molecular matters

Live++:
A Bag of Tricks

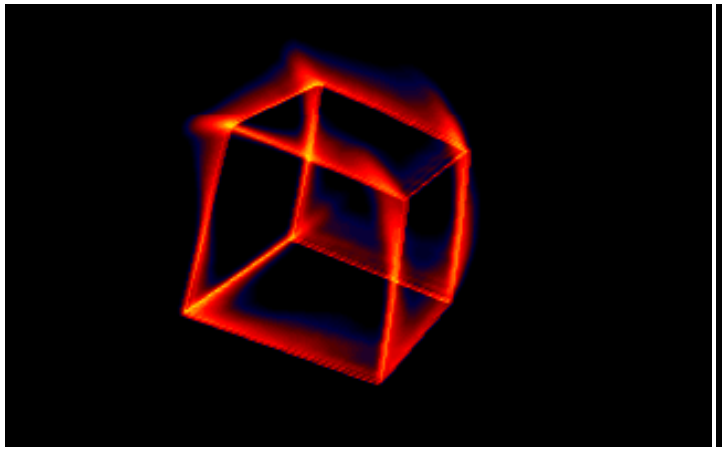
Guerrilla Games
April 10th, 2026

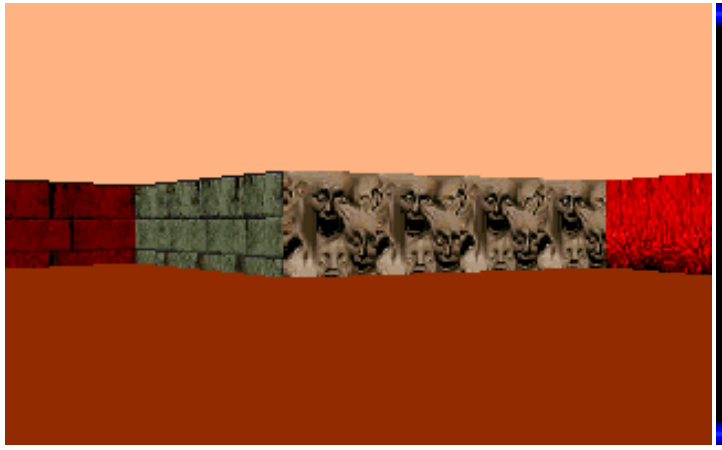
Stefan Reinalter, MSc – Founder/CTO @ Molecular Matters GmbH
https://liveplusplus.tech

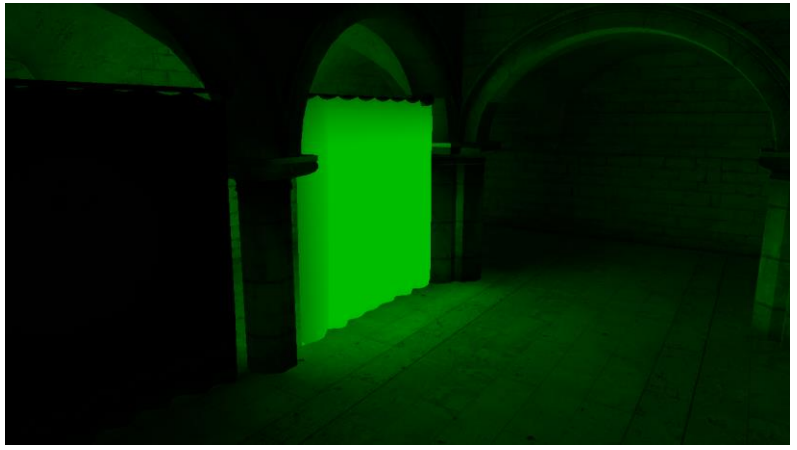
Introduction

- Stefan Reinalter
  - Founder/CTO/wearer of many hats @ Molecular Matters
- Started in the games industry in 2003 during original Xbox/PS2 era
  - 1989: First programming steps on C64, Logo, BASIC
  - 1995: Moved to MS-DOS, Pascal, 3d programming, x86 assembler
  - 1999: Moved to Windows, Delphi, DirectX programming
  - 2002: Started studying at Vienna University of Technology, finished in 2010
- Officially founded Molecular Matters in 2012
  - Worked on my own engine and a real-time radiosity solution
  - Freelance work (Ubisoft, Hitman PS3/Xbox360, undisclosed XB1 project)
  - Started teaching, blogging, writing for AltDevBlogADay
  - Contributed to Game Engine Gems 3
  - Launched Live++ in March 2018

Introduction







Technical

Scoped resources

- Let’s start with something easy
- RAII (Resource Acquisition Is Initialization)
  - Conceptually nice
  - Maps well to C++ using constructors & destructors
  - Useful for code with many return paths
    - Especially for locking/blocking resources such as synchronization primitives
  - (A bogus name for what it actually does…)

