libpas/bmalloc

Background

Problems:

- Performance ⇒ Browsers allocated/deallocated millions of objects so malloc must be fast
- Security ⇒ Lots of exploits battle it out on the heap (UAF, Type Confusion, etc.)
- Fragmentation ⇒ Long-running processes such as browser tabs can lead to fragmented memory

Legacy bmalloc?

Originally a bump-allocator optimized for pure speed

Allocation: Increment a pointer (ptr + size)

Deallocation: More complicated batch freeing that maintains the fast path

Problem: Lacks more granular isolation and making bmalloc more susceptible to heap memory attacks

Frontend memory allocator for WebKit (JavascriptCore)

libpas

Designed by Phil Pizlo

Goals: Beat bmalloc in single-threaded performance while being more memory efficient and secure!

How:

- Isoheaps (isolated heaps)
- External metadata
- Many many heaps

Type Safety

One heap per type prevents use-after-free exploits

High Performance

2% speed improvement on Speedometer2

Memory Efficient

8% memory reduction across benchmarks

Isoheaps

The Problem

Traditional memory allocators reuse freed memory for any type, enabling use-after-free vulnerabilities where an attacker can manipulate object A through a dangling pointer after its memory is reallocated as object B.

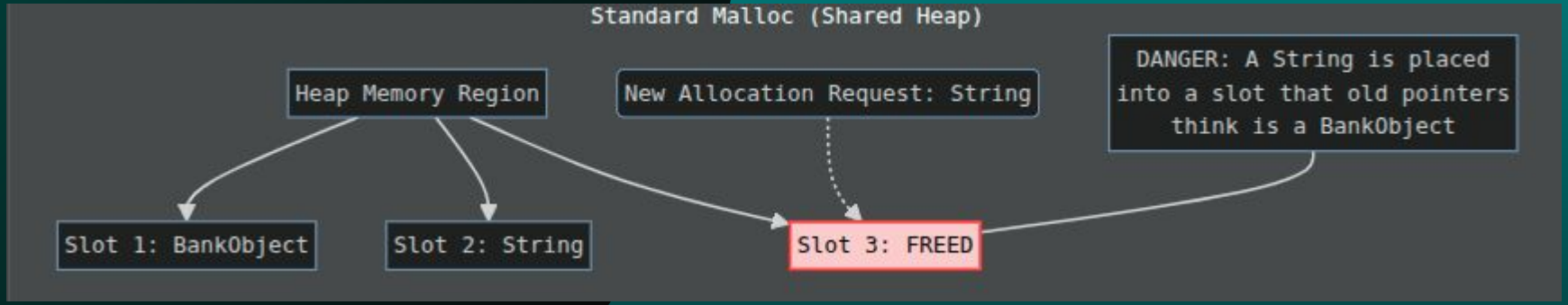
The Solution

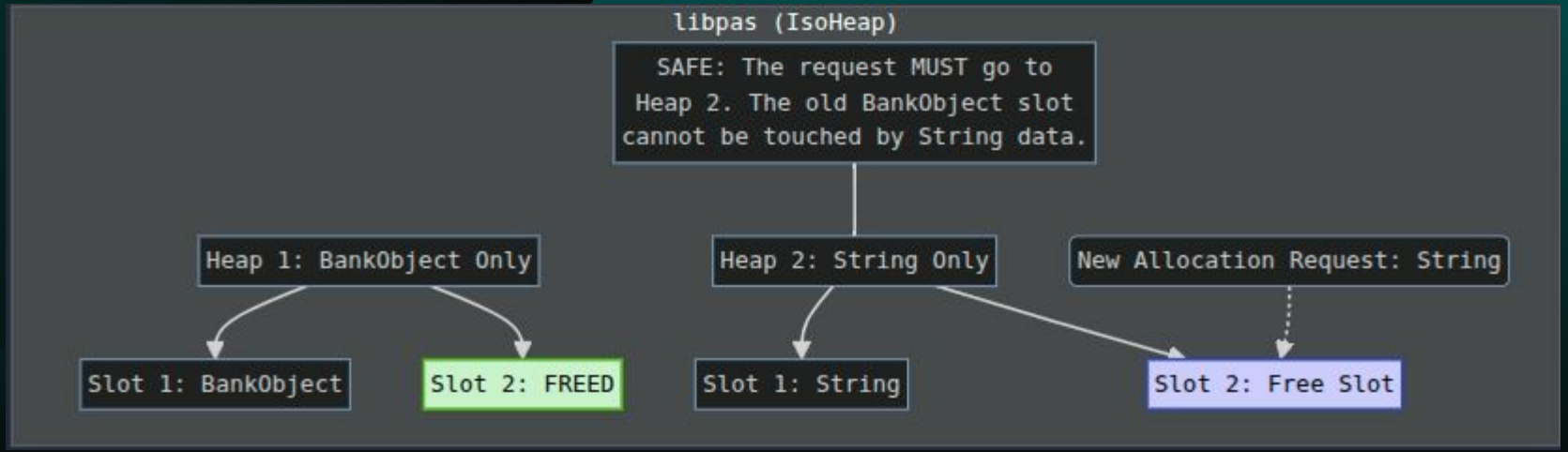
IsoHeap ensures that once a virtual address has been used for a specific type, only objects of that exact type can ever exist at that address for the program's lifetime. This makes type confusion attacks impossible.

