Persistent UMA

by Brian White <bcwhite@google.com>

Energy and persistence conquer all things. —Benjamin Franklin

Problem

While UMA's histograms have proved invaluable in the monitoring of Chrome's performance in the field, there are places it does not currently reach because it is limited to volatile storage that is only uploaded every 30 minutes. This prevents gathering information while Chrome is shutting down, inside auxiliary tasks such as the installer, and from tasks that end before having the opportunity to upload their data (the Renderer being the best example but Browser crashes and system resets as well).

Solution

Store histograms in some form of persistent memory separate from the main process heap. There are several ways to do this and which method is best may vary from process to process.

Shared memory can be used for sub-processes of the Browser. All modern operating systems provide shared memory segments. By allocating histogram data structures inside such storage, the data acquired will be maintained across many situations where it would otherwise be lost. The Renderer, for example, can update data in this area without worrying about sudden-death causing it to be lost. As long as the Browser doesn't crash, the data will still be available when the time comes to send the data upstream. Secure processes (e.g. Browser) must not store *their* histograms in the same space as insecure processes (e.g. Renderer) and even having multiple Renderers share information is not a good idea as it could leak information from other, uncompromised, operations into a compromised one.

Memory-mapped files can be used for processes that have no controlling parent and/or need to persist data from one run to the next. The OS will ensure that all data gets written to persistent storage and be available when the process next starts or readable by some other process. A power-failure or hard-reset may leave some recent data un-flushed but most of it will be saved.

Death-rattle handlers (such as *breakpad*) can detect when the process is dying and do a raw dump of the memory segment to a file on disk. There is no need to process or convert the data before writing which greatly simplifies the actions to open, write, close.

There is a complication due to associating histograms with a specific instance of a browser. Each run has a unique instance identifier and it's important to associate data persisted from a previous run with that previous identifier and not the current identifier. This is accomplished by adding a record to the memory segment describing the browser state when the segment was mapped. All histograms found after it can be associated with that state, uploaded, and then the segment cleared and new state data written to describe the new run.

Design

UMA histograms are already designed to be thread-safe so the solution breaks down into three basic blocks: Persistence, Allocation, and Association. The Histogram classes will need to be updated to be safe in shared memory and without the use of any direct pointers.

Persistence

A shared memory segment can persist until it is explicitly released or the system shuts down. This covers most of the existing “black” spaces where histogram data cannot be reported.

Chrome has an existing `MemoryMappedFile` class but is limited to read-only mapping of files. It should be possible to extend or adapt this class to support full create/read/write access and use that for improved persistence.

Alternatively, an additional process could be created or an existing process reused (“breakpad”?), now or at some arbitrary point in the future, that attaches to the shared memory segment and persist its contents to disk, either periodically or when it notices that no other handles to the memory are open. The Browser would read that, if necessary, next time it starts.

Allocation

Regardless of how data is persisted, allocation of histogram data within a memory block is the same. Though the segment may be mapped by a process at an arbitrary base address, it’s assumed that it will not move from that address for the lifetime of the process. Pointers acquired to data within the segment will remain valid so long as the referenced object is not deallocated either by this process or another (no different from any object pointer).

Because the address at which the file is mapped is not constant, no pointers can be stored within the mapped block; only offsets from the start can be used. Some changes to the Histogram objects will be required to support this but the run-time overhead of adding a base pointer to an offset should not affect performance.

To work well should the memory be a memory-mapped file, allocation will support “pages” across which allocations will not span (to ensure a consistent state on disk regardless of page flushing order or timing) and will not write to any memory until absolutely necessary (to prevent an auto-growing disk file from increasing size for no good reason) -- no zeroing memory or writing meta-data throughout the segment. The allocator assumes that memory is zero’d before being passed to it.

Sharing memory with other processes means it’s only as secure as the least-secure process, typically the Renderer. The allocation system and use of any data held in the shared block needs to be hardened against malicious intent. Each “offset dereference” must be validated so that it isn’t possible for corruption from one process to cause segmentation-faults in another process.