Scoped resources: RAII example

```
Concurrency::CriticalSection::ScopedLock::ScopedLock(Handle& handle) LPP_NO_EXCEPT
    : m_criticalSection(handle)
{
    Enter(handle);
}


Concurrency::CriticalSection::ScopedLock::~ScopedLock(void) LPP_NO_EXCEPT
{
    Leave(m_criticalSection);
}
```

```
// take socket ownership globally
synchronizedSocket->AddReference();
{
    Concurrency::CriticalSection::ScopedLock lock(g_accessSynchronizedSocketsCS);

    g_synchronizedSockets->InsertLastByCopy(synchronizedSocket);
}
```

Scoped resources

- While RAII is nice, it also:
  - Is tedious to write small classes for every object
  - Leads to lots of boilerplate
- Would be nice to decouple “resource acquisition” from “resource cleanup”
- Solution: Defer
  - Proposal for next C standard
  - Not available in C++, but other languages (Go, Zig)
  - Let’s craft our own minimal-overhead version

Defer: preprocessor helpers

```
// joins two tokens, even when the tokens are macros
#define LPP_PP_JOIN_HELPER_HELPER(_0, _1)    _0##_1
#define LPP_PP_JOIN_HELPER(_0, _1)           LPP_PP_JOIN_HELPER_HELPER(_0, _1)
#define LPP_PP_JOIN(_0, _1)                  LPP_PP_JOIN_HELPER(_0, _1)

// stringizes a token, even when the token is a macro
#define LPP_PP_STRINGIZE_HELPER(_0)          #_0
#define LPP_PP_STRINGIZE(_0)                 LPP_PP_STRINGIZE_HELPER(_0)

// generates a unique name
#define LPP_PP_UNIQUE_NAME(_name)            LPP_PP_JOIN(_name, __LINE__)
```

Defer: implementation (C++17)

```cpp
template <class T>
class LPP_NO_DISCARD DeferrableFunction
{
public:
    inline DeferrableFunction(T&& closure) LPP_NO_EXCEPT
        : m_closure(LPP_MOVE(closure))
    {
    }

    // Calls the deferrable function.
    inline ~DeferrableFunction(void) LPP_NO_EXCEPT
    {
        m_closure();
    }

private:
    T m_closure;

    LPP_DISABLE_COPY_MOVE(DeferrableFunction);
};

// CTAD deduction guide
template <class T>
DeferrableFunction(T&& closure) -> DeferrableFunction<T>;


// Macro that imitates Go's "defer" statement, calling the deferred statement at scope exit automatically.
#define LPP_DEFER const DeferrableFunction LPP_PP_UNIQUE_NAME(autoDefer) = [&](void)
```

Defer: slighty different implementation (C++11)

```cpp
template <class T>
class LPP_NO_DISCARD DeferrableFunction
{
public:
    inline explicit DeferrableFunction(T&& closure) LPP_NO_EXCEPT
        : m_closure(LPP_MOVE(closure))
    {
    }

    // Calls the deferrable function.
    inline ~DeferrableFunction(void) LPP_NO_EXCEPT
    {
        m_closure();
    }

private:
    T m_closure;

    LPP_DISABLE_COPY_MOVE(DeferrableFunction);
};

// Helper struct that creates deferrable functions using operator<<.
// This allows us to not having to know the closure's type in LPP_DEFER macro invocations, effectively "erasing" the closure's type.
struct LPP_NO_DISCARD DeferrableFunctionMaker
{
    template <class T>
    LPP_NO_DISCARD inline constexpr DeferrableFunction<T> operator<<(T&& closure) const LPP_NO_EXCEPT
    {
        return DeferrableFunction<T>(LPP_FORWARD(closure));
    }
};


// Macro that imitates Go's "defer" statement, calling the deferred statement at scope exit automatically.
#define LPP_DEFER const auto LPP_PP_UNIQUE_NAME(autoDefer) = DeferrableFunctionMaker() << [&](void)
```

Defer: usage

- Can be nicely mixed with C-like code, e.g. Win32 API
- Especially when returning RVO-ed objects from C-like data:

```
LPP_NO_DISCARD Filesystem::WidePath UserProfile::GetLocalAppDataDirectory(void) LPP_NO_EXCEPT
{
    wchar_t* localAppDataPath = nullptr;
    OS::Windows::SHGetKnownFolderPath(OS::Windows::FOLDERID_LocalAppData, 0u, nullptr, &localAppDataPath);

    LPP_DEFER
    {
        OS::Windows::CoTaskMemFree(localAppDataPath);
    };

    return Filesystem::WidePath(localAppDataPath);
}
```

