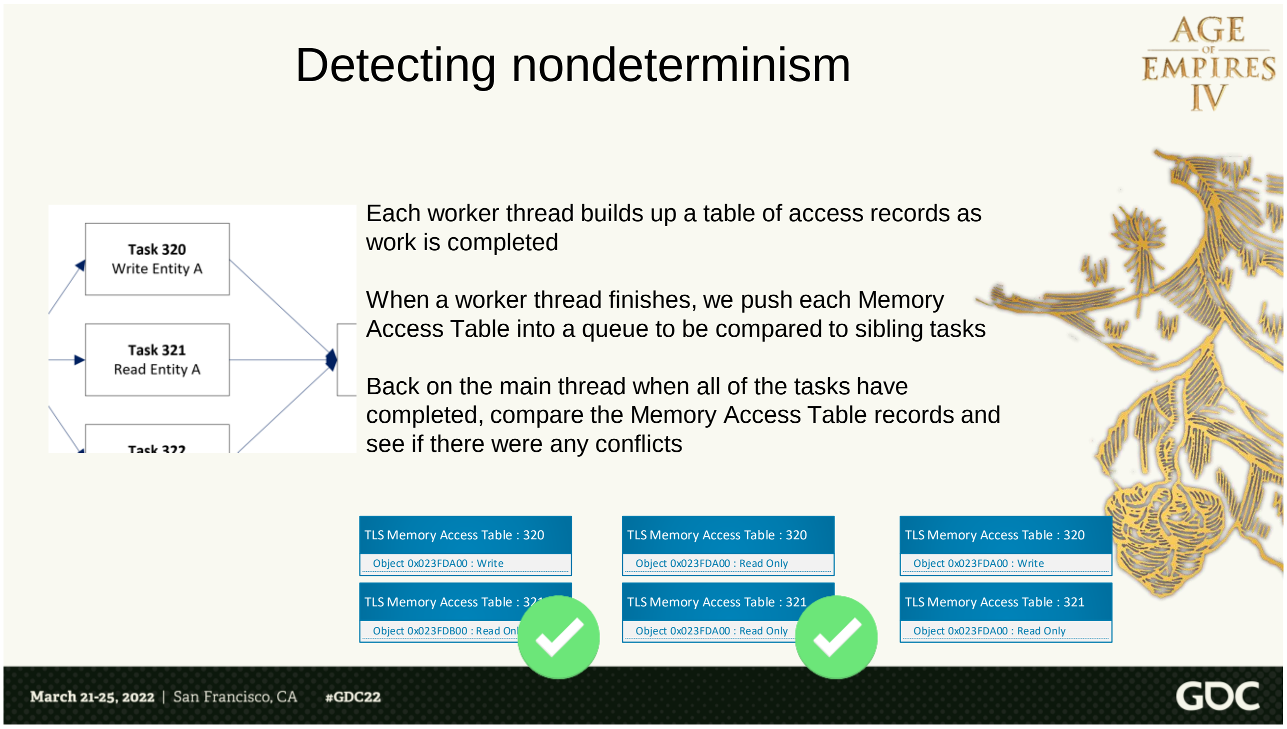
Back on the main thread when all of the tasks have completed, compare the Memory Access Table records and see if there were any conflicts

TLS Memory Access Table : 320
Object 0x023FDA00 : Write
TLS Memory Access Table : 321
Object 0x023FDB00 : Read Only ✓

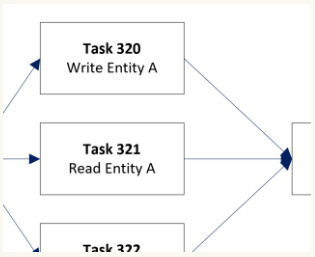
TLS Memory Access Table : 320
Object 0x023FDA00 : Read Only
TLS Memory Access Table : 321
Object 0x023FDA00 : Read Only ✓

March 21-25, 2022 | San Francisco, CA #GDC22

That’s OK too when its read – only.



Detecting nondeterminism



Each worker thread builds up a table of access records as work is completed

When a worker thread finishes, we push each Memory Access Table into a queue to be compared to sibling tasks

Back on the main thread when all of the tasks have completed, compare the Memory Access Table records and see if there were any conflicts

TLS Memory Access Table : 320
Object 0x023FDA00 : Write

TLS Memory Access Table : 32
Object 0x023FDB00 : Read Onl

✓

TLS Memory Access Table : 320
Object 0x023FDA00 : Read Only

TLS Memory Access Table : 321
Object 0x023FDA00 : Read Only

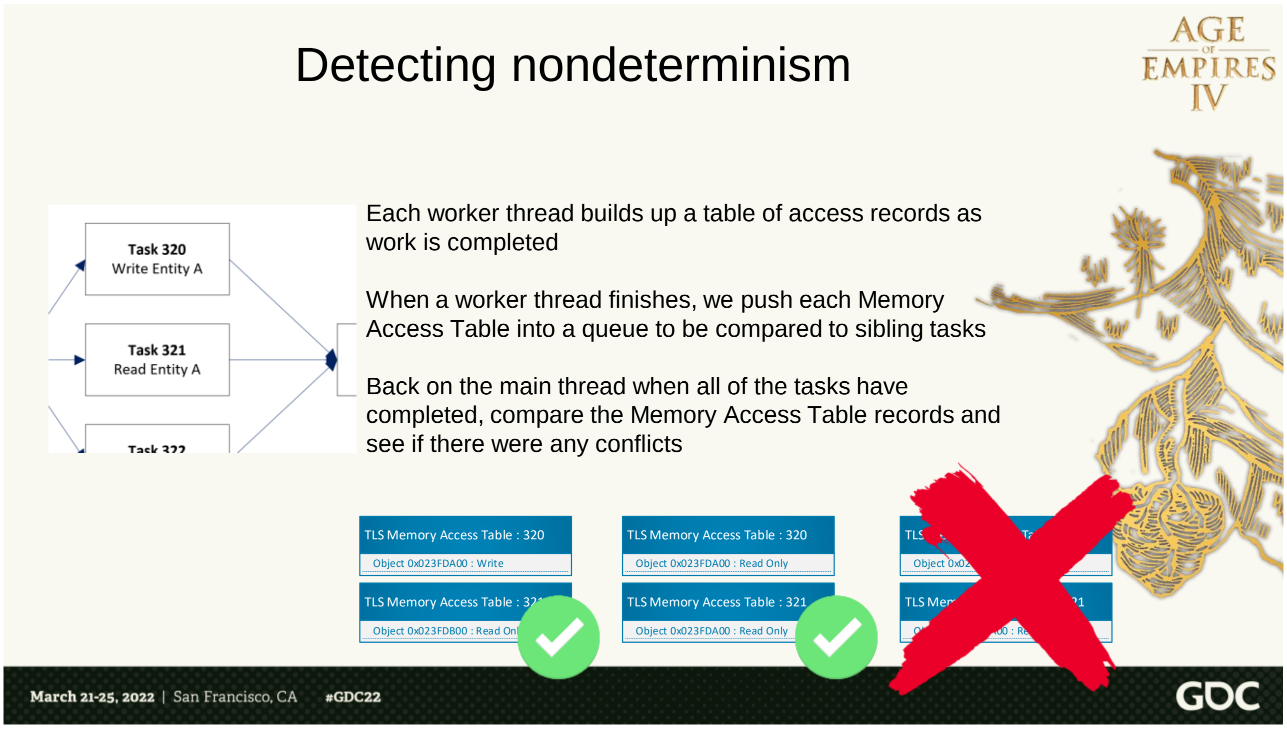
✓

TLS Memory Access Table : 320
Object 0x023FDA00 : Write

TLS Memory Access Table : 321
Object 0x023FDA00 : Read Only

March 21-25, 2022 | San Francisco, CA #GDC22

Two competing accesses for the same object?