The ability to iterate through allocated blocks will also be supported. This will allow readers to access data put down by writers without the latter having to communicate to the former that it has done so.

Simplicity and safety are the primary goals. Speed (referring only to the Allocator) is considered a secondary as it is expected that allocation requests will be uncommon at best since objects are expected to live forever and there are limits to the size of the allocation.

For this project, only “allocate” requests need be supported but the interface should not preclude future extensions that allow the “release” of allocated memory for later reuse.

All this will be accomplished so that it is concurrency-safe using only atomic memory operations. No mutex locks will be required.

Association

When the Browser starts, any histograms existing in the Browser’s persistent memory segment are from the previous incarnation or process that ran in between (such as setup). These can be copied out and the shared memory segment reset for the current run while the old data is asynchronously pushed out to servers and then deleted.

Concerns

All designs have problems and limitations. This section attempts to address the biggest of them.

Security

Sharing a memory segment between processes, especially if a process (e.g. the Renderer) is not considered to be a secure process, has the danger of misbehavior in one causing issues in another.

Histogram data is important but not critical. In the last resort, it is acceptable lose some data rather than suffer a catastrophic failure.

Metadata Corruption

A process could overwrite (accidentally or maliciously) the metadata that describes which areas of a shared segment are used and which are free.

- Validate all offsets before use: Pointers cannot be stored within shared memory because each process may map it at a different base address. Thus, all intra-mem references must be held as offsets from that base address. When converting an offset to a pointer for actual use, verify that all access will remain within the memory segment. Also check that a valid block header exists at that offset. Return NULL if there is a problem. Callers *must* handle NULL return values gracefully.
- Validate metadata during allocations and iteration to ensure that returned regions are valid. Return NULL if there is a problem. Callers *must* handle NULL return values gracefully.

Data Corruption

A process could overwrite (accidentally or maliciously) the data inside allocated blocks that are then accessed by other processes.

- Check all values in the histogram for validity against safe values (e.g. `sizeof`) before using them. Fail gracefully if there is a problem.
- Ignore corruption of histogram “counts”. This is the existing policy and is not a problem as such errors do not affect the general result when collated with the entirety of reported metrics.
- Each Renderer should have a distinct memory segment to increase isolation though they could share a memory segment to reduce RAM usage. Unfortunately, such a shared space would mean that a compromised Renderer could access information stored by other Renderers. Though most of that data is innocuous, some measurements or indications of capabilities could end up giving away semi-private information.

Implementation

Development of this feature will happen in three phases:

1. Setup: There is currently no way to export histograms from setup.exe to UMA. This will be accomplished by having setup write its histograms to a file which will be later read by Chrome and uploaded. Success will be indicated by metrics from setup.exe appearing on the UMA dashboard. Getting to this point will be roughly 70% of the total effort.
2. Renderer: While renderers create and export histograms to the browser, they only do so on intervals (every 30 minutes) and lose at exit all information collected since the last export, including information that could indicate why it may exit unexpectedly. Including that "tail data" will be accomplished by creating shared memory segments known to both the renderer and the browser which is only deleted after the next export cycle after the renderer has exited. Success will be indicated by seeing UMA information for things that only happen at renderer-exit, such as navigation to a different domain. Adding this will be roughly 10% of the total effort.
3. Browser: Similarly, the browser will lose at exit all information collected since the last export. Capturing that data will involve adding support in the browser's crash-handler to dump the histograms to disk which will then be loaded and reported during the next start-up. Success will be indicated by seeing on UMA a browser histogram managed by CrashPad. Adding this will be roughly 20% of the total effort.

Inclusion of this by insecure processes such as the Renderer means that security and fail-safety are essential so that malicious intent cannot result in failures by the Browser.

Allocation

Management of UMA histograms in memory falls into two pieces: Memory Management and Histogram Management.

