Oilpan: GC for Blink

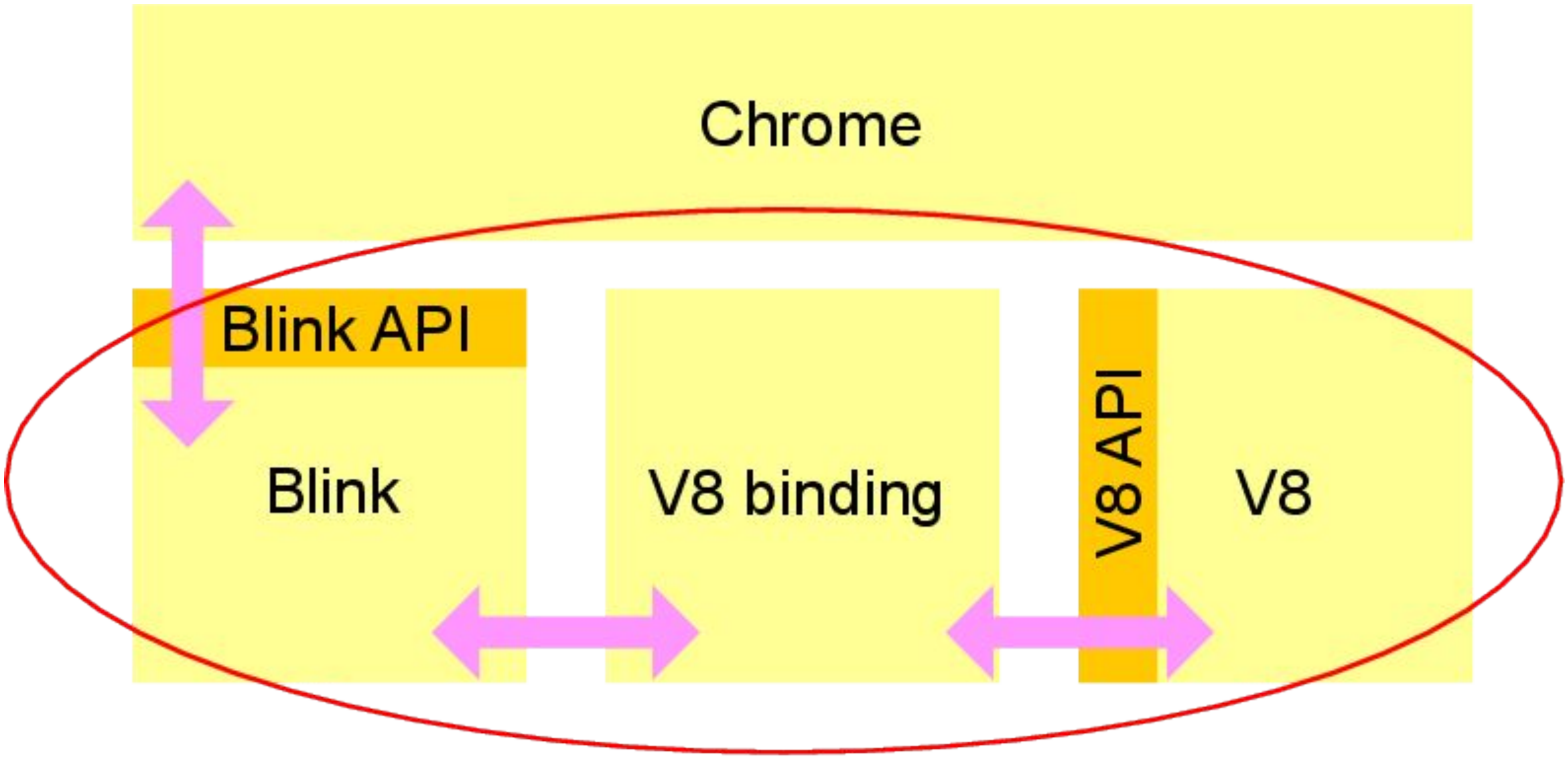
No more crashes, No more leaks

Kentaro Hara (haraken@chromium.org)

Team

- Mads Ager (TL)
- Vyacheslav Egorov
- Erik Corry
- Kentaro Hara
- Ian Zerny
- Gustav Wibling
- Kouhei Ueno

What is Oilpan?



This is a story of memory management around here

What is Oilpan?

- Blink has 350 K lines of C++ code with 5500 files

- The lifetime of the C++ objects are managed by manual reference counting

- The reference counting has been causing a ton of problems

- Oilpan replaces the reference counting with a cool GC

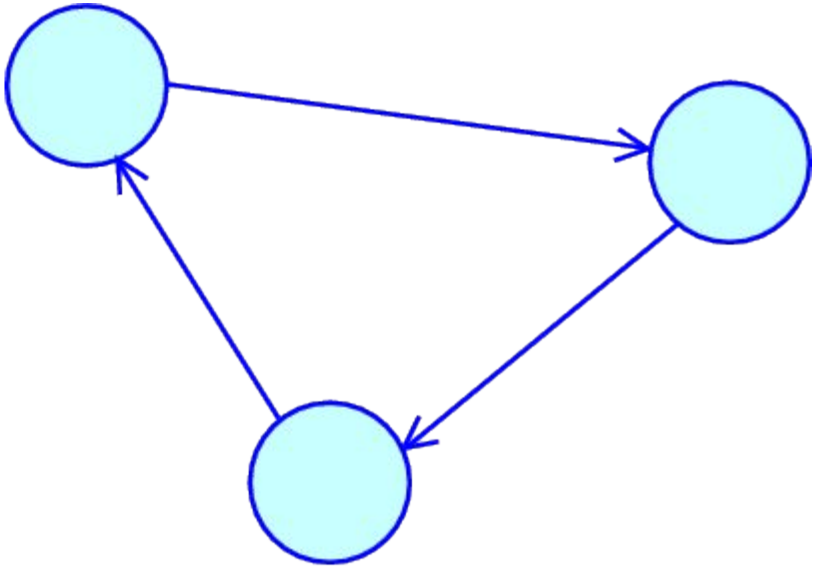
Agenda of this talk

- Motivations
- Goals of Oilpan
- Programming model
- Implementation details
- Performance results

Motivations

A disadvantage of reference counting

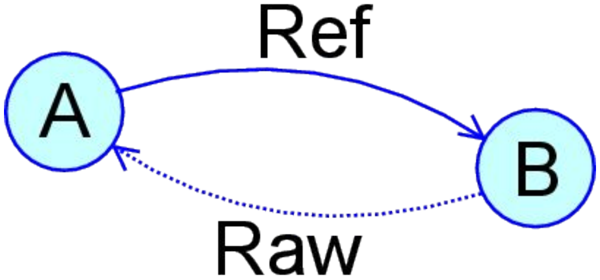
- In reference counting, reference cycles are not allowed
    - Cycles leak memory