Key Principle: One Heap Per Type

libpas was designed with the goal of enabling comprehensive isoheaping of *all* malloc/new callsites in C/C++ programs, obviating the need for complex ownership type systems.

Isoheaps


Standard Malloc (Shared Heap)
Heap Memory Region
New Allocation Request: String
DANGER: A String is placed into a slot that old pointers think is a BankObject
Slot 1: BankObject
Slot 2: String
Slot 3: FREED


libpas (IsoHeap)
SAFE: The request MUST go to Heap 2. The old BankObject slot cannot be touched by String data.
Heap 1: BankObject Only
Heap 2: String Only
New Allocation Request: String
Slot 1: BankObject
Slot 2: FREED
Slot 1: String
Slot 2: Free Slot

3 Allocator Algorithms

Segregated Heap

Simple segregated storage allocator. Each page contains objects of a single size class. Extremely fast with no atomic operations needed for typical allocations.

Bitfit Heap

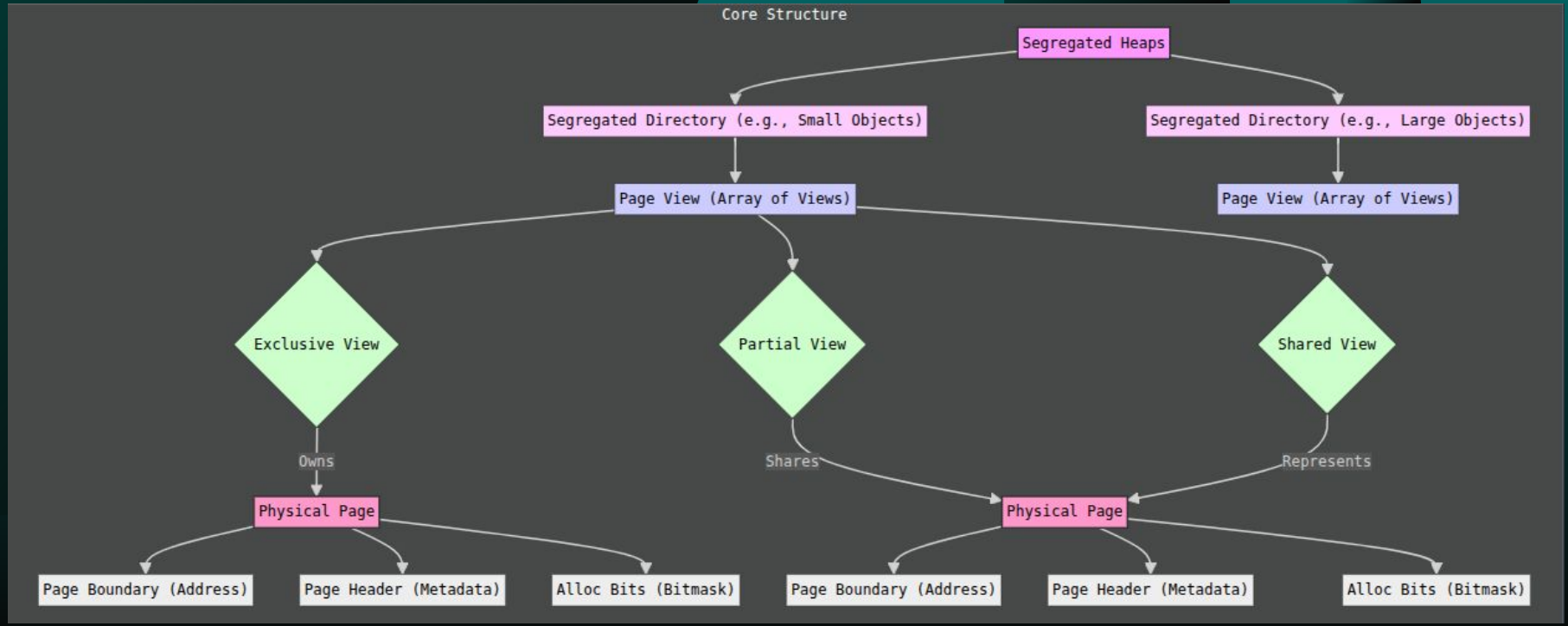
First-fit allocation using bit-per-granule tracking. Highly space-efficient for medium objects but not suitable for isoheaps due to type mixing.