Memory Management

Memory management is a very simple linear block layout. A pointer to the "free" arena. With each allocation, the "free" pointer is extended by the allocation amount (plus block header) and written after which the block is filled in with its size, a "valid" cookie, and any other information. The cookie indicates if the block is empty zero (unallocated), wasted (used to pad the end of a block), or allocated. In the case of a failure in between, analysis can determine that the block is empty by its zero contents.

```
int32 Allocate(int bytes):
      round up bytes to required alignment size
      add size of block header
      if required size > max block size:
            return NULL
      repeat:
            freeptr = atomic read of global_freeptr
            if freeptr + sizeof(block header) + alignment size >= BASE + SIZE:
                  // full
                  return NULL
            if freeptr->cookie != zero:
                  // something was allocated but not recorded as used
```

```
            if freeptr->size > max block size:
                  // corruption detected
                  return NULL
            // could happen from partial persistence where data page changes
            // were saved but not the updated global_freeptr
            newfreeptr = freeptr + freeptr->size
            if newfreeptr > BASE + SIZE:
                  // corruption detected
                  return NULL
            compare-to freeptr and-write newfreeptr at-address &global free
            continue
      // can't cross page boundaries; create empty alloc if necessary
      if page of freeptr != page of freeptr + required size - 1:
            wastesize = remaining in current page
            newfreeptr = freeptr + wastesize
            if not compare-to freeptr and-write newfreeptr at-addr &global_freeptr:
                  // another thread beat us too it; try again
                  continue
            freeptr->size = wastesize
            freeptr->cookie = waste cookie
            continue
      if remaining on page - required size < sizeof(block header) + alignment size:
            // don't leave slice at end of page too small for an allocation
            required size = remaining on page
      newfreeptr = freeptr + required size
      if newfreeptr > BASE + SIZE:
            // corruption detected
            return NULL
      if not compare-to freeptr and-write newfreeptr at-address &global_freeptr:
            // another thread has allocated something; try again
            continue
      freeptr->size = required size
      freeptr->cookie = "allocated" cookie
      return freeptr->data as offset
```

Allocation operations return offsets because those values can be stored in other objects also in the memory segment and used by other processes sharing the memory. Pointers will fail because different processes will likely map the segment at different base addresses. Offsets converted to pointers can be saved locally since the address within a process is guaranteed not to change. Callers **must** gracefully handle NULL results indicative of a bad offset so that damage by one process will not cause segmentation faults on another process.

```
T* GetObject(int32 offset):
      if offset < sizeof(segment header):
            return NULL
      if offset + sizeof(T) > global freeptr:
            return NULL
      return (T*) (BASE + offset + sizeof(block header))
```

Iteration involves keeping a "current" block pointer with an ability to get the "next" block pointer. It is guaranteed, at least in the current implementation of append-only, not to skip and can be continued after reaching the end if additional blocks get added.

Allocated blocks are not iterable by default. If the creating process wants an object to be iterable, it makes a call to set it so which adds that object to the tail of the internal list of such objects.

```
void MakeIterable(int32 offset):

int32 GetFirstIterable(State* state):
      state->last = queue head
      return getNextIterable(state)

int32 GetNextIterable(State* state):
      check for loops
      ...
```

Histogram Management

For performance reasons, direct access to histograms is preferred as it eliminates a costly lookup of a data structure by its name. To accomplish this, most histogram code keeps local “static” pointers around that are only resolved on first use; future use accesses the histogram directly through the static pointers. Histograms are never released from memory once created; they die only when all processes using them have gone.

Each process keeps in local memory a mapping of histogram known names to their respective objects. Each process also has a local iterator to the objects of the shared memory segment. When a request for a histogram cannot be fulfilled by the local mapping, the iterator is advanced to add shared histograms to that local store until the desired histogram is found or the no more fully-initialized histograms are found. By never needing to reset the iterator to the beginning, this step has linear complexity. If the desired histogram is still not found, an allocation request is made and details filled in so other processes will find this new histogram when they next iterate.

It’s possible for iteration to find a histogram that has been allocated but not yet fully initialized. This will have to be checked and the object skipped if that is the case. A process cannot block waiting for for another process to complete the initialization; the other process could have died and never complete the operation.

