Elsewhere Memory

Niall Douglas

Contents:

1. Background:
   a. The C++ 20 object and memory model (addressing, not concurrency, scheme)
   b. Virtual Memory (simplified)
   c. Memory Elsewhere to ‘here’
2. Possible futures for C 2x and C++ 2y:
   a. WG21 P1631 *Object attachment and detachment*
   b. *WG21 P???? Page-based object storage*
   ~~c. WG14 N2362 *Moving to a provenance-aware memory model for C2x*~~

The C++ 20 object and memory model

(addressing not concurrency memory model)

C++ 20 abstract machine

- (Unfair but not untrue) Basically requires that implementations emulate the *observable behaviour* of the original PDP-11/20 mainframe:
  - Memory is a single flat space with equal access latency, and all parts are [equally] reachable
  - Every live object has a single, unique address within that memory which can be referred to by a pointer to that type (or `void*`)

C++ 20 abstract machine

- The totality of the live objects within that memory represent the program's current, valid, state
- Programs are a time-incremental stream of *sequence points* where objects are transformed from one *valid state* into another *valid state* through the application of *operations* upon those objects
  - Sequence points are barriers preventing reordering of operations IF those operations have observable side effects

C++ 20 abstract machine

- Only the program may apply operations to objects
  - *There is only one C++ program*
  - *Objects cannot be the program itself*
- Memory consists of bytes, therefore object storage is always [zero or] one *or more* arrays of bytes
- Thus objects are not stored contiguously, though arrays of objects are *indexable* contiguously
  - Thus we get structure padding!
  - Thus implementation of some types of object is utterly dependent on the running C++ program

C++ 20 abstract machine

- Objects have (paraphrased) lifetime one of:
  - Program-lifetime duration
  - Thread-lifetime duration
  - Stack frame-lifetime duration
    - Note: no stack is defined in C++, but *stack unwinding* is!
  - Expression-lifetime (temporary) duration
  - Programmer-managed (i.e. `malloc`)
- Thus no object lifetime exceeds the program

C++ 20 abstract machine

- Concurrency model assumes:
  - Data shared between program threads could always mutate (i.e. const is *soft-const*, can always be cast off)
  - There is only one C++ program at a time
- Thus if given the same i/o, all valid C++ programs always have identical *observable* behaviour
- BUT may have different valid states at sequence points between program start and end
  - I.e. road destination != road travelled

Further reading

- CppCon 2017: Patrice Roy “Which Machine Am I Coding To?” https://www.youtube.com/watch?v=KoqY50HSuQg

Virtual Memory (simplified)

Virtual Memory - Overview

- The default memory abstraction mechanism on all the major desktop and mobile operating systems for over 20 years
  - And a large chunk of high end embedded systems e.g. QNX
- Originates from the 1970s when low latency (10e-6) memory was expensive (~$4/byte), and high latency (10e-1) memory was much less so (~$0.001/byte)
  - The ability to just-in-time substitute high latency for low latency memory was key to building reliable systems
- Was controversial when first proposed (1970), it has 'won' and is now fully standardised into POSIX.2017

Virtual Memory - Pages

- The 64-bit address space is composed of varying size *pages*
  - On x64: 4Kb, 2Mb, 1Gb, (512Gb)
- The CPU’s MMU maps a physical page of RAM to an address, using kernel-maintained *page tables* to look up each mapping
- A page can be marked fault-never, fault-on-read, fault-on-write, fault-always.

