Mimalloc

Oleg Lazari

Agenda

1. Problem With Malloc
2. Glibc Malloc Deep Dive
3. Mimalloc Architecture Overview
4. Free List Sharding
5. Multi-Sharding / Thread Safety
6. Temporal Cadence / Deferred Ops
7. Performance Benchmarks
8. Security Features & Takeaways

Why Do We Need Better Allocators?

4 Competing Demands

Performance

Allocation/free must be fast. Microseconds matter at scale for many operations.

Concurrency

Multi-threaded apps need lock-free or low-contention allocation.

Fragmentation

Memory must be used efficiently. Internal & External fragmentation waste resources.

Security

Heap exploits are common. Address space randomization, guard pages, and metadata protection is required.

Glibc Malloc (ptmalloc2) Architecture

Arenas
- Independent Memory Regions (Thread Scalability)
- Main arena uses sbrk(), secondary uses mmap()
- Each arena has its own mutex
- Default: 8 x CPU cores maximum

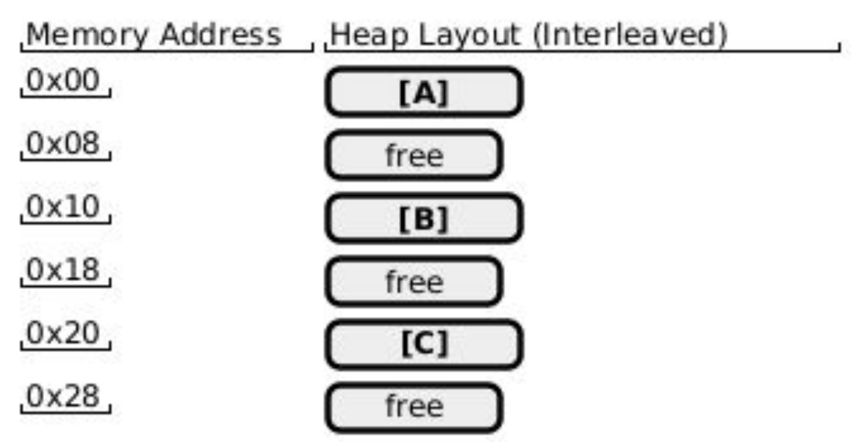
Chunks
- Fundamental allocation unit
- Contains metadata (size, flags) + user data
- Flags in low bits: IS_MMAPED, NON_MAIN_ARENA, PREV_INUSE, etc.

Bins (Free Lists)
- Freed Chunks organized by Size class into Linked Lists
- 128 bins total

Tcache (glibc 2.26+)
- Per-thread cache for small allocations
- 64 bins, max 7 chunks each
- Bypass Arena Locks

Bin Types Table (64-bit)	
Fastbins	32-176 bytes | LIFO, singly-linked, no coalescing
Smallbins	≤1024 bytes | FIFO, doubly-linked, exact size
Largebins	>1024 bytes | Sorted by size, best-fit search
Unsorted	Any size | Staging area, sorted on demand


glibc malloc Allocation Path
1. tcache
per-thread cache
miss
2. fastbin
≤176 bytes
miss
3. smallbin
≤1024 bytes
miss
4. unsorted
coalesce + sort
miss
5. large/top
best-fit search
Lock-free
O(1)
Arena lock
O(n) search




```c
void* malloc(size_t size) {
    // 1. Check tcache first (no lock)
    if (tcache && size <= TCACHE_MAX) {
        if (tcache->entries[idx]) return pop();
    }
    // 2. Get arena lock (!)
    lock(arena->mutex);
    // 3-5. Search bins...
    unlock(arena->mutex);
}
```

Mimalloc’s Design Philosophy

- Instead of one large free list per size class, have many smaller lists per page.
- Things allocated close in time get allocated close in memory.