Problem 1: Poor programmability

- You have to be very careful not to produce cycles

```
class A {
  RefPtr<B> m_b;
};

class B {
  A* m_a;  // This is a back pointer to A.
           // You need to use a raw pointer in order not to
           // produce a cycle.
};
```



Problem 1: Poor programmability

- You need to understand relationships between a lot of Blink objects to make sure that the reference you're going to introduce won't produce a cycle somewhere

- You need to wait for review from Blink experts

- This has slowed down productivity of Blink development

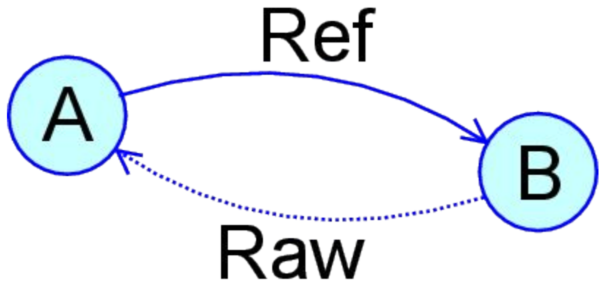
Problem 2: Memory leaks & Poor security

- Relationships of Blink objects are not as easy as humans can understand

- Regardless of our careful development:
    - there are a lot of memory leaks
        - 10% of our tests are leaking
    - there are a lot of use-after-free crashes (security bugs)
        - Almost all crash reports are due to use-after-frees

Problem 2: Memory leaks & Poor security

```
class A {
  ~A() { m_b->clear(); } // If B outlives A, you need to clear
                         // B::m_a when A is destructed.
                         // If you forget to clear, B::m_a will
                         // cause use-after-free.
  RefPtr<B> m_b;
};

class B {
  void clear() { m_a = 0; }
  A* m_a;
};
```



Problem 2: Memory leaks & Poor security

```
class A {
  ~A() { m_b->unregisterA(this); } // If B outlives A and you
                                   // forget to call unregisterA(),
                                   // B::m_weakSet will cause
                                   // use-after-free.
  RefPtr<B> m_b;
};

class B {
  void unregisterA(A* a) { m_weakSet.remove(a); }
  HashSet<A*> m_weakSet;
};
```

Problem 2: Memory leaks & Poor security

- To break a cycle, you need to use raw pointers

    => If you miss it, it will cause memory leaks

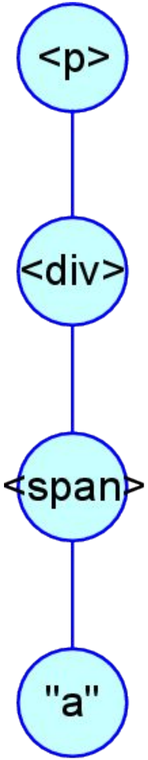
- You need to manually maintain the lifetime of the raw pointers (i.e., back pointers, weak pointers)

    => If you miss it, it will cause use-after-free crashes

Problem 3: Correctness is broken

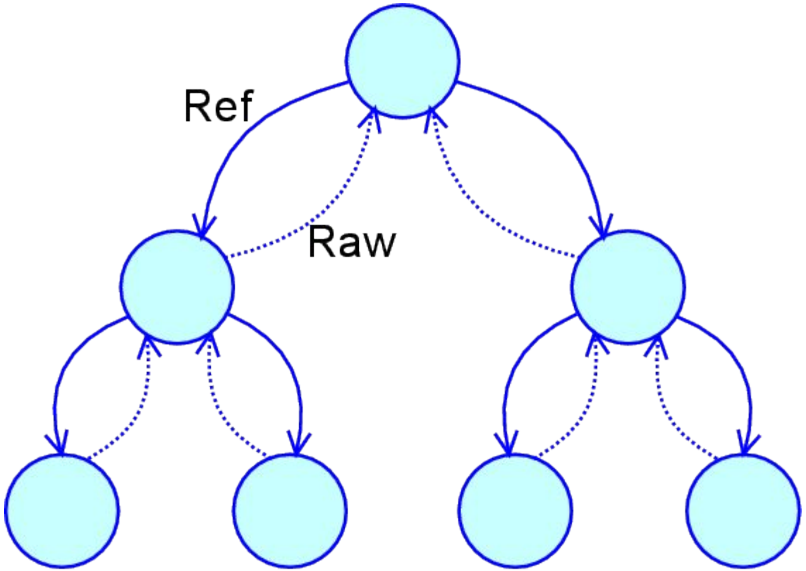
- In some cases, we're intentionally accepting wrong behavior in order to break cycles

```
p = document.createElement("p");
p.innerHTML ='<div><span>a</span></div>';
span = p.querySelector("span");
p.innerHTML = "";
alert(span.parentNode);
// The expected result is <div>,
// but the actual result is null.
```



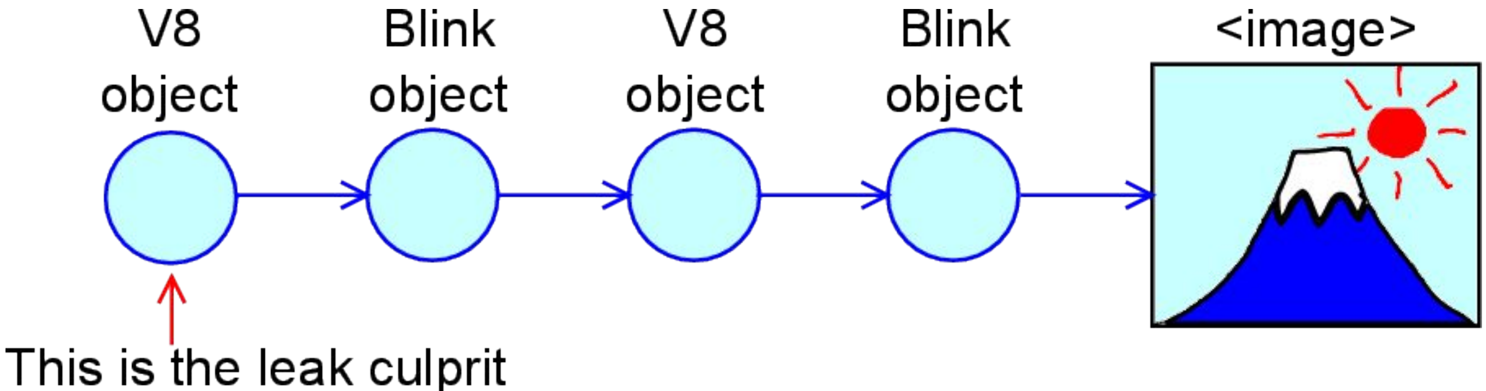
Problem 3: Correctness is broken

- This is because Blink doesn't hold a reference to parent nodes in order to break a cycle
    - This is just one example of existing wrong behaviors