Detecting nondeterminism

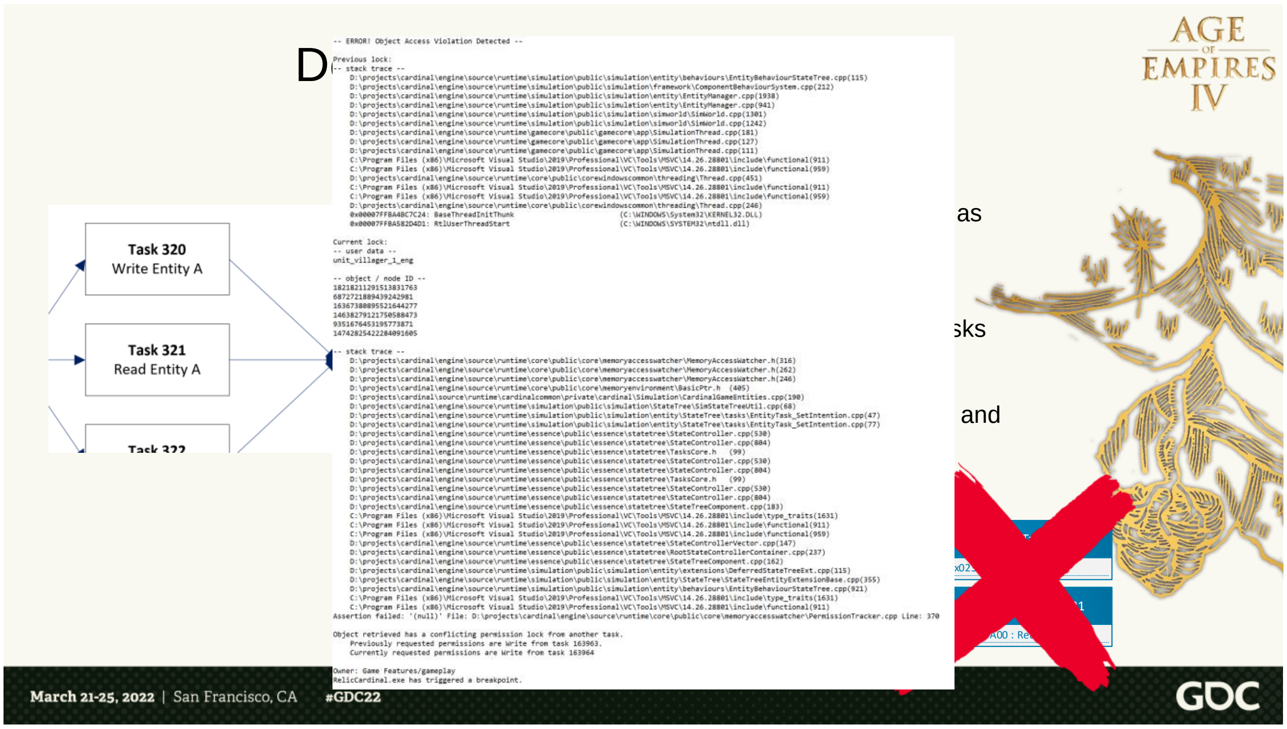
Each worker thread builds up a table of access records as work is completed

When a worker thread finishes, we push each Memory Access Table into a queue to be compared to sibling tasks

Back on the main thread when all of the tasks have completed, compare the Memory Access Table records and see if there were any conflicts

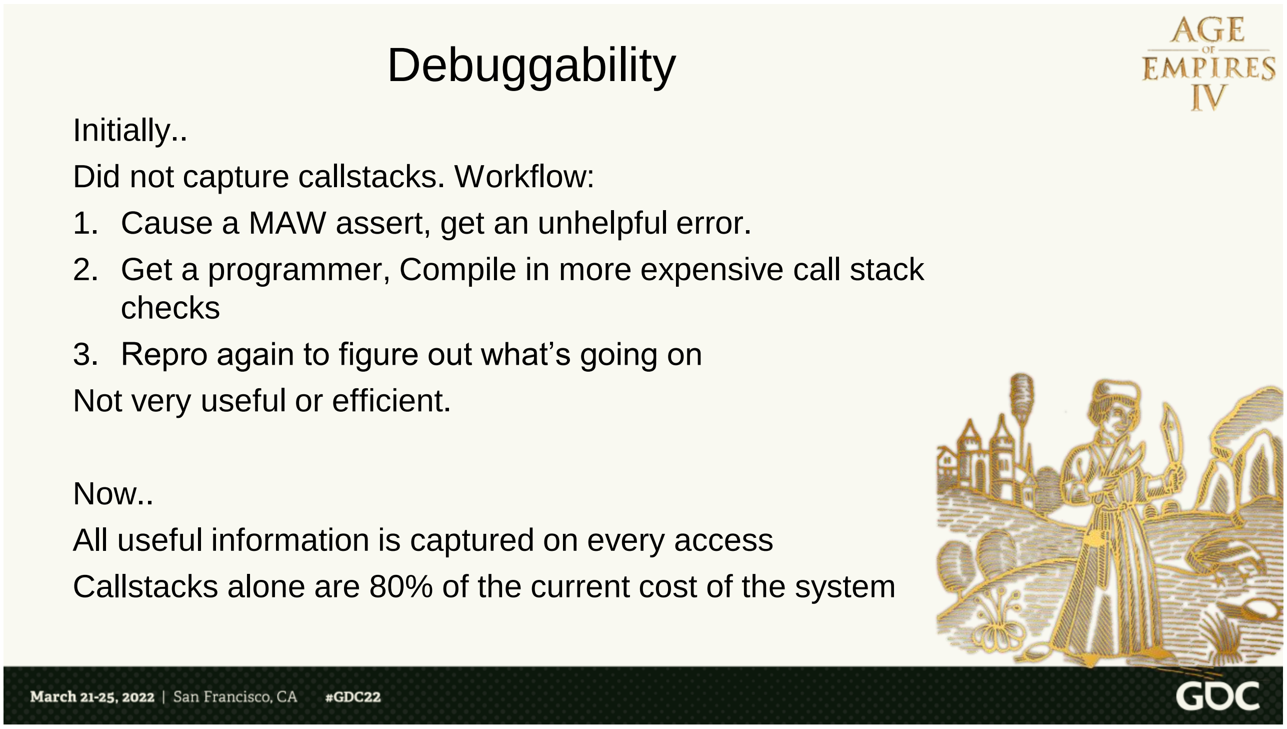
Task 320
Write Entity A
Task 321
Read Entity A
Task 322
TLS Memory Access Table : 320
Object 0x023FDA00 : Write
TLS Memory Access Table : 32
Object 0x023FDB00 : Read Onl
TLS Memory Access Table : 320
Object 0x023FDA00 : Read Only
TLS Memory Access Table : 321
Object 0x023FDA00 : Read Only
March 21-25, 2022 | San Francisco, CA #GDC22
GDC
AGE OF EMPIRES IV

Uh oh. This would also be true for write/write. Doesn’t matter, once a single write happens no other task can touch the object.



```
-- ERROR! Object Access Violation Detected --

Previous lock:
-- stack trace --
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\behaviours\EntityBehaviourStateTree.cpp(115)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\framework\ComponentBehaviourSystem.cpp(212)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\EntityManager.cpp(1938)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\EntityManager.cpp(941)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\simworld\SimWorld.cpp(1301)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\simworld\SimWorld.cpp(1242)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(181)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(127)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(111)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\core\public\corewindowscommon\threading\Thread.cpp(451)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\core\public\corewindowscommon\threading\Thread.cpp(246)
    0x00007FFBA4BC7C24: BaseThreadInitThunk                        (C:\WINDOWS\System32\KERNEL32.DLL)
    0x00007FFBA582D4D1: RtlUserThreadStart                         (C:\WINDOWS\SYSTEM32\ntdll.dll)

Current lock:
-- user data --
unit_villager_1_eng

-- object / node ID --
18218211291513831763
6872721889439242981
16367388895521644277
14638279121750588473
9351676453195773871
14742825422284091605

-- stack trace --
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(316)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(262)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(246)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryenvironment\BasicPtr.h  (405)
    D:\projects\cardinal\source\runtime\cardinalcommon\private\cardinal\Simulation\CardinalGameEntities.cpp(190)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\StateTree\SimStateTreeUtil.cpp(68)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\tasks\EntityTask_SetIntention.cpp(47)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\tasks\EntityTask_SetIntention.cpp(77)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\TasksCore.h   (99)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\TasksCore.h   (99)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateTreeComponent.cpp(183)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\type_traits(1631)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateControllerVector.cpp(147)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\RootStateControllerContainer.cpp(237)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateTreeComponent.cpp(162)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\extensions\DeferredStateTreeExt.cpp(115)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\StateTreeEntityExtensionBase.cpp(355)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\behaviours\EntityBehaviourStateTree.cpp(921)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\type_traits(1631)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
Assertion failed: '(null)' File: D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\PermissionTracker.cpp Line: 370

Object retrieved has a conflicting permission lock from another task.
    Previously requested permissions are Write from task 163963.
    Currently requested permissions are Write from task 163964

Owner: Game Features/gameplay
RelicCardinal.exe has triggered a breakpoint.
```

And you get hit in the face with this. It tells you the bad happened. What is all of this information? There are 4 important bits of information here and we'll break down what it all means so don't bother squinting and trying to read this.



Debuggability

Initially..