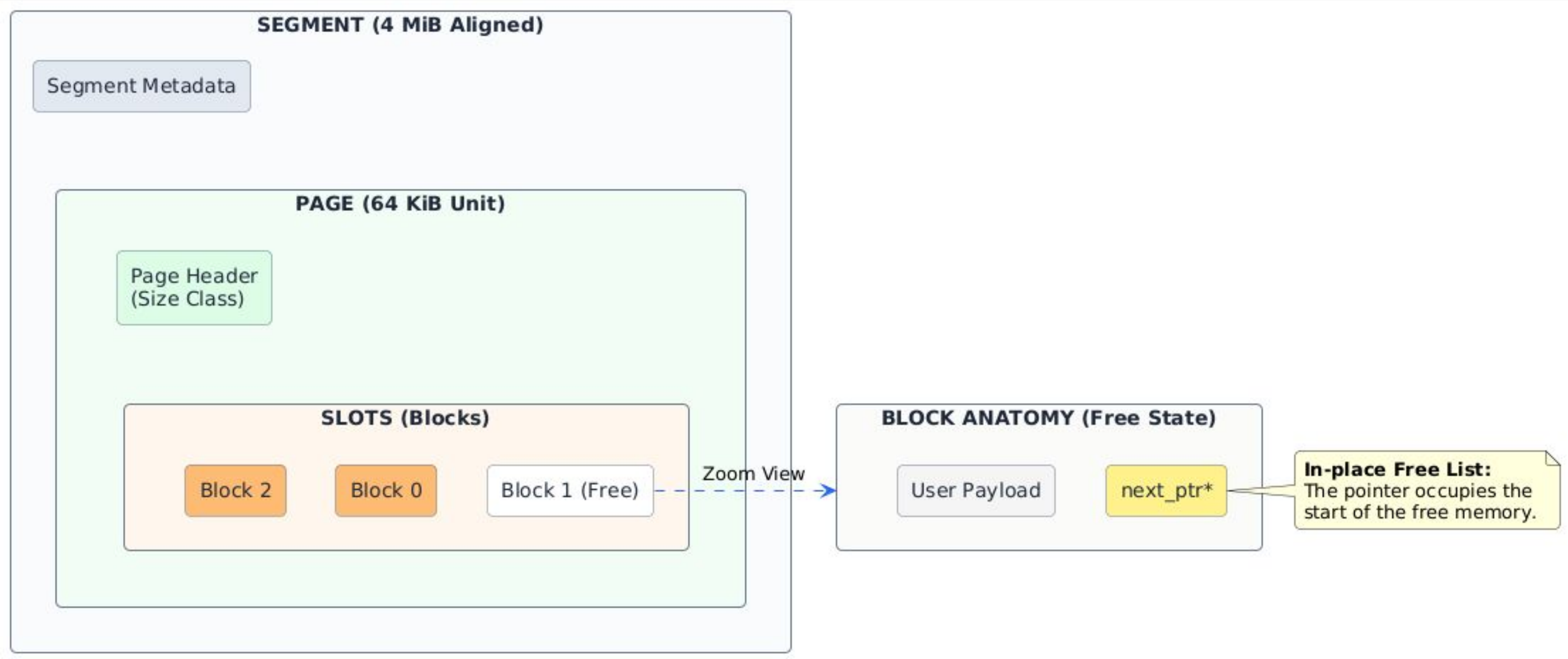
Design Goals

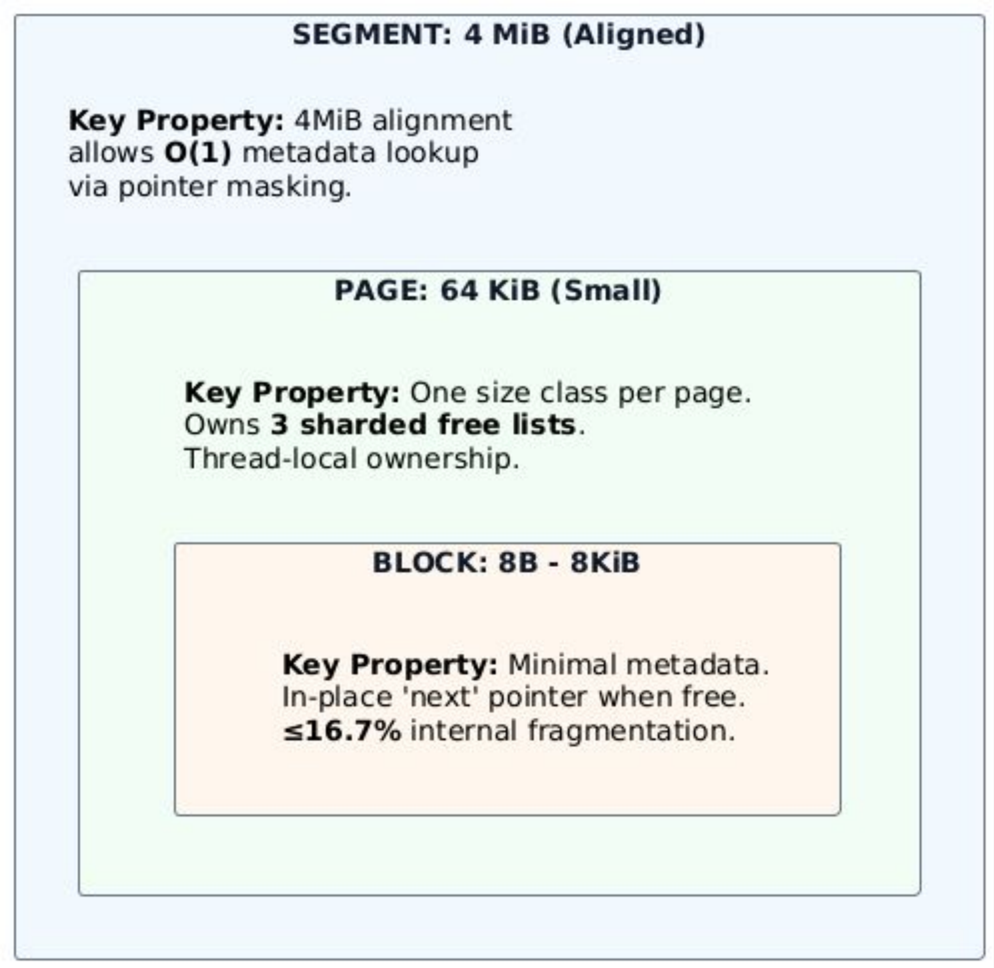
- Small & consistent (~10k LOC vs ~25k for glibc)
- No locks (atomic operations only, lock-free design)
- Bounded (worst-case O(1) alloc time, ~0.2% metadata)
- Reference counting friendly (deferred free support)
- Single fast-path conditional (highly optimized)

Features:

- Free List Sharding
- Free List Multi-Sharding
- Temporal Cadence
- Eager Page Purging