Defer: usage

- Turns RAII into:
  - if (Acquire())
    - Defer { Cleanup(); }

```
BuildSystem::DependencyFile dependencyFile;
if (dependencyFile.Open(dependencyFilePath.GetString()))
{
    LPP_DEFER
    {
        dependencyFile.Close();
    };

    if (dependencyFile.ParseOutputFile())
    {
        const NarrowStringView outputFilePath = dependencyFile.GetOutputFile();
        const Filesystem::NarrowPath outputFilename = Filesystem::GetFilename(outputFilePath);

        m_filenameToAbsolutePath.InsertByMoveCopy(NarrowString(outputFilename.GetString(), outputFilename.GetLength()), absolutePath);
    }
    else
    {
        LPP_LOG_ERROR(LPP_TID(log_failed_to_parse_output_file_in_dependency_file), dependencyFilePath.GetString());
    }
}
```

Self-relative data structures: what?

- A data structure with one or more members which are relative to:
  - The base of the data structure
    - Often seen in file formats: every offset is relative to start of the file
    - Flexible array member in C (and in C++, but not standardized)

```
struct BinaryTexture2D
{
    RHI::TextureFormat format;
    unsigned int channelCount;
    unsigned int width;
    unsigned int height;
    unsigned int mipMapCount;
    DM_FLEXIBLE_ARRAY_MEMBER(std::byte, data);
};
```

```
// Defines a C-like flexible array member.
#define DM_FLEXIBLE_ARRAY_MEMBER(_type, _name)                \
    DM_PUSH_WARNING_MSVC                                      \
    DM_PUSH_WARNING_CLANG                                     \
    DM_DISABLE_WARNING_MSVC(4200)                             \
    DM_DISABLE_WARNING_CLANG("-Wc99-extensions")              \
    DM_DISABLE_WARNING_CLANG("-Wmicrosoft-flexible-array")    \
    _type _name[]                                             \
    DM_POP_WARNING_MSVC                                       \
    DM_POP_WARNING_CLANG
```

  - The address of the member itself

Self-relative data structures: why?

- Advantages:
  - Dynamically sized, but require only a single allocation
  - Trivially relocatable
    - memcpy
  - Trivially serializable
    - memcpy
- Disadvantages:
  - Often require pointer fix-ups
    - E.g. convert offsets in file format to pointers to actual data structure
- Can we have the advantages without requiring any pointer fix-up?
  - Yes!
    - By using this-relative pointers

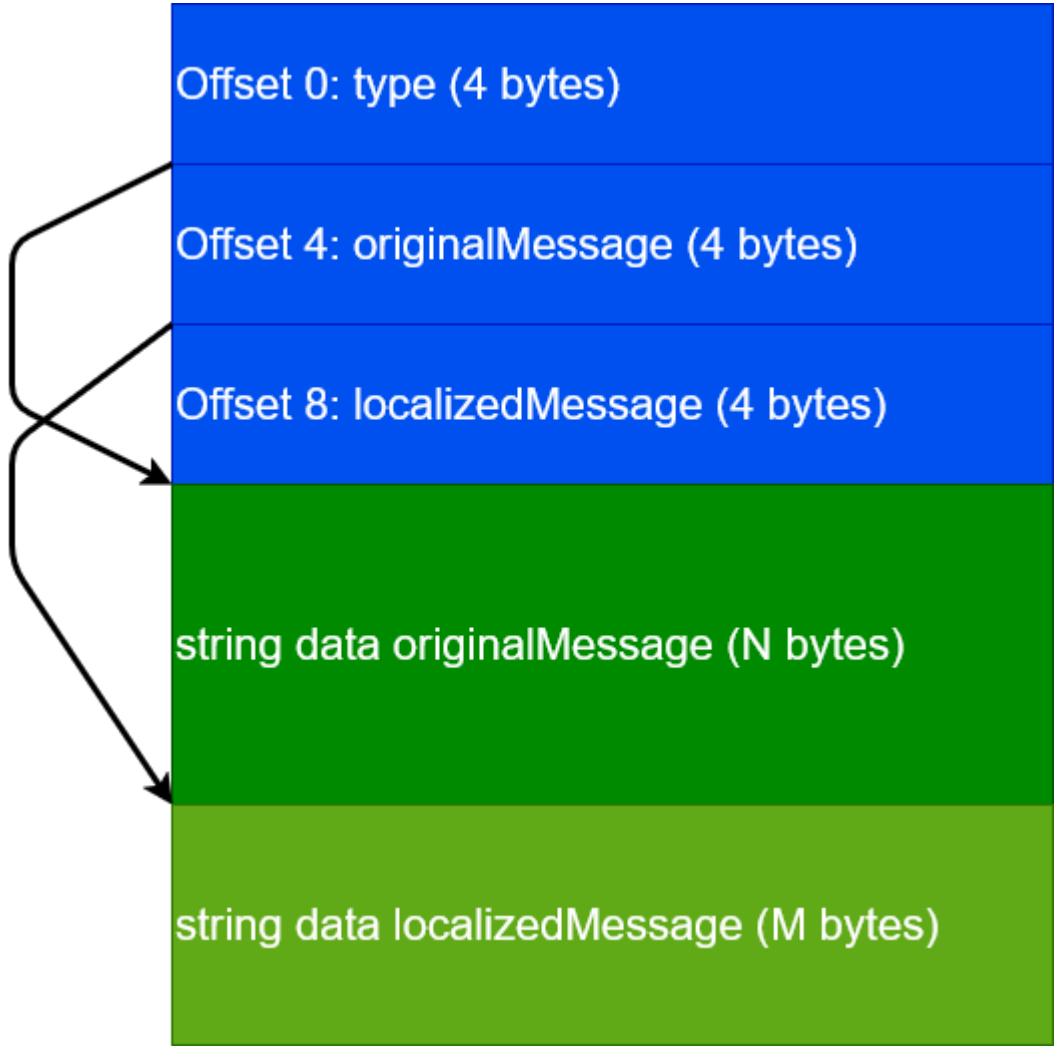
Self-relative data structures: how?