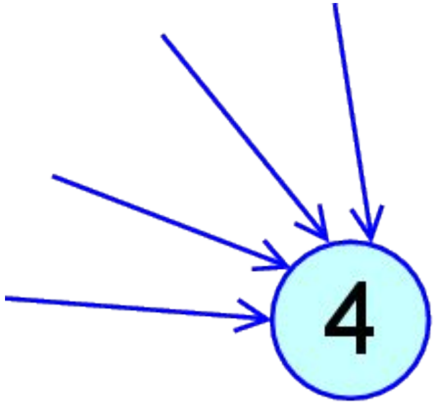
Problem 4: Objects are not traceable

- When memory leaks, it is important to identify the culprit object which holds memory behind it
    - Even if <image> leaks 200 MB of memory, it does not mean that the <image> is the culprit of the leak
    - Blink and V8 objects have complicated relationships



Problem 4: Objects are not traceable

- To find the culprit, you need to trace objects
- However, reference counting has no information about object graphs



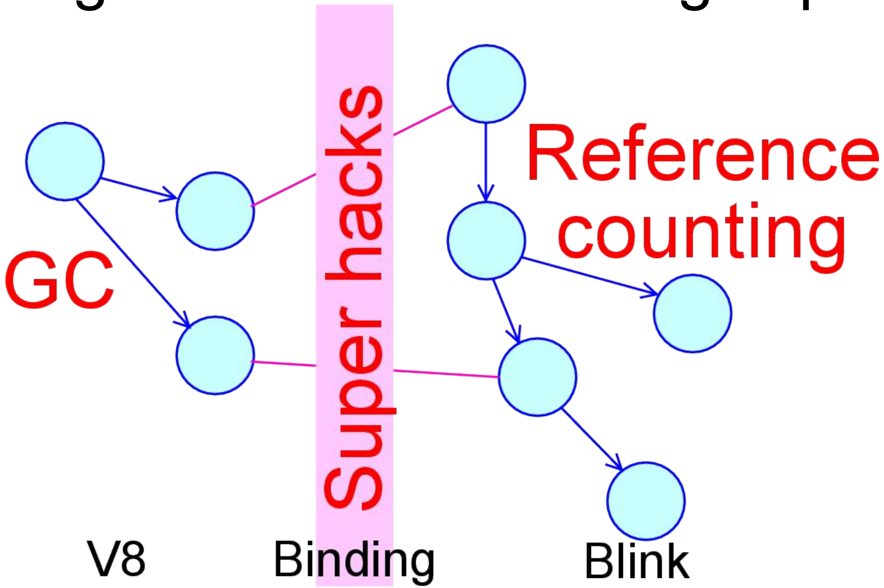
I know I am referenced from 4 objects
but I don't know what the 4 objects are...

Problem 4: Objects are not traceable

- As a result:
  - You cannot find the cause of memory leaks easily
  - Chrome cannot provide cool developer tools which could be provided if object graphs are available

Problem 5: Complicated language bindings

- V8 is working with a generational GC
- Blink is working with reference counting
- V8-Blink binding connects them using super hacks



Problem 5: Complicated language bindings

- It will take 10 mins to explain how V8's major GC interacts with Blink's reference counting

- It will take 20 mins to explain how V8's minor GC interacts with Blink's reference counting

- It will take 30 mins to explain exceptional cases

Here is Oilpan!

Goals of Oilpan

For Problem 1: Poor programmability

- In Oilpan, reference cycles are OK
    - Oilpan's GC takes care of cycles

- You no longer need to introduce a complicated architecture just for breaking cycles

For Problem 2: Memory leaks & Poor security

- In Oilpan, reference cycles are OK
  => There will be no memory leaks

- You no longer need to use raw pointers
  => There will be no use-after-free crashes

For Problem 3: Correctness is broken

- In Oilpan, reference cycles are OK

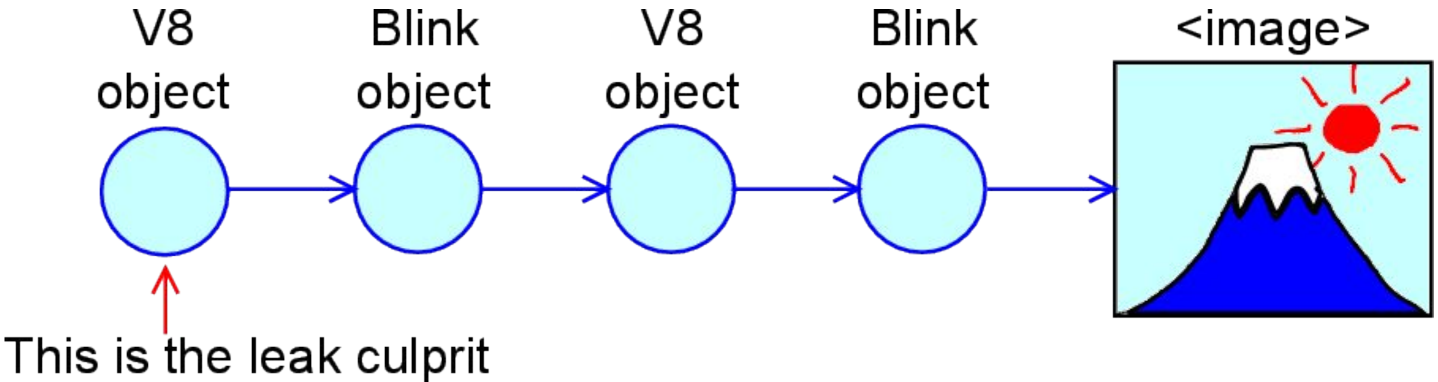
  => You no longer need to give up correctness just for breaking cycles

For Problem 4: Objects are not traceable

- In Oilpan, objects become traceable because Oilpan knows a complete graph of Blink objects

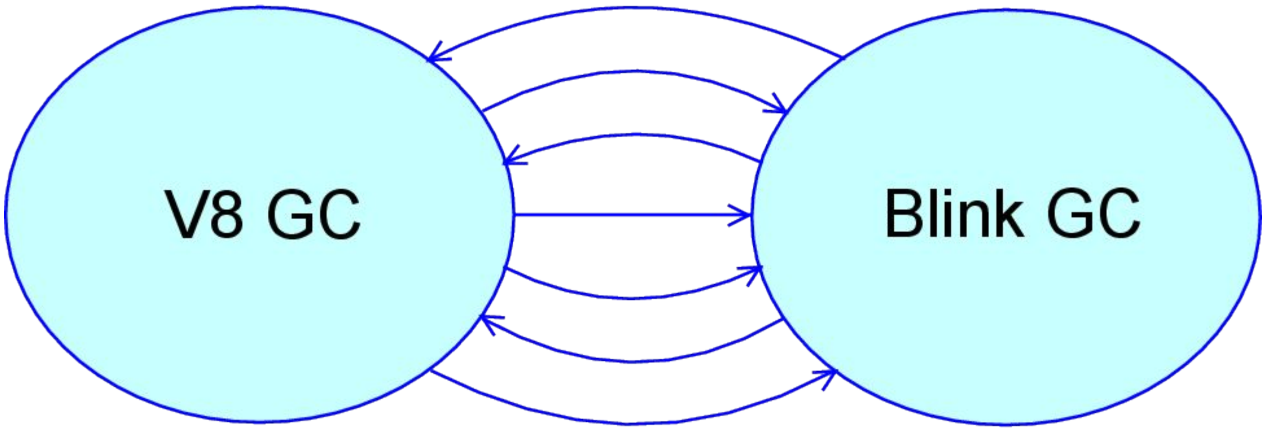
=> You can easily find the culprit object of memory leaks