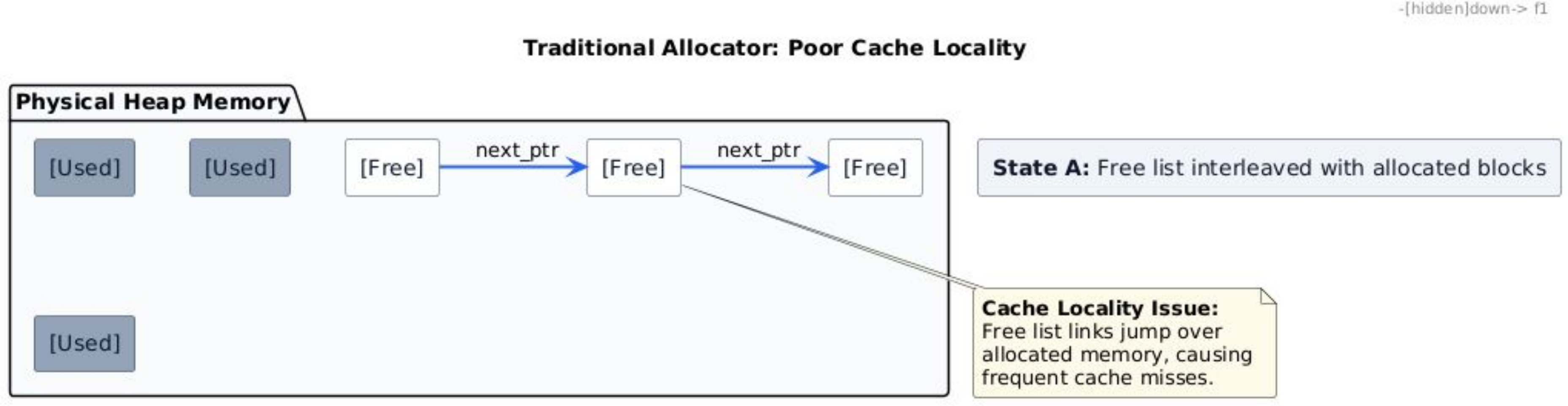
Memory Hierarchy

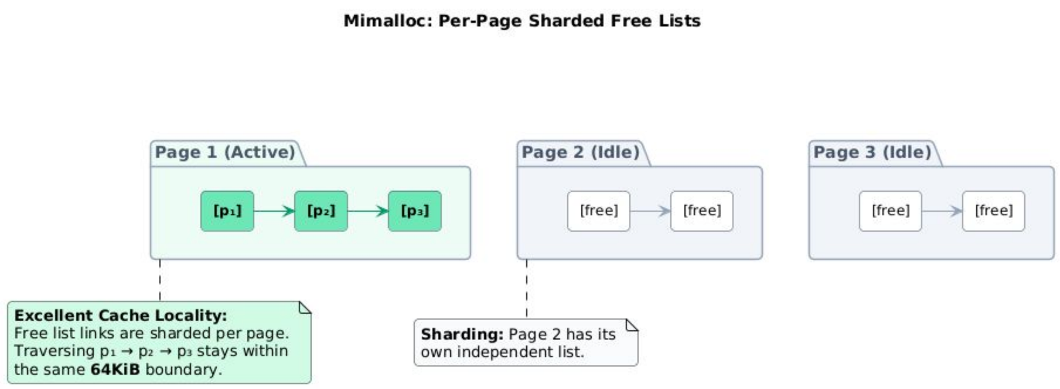




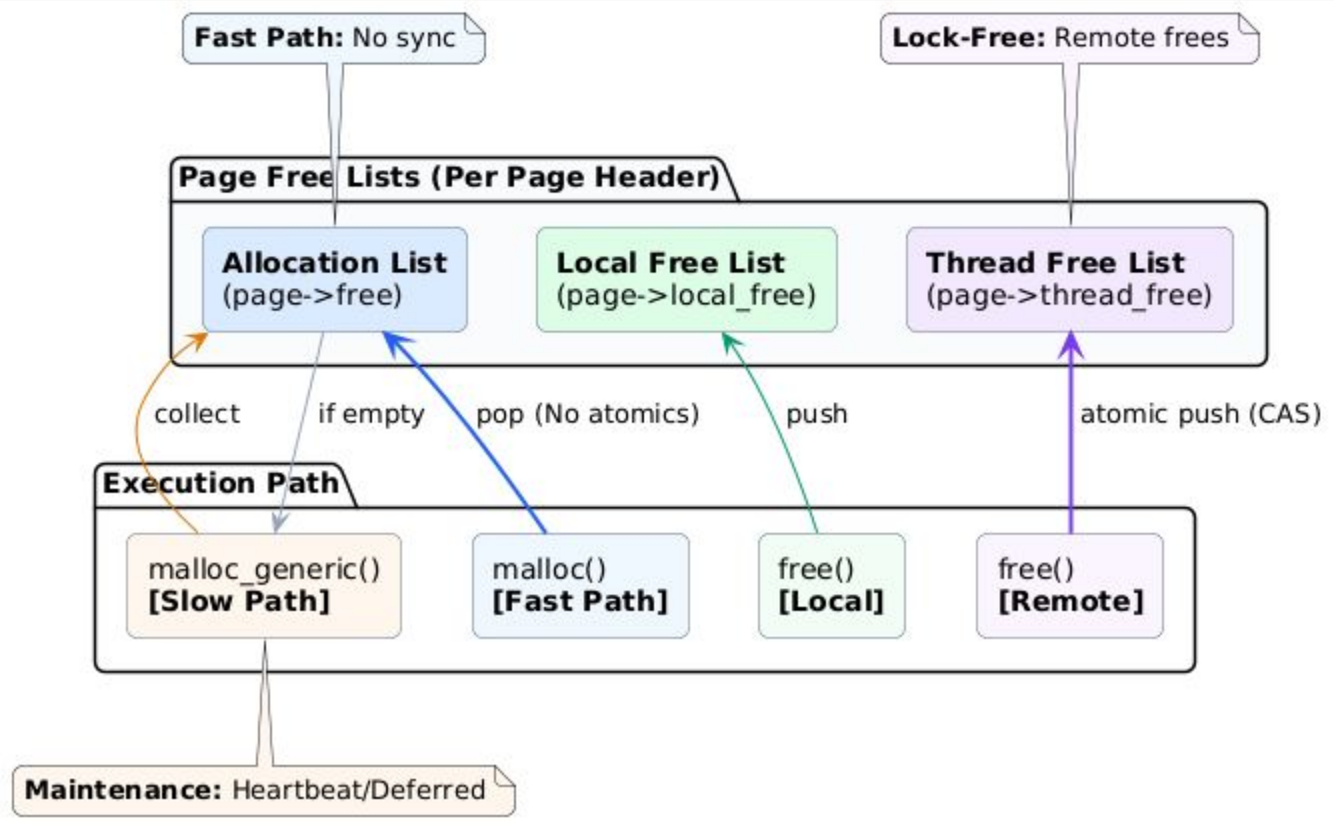

```
segment = (ptr & ~(4MB-1));
page = segment->pages[(ptr - segment) >> page_shift];
// No pointer chasing or hash tables!
```