- This-relative pointer
  - A **relative** pointer, relative to **itself**
  - Offset is calculated based on its own location in memory

Self-relative data structures: how?

- Memory layout

```
LPP_DEFINE_BLOB
(
    LogNarrow,

    enum class Type : uint32_t
    {
        Debug,
        Success,
        Info,
        Warning,
        Error,
        Panic
    };

    Type type;
    ThisRelativePointer<char> originalMessage;
    ThisRelativePointer<char> localizedMessage;
);
```



Self-relative data structures: how?

- This-relative pointer
  - A **relative** pointer, relative to **itself**
  - Offset is calculated based on its own location in memory

```cpp
template <typename T>
class LPP_NO_DISCARD ThisRelativePointer
{
public:
    inline explicit ThisRelativePointer(const T* ptr) LPP_NO_EXCEPT
        : m_offset(0u)
    {
        LPP_ASSERT(reinterpret_cast<uintptr_t>(ptr) >= reinterpret_cast<uintptr_t>(this), "Offset needs to be positive.");

        m_offset = static_cast<uint32_t>(reinterpret_cast<uintptr_t>(ptr) - reinterpret_cast<uintptr_t>(this));
    }

    LPP_NO_DISCARD inline T* Resolve(void) LPP_NO_EXCEPT
    {
        return reinterpret_cast<T*>(reinterpret_cast<uintptr_t>(this) + m_offset);
    }

private:
    // the offset from the 'this' pointer to the actual address in memory
    uint32_t m_offset;
};
```

  - **Resolve()** is remarkably simple: **mov** from memory, followed by **add**

Self-relative data structures

- Used for all network blobs in Live++

```
using BlobType = Blob::LogNarrow;
Owner<Blob::TypedBlob<BlobType>> typedBlob = Blob::Builder::Create<BlobType>
(
    Blob::Builder::DefineMember(&BlobType::type, type),
    Blob::Builder::DefineStringMember(&BlobType::originalMessage, originalView.Decay()),
    Blob::Builder::DefineStringMember(&BlobType::localizedMessage, localizedView.Decay())
);

LPP_DEFER
{
    Blob::Builder::Destroy(typedBlob);
};

SendToBridge(LPP_MOVE(typedBlob));
```

- Building blobs is performed by a variadic helper template
  - **Blob::Builder::Create<T>**
    - Calculates total size of blob
    - Initializes blob members one by one using fold expressions

String interning: why?

- In Live++, we need to:
  - Store millions of (very long) strings
    - Paths
    - C++ symbol names
  - Compare millions of (very long) strings
    - “Does symbol N in the patch already exist in the original executable?”
    - Large performance overhead in Live++ 1.x.x
- There is no upper limit to the number of strings
  - Indie title vs. Fortnite
- There is no upper limit to how many processes and executables/modules are registered with Live++
  - Processes are likely to share paths
  - Modules are likely to share paths and symbol names

String interning: how?