Virtual Memory - Private pages

- Virtual Memory is *lazy*:
  - Unallocated address space is simply always-fault pages, fault handler kills the process
  - New memory allocations from the system simply map the single all-bits-zero system page repeatedly
  - First write faults, causing a real empty page to be allocated and mapped into the process
  - Actual RAM (data) consumption of a process is the total number of written-at-least-once pages

Virtual Memory - Files

- Kernels read and write files from storage in whole (4Kb, occasionally 2Mb) pages
- The kernel page cache is those parts of files currently cached in RAM
- The kernel page cache can be mapped into processes (memory mapped files)
- Single, unified, page cache architecture has ‘won’, all major kernels use it

Virtual Memory - Files

- Same lazy fault-driven mechanism applies as for allocating new memory
  - At process start, *none* of an executable binary may actually be mapped into the process (pages all marked fault-on-read)
  - Each time a page is first read, only *then* is that page read from the storage device
  - Every filled page is marked fault-on-write. On the first write, the *dirty* bit is set for that page

Virtual Memory - Files

- Every X seconds, all dirty pages are written to storage, reset to fault-on-write and dirty bit cleared
- `read`() and `write`() do `memcpy`() to the exact same kernel page cached pages as when a process does `memcpy`() to memory mapped files
- Actual RAM (code) consumption of a process is the total number of read-at-least-once pages

Virtual Memory - Swap file

- Private pages (fresh memory allocations), on first write and therefore actual allocation, usually have space reserved for them in the system swap file
- Then if physical RAM runs low, a not-recently used private page can be placed in swap, and the physical RAM page used more productively elsewhere

Virtual Memory - Swap file

- One only ‘runs out of memory’ when all the space in the swap file is consumed
  - Long before that the system may slow to a crawl
  - Even on very fast storage (12Gb/sec), anything PCIe connected will be at least 20x worse latency than main memory (~500ns vs ~25ns)
  - And all the memory copying (2µs/page) and TLB shootdown (1µs)!
  - And spinning rust storage is more like 10,000x higher latency again!

Further reading

- ‘What every programmer should know about memory’ by Ulrich Drepper

  https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

Elsewhere Memory

Elsewhere Memory - What is it?

- There is an increasing trend in computing of *ever more dedicated-purpose CPUs*
  - SSDs and shingled hard drives have two or three medium spec ARM CPUs each equal to a high end mobile phone a few years ago
  - Graphics CPUs long ago became very powerful
  - >= 10Gbps NICs tend to have non-trivial CPUs
  - Even a USB controller has significant memory bandwidth - think USB 3.0

Q: What is becoming ever more important in these bundles of dedicated-purpose CPUs?

A: Getting state between these CPUs

How do we share memory across CPUs?

Elsewhere Memory - Accessing it

Main memory sharing mechanisms:

1. Copy-based (usually kernel-implemented)
   - Memory elsewhere is copied to local memory by software using PIO or DMA as needed
   - Optionally ‘oplock’ cached in the local CPU’s memory
     - With optional remote invalidation (i.e. remote knows who has/could have a copy of its memory, and says when it has been modified)

Elsewhere Memory - Accessing it

Main memory sharing mechanisms:

2. Directly mapped (as-if memory)
    - Memory elsewhere appears as-if main memory to the CPU
    - Some memory is higher latency (‘further away’) than other memory
    - Fits well into NUMA software design strategies
    - BUT makes atomic operations *particularly* expensive - coarse grained synchronisation is best

Elsewhere Memory - Accessing it

Main memory sharing mechanisms:

3. Offload of whole chunks of kernel implementation to custom hardware (i/o)
    - Move graphics rasterisation into dedicated device (GPUs)
    - Move kernel page cache into dedicated device (Intel Optane SCM, 50ns vs 2000ns)
    - Move file system into dedicated device (Samsung KV-SSD edition of Z-SSD, 30μs vs 170μs)

(Elsewhere Memory - Confounding)

- (The OS kernel may emulate Directly Mapped using PIO/DMA and a kernel cache, so user mode code sees Directly Mapped
  - E.g. Memory mapped files)
- (OR the OS kernel may hide Directly Mapped, so user mode code sees only copy-based
  - E.g. socket i/o on high end NICs)

Problem to solved:

The C++ abstract machine needs to be taught:

1. That some memory is shareable
2. Shared memory can be modified outside the current C++ program by others
3. Some objects are shareable
   - Either trivially, or by user-defined customisation point
   - Must handle different memory locations in each C++ program sharing the object

Problem to solved:

Other problems not addressed for now:

1. Cache coherency
2. Synchronisation
   - SMP threading model is insufficient
3. Inter-process communication
4. Lots of other stuff …

(Baby steps first!)