=> Chrome can provide cool developer tools that trace objects between Blink and V8



For Problem 5: Complicated language bindings

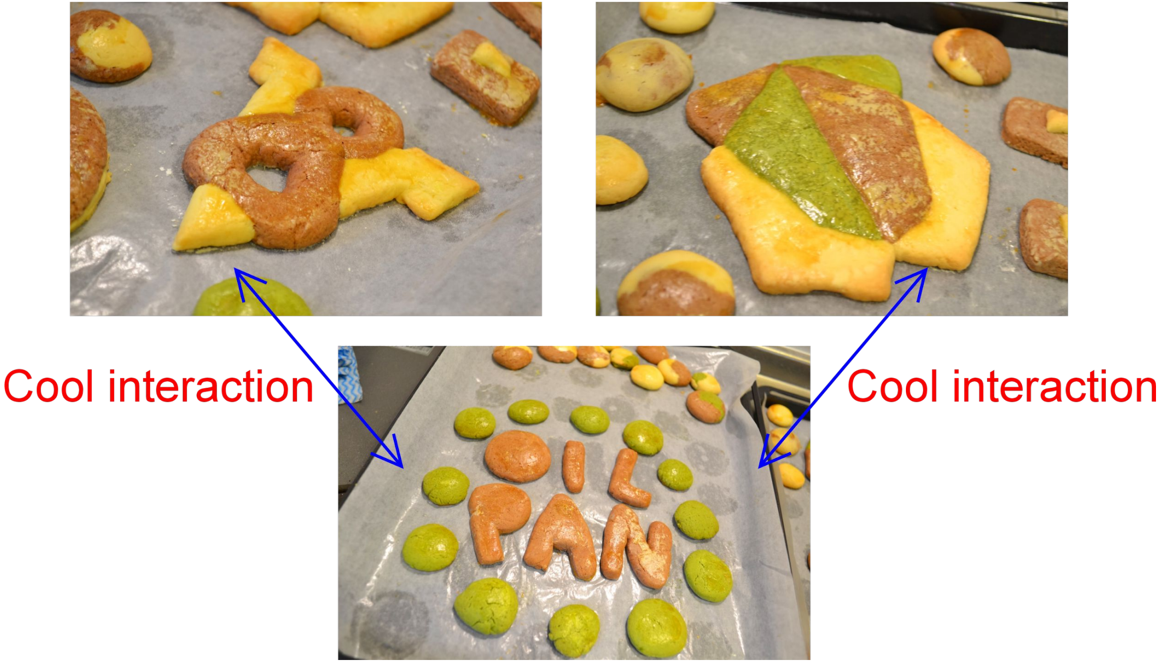
- In Oilpan, you just need to consider how to connect two GCs
  - This is easy: Two GCs just need to exchange out-going pointers iteratively until they mark everything



Cool interaction

For Problem 5: Complicated language bindings

- It’s even possible to integrate Dart
    - Historically, this was the reason we started Oilpan :)



Summary of the goals

- Better programming model
- No memory leaks
- No use-after-free crashes (i.e., Better security)
- Correctness
- Object traceability
- Better language bindings

Sounds cool!

Programming model

Overview

- In the reference counting world:
    - class X : public RefCounted<X> { … };
    - RefPtr<X>, X*

- In the Oilpan world:
    - class X : GarbageCollected<X> { ... };
    - Member<X>, WeakMember<X>, Persistent<X>, X*

Programming rule 1: Object allocation

- Use GarbageCollected<X> to allocate an object on Oilpan’s heap

```
class X : public GarbageCollected<X> {
    …;
};
```

Programming rule 2: Pointers

- To add a reference to X, use either of Member<X> or WeakMember<X> or Persistent<X> or RefPtr<X> or X*

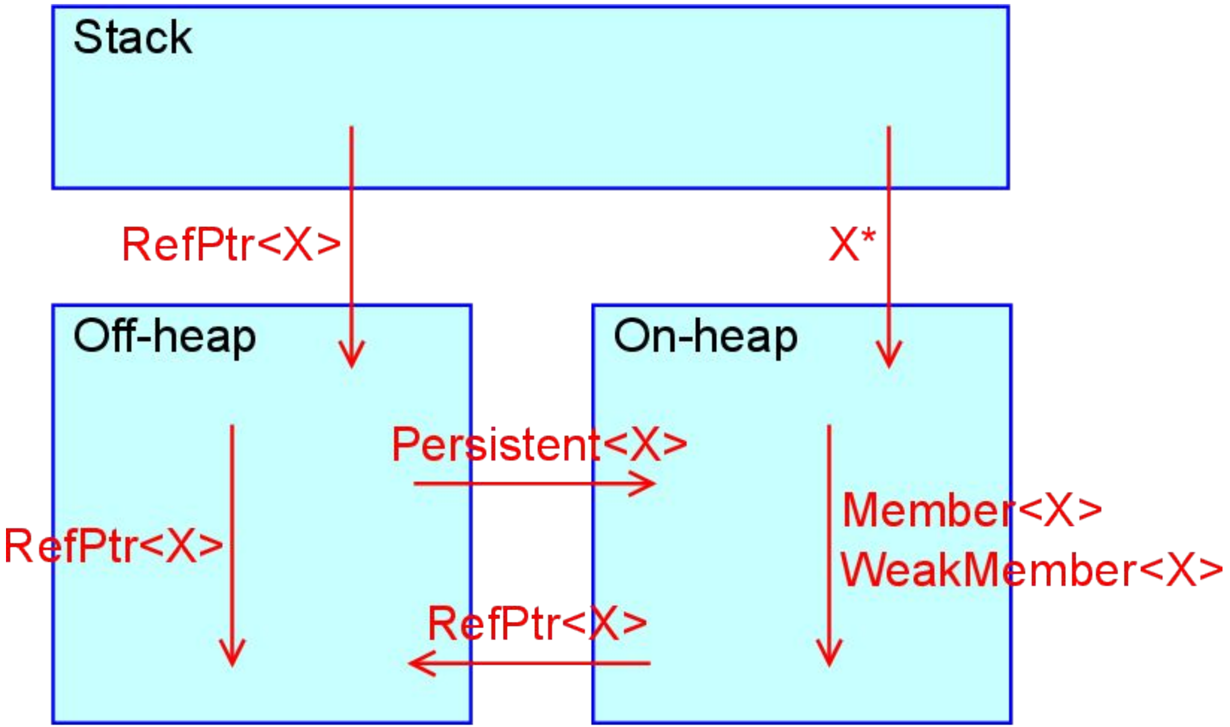
- “Which one should I use???”
    - It depends on where the reference source exists and where the reference destination exists