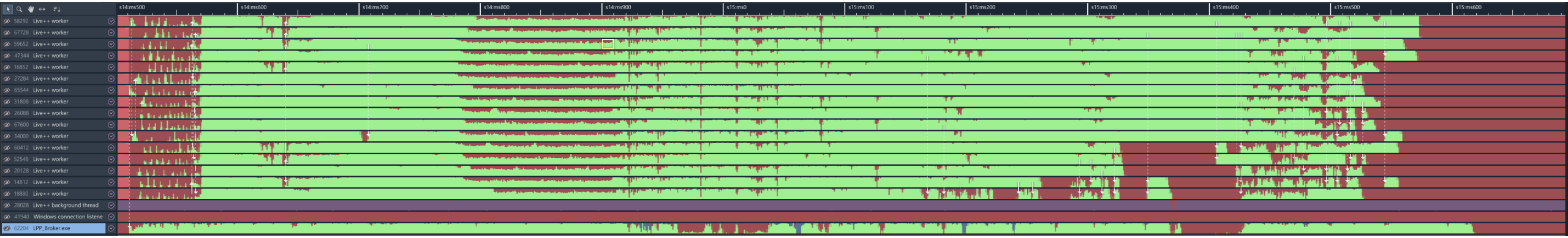
- Important observation #1:
  - The internal databases for a process can only grow
  - Nothing ever gets deleted
    - Original executable has N symbols
    - Patch has much fewer symbols…
    - …but every future patch can refer to any of the N symbols
- Important observation #2:
  - Databases are “initialize once, lookup & compare often”
  - Initialization performed during PDB loading
  - Lookup & compare performed for each symbol in every patch
- Basic idea:
  - Make lookup & compare as fast as possible, O(1)
  - Make initialization as fast as possible given our constraints

String interning: how?

- A hash map <name, symbol> is an obvious choice for symbol lookup
  - Resolving collisions requires (potentially expensive) symbol name comparison
  - Interned strings can give us O(1) comparison
- String interning requires us to identify unique strings
  - Which in turn requires a hash set
- Basic steps of **Intern(string)** operation:
  - Ask hash set: does the pool of interned strings already contain **string**?
    - Yes: nothing to store
      - Return a value that uniquely identifies the existing interned string
    - No: allocate space in the pool for storing the interned string
      - Return a value that uniquely identifies the newly interned string

String interning: difficulties

- Live++ is heavily threaded, especially initial loading



- String interning has at least two choke points
  - Hash set lookup and insertion for checking uniqueness of strings
    - Insertion needs to lock the container
    - Major source of contention
  - Memory allocation for storing interned strings
    - Generic allocator another source of contention

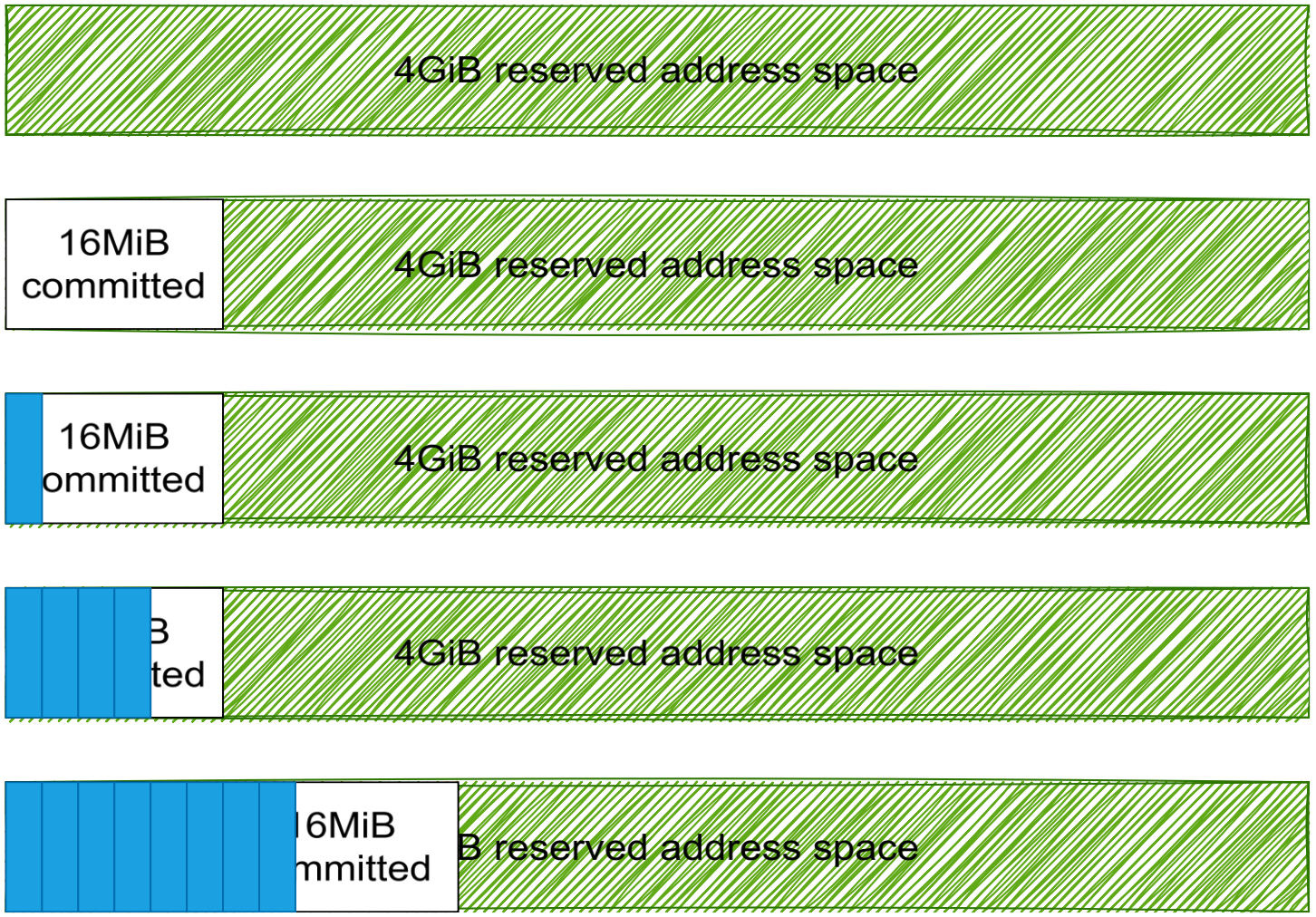
String interning: resolving hash set contention