Large Heap

Cartesian-tree-based first-fit for arbitrary sizes. Remembers type of every free object and original boundaries for strong type safety.

Segregated Heaps and TLCs

Provided great performance by leveraging Thread-local caches (TLCs)



Segregated Heaps and TLCs

Fast Path Operation

Typical allocation:

- libpas checks a TLC first
- If empty, it grabs a ready-to-use page from the directory.
- It uses a bit-vector to find the next free slot (CPU cache friendly)

No atomic operations needed

Why It's Fast

Segregated storage means each thread can work independently on its size classes without coordination.

The allocator beats the original bmalloc at object churn performance through this lock-free design.

Segregated Heap Allocation

```
// From pas_local_allocator_inlines.h lines 1547-1574
remaining = allocator->remaining;
object_size = allocator->object_size;

if (remaining) {
    uintptr_t result;

    /* This is the fastest fast path for allocation. It will happen if:
       - We found a totally empty page.
       - We are in alloc_bits_in_page refill mode. */
    allocator->remaining = remaining - object_size;
    result = allocator->payload_end - remaining;

    return pas_allocation_result_create_success(result);
}
```

Segregated Heap Allocation

```
// From pas_local_allocator_inlines.h lines 113-130
static inline void pas_local_allocator_set_up_bump(pas_local_allocator* allocator,
                                                   uintptr_t page_boundary,
                                                   uintptr_t begin,
                                                   uintptr_t end,
                                                   pas_segregated_heap*
segregated_heap,
                                                   pas_segregated_page_config
page_config)
{
    allocator->payload_end = end;
    allocator->remaining = (unsigned)(end - begin);
    allocator->current_offset = 0;
    allocator->end_offset = 0;
    allocator->current_word = 0;
    pas_compiler_fence();
    allocator->page_ish = page_boundary;
}
```

Segregated Heap Allocation

```
// From pas_local_allocator_inlines.h lines 1402-1528
static PAS_ALWAYS_INLINE pas_allocation_result
pas_local_allocator_try_allocate_with_free_bits(
   pas_local_allocator* allocator,
   pas_allocation_mode allocation_mode,
   pas_segregated_page_config page_config)
{
        . . .

   // Get current word of free bits from local allocator cache
   current_word = allocator->current_word;
   page_ish = allocator->page_ish;

   if (PAS_UNLIKELY(!current_word)) {
       // Advance to next word that has free bits
       end_offset = allocator->end_offset;
       current_offset = allocator->current_offset;

       do {
           ++current_offset;
           page_ish += 64u << page_config.base.min_align_shift;
           if (current_offset >= end_offset)
               return pas_allocation_result_create_failure(); // Need refill
           current_word = allocator->bits[current_offset];
       } while (!current_word);

       allocator->current_offset = current_offset;
       allocator->page_ish = page_ish;
   }
   // Find first free bit and clear it
   found_bit_index = (uintptr_t)__builtin_ctzll(current_word);
   current_word &= ~PAS_BITVECTOR_BIT_MASK64(found_bit_index);
   allocator->current_word = current_word;

   result = page_ish + (found_bit_index << page_config.base.min_align_shift);

   return pas_allocation_result_create_success(result);
}
```