Free List Sharding





Three Free Lists Per Page



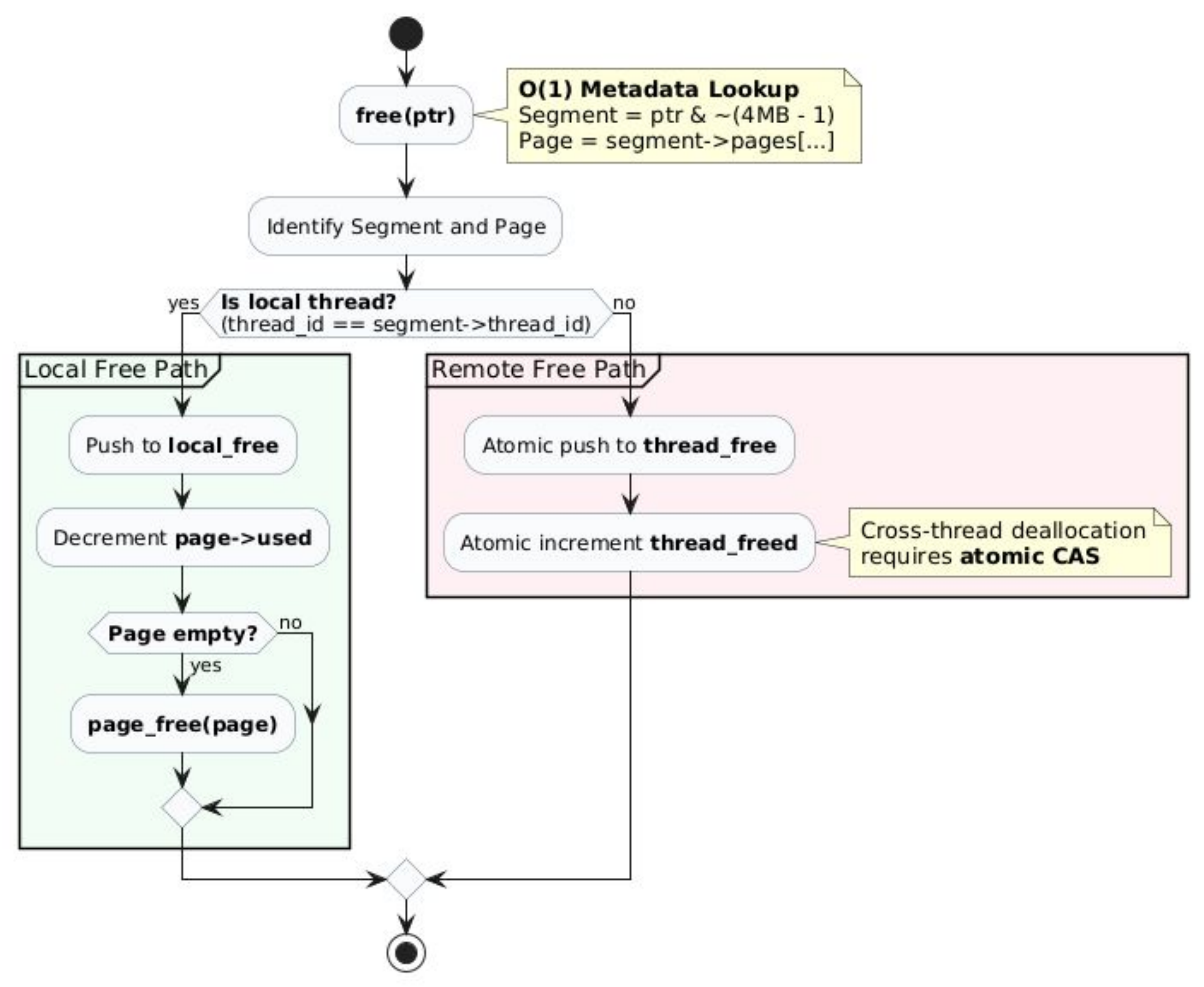
List	Purpose	Operations
page->free	Primary allocation source	Pop (fast path, no atomics)
page->local_free	Local thread's frees	Push (no atomics)
page->thread_free	Remote thread's frees	Atomic push (single CAS)