- Hash set checking for uniqueness can be **sharded**
  - Remember:
    - Same strings == same hashes
    - Different strings == probably different hashes, but could be the same (collision)
- Sharding doesn't break what we are testing for
- Use N hash sets instead of a single global hash set
  - As long as different threads ask different sets, there is no contention
  - Determine shard index based on string hash
    - The hash is required for lookup and insertion anyway, so no additional overhead
  - Careful!
    - How are your hashes calculated?
    - How does your hash container work internally?
      - E.g. Swiss Tables use the lowest 7 bits of the hash for metadata, so **do not use** the hash's LSB as shard index

String interning: resolving allocation contention

- Start with the fastest allocator possible: a bump/linear allocator
  - Trivial to implement lock-free using atomic operations
- Do not know amount of required memory a priori...
- ...but can probably define a reasonable upper bound?
- Idea:
  - Reserve upper bound of address space for the whole string pool
  - Grow committed memory in N pages on demand
  - Use 4GiB (2^{32}) as upper bound
    - Each interned string lives somewhere in this 4GiB range
    - Each interned string is **uniquely identified by a 4-byte offset**
    - Testing two interned strings for equality becomes a single 4-byte comparison
  - Reminder: we would need >4GiB of **unique strings** to break this limit

String interning: resolving allocation contention



String interning: resolving allocation contention

- Implemented using Windows' virtual memory API
  - **VirtualAlloc(MEM_RESERVE)** for reservation
  - **VirtualAlloc(MEM_COMMIT)** for committing pages
- Bump allocator is basically lock-free except when committing new pages
- Initialization:

```
StringPoolAllocator::StringPoolAllocator(void) LPP_NO_EXCEPT
    : m_srwLock(Concurrency::SlimReaderWriterLock::INVALID_HANDLE)
    , m_memory(static_cast<char*>(VirtualMemory::ReserveAddressSpace(Unit::GiB(4ull))))
    , m_neededMemory(0u)
    , m_committedMemory(GrowSize)
{
    Concurrency::SlimReaderWriterLock::Create(m_srwLock, "StringPool allocator");

    // allocate a few MB of memory right away. it doesn't make sense to let the pool grow slowly
    if (!VirtualMemory::AllocatePhysicalMemory(m_memory, static_cast<uint64_t>(m_committedMemory)))
    {
        LPP_LOG_ERROR(LPP_TID(log_failed_to_ensure_allocated_memory), static_cast<uint64_t>(m_committedMemory));
    }
}
```