Segregated Heap Allocation

```
// From pas_local_allocator_inlines.h - refill flow (conceptual)
static PAS_ALWAYS_INLINE bool
pas_local_allocator_refill_with_known_config(
    pas_local_allocator* allocator,
    pas_allocation_mode allocation_mode,
    pas_allocator_counts* counts,
    pas_segregated_page_config page_config)
{
    // 1. Stop allocating in current view
    pas_segregated_view_did_stop_allocating(current_view, page, page_config);

    // 2. Find an eligible view with free objects
    pas_segregated_view new_view =
        pas_segregated_directory_take_first_eligible(directory, pas_lock_is_not_held);

    // 3. Prepare the view (may commit memory if needed)
    new_view = pas_segregated_view_will_start_allocating(new_view, page_config);

    // 4. Set up the local allocator with free bits from the page
    pas_local_allocator_prepare_to_allocate(allocator, view_kind, view, page, directory, page_config);

    return true;
}
```

Segregated Heap Allocation

```
// From pas_segregated_view_allocator_inlines.h lines 56-234
static PAS_ALWAYS_INLINE pas_segregated_view
pas_segregated_view_will_start_allocating(pas_segregated_view view,
                                          pas_segregated_page_config page_config)
{
    switch (pas_segregated_view_get_kind(view)) {
    case pas_segregated_exclusive_view_kind:
    case pas_segregated_ineligible_exclusive_view_kind: {
        exclusive = (pas_segregated_exclusive_view*)pas_segregated_view_get_ptr(view);

        if (!exclusive->is_owned) {
            // Allocate or commit a page
            if (!exclusive->page_boundary) {
                // Allocate new page from page allocator
                exclusive->page_boundary = page_config.page_allocator(
                    size_directory->heap,
                    heap_lock_hold_mode ? NULL : &transaction,
                    pas_segregated_page_exclusive_role);

                page_config.base.create_page_header(
                    exclusive->page_boundary,
                    pas_page_kind_for_segregated_variant_and_role(...),
                    pas_lock_is_held);
            } else {
                // Recommit existing page
                pas_page_malloc_commit(exclusive->page_boundary,
                                       page_config.base.page_size,
                                       page_config.base.heap_config_ptr->mmap_capability);
            }

            // Construct the page structure
            pas_segregated_page_construct(
                pas_segregated_page_for_boundary_unchecked(exclusive->page_boundary, page_config),
                ineligible_owning_view,
                was_stolen,
                (const pas_segregated_page_config*)page_config.base.page_config_ptr);

            exclusive->is_owned = true;
        }
        return view;
    }
    //... partial view handling
    }
}
```

Segregated Heap Allocation

```
// From pas_local_allocator_inlines.h lines 429-500
static PAS_ALWAYS_INLINE void
pas_local_allocator_prepare_to_allocate(
    pas_local_allocator* allocator,
    pas_segregated_view_kind view_kind,
    void* view,
    pas_segregated_page* page,
    pas_segregated_size_directory* directory,
    pas_segregated_page_config page_config)
{
    page_boundary = (uintptr_t)pas_segregated_page_boundary(page, page_config);

    // If page is empty, use bump allocation
    if (view_kind == pas_segregated_exclusive_view_kind && !page->emptiness.num_non_empty_words) {
        payload_begin = page_boundary + data->offset_from_page_boundary_to_first_object;
        payload_end = page_boundary + data->offset_from_page_boundary_to_end_of_last_object;

        page->emptiness.num_non_empty_words = data->full_num_non_empty_words;

        pas_local_allocator_make_bump(allocator, page_boundary, payload_begin, payload_end,
                                      directory->heap, page_config);

        // Copy full alloc bits to page
        memcpy(page->alloc_bits,
               pas_compact_tagged_unsigned_ptr_load_non_null(&data->full_alloc_bits),
               pas_segregated_page_config_num_alloc_bytes(page_config));
        return;
    }

    // Otherwise scan existing bits for free slots
    pas_local_allocator_set_up_free_bits(allocator, view_kind, view, directory,
                                         page, page_boundary, page_config);
}
```

Segregated Heap Deallocation

```
// From pas_segregated_page_inlines.h lines 510-526
static PAS_ALWAYS_INLINE void pas_segregated_page_deallocate(
    uintptr_t begin,
    pas_lock** held_lock,
    pas_segregated_deallocation_mode deallocation_mode,
    pas_thread_local_cache* thread_local_cache,
    pas_segregated_page_config page_config,
    pas_segregated_page_role role)
{
    pas_segregated_page* page;

    // Find the page from the address
    page = pas_segregated_page_for_address_and_page_config(begin, page_config);

    // Switch to the page's lock
    pas_segregated_page_switch_lock(page, held_lock, page_config);

    // Do the actual deallocation
    pas_segregated_page_deallocate_with_page(
        page, begin, deallocation_mode, thread_local_cache, page_config, role);
}
```