Did not capture callstacks. Workflow:

1. Cause a MAW assert, get an unhelpful error.
2. Get a programmer, Compile in more expensive call stack checks
3. Repro again to figure out what's going on

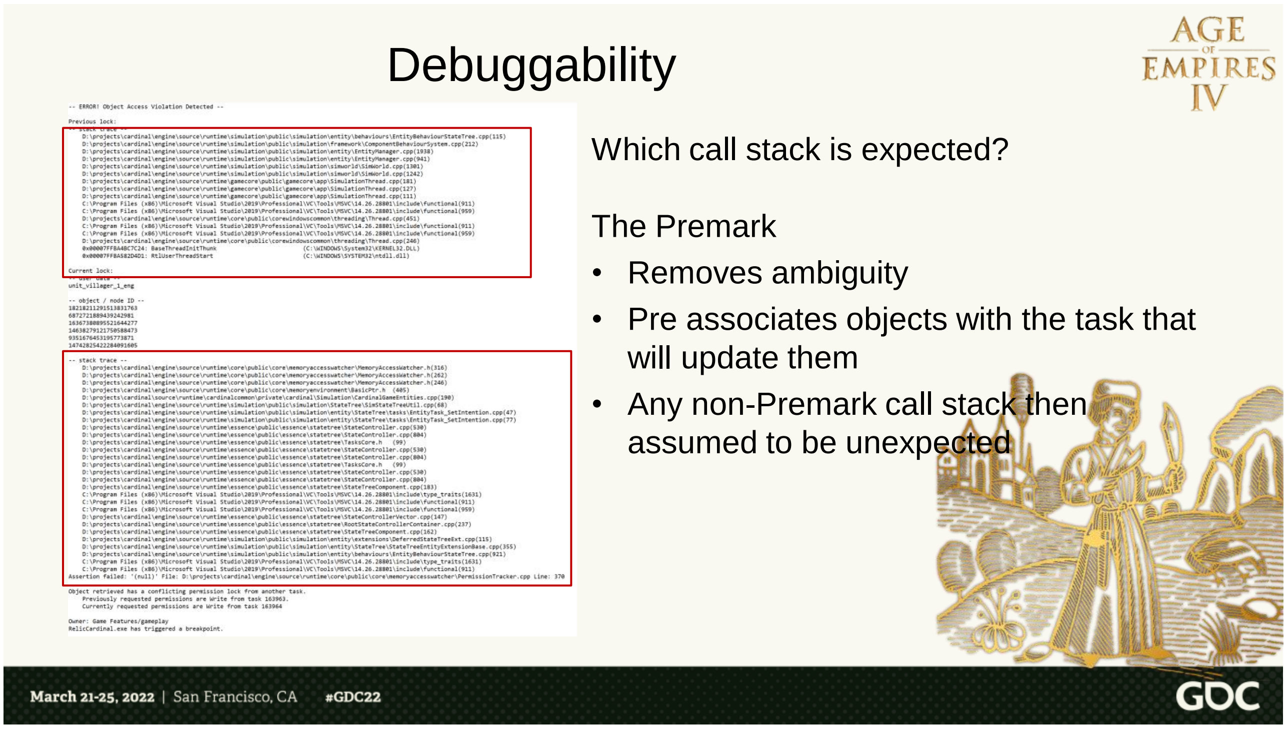
Not very useful or efficient.

Now..

All useful information is captured on every access

Callstacks alone are 80% of the current cost of the system

How'd we get to this big ol message? It didn't start that way. We still just keep the callstack as void*s until an actual violation is detected. Then we load the symbols and convert to something human readable. Its also why I didn't spend too much time optimizing the rest of the system.. Could make record tracking/comparison 2x faster but no one would notice. Everything I am about to cover is very engine and game specific, but I'm sure you'll see its worth considering for any implementation.



Debuggability

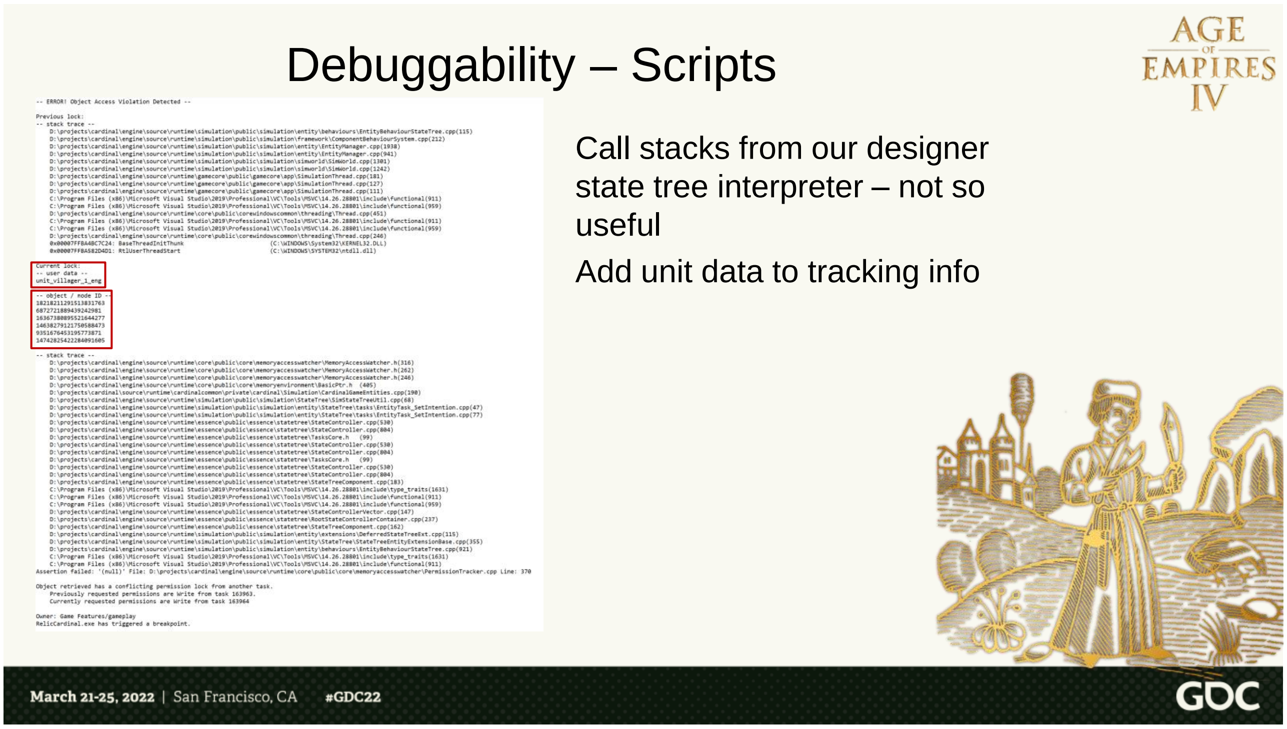
```
-- ERROR! Object Access Violation Detected --

Previous lock:
-- stack trace --
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\behaviours\EntityBehaviourStateTree.cpp(115)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\framework\ComponentBehaviourSystem.cpp(212)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\EntityManager.cpp(1938)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\EntityManager.cpp(941)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\simworld\SimWorld.cpp(1301)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\simworld\SimWorld.cpp(1242)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(181)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(127)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(111)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\core\public\corewindowscommon\threading\Thread.cpp(451)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\core\public\corewindowscommon\threading\Thread.cpp(246)
    0x00007FFBA4BC7C24: BaseThreadInitThunk                          (C:\WINDOWS\System32\KERNEL32.DLL)
    0x00007FFBA582D4D1: RtlUserThreadStart                           (C:\WINDOWS\SYSTEM32\ntdll.dll)

Current lock:
-- user data --
unit_villager_1_eng

-- object / node ID --
1821821129151383l763
6872721889439242981
16367380895521644277
14638279121750588473
9351676453195773871
14742825422284091605

-- stack trace --
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(316)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(262)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(246)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryenvironment\BasicPtr.h  (405)
    D:\projects\cardinal\source\runtime\cardinalcommon\private\cardinal\Simulation\CardinalGameEntities.cpp(190)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\StateTree\SimStateTreeUtil.cpp(68)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\tasks\EntityTask_SetIntention.cpp(47)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\tasks\EntityTask_SetIntention.cpp(77)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\TasksCore.h   (99)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\TasksCore.h   (99)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateTreeComponent.cpp(183)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\type_traits(1631)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateControllerVector.cpp(147)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\RootStateControllerContainer.cpp(237)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateTreeComponent.cpp(162)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\extensions\DeferredStateTreeExt.cpp(115)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\StateTreeEntityExtensionBase.cpp(355)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\behaviours\EntityBehaviourStateTree.cpp(921)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\type_traits(1631)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
Assertion failed: '(null)' File: D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\PermissionTracker.cpp Line: 370
Object retrieved has a conflicting permission lock from another task.
    Previously requested permissions are Write from task 163963.
    Currently requested permissions are Write from task 163964

Owner: Game Features/gameplay
RelicCardinal.exe has triggered a breakpoint.
```

Which call stack is expected?