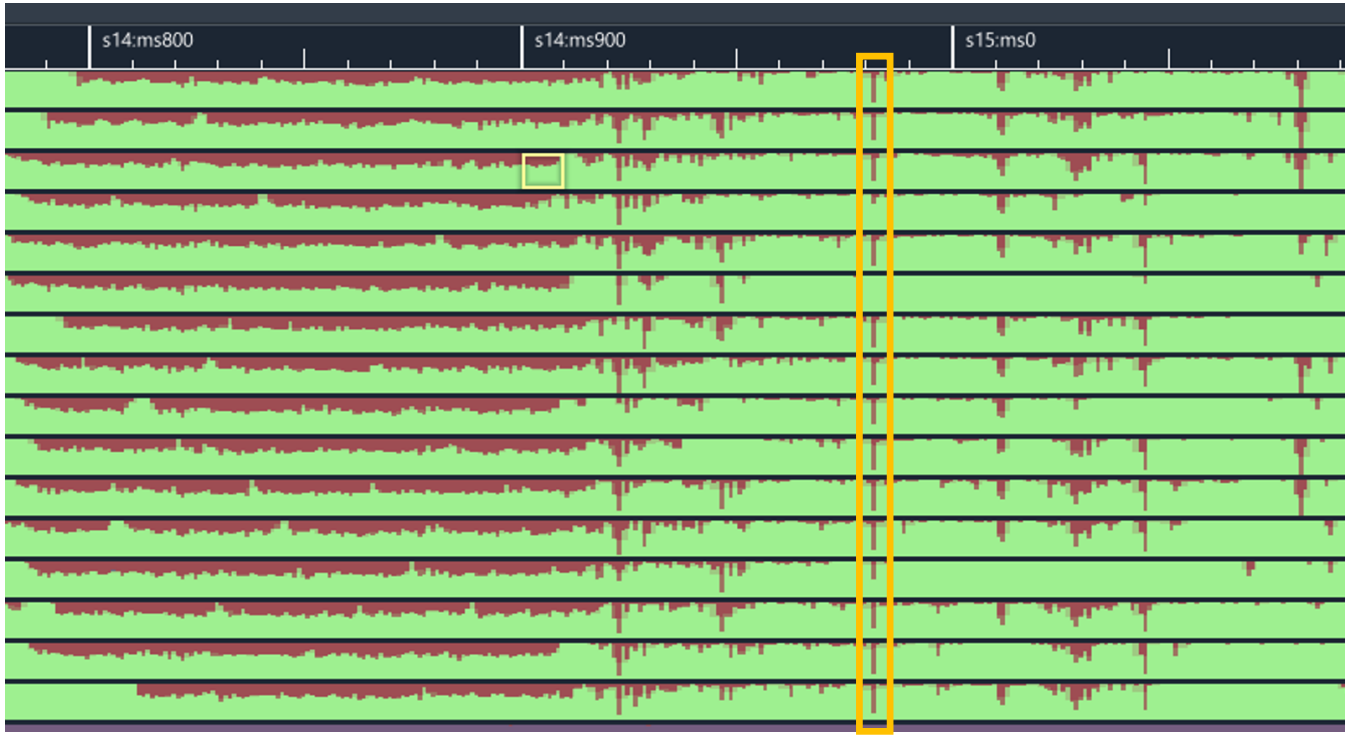
String interning: resolving allocation contention

- Allocation:

```
LPP_NO_DISCARD void* StringPoolAllocator::Allocate(size_t size) LPP_NO_EXCEPT
{
    // as long as there is still enough memory committed, the allocation is lock-free
    const uint64_t currentOffset = m_neededMemory.AddSequentiallyConsistent(size);
    const uint64_t currentlyCommittedMemory = static_cast<uint64_t>(m_committedMemory);
    if (currentOffset + size > currentlyCommittedMemory)
    {
        // need to commit additional memory, so take the lock
        Concurrency::SlimReaderWriterLock::ScopedLockExclusive lock(m_srwLock);

        // several threads might have tried to allocate at the same time, with one of them entering this code path, and the others waiting for the lock.
        // since the thread making progress will take care of committing more memory, we need to check again whether the original condition is still true,
        // to avoid calling VirtualMemory::AllocatePhysicalMemory too often.
        while (currentOffset + size > static_cast<uint64_t>(m_committedMemory))
        {
            const uint64_t value = static_cast<uint64_t>(m_committedMemory);
            if (!VirtualMemory::AllocatePhysicalMemory(m_memory, value + GrowSize))
            {
                LPP_LOG_ERROR(LPP_TID(log_failed_to_ensure_allocated_memory), value + GrowSize);
                return nullptr;
            }

            // signal the change to other threads only *after* the memory has been committed
            m_committedMemory.AddSequentiallyConsistent(GrowSize);
        }
    }

    return m_memory + currentOffset;
}
```

String interning

- Live++ uses:
  - 256 hash set shards
    - Each shard has its own slim reader-writer lock
    - Hash set optimized for insertion & lookup, does not support deletion
  - Bump allocator
    GrowSize of 16 MiB



Non-technical

How I (try to) work

- Mostly work on difficult technical problems these days
  - Both Live++ and my Next Thing
- Used to **really** get my teeth into these problems
  - Returning to the office after dinner with friends, staying until 4am
  - Working 80+ hours per week for **years**
- I am no longer in a position to work even 8 hours a day
- Need to make the best of my time
- What to do?