Three kinds of memory regions

- There are three kinds of memory regions in Oilpan
    - Stack region: Call stack of program execution
    - On-heap region: The region managed by Oilpan’s GC
    - Off-heap region: The region (still) managed by tcmalloc
        - Reference-counted objects
        - Static region of program execution

- Our goal is to migrate reference-counted objects from off-heap to on-heap

Which pointer you should use



- Once we complete Oilpan, * => Off-heap will be gone

Which pointer you should use

- Stack => On-heap: X*
- Stack => Off-heap: RefPtr<X> (We’re already doing)
- On-heap => On-heap: Member<X> or WeakMember<X>
- On-heap => Off-heap: RefPtr<X>
- Off-heap => Off-heap: RefPtr<X> (We’re already doing)
- Off-heap => On-heap: Persistent<X>

Which pointer you should use

- Stack => On-heap: X*
- Stack => Off-heap: RefPtr<X> (We're already doing)
- On-heap => On-heap: Member<X> or WeakMember<X>
- On-heap => Off-heap: RefPtr<X>
- Off-heap => Off-heap: RefPtr<X> (We're already doing)
- Off-heap => On-heap: Persistent<X>

Stack => On-heap

- You can just use raw pointers on a stack
  - Oilpan's conservative GC will automatically find on-stack raw pointers

```
Node* Node::parentNode() {
  Node* parent = this;
  while (parent->parentNode)
    parent = parent->parentNode;
  return parent;
}
```

Stack => On-heap

- You can remove all on-stack RefPtr<X>s
    - This produces a big performance win!
    - This is the biggest reason why Oilpan performs better than the current reference counting

Stack => On-heap

- You no longer need to write “RefPtr<X> protect(this);”

```
void Frame::callV8Function(Function v8Function) {
  RefPtr<Frame> protect(this);  // You had to protect the Frame
                                // because the V8 execution might
                                // lose the last reference to
                                // the Frame. You don’t need this
                                // protection in Oilpan.
  v8Function->call();
}
```

Which pointer you should use

- Stack => On-heap: X*
- Stack => Off-heap: RefPtr<X> (We’re already doing)
- On-heap => On-heap: Member<X> or WeakMember<X>
- On-heap => Off-heap: RefPtr<X>
- Off-heap => Off-heap: RefPtr<X> (We’re already doing)
- Off-heap => On-heap: Persistent<X>

On-heap => On-heap

- Use Member<X> or WeakMember<X> depending on lifetime
- If it’s a strong reference, use Member<X>

```
class Node : public GarbageCollected<Node> {
  Member<Document> m_document;
};

class Document : public GarbageCollected<Document> {
  Member<Node> m_focusedNode;
};
```

On-heap => On-heap

- If it’s a weak reference, use WeakMember<X>

```
class Node : public GarbageCollected<Node> { … };

class LiveNodeList : public GarbageCollected<LiveNodeList> {
  WeakMember<Node> m_cachedNode; // m_cachedNode will be
                                 // cleared when the Node is
                                 // destructed.
};
```

Which pointer you should use

- Stack => On-heap: X*
- Stack => Off-heap: RefPtr<X> (We're already doing)
- On-heap => On-heap: Member<X> or WeakMember<X>
- On-heap => Off-heap: RefPtr<X>
- Off-heap => Off-heap: RefPtr<X> (We're already doing)
- Off-heap => On-heap: Persistent<X>

On-heap => Off-heap

```
class CSSParser : public RefCounted<CSSParser> { … };

class Node : public GarbageCollected<Node> {
  RefPtr<CSSParser> m_parser;
};
```

- Once we move CSSParser to Oilpan’s heap, the RefPtr<CSSParser> will become Member<CSSParser>

Which pointer you should use

- Stack => On-heap: X*
- Stack => Off-heap: RefPtr<X> (We're already doing)
- On-heap => On-heap: Member<X> or WeakMember<X>
- On-heap => Off-heap: RefPtr<X>
- Off-heap => Off-heap: RefPtr<X> (We're already doing)
- Off-heap => On-heap: Persistent<X>

Off-heap => On-heap

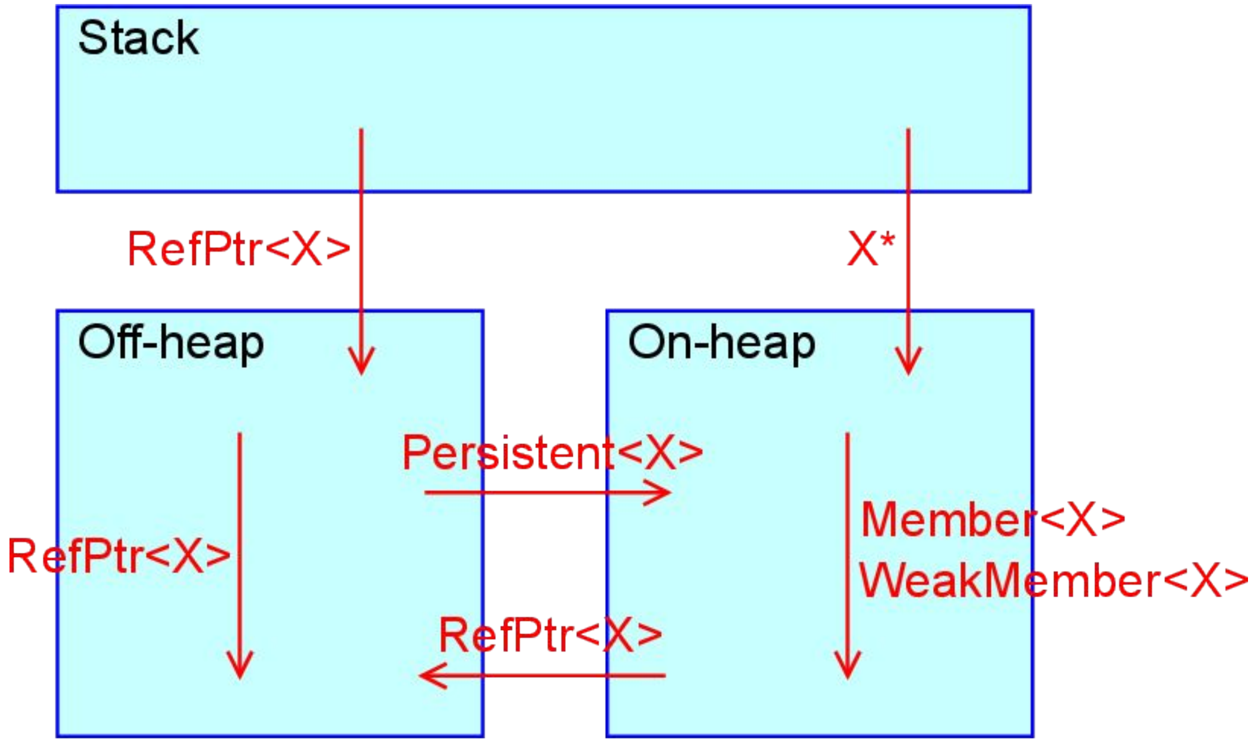
```
class Node : public GarbageCollected<Node> { … };

class CSSParser : public RefCounted<CSSParser> {
  Persistent<Node> m_node;
};

Node* someFunction() {
  static Persistent<Node> cachedNode = new Node();
  return cachedNode;
}
```