The Premark

- Removes ambiguity
- Pre associates objects with the task that will update them
- Any non-Premark call stack then assumed to be unexpected

Initially the system would throw two gnarly callstacks at you. How would you know which one of the two was correct? It could be that neither was expected and the collision was detected. Or it could be that somehow both were expected and this is an architectural issue that needs resolving. You were always in a position where you'd need the author of the both bits of code to come look over the log and figure out which of the two was 'right'. To simplify this we added the Premark phase. The top callstack is the obvious in the log is always from this block of code, anyone could look at it and know it was the intended access. Well, the intended task owner. Any other callstack was not expected. Ok maybe this isn’t so game specific.



Debuggability – Scripts

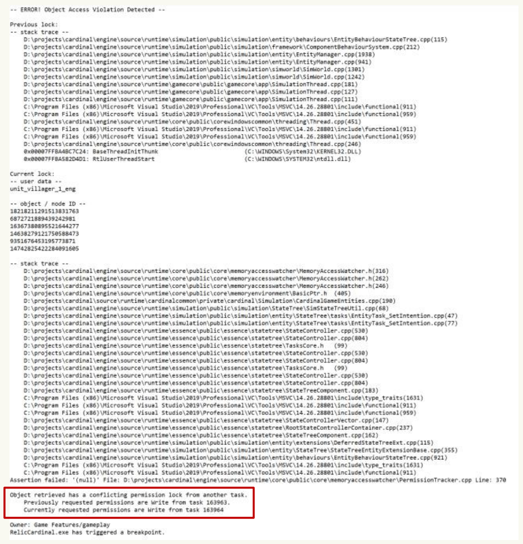
```
-- ERROR! Object Access Violation Detected --

Previous lock:
-- stack trace --
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\behaviours\EntityBehaviourStateTree.cpp(115)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\framework\ComponentBehaviourSystem.cpp(212)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\EntityManager.cpp(1938)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\EntityManager.cpp(941)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\simworld\SimWorld.cpp(1301)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\simworld\SimWorld.cpp(1242)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(181)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(127)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(111)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\core\public\corewindowscommon\threading\Thread.cpp(451)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\core\public\corewindowscommon\threading\Thread.cpp(246)
    0x00007FFBA4BC7C24: BaseThreadInitThunk                        (C:\WINDOWS\System32\KERNEL32.DLL)
    0x00007FFBA582D4D1: RtlUserThreadStart                         (C:\WINDOWS\SYSTEM32\ntdll.dll)

Current lock:
-- user data --
unit_villager_1_eng

-- object / node ID --
18218211291513831763
6872721889439242981
16367380895521644277
14638279121750588473
9351676453195773871
14742825422284091605

-- stack trace --
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(316)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(262)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(246)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryenvironment\BasicPtr.h  (405)
    D:\projects\cardinal\source\runtime\cardinalcommon\private\cardinal\Simulation\CardinalGameEntities.cpp(190)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\StateTree\SimStateTreeUtil.cpp(68)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\tasks\EntityTask_SetIntention.cpp(47)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\tasks\EntityTask_SetIntention.cpp(77)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\TasksCore.h   (99)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\TasksCore.h   (99)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateTreeComponent.cpp(183)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\type_traits(1631)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateControllerVector.cpp(147)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\RootStateControllerContainer.cpp(237)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateTreeComponent.cpp(162)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\extensions\DeferredStateTreeExt.cpp(115)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\StateTreeEntityExtensionBase.cpp(355)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\behaviours\EntityBehaviourStateTree.cpp(921)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\type_traits(1631)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
Assertion failed: '(null)' File: D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\PermissionTracker.cpp Line: 370

Object retrieved has a conflicting permission lock from another task.
    Previously requested permissions are Write from task 163963.
    Currently requested permissions are Write from task 163964

Owner: Game Features/gameplay
RelicCardinal.exe has triggered a breakpoint.
```

Call stacks from our designer state tree interpreter – not so useful

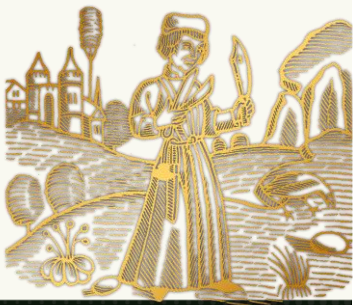
Add unit data to tracking info

This block of code associates the unit who’s statetree script is running with any subsequently captured callstacks on this worker thread. This is designer created data, and callstacks generated by it are generic state tree interpreter call stacks. Not so useful. This macro associates the unit name with any subsequently captured callstacks so we can actually figure out what unit was running. There are similar macros that log the state tree node UIDs as they execute, so we can work back from a MAW log to actual designer data.

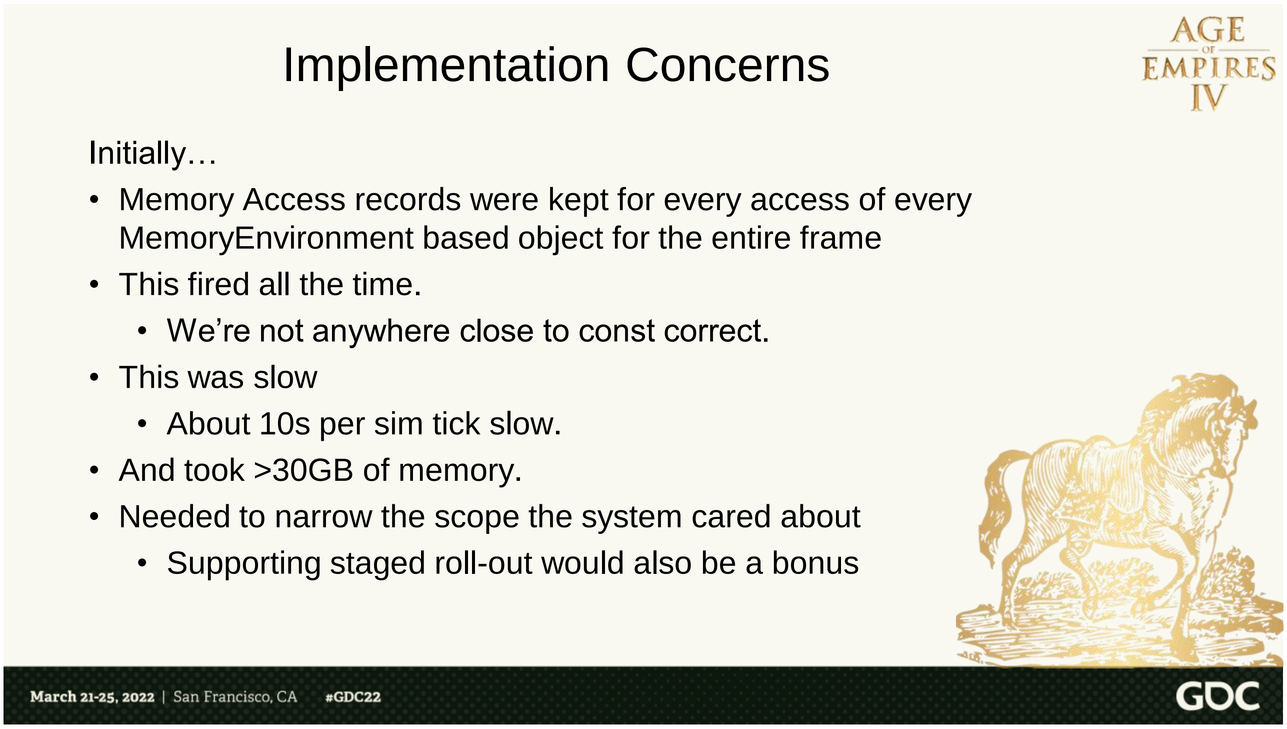
Debuggability



```
-- ERROR! Object Access Violation Detected --

Previous lock:
-- stack trace --
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\behaviours\EntityBehaviourStateTree.cpp(115)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\framework\ComponentBehaviourSystem.cpp(212)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\EntityManager.cpp(1938)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\EntityManager.cpp(941)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\simworld\SimWorld.cpp(1301)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\simworld\SimWorld.cpp(1242)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(181)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(127)
    D:\projects\cardinal\engine\source\runtime\gamecore\public\gamecore\app\SimulationThread.cpp(111)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\core\public\corewindowscommon\threading\Thread.cpp(451)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\core\public\corewindowscommon\threading\Thread.cpp(246)
    0x00007FFBA4BC7C24: BaseThreadInitThunk                        (C:\WINDOWS\System32\KERNEL32.DLL)
    0x00007FFBA582D4D1: RtlUserThreadStart                         (C:\WINDOWS\SYSTEM32\ntdll.dll)

Current lock:
-- user data --
unit_villager_1_eng

-- object / node ID --
18218211291513831763
6872721889439242981
16367388895521644277
14638279121750588473
9351676453195773871
14742825422284091605

-- stack trace --
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(316)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(262)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\MemoryAccessWatcher.h(246)
    D:\projects\cardinal\engine\source\runtime\core\public\core\memoryenvironment\BasicPtr.h  (405)
    D:\projects\cardinal\source\runtime\cardinalcommon\private\cardinal\Simulation\CardinalGameEntities.cpp(190)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\StateTree\SimStateTreeUtil.cpp(68)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\tasks\EntityTask_SetIntention.cpp(47)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\tasks\EntityTask_SetIntention.cpp(77)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\TasksCore.h   (99)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\TasksCore.h   (99)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(530)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateController.cpp(804)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateTreeComponent.cpp(183)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\type_traits(1631)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(959)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateControllerVector.cpp(147)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\RootStateControllerContainer.cpp(237)
    D:\projects\cardinal\engine\source\runtime\essence\public\essence\statetree\StateTreeComponent.cpp(162)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\extensions\DeferredStateTreeExt.cpp(115)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\StateTree\StateTreeEntityExtensionBase.cpp(355)
    D:\projects\cardinal\engine\source\runtime\simulation\public\simulation\entity\behaviours\EntityBehaviourStateTree.cpp(921)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\type_traits(1631)
    C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.28801\include\functional(911)
Assertion failed: '(null)' File: D:\projects\cardinal\engine\source\runtime\core\public\core\memoryaccesswatcher\PermissionTracker.cpp Line: 370

Object retrieved has a conflicting permission lock from another task.
    Previously requested permissions are Write from task 163963.
    Currently requested permissions are Write from task 163964

Owner: Game Features/gameplay
RelicCardinal.exe has triggered a breakpoint.
```