Questions before the proposal papers?

WG21 P1631 *Object attachment and detachment*

Object attachment/detachment

- C++ objects are stored in zero, one or many arrays of bytes
- One can reinterpret cast an object into its byte array only for trivially copyable types
- It is *undefined behaviour* to do this for any other kind of C++ object
  - And no defined mechanism exists for reinterpreting such C++ objects as an array of bytes

Object attachment/detachment

This means:

- Well defined code must therefore employ *serialisation* i.e. *memory copying* for non-trivially copyable types
    - I.e. translate objects into a trivially copyable representation
- Even then, how do you ‘release’ a set of bytes from the abstract machine?

Object detachment

P1631 proposes two new operations:

1. Detachment, which is the in-place conversion of C++ objects into a byte array representing the formerly live object
    - Read/write reordering barrier (compiler only)
    - Object lifetime ends
    - No memory copying
    - Objects without reference to other objects are by default *trivially detachable*

Detachment operations

```
// A "one-way" reinterpret cast operator without possibility
// of aliasing. Upon return, input array of T is now an array of byte.
// It is UB to "speak T" to the output byte array
span<byte> detach_cast(span<T>)

// Main customisation point (free function)
// For trivially detachable T's (i.e. not containing pointers nor
// references - including vptrs!), it has a default implementation
// equal to detach_cast()
template<class T> [constexpr|consteval]
span<byte> in_place_detach(span<T>) [noexcept|throws];
```

Object attachment

2. **Attachment**, which is the in-place conversion of a previously detached object representation into a live object
    - Lifetime begins
    - Only *reachable C++ programs* may reattach object representations that they previously detached
    - Constexpr global static data init at process launch becomes redefined into detachment and attachment

Attachment operations

```
// A "one-way" reinterpret cast operator without possibility
// of aliasing. Upon return, input array of byte is now an array of T.
// It is UB to "speak byte" to the output T array
span<T> attach_cast(span<byte>)

// Main customisation point (free function)
// For trivially attachable T's (i.e. not containing pointers nor
// references - including vptrs!), it has a default implementation
// equal to attach_cast()
template<class T> [constexpr|consteval]
span<T> in_place_attach(span<byte>) [noexcept|throws];
```

‘Reachable C++ programs’

Reachable C++ programs 1/3

Required to be one of the following:

1. The currently running C++ program only. In this definition, all modifications to storage instances are lost when the C++ program's execution ends

   (this is the existing model in C++ 20)

Reachable C++ programs 2/3

2. Sequential executions of the unmodified current C++ program over time, where at least one modification to ***shared*** storage instances by one execution is made available to subsequent executions of the same C++ program, so long as each execution forms a total sequential ordering.

Reachable C++ programs 3/3

3. Concurrent executions of many instances of the current, unmodified, C++ program, where modified ***shared*** storage instances can be passed between those concurrently executing instances, including across heterogeneous compute.

But this *still* isn't enough to implement abstract machine support for mapped memory …

Potential std::detach to elsewhere

```
template<class T>
span<byte> std::detach_to_other_program(T &v) {
  // Step 1: Turn object into array of bytes
  span<byte> bytearray = in_place_detach(span<T>{&v, 1});
  // Step 2: Prevent dead store elimination
  atomic_signal_fence(memory_order_seq_cst);
  // Step 3: Tell abstract machine to treat this byte array as
  // indeterminate from now onwards
  for(byte &b : bytearray)
    b.~byte();  // assumes this does not modify memory!!!
  return bytearray;
}
```

Potential std::attach from elsewhere

```
template<class T>
span<T> std::attach_from_other_program(span<byte> bytearray) {
  // Step 1: Tell abstract machine the contents of this byte
  // array is not indeterminate, and contains valid objects
  memmove(bytearray.data(), bytearray.data(), bytearray.size());

  // Step 2: Turn byte array into live objects
  return in_place_attach<T>(bytearray);
}
```