The malloc() Fast Path

```
void* malloc_small(size_t n) {  // 0 < n <= 1024
    heap_t* heap = tlb;                            // Thread-local heap
    page_t* page = heap->pages_direct[(n+7)>>3];   // Direct array lookup
    block_t* block = page->free;                   // Get head of free list

    if (block == NULL)
        return malloc_generic(heap, n);            // SINGLE conditional!

    page->free = block->next;                      // Pop from list
    page->used++;                                  // Track usage
    return block;
}
```

The free() Implementation

```c
void free(void* p) {
    // O(1) segment lookup via pointer masking
    segment_t* segment = (segment_t*)((uintptr_t)p & ~(4*MB));
    if (segment == NULL) return;

    // O(1) page lookup via shift
    page_t* page = &segment->pages[(p - segment) >> segment->page_shift];
    block_t* block = (block_t*)p;

    if (thread_id() == segment->thread_id) {
        // LOCAL FREE - no atomics!
        block->next = page->local_free;
        page->local_free = block;
        page->used--;
        if (page->used - page->thread_freed == 0)
            page_free(page);
    } else {
        // REMOTE FREE - single CAS
        atomic_push(&page->thread_free, block);
        atomic_incr(&page->thread_freed);
    }
}
```



free(ptr)
O(1) Metadata Lookup
Segment = ptr & ~(4MB - 1)
Page = segment->pages[...]
Identify Segment and Page
Is local thread?
(thread_id == segment->thread_id)
yes
no
Local Free Path
Push to local_free
Decrement page->used
Page empty?
yes
no
page_free(page)
Remote Free Path
Atomic push to thread_free
Atomic increment thread_freed
Cross-thread deallocation
requires atomic CAS