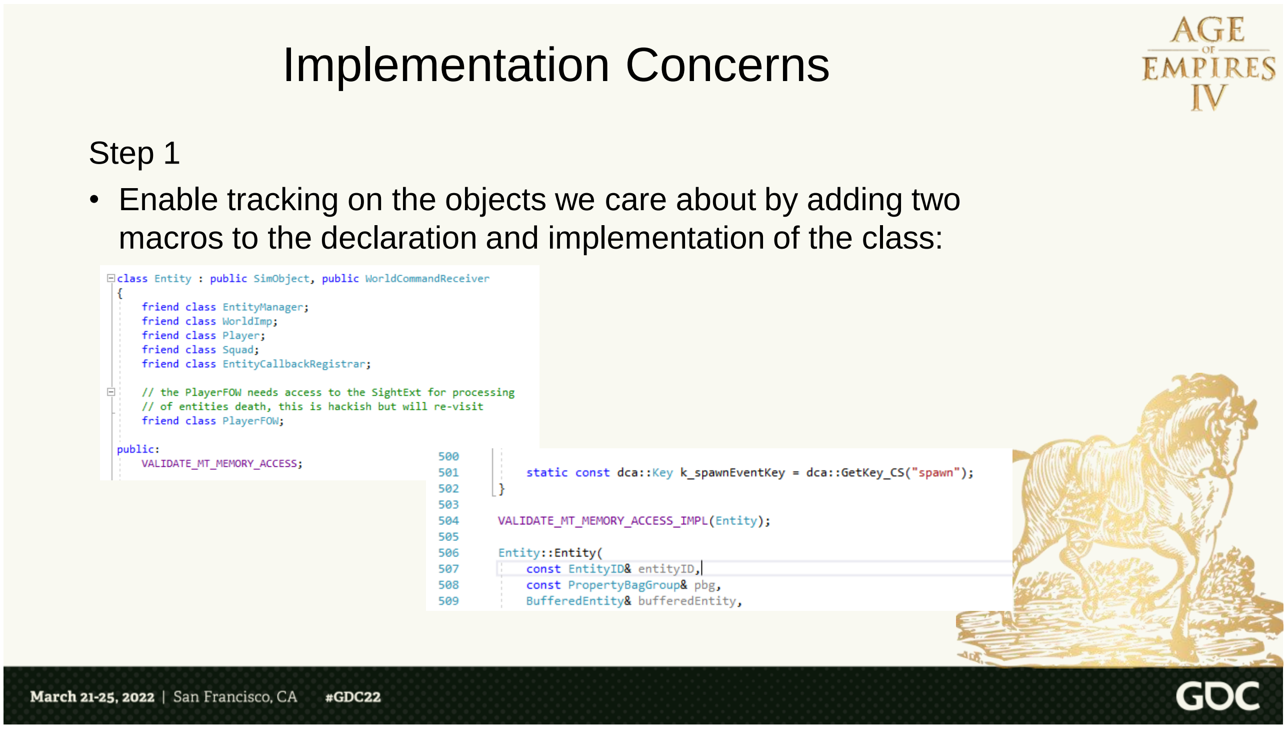
The final block here has the internal job ID and violating permission types being requested.



Implementation Concerns

Initially…

- Memory Access records were kept for every access of every MemoryEnvironment based object for the entire frame
- This fired all the time.
  - We’re not anywhere close to const correct.
- This was slow
  - About 10s per sim tick slow.
- And took >30GB of memory.
- Needed to narrow the scope the system cared about
  - Supporting staged roll-out would also be a bonus



Implementation Concerns

Step 1

- Enable tracking on the objects we care about by adding two macros to the declaration and implementation of the class:

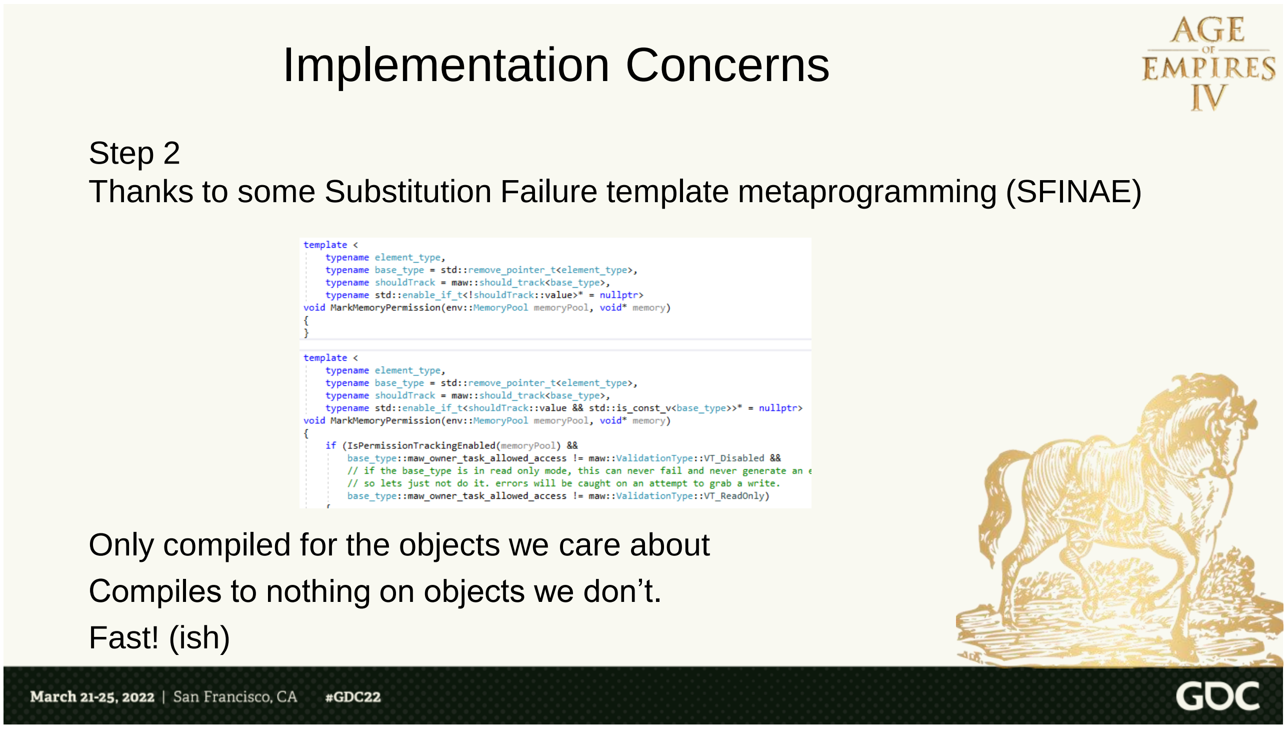
```
class Entity : public SimObject, public WorldCommandReceiver
{
    friend class EntityManager;
    friend class WorldImp;
    friend class Player;
    friend class Squad;
    friend class EntityCallbackRegistrar;

    // the PlayerFOW needs access to the SightExt for processing
    // of entities death, this is hackish but will re-visit
    friend class PlayerFOW;

public:
    VALIDATE_MT_MEMORY_ACCESS;
```

```
        static const dca::Key k_spawnEventKey = dca::GetKey_CS("spawn");
}

VALIDATE_MT_MEMORY_ACCESS_IMPL(Entity);

Entity::Entity(
        const EntityID& entityID,
        const PropertyBagGroup& pbg,
        BufferedEntity& bufferedEntity,
```

March 21-25, 2022 | San Francisco, CA #GDC22

Easy to integrate. We got enabling the system down to these two lines. This is all it took to enable MAW tracking on any object in our engine.