Segregated Heap Deallocation

```
// From pas_segregated_page_inlines.h lines 325-508
static PAS_ALWAYS_INLINE void
pas_segregated_page_deallocate_with_page(pas_segregated_page* page,
                                         uintptr_t begin,
                                         pas_segregated_deallocation_mode deallocation_mode,
                                         pas_thread_local_cache* thread_local_cache,
                                         pas_segregated_page_config page_config,
                                         pas_segregated_page_role role)
{
    // Calculate bit index from address
    bit_index_unmasked = begin >> page_config.base.min_align_shift;
    word_index = pas_modulo_power_of_2(
        (begin >> (page_config.base.min_align_shift + PAS_BITVECTOR_WORD_SHIFT)),
        page_config.base.page_size >> (page_config.base.min_align_shift + PAS_BITVECTOR_WORD_SHIFT));

    word = page->alloc_bits[word_index];

    // Check for double-free (platform-specific optimization)
    if (page_config.check_deallocation) {
#if !PAS_ARM && !PAS_RISCV
        // x86: Use atomic BT (bit test) instruction
        __asm__ volatile (
            "btrl %1, %0\n\t"
            "jc 0f\n\t"
            "movq %2, %%rdi\n\t"
            "call pas_segregated_page_deallocation_did_fail\n\t"
            "0:"
            : "+r"(new_word)
            : "r"((unsigned)bit_index_unmasked), "r"(begin)
            : "memory");
#else
        // ARM/RISC-V: Manual check
        unsigned bit_mask = PAS_BITVECTOR_BIT_MASK(bit_index_unmasked);
        if (PAS_UNLIKELY(!(word & bit_mask)))
            pas_segregated_page_deallocation_did_fail(begin);
        new_word = word & ~bit_mask;
#endif
    } else
        new_word = word & ~PAS_BITVECTOR_BIT_MASK(bit_index_unmasked);

    // Clear the allocation bit
    page->alloc_bits[word_index] = new_word;
```

Segregated Heap Deallocation

```
// Continuing from pas_segregated_page_deallocate_with_page...

// If word became zero, we may need to notify eligibility
if (!pas_segregated_page_config_enable_empty_word_eligibility_optimization_for_role(page_config, role)
    || !new_word) {
    pas_segregated_view owner = page->owner;

    switch (role) {
    case pas_segregated_page_exclusive_role: {
        if (pas_segregated_view_get_kind(owner) != pas_segregated_exclusive_view_kind) {
            // Notify that this page now has free space
            pas_segregated_exclusive_view_note_eligibility(
                pas_segregated_view_get_exclusive(owner),
                page, deallocation_mode, thread_local_cache, page_config);
        }
        break;
    }

    case pas_segregated_page_shared_role: {
        // For partial views, notify the specific partial view
        pas_segregated_shared_handle* shared_handle;
        pas_segregated_partial_view* partial_view;

        shared_handle = pas_segregated_view_get_shared_handle(owner);
        partial_view = pas_segregated_shared_handle_partial_view_for_index(
            shared_handle, bit_index, page_config);

        if (!partial_view->eligibility_has_been_noted)
            pas_segregated_partial_view_note_eligibility(partial_view, page);
        break;
    }
    }
}
```

Segregated Heap Deallocation

```
    // For medium pages with sub-page granules
    if (page_config.base.page_size > page_config.base.granule_size) {
        bool did_find_empty_granule;

        // Decrement use counts for the freed object's granule range
        did_find_empty_granule = pas_page_base_free_granule_uses_in_range(
            pas_segregated_page_get_granule_use_counts(page, page_config),
            offset_in_page,
            offset_in_page + object_size,
            page_config.base);

        // If a granule became empty, mark page for potential decommit
        if (did_find_empty_granule)
            pas_segregated_page_note_emptiness(page,
pas_note_emptiness_keep_num_non_empty_words);
    }
```