Temporal Cadence

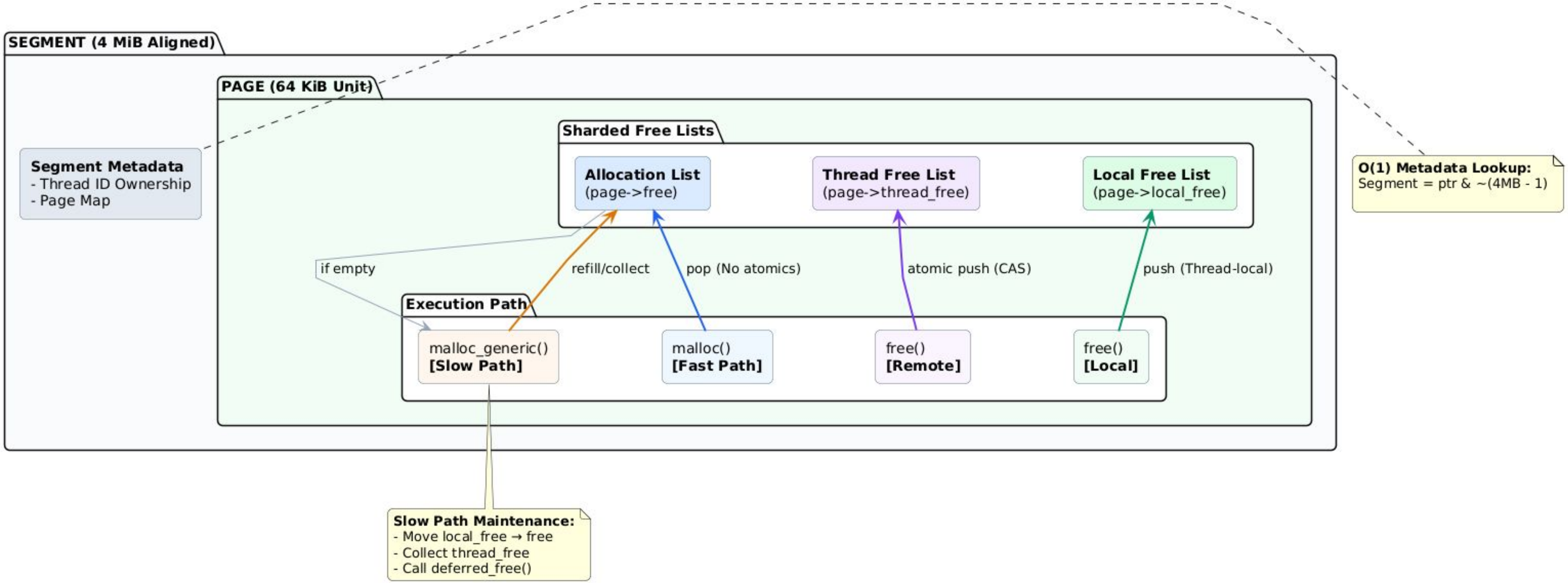
- How do you guarantee maintenance tasks run regularly?
- Deferred reference count decrements
- Collecting cross-thread frees
- Heartbeat for timeouts

Solution

Since we allocate from `page->free` and free to `page->local_free`, the allocation list must become empty after a fixed number of allocations (at most the page capacity).

When empty → `malloc_generic()` runs → maintenance happens!

Mimalloc Consolidated Architecture: Segment to Block Flow



malloc_generic() Does

1. Move `local_free` → `free`
2. Atomically swap entire `thread_free` list
3. Call user-defined `deferred_free()` callback
4. Provide deterministic heartbeat

Handling Full Pages

GCC Performance Bottleneck

- **Persistence of Large Objects:** GCC allocates numerous large objects that remain live throughout execution, resulting in over 18,000 fully occupied pages.
- **Linear Search Inefficiency:** Originally, `malloc_generic()` performed a linear search through every page—including full ones—leading to a significant **30% performance slowdown**.

The Solution: Intelligent Page Management

- **The `full_list` Separation:** Fully occupied pages are now moved to a dedicated `full_list`.
- **Search Optimization:** These pages are bypassed during the allocation search, keeping the fast path efficient.
- **Dynamic Re-integration:** Once an object is freed and the page is no longer full, it is moved back into the searchable heap.

Multi-threading Complexity: Remote Freeing

- **Ownership Constraints:** A remote thread cannot unilaterally move a page to a different list because it does not own the metadata.
- **Signaling Requirement:** When a remote thread frees an object in a "full" page, it must signal the owning heap to re-evaluate the page's status.
- **Atomic Synchronization:** This process utilizes atomic CAS (Compare-and-Swap) to safely return the block to the owner’s `thread_free` list for later collection.