This macro serves two purposes. It creates an uint UID that is used under the hood during allocation record creation. This represents the objects slot in the allocation tables. Every object is given a unique ID the first time it is used in the MAW system. Lazy assignment. Prevents creation of loads of unused IDs. We also support enabling and disabling tracking of object types for different parts of the update, but this has a performance penalty over disabling by removing the macro as the latter causes the system to never be compiled into the object to begin with.



Implementation Concerns

AGE OF EMPIRES IV

Step 2
Thanks to some Substitution Failure template metaprogramming (SFINAE)

```
template <
    typename element_type,
    typename base_type = std::remove_pointer_t<element_type>,
    typename shouldTrack = maw::should_track<base_type>,
    typename std::enable_if_t<!shouldTrack::value>* = nullptr>
void MarkMemoryPermission(env::MemoryPool memoryPool, void* memory)
{
}
```

```
template <
    typename element_type,
    typename base_type = std::remove_pointer_t<element_type>,
    typename shouldTrack = maw::should_track<base_type>,
    typename std::enable_if_t<shouldTrack::value && std::is_const_v<base_type>>* = nullptr>
void MarkMemoryPermission(env::MemoryPool memoryPool, void* memory)
{
    if (IsPermissionTrackingEnabled(memoryPool) &&
        base_type::maw_owner_task_allowed_access != maw::ValidationType::VT_Disabled &&
        // if the base_type is in read only mode, this can never fail and never generate an e
        // so lets just not do it. errors will be caught on an attempt to grab a write.
        base_type::maw_owner_task_allowed_access != maw::ValidationType::VT_ReadOnly)
```

Only compiled for the objects we care about

Compiles to nothing on objects we don't.

Fast! (ish)

March 21-25, 2022 | San Francisco, CA #GDC22

GDC

But having to check whether MAW is enabled on a per object basis added a lot of overhead, just due to the number of objects and dereferences going on in our engine. Templates to the rescue (?)

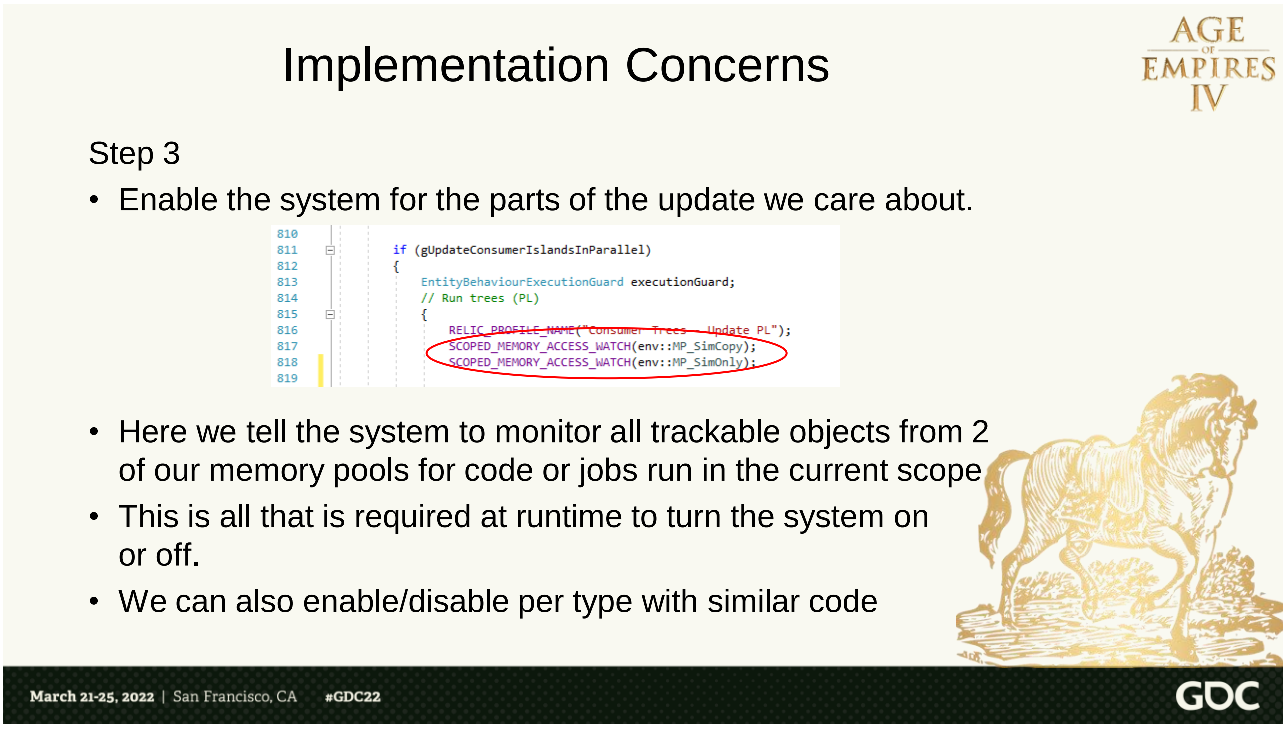
Using SFINAE we were able to only compile MAW in on objects that had the required tracking variables. MAW was completely compiled out on all of the rest of our game objects. Can't get lower overhead than that.

```
template <

typename element_type,

typename base_type =
std::remove_pointer_t<element_type>,

typename shouldTrack =
maw::should_track<base_type>,

typename
std::enable_if_t<!shouldTrack::value>* =
nullptr>

void MarkMemoryPermission(env::MemoryPool,
void*)

{

}


template <

typename element_type,

typename base_type =
std::remove_pointer_t<element_type>,

typename shouldTrack =
maw::should_track<base_type>,

typename
```

```
std::enable_if_t<shouldTrack::value &&
std::is_const_v<base_type>>* = nullptr>
void MarkMemoryPermission(env::MemoryPool
memoryPool, void* memory)
{
if
(IsPermissionTrackingEnabled(memoryPool)
&&
base_type::maw_owner_task_allowed_access
!= maw::ValidationType::VT_Disabled &&
// if the base_type is in read only mode,
this can never fail and never generate an
error.
// so lets just not do it. errors will be
caught on an attempt to grab a write.
base_type::maw_owner_task_allowed_access
!= maw::ValidationType::VT_ReadOnly)
{
base_type* object =
reinterpret_cast<base_type*>(memory);
if
(ValidateReadOnlyPermissionRequest(memoryP
ool, object))
{
```

```
MarkMemoryAsReadOnly(memoryPool, object);
}
}
}

template <
typename element_type,
typename base_type =
std::remove_pointer_t<element_type>,
typename shouldTrack =
maw::should_track<base_type>,
typename
std::enable_if_t<shouldTrack::value &&
!std::is_const_v<base_type>>* = nullptr>
void MarkMemoryPermission(env::MemoryPool
memoryPool, void* memory)
{
if
(IsPermissionTrackingEnabled(memoryPool)
&&
base_type::maw_owner_task_allowed_access
!= maw::ValidationType::VT_Disabled)
{
```

```
base_type* object =
reinterpret_cast<base_type*>(memory);
if
(ValidateWritePermissionRequest(memoryPool
, object))
{
MarkMemoryAsWriteable(memoryPool, object);
}
}
}
```



Implementation Concerns

AGE OF EMPIRES IV

Step 3

- Enable the system for the parts of the update we care about.

```
    if (gUpdateConsumerIslandsInParallel)
    {
        EntityBehaviourExecutionGuard executionGuard;
        // Run trees (PL)
        {
            RELIC_PROFILE_NAME("Consumer Trees - Update PL");
            SCOPED_MEMORY_ACCESS_WATCH(env::MP_SimCopy);
            SCOPED_MEMORY_ACCESS_WATCH(env::MP_SimOnly);
```