Off-heap => On-heap

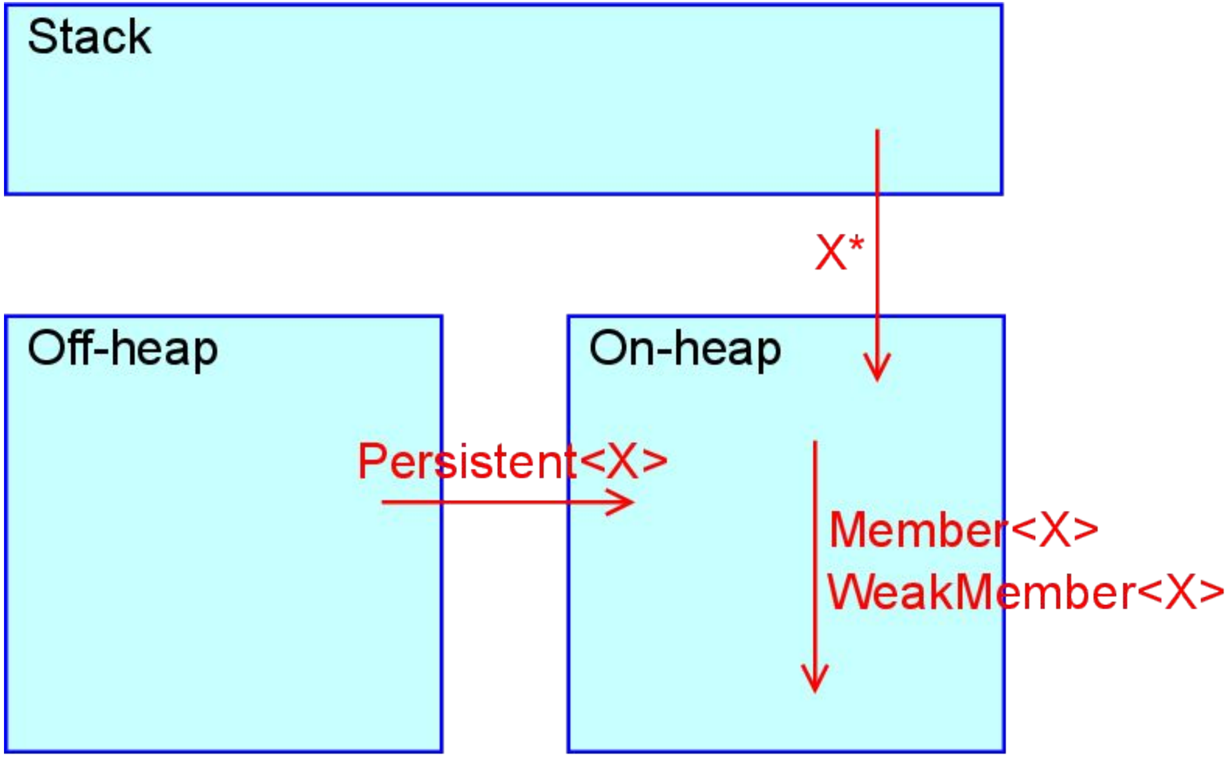
- From the perspective of GC, Persistent<X>s are treated as root objects in the marking phase

- You must not use Persistent<X>s for On-heap => On-heap references
    - It will cause memory leaks

Summary (in the transition period)



Summary (in the future)



- Once we complete Oilpan, * => Off-heap will be gone

Programming rule 3: Destructors

- Oilpan doesn’t call destructors for GarbageCollected objects
    - because destructors are bad things for various reasons
        - destructors waste time in fixing dead objects
        - destructors cause ordering issues (explained later)
        - ...etc

- We’re planning to get rid of most of all destructors eventually

If you do need destructors...

- If you want Oilpan to call destructors, you need to use GarbageCollectedFinalized<X> instead of GarbageCollected<X>

```
class X : public GarbageCollectedFinalized<X> {
  ~X() {}  // ~X() is necessary because...
  String m_string;  // String needs destructor.
  RefPtr<Y> m_y;  // RefPtr needs destructor.
  Vector<Z> m_vector;  // Vector needs destructor.
};
```

However, be careful

- However, there is no guarantee about the order in which destructors are called
    - You can never expect that a DOM tree is destructed from top to down

However, be careful

- Thus you must not touch other on-heap objects in destructors
    - That's why we want to get rid of destructors :)

```
class X : public GarbageCollected<X> { … };

class Y : public GarbageCollectedFinalized<Y> {
  ~Y() { m_x->clear(); }  // This will crash depending on
                          // the destruction order.
  Member<X> m_x;
};
```

Programming rule 4: Tracing objects

- Each GarbageCollected object has to have a trace() method which traces all on-heap members

```
class X : public GarbageCollected<X> {
  void trace(Visitor* v) { v->trace(m_a); }
  Member<A> m_a;
};
class Y : public X {
  void trace(Visitor* v) { v->trace(m_b); X::trace(v); }
  Member<B> m_b;
};
```

Why do we have to hand-write trace()?

- Oilpan’s GC uses the trace() methods to mark all reachable objects

- In C++, it’s hard to identify what is on-heap pointer just by looking at raw memory layout (esp., complicated on-heap collections)

- We will provide a clang plugin that verifies that trace() methods are correctly written for all Members

Programming rule 5: On-heap collections

- Oilpan provides on-heap collections
    - Vector<Member<X>> => HeapVector<Member<X>>
    - HashSet<Member<X>> => HeapHashSet<Member<X>>
    - HashMap<Member<X>> => HeapHashMap<Member<X>>

- You can treat on-heap collections as normal on-heap objects
    - Persistent<HeapVector<Member<X>>>

Summary

Programming rules:

- Use GarbageCollected<X>
- Use either of Member<X> or WeakMember<X> or Persistent<X> or RefPtr<X> or X*
- Be careful about the destruction order
- Write trace() methods
- Use on-heap collections

Sounds complicated??

Sounds complicated?

- The programming rules might look complicated at first

- However:
  - The rules are consistent
  - The rules are less error-prone than the current reference counting, and thus better in long-term
    - Less memory leaks
    - Less use-after-frees

Sounds complicated?