Segregated Heap Deallocation

```
    // If dword became zero, update emptiness tracking
    if (!new_word) {
        // Guard against double-free
        if (PAS_UNLIKELY(!word))
            pas_segregated_page_deallocation_did_fail(begin);

        uintptr_t num_non_empty_words =
page->emptiness.num_non_empty_words;
        if (!--num_non_empty_words) {
            // Page is now completely empty - can be decommitted
            pas_segregated_page_note_emptiness(page,
pas_note_emptiness_clear_num_non_empty_words);
        } else
            page->emptiness.num_non_empty_words = num_non_empty_words;
    }
}
```

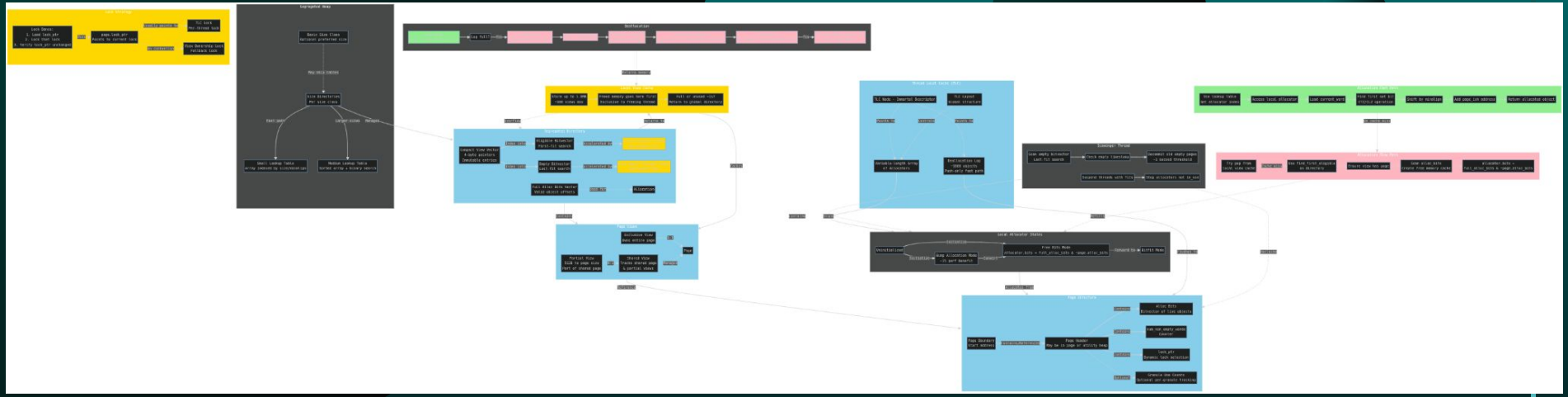
Segregated Heaps: Key Data Structs

```
// From pas_segregated_page.h lines 71-122
struct pas_segregated_page {
    pas_page_base base;

    bool is_in_use_for_allocation;       // Exclusive pages only
    bool eligibility_notification_has_been_deferred;
    bool is_committing_fully;

    unsigned object_size;                // Cached for performance
    pas_lock* lock_ptr;

    pas_segregated_page_emptiness emptiness;  // Tracks empty words

    pas_segregated_view owner;           // exclusive or shared_handle view
    pas_allocator_index view_cache_index;

    unsigned alloc_bits[1];              // Bitmap: 1 = allocated, 0 = free
};
```

Segregated Heaps and TLCs

```
// From pas_segregated_heap.h lines 83-110
struct pas_segregated_heap {
    pas_heap_runtime_config* runtime_config;
    pas_heap* parent_heap;

    // Fast lookup for small sizes
    pas_allocator_index* index_to_small_allocator_index;
    pas_compact_atomic_segregated_size_directory_ptr* index_to_small_size_directory;

    // Head of size directory linked list
    pas_compact_atomic_segregated_size_directory_ptr basic_size_directory_and_head;

    pas_segregated_heap_rare_data_ptr rare_data;  // For medium sizes
    pas_compact_atomic_bitfit_heap_ptr bitfit_heap;

    unsigned small_index_upper_bound;  // Size threshold for fast lookup
};
```

Segregated Heaps and TLCs



Metadata Protection

Common Attacks: Attackers can corrupt chunk metadata (prev_size, etc.)

libpas Defense:

- Metadata is stored in a separate memory region from the user's object
- Compact pointers allows efficient metadata lookup

The Scavenger

The scavenger is a dedicated background thread responsible for returning memory back to the OS.

Resources

- https://github.com/WebKit/WebKit/blob/main/Source/bmalloc/libpas/Documentation.md