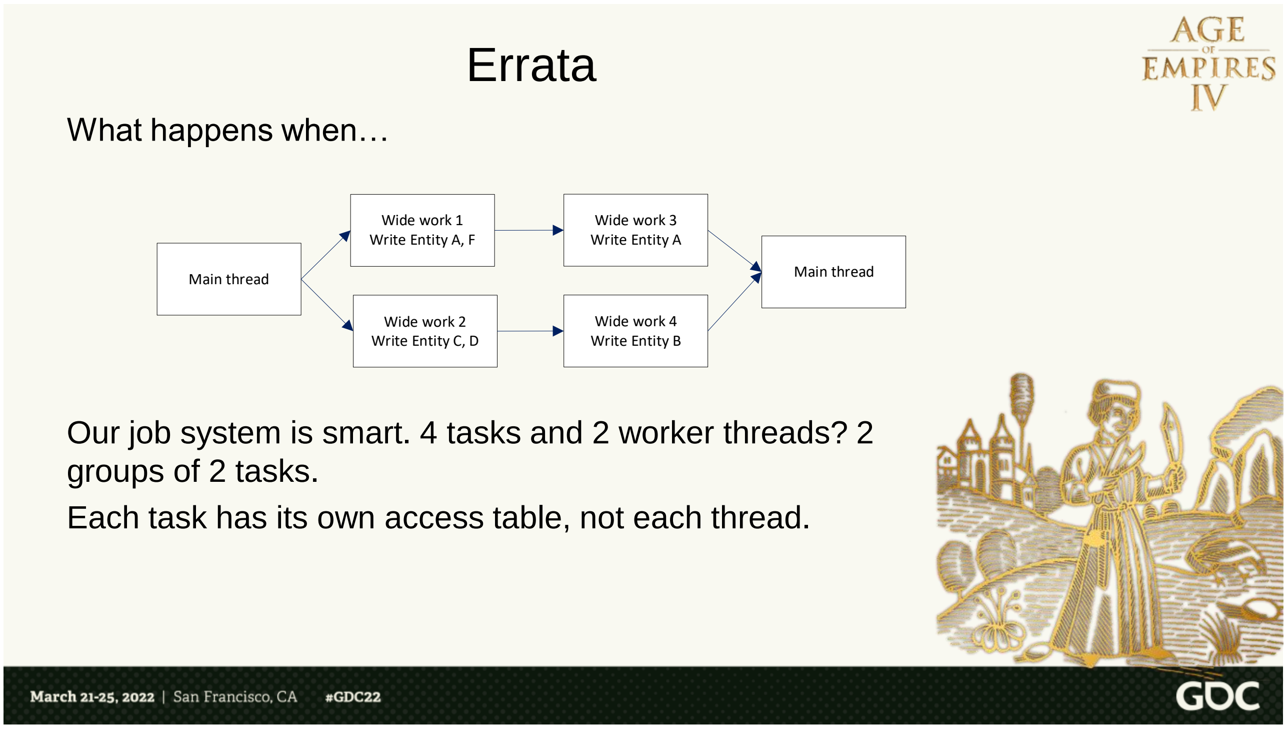
- Here we tell the system to monitor all trackable objects from 2 of our memory pools for code or jobs run in the current scope
- This is all that is required at runtime to turn the system on or off.
- We can also enable/disable per type with similar code

March 21-25, 2022 | San Francisco, CA #GDC22 GDC

MAW is off by default. If you were eagle eyed you'd have noticed in that last block of code, the very first thing MAW checks is whether its even enabled. These macros are how you toggle MAW. These enable MAW for the current scope on the specified pool and any tasks spawned during it. There are also similar macros to disable MAW on a per object type basis.

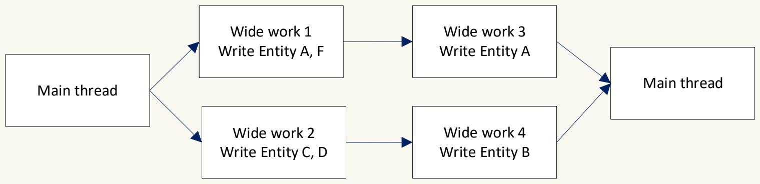
When the team was getting ready to

tackle parallelization for a new part of the update, we just had to drop these macros into our code for each new section as we were ready to make it thread safe. We’d tackle all the bugs that MAW highlighted then would move on to the next section.



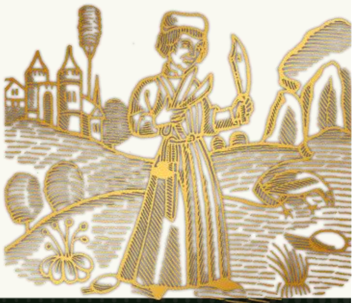
Errata

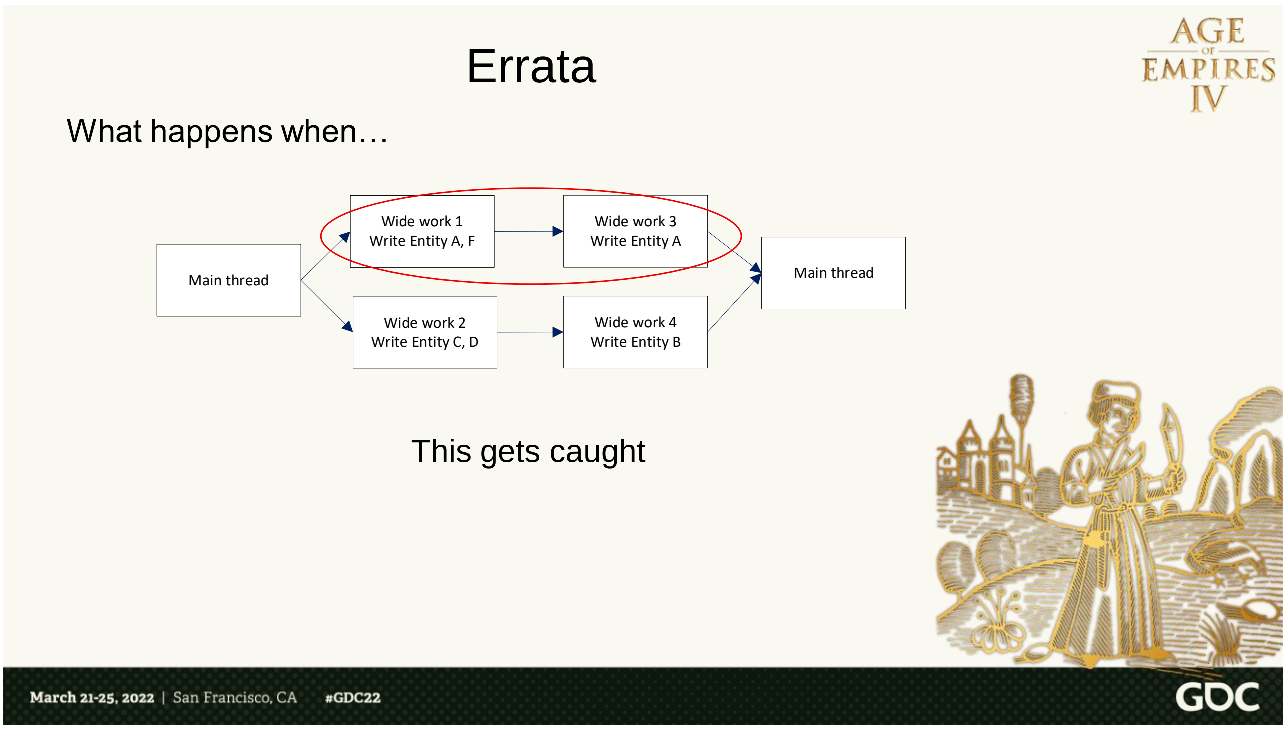
What happens when…



Our job system is smart. 4 tasks and 2 worker threads? 2 groups of 2 tasks.

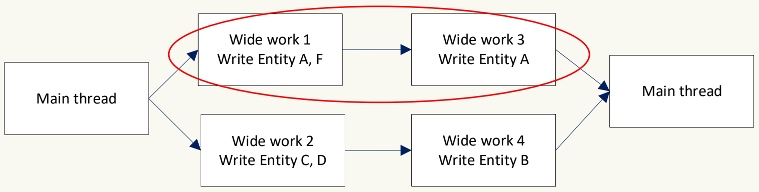
Each task has its own access table, not each thread.