- We migrated the Node & CSS hierarchy to Oilpan’s heap and confirmed that the rules will scale up to the entire code base

Verification tools

- Oilpan provides a clang plugin to verify that your code follows Oilpan's programming rule
    - Are trace() methods written correctly?
    - Is Persistent<X> not used for an On-heap => On-heap reference?
    - ...etc

Verification tools

- Oilpan provides ASan for Oilpan's heap
    - ASan can detect use-after-frees
    - ASan can detect destructors that rely on destruction order

- Oilpan provides a leak detector to verify that each Oilpan change won't introduce new memory leaks

Implementation details

Overview of Oilpan's GC

- Oilpan runs a mark-and-sweep GC

- Oilpan runs a conservative GC for on-stack objects

- Oilpan runs a precise GC for on-heap objects

How the mark-and-sweep GC works

(1) The GC marks root objects

- On-stack pointers (including false-positives)
- Persistent<X>

(2) The GC calls trace() methods and marks all reachable objects

(3) The GC sweeps unmarked objects

Why use a conservative GC on a stack?

Reason 1: Programmability

- In order to run a precise GC on a stack, we have to ask programmers to annotate what are Oilpan pointers
- We tried this approach but concluded that the programmability won't scale up to the entire code base

```
for(Handle<Node> node=firstChild(); node; node=nextSibling())
{
  HandleScope scope;
  node->foo();
}
```

Why use a conservative GC on a stack?

Reason 2: Performance

- Compilers are not smart enough to optimize operations on those annotated pointers into raw pointer operations
- If we adopt a conservative GC, we can remove all on-stack RefPtrs
    - This produces a big performance win (which will cover regressions introduced by Oilpan's GC)

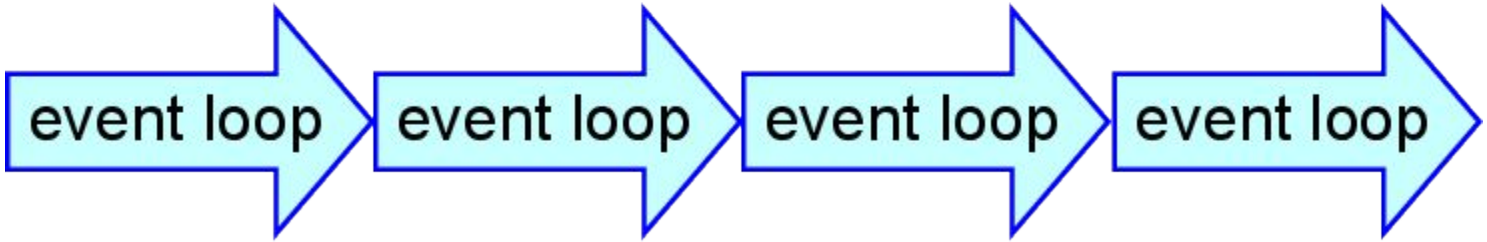
More about the conservative GC

Question: “However, a conservative GC works conservatively. Won’t it cause unexpected memory leaks??”

Answer: Hehe, there is a magic :)

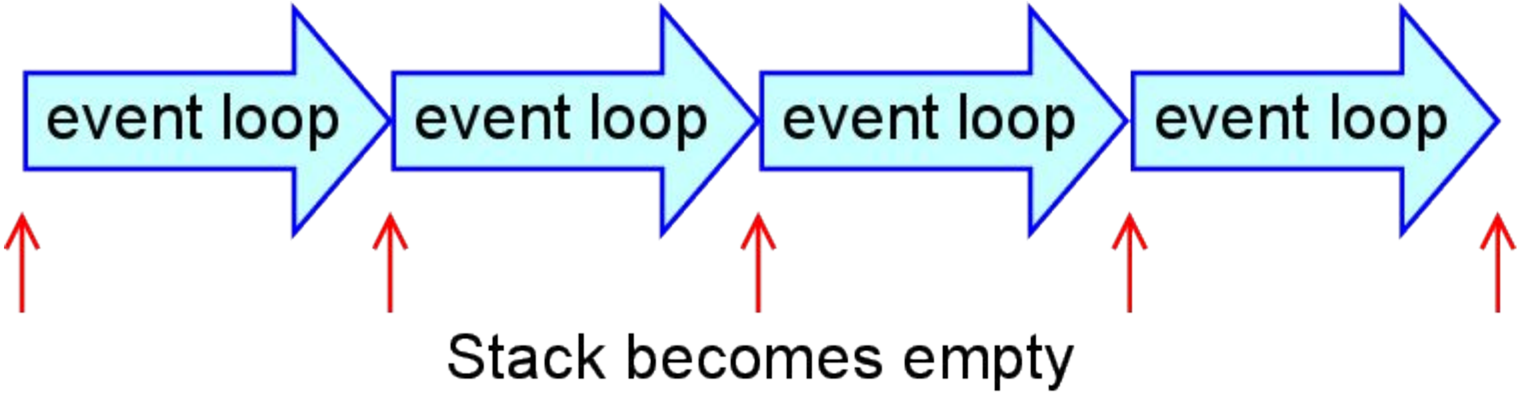
More about the conservative GC

- Blink and JavaScript are executed in event loops
    - An event loop is a unit of Blink and JavaScript execution
        - e.g., setTimeout() creates one event loop
    - Chrome runs Blink and JavaScript as a sequence of multiple event loops



More about the conservative GC

- At the end of each event loop, a stack becomes empty
- At that point, Oilpan can run a precise GC