Should two processes that request a named histogram at the same moment, it’s not guaranteed that both will receive pointers to the same. In some instances, two separate objects will be created and continue to be used by each. The Browser is responsible for merging during reporting all editions of the same histogram name into a single one.

Histograms

There are three ways forward with the Histograms themselves:

1. Replace the existing histograms with shared-memory versions. This isn’t likely practical because the change would simply be too big to go out as a single release.
2. Create parallel SharedHistogram classes that are nearly identical in use (and code) but operate on Sample vectors held in a different memory pool.
3. Change the Histogram classes so that they can operate on either local memory or shared memory depending on parameters during construction.

It's possible that, in the latter two cases, the original code for dealing with local histogram storage could be removed at some point in the future.

Development for option #2 would proceed something like this:

1. Copy existing histogram files into "shared" versions and update them.
2. Have current instantiations create both and compare them on upload to ensure correctness of implementation.
3. Undo duplication. Implement "sharing" mechanism and place selected histograms there for testing.
4. Migrate all appropriate histograms to new method.
5. Migrate remaining histograms and replace old code with new code. (optional)

Development for option #3 would proceed something like this:

1. Modify Histogram class to work as currently instantiated or be able create/access histograms in a shared memory segment.
2. Test new abilities by converting a few histograms to use the new method.
3. Migrate all appropriate histograms to new method.
4. Migrate remaining histograms and replace old code with new code. (optional)

Current histograms use virtual functions and pointers to vectors of bucket data. Neither of these are possible with shared histograms because true pointers cannot be stored in the shared segment -- only offsets which can be dereferenced by any process. Instead, both the histogram's metadata and it's bucket data will need to be held in simple structures; a new type of Histogram class can be loaded with pointers to both those in order to support the current interface.

Control of whether histograms follow the existing methodology or use dedicated memory segments will be controlled by a Finch flag.

Sparse Histograms

Most histograms use a simple vector for their sample counts with atomic increments within; this will be easy to adapt to a block of shared memory. Sparse histograms use a `std::map` and a separate lock for their actions; this will require significant refactoring to operate through shared memory because the "map" structure needs to be shared and there is no "lock" across processes on some platforms, notably Android. A "lockless map" that can make use of multiple arrays of entries, adding new blocks when existing ones fill up.

There are roughly 250 "sparse histograms" in the codebase, many used for returning arbitrary error codes from the operating system. The complexity of supporting sparse histograms means they will likely have to continuing using the existing mechanisms (and suffering their limitations) until work on the simpler versions is nearing completion.

Basic Design

Each process will have `SparseHistogram` object that contains a `std::map` of sample to count *pointer*, different from the existing implementation which has the count embedded directly in the map. The actual counts will reside in shared memory as needed but also need to be able to be found by other processes once created. Individual counts can be allocated from shared memory to make use of its existing lock-free

algorithms for allocation and iteration but waste the “header” space overhead on each sample (16 bytes of header for 4 bytes of count) or a new lockless algorithm created that works within bigger blocks, sharing the overhead across hundreds or even thousand of samples. The first option is likely the best choice to start, going to the more complicated solution if the memory waste is found to be an actual concern.

Testing

UMA is pervasive throughout Chrome. It's essential that changes to it do not introduce errors that could be difficult to detect or impact performance (especially on older hardware).

Deprecated

Don’t bother reading anything below this line. It was once part of the document but has been removed in favor of other content.

A special allocation module will be required that can return (and release) space from a dedicated block of memory, one that is somewhat separate from the main process heap and can be accessed in a read/write memory from multiple independent processes. This could be utilized directly via alloc/release calls or indirectly via C++'s new operator and a custom allocator.