How I (try to) work

- Our brains are wonderful problem solvers
- Let the brain do its thing on a background thread
  - You probably know these “had an idea under the shower” moments
- There is research behind this, called **“Incubation effect”**
- **Pre-incubation**: initial attempt at solving a problem, but you get stuck
- **Incubation**: taking a break from the problem
- **Post-incubation**: returning to the problem with a “fresh pair of eyes”
- But also:
  - Having an epiphany under the shower
  - Having an idea for solving a difficult problem at dinner
  - “My background thread just called” has become a thing in our household

How I no longer work

- Trying to solve a tremendously difficult problem over a span of several hours or even days
  - Come prepared, having done my research about the topic at hand
  - Having an initial idea, trying that
    - Failing
  - Having another idea, trying that
    - Failing again
  - …
  - More relentless hacking and trying
  - Not giving up
  - Not taking a break
  - “I need to solve this **today**!”

How I work nowadays instead

- Trying to solve a tremendously difficult problem over a span of several hours or even days
  - Come prepared, having done my research about the topic at hand
  - Having an initial idea, trying that
    - Failing
  - Having another idea, trying that
    - Failing again
  - Realizing “I won’t solve this **today**”
  - Going for a walk, going for a run
  - Playing with my cats
  - Scribbling ideas using pen and paper
  - Working on something entirely different
    - Even a completely different project
  - **Working in focused one-hour bursts**

Final thoughts

- Disclaimer: probably doesn't apply to everybody
  - Being an employee is a different setting than running a business
- Still, I **strongly** urge you to give it a try the next time you're stuck
  - If you can, switch to an easy task in a completely different part of the codebase
  - If you can, switch to a completely different project
  - Get out, enjoy a walk
    - This is not procrastinating, you're still working – your brain is doing the work subconsciously
  - All of these reduce mental fixation on a single particular problem
- It takes getting used to
  - But I truly believe this is a learnable skill

THANK YOU!

X (Twitter)
@molecularmusing
@liveplusplus

Bluesky
@molecularmusing.bsky.social
@liveplusplus.bsky.social

Mastodon
@molecularmusing@mastodon.gamedev.place
@liveplusplus@mastodon.gamedev.place

Bonus Technical

Canonical paths and file attributes: why?

- Paths are a mess
  - PDBs use a mix of absolute paths, relative paths, mixed-case paths, lower-case paths, …
  - “D:\Folder\A\..\file.bin”
  - “D:\Folder\A\B\..\..\file.bin”
  - “D:\folder\FILE.bin”
  - “E:\reparsePoint\file.BIN”
- They all point to the same path underneath
  - Redundant strings in our pool
  - The bigger problem: symbol lookup
    - Names for certain symbols with internal linkage are suffixed
    - Conceptually, suffix is path of the object file, e.g. **“@mySymbol%D:\folder\file.obj”**
    - Actually stored with offset of the interned string, e.g. **“@mySymbol%01234567”**

Canonical paths and file attributes: why?

- Live++ needs canonical paths for:
  - Source files
  - Object files
  - EXE/DLL/PDB files (on Xbox)
- Live++ has to perform potentially hundreds of thousands “Does this file exist?” tests during initialization for finding .cpp, .obj and .lib
  - There are no rules for how toolchains should store paths inside the PDB
  - Absolute paths? Those are usually fine
  - Relative paths? Need to try different combinations of:
    - Compiler/linker working directory
    - Relative to EXE/PDB
    - Static libraries?
    - Compiled translation unit vs. linked translation unit

Canonical paths and file attributes: how?

- Live++ also needs the creation time and modification time
  - File existence tests can be handled via timestamps
- Good news: there are Win32 APIs
  - **GetFileAttributesEx**
    - No handle required
  - **GetFinalPathNameByHandle**
    - As the name implies, requires a file handle
- Bad news: the Windows filesystem is notoriously slow
  - For AAA projects, these API calls add up
- Important observation:
  - We have a number of files F stored in a number of directories D
  - On average, #D is much smaller than #F

Canonical paths and file attributes

- Spelunking around at http://undocumented.ntinternals.net/index.html
  - **NtQueryDirectoryFile**
    - Requires a directory handle
    - Returns 64KiB worth of **FILE_DIRECTORY_INFORMATION** data
      - Case-correct filenames
      - Timestamps
- Feed two birds with one scone!
- Strategy employed by Live++ during initialization:
  - Gather unique directories from all PDB paths via hash set
  - Prime an **AttributeCache** with all unique directories
    - **GetFinalPathNameByHandle** used on directory path
    - **NtQueryDirectoryFile** used for directory files
  - Consult **AttributeCache** instead of Windows filesystem

Sidenote: Windows API regressions

- There have been several API performance regressions over the years
- Live++ now uses all of these:
  - NtReadVirtualMemory
  - NtWriteVirtualMemory
  - NtQuerySystemInformation
  - NtQueryInformationProcess
  - NtQueryVirtualMemory
  - NtQueryDirectoryFile
  - NtQueryInformationFile
- And also these, but for other reasons:
  - NtSuspendProcess
  - NtResumeProcess
  - NtContinue