Errata

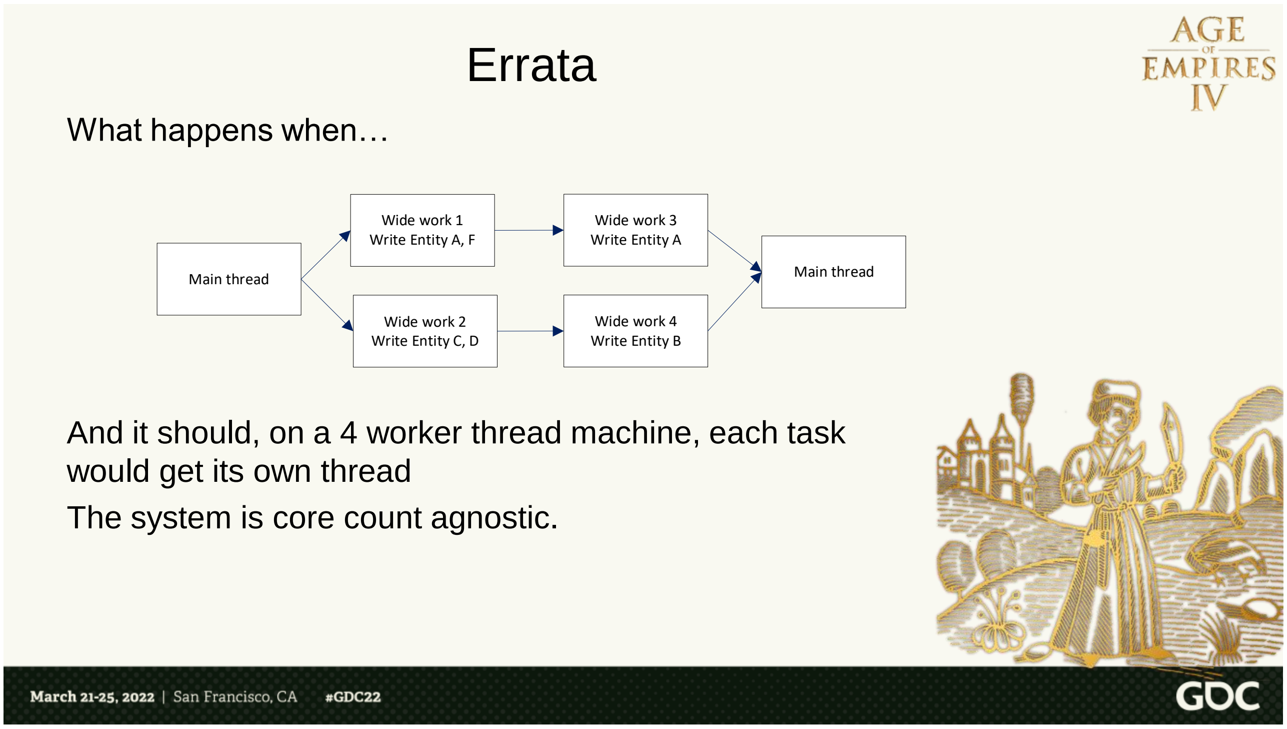
What happens when…



This gets caught

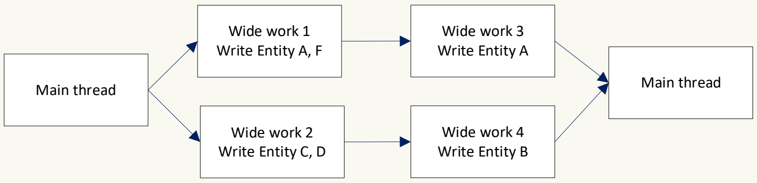
March 21-25, 2022 | San Francisco, CA #GDC22

Specifically, it gets caught after Tasks 316-319 finish executing and before 320 starts executing. This is when all of the access record tables are compared.



Errata

What happens when…

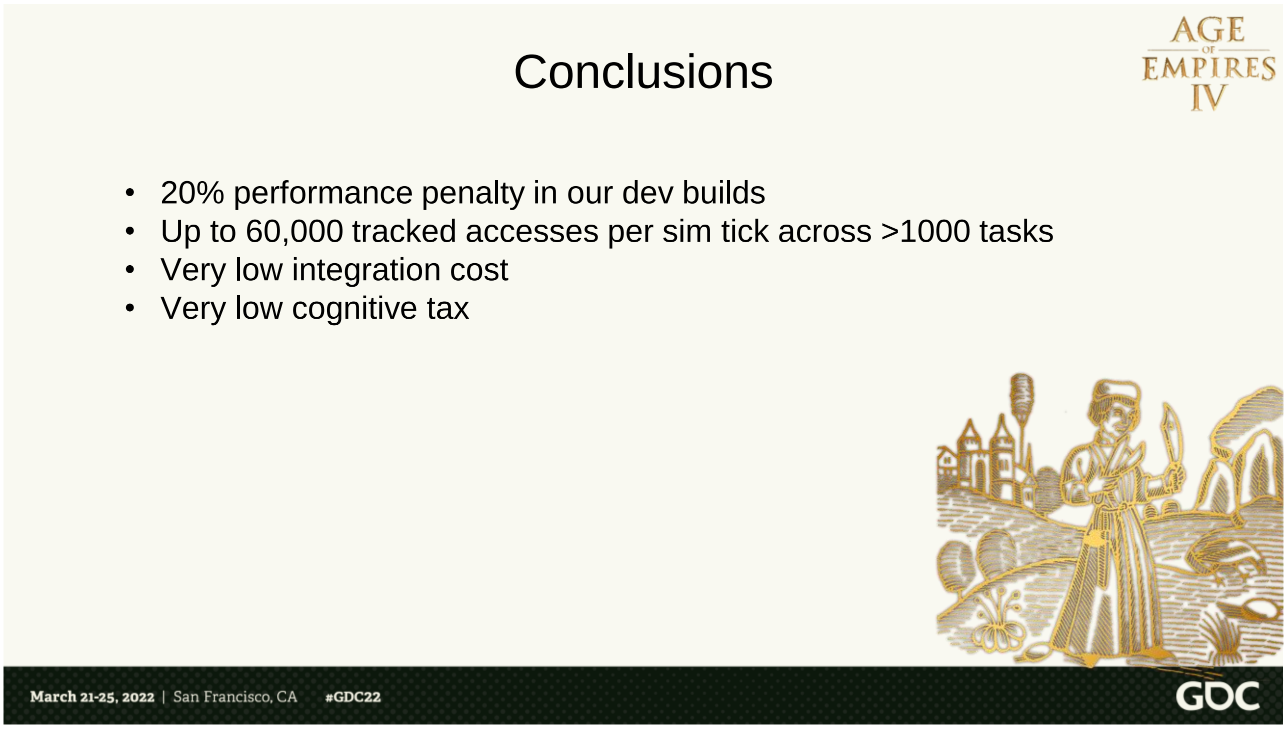


And it should, on a 4 worker thread machine, each task would get its own thread

The system is core count agnostic.

March 21-25, 2022 | San Francisco, CA #GDC22

You can extrapolate that the system would still catch errors if the tasks were processed in an entirely single threaded manner. This was an initial design goal, and we used it like this a lot to verify code could run in parallel before actually making it run in parallel. Barry Genova's 'Multithreading the entire Destiny engine' GDC talk he made a great point about not bringing down the entire team when you light up a system like this. So I made sure we could roll out incrementally in every way possible. We can start with a small subset of code and roll it out to new parts of the engine. We can also start with a small set of objects and add more as we get the original set under control.



Conclusions

- 20% performance penalty in our dev builds
- Up to 60,000 tracked accesses per sim tick across >1000 tasks
- Very low integration cost
- Very low cognitive tax

AGE OF EMPIRES IV
March 21-25, 2022 | San Francisco, CA #GDC22
GDC

Talking points
#1: dev builds were already about half as fast as RTM builds. So 20% more wasn't a big deal. Since our game was designed to run 4 player 800 unit games on a 10 year old ultrabook, we just limited dev build games on our super dev PCs to ~4 players and that worked mostly fine. One of the really important points of this talk, other than the fact the system worked is that we went from 10s per tick of overhead to single

digit milliseconds by making sure we paid as little for the systems existence as possible when we didn’t need it. It wasn’t compiled in on objects we never needed to track. And It was only enabled and doing heavy work during parts of our simulation update that we knew were hard to parallelize.
#2: Point number 2 here is just to illustrate the kind of volume the system was handling on average. There was no theoretical limit on the number of tracked accesses. That’s just the most I ever saw going through the system in a log or when it broke.

#3: The system was integrated at the lowest levels of our job and smart pointer systems. Required minimal setup. It integrated into the engine by a guy who wasn’t me, and we rarely had to go and modify the code in our sim loop that modify that.

#4: Gameplay engineers just wrote code, designers just made state tree logic and MAW told them when it was unsafe. Most engineers who started after we put the system in had no idea it was even there until it tripped.

Questions?

The Age family is growing!

WORLD'S EDGE

https://aoe.ms/careers

relic ENTERTAINMENT

https://www.relic.com/#careers

XBOX GAME STUDIOS

AGE OF EMPIRES IV

AGE of EMPIRES IV

End of talk