While the module that handles memory management within the shared memory segment is not expected to manage parallel access, it will nonetheless require some sort of mutex so that multiple allocation/deallocation requests cannot corrupt the internal data structures. Since allocation requests are expected to be rare, we can export this same mutex for use by calling processes; it should be able to acquire the lock once and then make multiple allocation requests, releasing the lock once they're all done.

```
bool is_locked = false;

void Lock():
     if is_locked: return
     acquire mutex
     is_locked = true

void Unlock():
     ASSERT(not is_locked)
     is_locked = false
     release mutex
```

The module will keep its shared internal values starting at offset zero. This ensures that no memory request will ever return an offset of zero (NULL). As well as malloc support, there should be support for a static structure held just after the internal values. This will allow processes to store a "starting values" descriptor used by other processes to locate allocated objects. A version number associated with this structure will be kept so that a newly upgraded binary can detect data from the previous binary.

```
bool GetDescriptor(Descriptor** desc, size_t* desc_size, int32* desc_version):
     if shared_current_desc_version == 0:
          return false
     *desc = (Descriptor*) (BASE + sizeof(internal values))
     *desc_size = shared_current_desc_size
     *desc_version = shared_current_desc_version
     heap_base = BASE + sizeof(internal values) + shared_current_desc_size
     heap_size = SIZE - sizeof(internal values) - shared_current_desc_size
     return true

void CreateDescriptor(Descriptor* desc, size_t desc_size, int32 desc_version):
     ASSERT(desc_version > 0)
     ASSERT(shared_current_desc_version == 0)
     shared_current_desc_version = desc_version
     shared_current_desc_size = desc_size
     copy *desc to *(BASE + sizeof(internal values))
```

```
Descriptor* GetOrCreateDescriptor(Descriptor* desc, size_t* desc_size,
                                  int32* desc_version):
      Lock()
      Descriptor* existing_desc
      if not GetDescriptor(&existing_desc, desc_size, desc_version):
            CreateDescriptor(desc, *desc_size, *desc_version)
            GetDescriptor(&existing_desc, desc_size, desc_version)
      Unlock()
      return existing_desc
```

Initializing the module means passing it a shared memory segment and a mutex. After acquiring the mutex lock, it then initializes the memory if not already done followed by its own internal state based on the shared memory values. Detection of already-initialized memory will depend on validating the state of the "internal values" consisting of a cookie, the shared-memory-segment size, and perhaps others. The internal state of the heap allocation system needs to also be checked.

If the segment is *not* valid, it needs to initialized. However, if the segment is already open by other processes (i.e. it was previously initialized but has been corrupted) then there is a problem. Re-initializing the segment will likely lead to new corruption as other processes write via pointers (temporary or static) that are no longer valid. Quite simply, as long as the segment is open by others, it cannot be initialized and all attempts to access it must safely fail with those failures being handled gracefully by the caller.

```
bool IsInitialized():
      return (internal values are reasonable)

bool ValidateAll():
      Lock()
      is_valid = ValidateInternalValues() && ValidateHeap()
      Unlock()
      return is_valid

bool Initialize(SharedMemory mem, Mutex m):
      BASE = mem.base
      SIZE = mem.size
      mutex = m
      if not IsInitialized():
            InitializeInternal()
            InitializeHeap()
            return true
      return ValidateAll()
```

Pointers cannot be stored inside the shared memory segment because there is no knowing at what base address the segment will be mapped within a process's address space. All references must be done with *offsets* that are then converted to local pointers. Those pointers can be stored process-local with the knowledge that the base address will not change once mapped **but only if** they are confident that the object will never be *deallocated* by either the local process or any other process (standard multi-processing things).

It's not possible for this code to guarantee that offsets point to valid objects but can at least ensure it points to valid memory and thus not cause segmentation violations. The data and meta-data of the shared segment could still be corrupted by a broken process so calling code will have to be cautious and not be overly reliant if they are worried about such. Additional meta-data (cookie, object-size, etc.) could be stored before/after the object to provide additional validity checks.

```
// Callers MUST check for NULL return values and handle appropriately.
T* GetObject(int offset, T* not_used):
     if not is_valid:
          return NULL
     if offset < heap_base or offset + sizeof(T) > heap_base + heap_size:
          return NULL
     // other checks?
     return (T*) (BASE + offset);
```

Heap management is a simple version of first-fit malloc with a global lock. This is expected to be "good enough" for the current intended use.