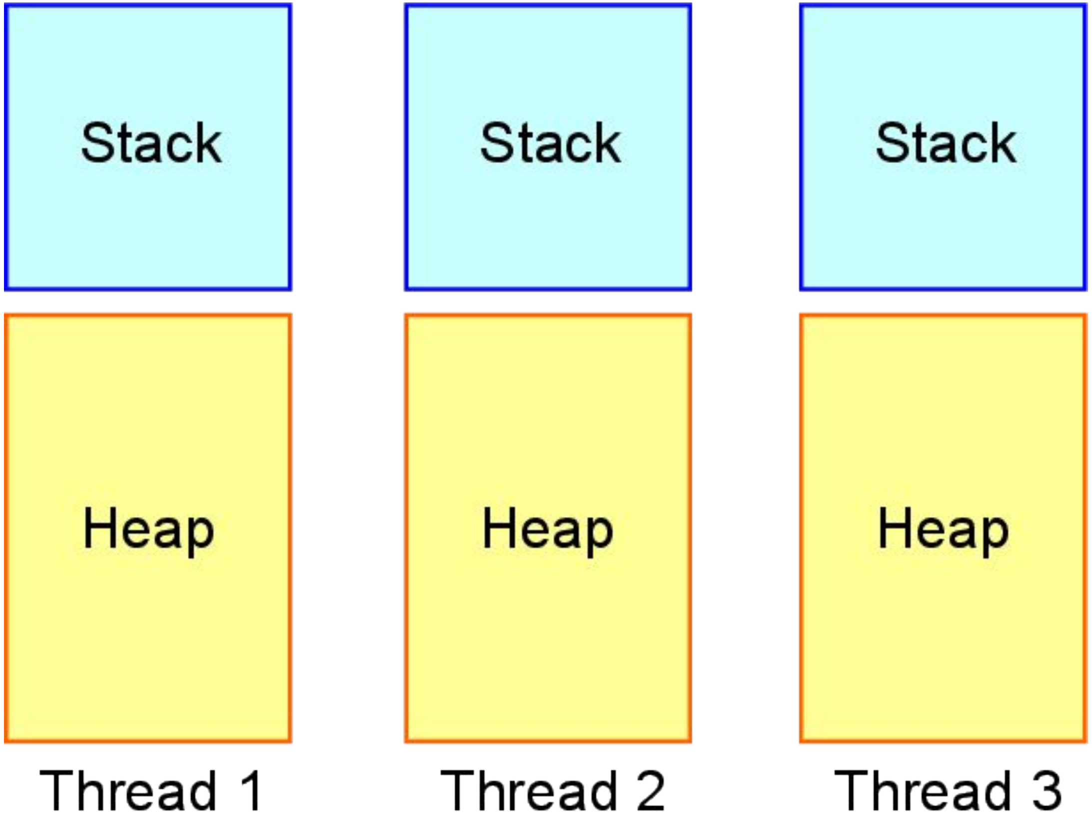
More about the conservative GC

- Oilpan tries its best to run a GC at the end of event loops

- A conservative GC comes in to a play only when Oilpan really needs to trigger a GC during an event loop

- Thus Oilpan's GC is not that conservative in practice
    - We confirmed that the conservativeness won't become a problem in practice

Heap structure



Heap structure

- Each thread has its own heap and stack

- Each thread can touch other threads' heaps

- Objects allocated by a thread X is guaranteed to be destructed by the thread X

Threading

- A GC runs in a stop-the-world manner

- When a GC runs, all threads have to be in safe points
  - Safe points = Places where it is guaranteed that the thread doesn't touch any memory on heaps
  - This doesn't mean that all threads have to "stop"
    - OK: The thread is executing JavaScript
    - OK: The thread is executing blocking I/O

Threading

- When a GC runs, all threads have to be in safe points

- This means that the GC has to wait for all threads to enter safe points

- Thus it is important to make sure that each thread enters safe points very frequently

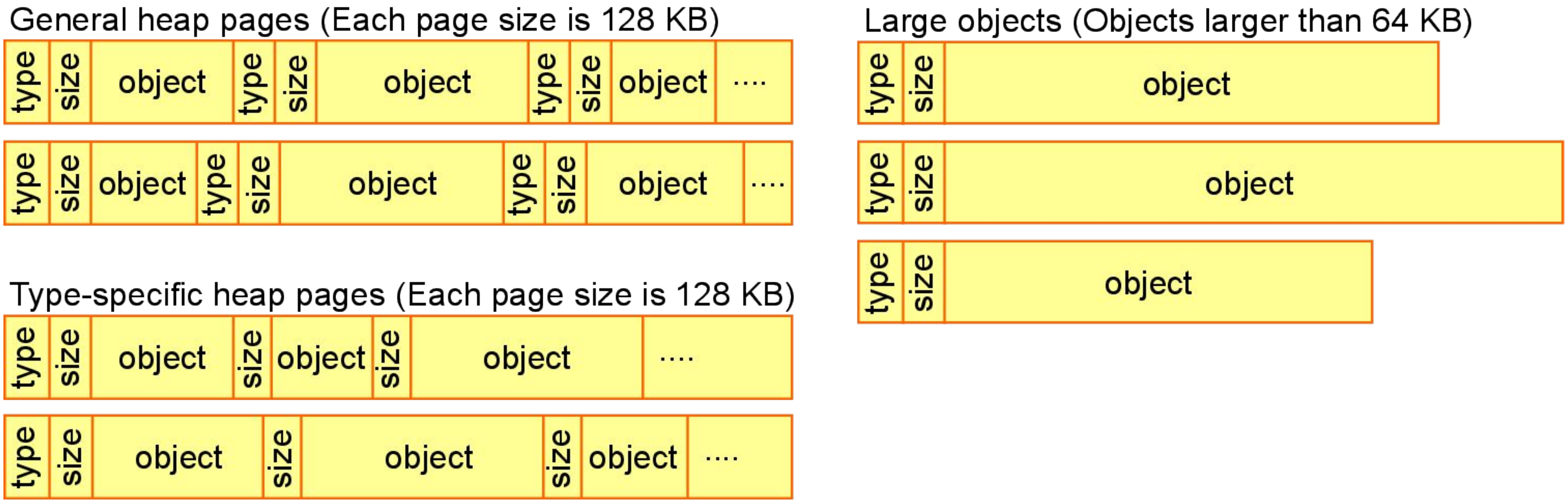
Threading

- Oilpan inserts safe points at the end of event loops

- Oilpan inserts safe points to various places in V8 (loops, function calls) using existing interruption mechanism of V8
    - Thus long-running JavaScript is no problem

- Oilpan inserts safe points to long-running C++ code in Blink (image processing, network I/O)

Memory layout



Memory layout

- Oilpan will add 0 or 1 extra word to each DOM object:
  - Oilpan removes one word for reference counting
  - For normal DOM objects, Oilpan adds two words
  - For type-specific DOM objects (e.g., Node), Oilpan adds one word

- These extra words won’t be a problem, since the size of DOM objects is not a dominant factor of total memory usage
  - V8 heap and Strings consume much more memory

For better security

- Oilpan supports ASan

- Oilpan defers reusing freed pages for a couple of GC cycles
    - This is effective to prevent security exploits that exploit use-after-frees

Performance results

Performance

- Results

- Some benchmarks are better, some benchmarks are worse
    - Performance gain comes from the fact that we removed all on-stack RefPtrs
    - Performance loss comes from the GC overhead

Performance

- Dromaeo (A popular micro benchmark for DOM)
  - dom-attr: -5%
  - dom-modify: +4%
  - dom-query: +18%
  - dom-traverse: +7%

Memory

- Results

- We confirmed that there is no observable memory increase in page cyclers and top 25 web sites

- We confirmed that the conservativeness won't become a problem in practice

Notes

- We’ve been developing Oilpan based on an 8 month old branch

- The results compare the branching point vs. the current point

- We will re-evaluate performance & memory more in detail when upstreaming Oilpan to the trunk incrementally

Conclusions

Summary

- Oilpan solves a bunch of problems of the current reference counting in Blink
    - Better programming model
    - No memory leaks
    - No use-after-free crashes (i.e., Better security)
    - Correctness
    - Object traceability
    - Better language bindings

Summary

- The programming rules might look complicated at first, but the rules are consistent and less error-prone

- Performance and memory results look good

Shipping plan

- In January, we will start shipping Oilpan for simple DOM objects
    - Objects in modules/ (IndexedDB, WebSockets etc)
    - Objects in core/ that have simple, self-contained hierarchies

- We will start shipping Oilpan for the Node & CSS objects behind a compile time flag

We’re hiring!

- The transition period where reference counting and GC coexist will be a bit confusing and buggy

- We want to make the transition period as short as possible

- Your help is appreciated!

Links

- Design document by ager@

- oilpan-team@google.com