Observable side effects 1/2

1. Shared memory
   - Detach, IPC, other side attach
2. Memory mapped storage
   - Detach, program end, program begin, attach
3. Object relocation
   - Detach, `memcpy`, Attach
4. More powerful object moves
   - Many more types gain defaultable move constructors and move assignment

Observable side effects 2/2

5. New type category: `ByteCopyable`
   - Solves small-value C++ ABI inefficiency!
   - Superset of `TriviallyCopyable`
     - Can include types with non-trivial destructors
   - Enables CPU register storage for all (trivially? constexpr?) detachable and attachable types
     - (though this would be an ABI break)
   - C++ objects can pass from C++, through C code, back into C++, a major gain for the ecosystem

Missing parts

- How best to make types containing dynamic memory allocations `ByteCopyable`? E.g. `std::vector`, P0709 `std::error`, etc
  - Move constructor = `relocate`?
  - Split attach/detach into sub-operations e.g. reanimate/zombify?
  - Something else entirely? Perhaps merge aspects of WG21 P1144 *Object relocation in terms of move plus destroy*

Links

- WG21 P1631 will be in the Cologne (July) mailing
    - NOT available yet (so the URL in these slides will 404 until June 2019!)

WG21 P???? *Page-based object storage*

Uniquely identifying object storage

- How does reachable C++ program A uniquely identify a detached object in shared memory to reachable C++ program B, so B can reattach it?
  - Must be as fast as possible!
  - Must work well over a network-connected HPC cluster! (i.e. high latency high bandwidth fabric)
  - Ideally must be amenable to *pointer provenance* validation

Uniquely identifying object storage

- Objects are stored in one or more memory pages
- Figuring out which memory pages store an object is *fast* (page table walk)
- Figuring out which storage duration (program, thread, automatic, stack), and thus its *shareability*, a page is associated with *could* be fast

Page-based object storage

- It is probably unavoidable that the C++ abstract machine needs to be taught something about memory pages
  - Shared pages => OK for multiple reattach
  - Static init copy-on-write pages => OK for *once-off* reattach
  - Stack and thread-local pages => NEVER ok
  - Private anonymous pages => PROBABLY NOT ok?

Page-based object storage

And then you might as well expose:

- `span<T>` to memory pages (i.e. query page tables)
- Query memory pages sizes, protections, copy-on-write, dirty bits etc
- Remap pages from address A to address B
- Throw away contents of pages
- Kick pages to swap file
- Allocate prefaulted, committed and non-committed pages
- …

Page-based object storage

- Most of this is stuff for the standard library e.g. WG21 P1031 *Low level file i/o* which already implements most of the above
  - Pages mapped in from a shared file are shared pages
  - Objects in the shared file can be uniquely identified across all reachable C++ programs with access to the shared file

Page-based object storage

- So what is the bare minimum which the C++ abstract machine needs to be taught?
- I *think* the only parts are:
  a. There are pages of memory of varying sizes
  b. All objects are stored in pages
  c. Pages can be one of (i) indeterminate, (ii) private, (iii) shared, (iv) copy-on-write, (v) clean (all bits zero)

Page-based object storage

- And maybe now or later?
  d. Whole arrays of pages can be attached and detached, should be treated as structures full of objects (assumes Reflection is up to the job)
     - Why? Actor based concurrency
     - Why? Gifting/splicing pages to zero copy i/o
     - Why? Dynamically loadable Modules
     - Why? Direct manipulation of process page tables by userspace - far more efficient malloc(), memcpy(), etc

~~WG14 N2362 *Moving to a provenance-aware memory model for C2x*~~

Provenance-aware memory model

I will say nothing on this, lest I make a fool of myself on internet published video, except:

- I *think* this essentially brings C's memory model to parity with C++ 20's memory model
    - Devil is in the corner cases though
- No Rust-style memory hygiene enforcement here (sad panda)

Thank you

And let the questions begin!