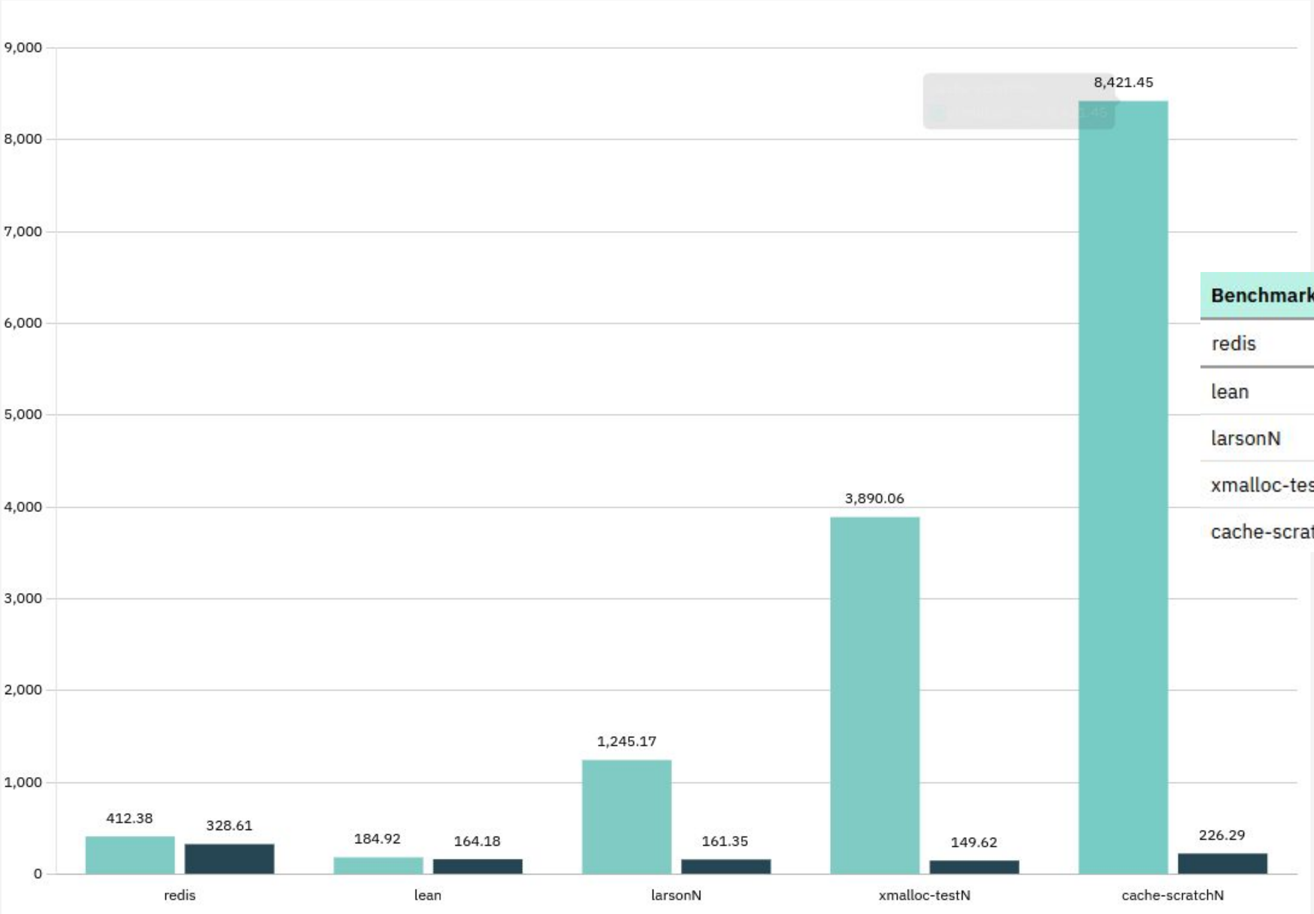
Heap-owned thread_delayed_free

thread_free pointer uses 2 low bits for state:

- NORMAL: push to page's thread_free
- DELAYED: push to heap's delayed_free
- DELAYING: transition state (heap validity)

After one delayed free → state returns to NORMAL (only need one signal).

Performance Benchmarks



Benchmark	ptmalloc2_ms	mimalloc_ms	Speedup_Facto
redis	412.38	328.61	1.25
lean	184.92	164.18	1.13
larsonN	1245.17	161.35	7.72
xmalloc-testN	3890.06	149.62	26.0
cache-scratchN	8421.45	226.29	37.22

Memory Usage

mimalloc Memory Characteristics:

- **~0.2% metadata overhead**
- **≤16.7% internal fragmentation** (1/8th size class rounding)
- Bounded worst-case allocation time
- No blowup (memory usage scales with actual allocation)

Eager Page Purging:

1. Memory marked to OS as unused (`madvise(MADV_DONTNEED)`)
2. Physical memory can be reclaimed
3. Virtual address space retained (fast reuse)
4. Configurable delay via `MIMALLOC_PURGE_DELAY`

vs Other Allocators:

- Similar or better memory usage than jemalloc
- tcmalloc often uses 1.5-2× more memory
- Hoard can have 4× memory usage on some workloads

Security Features

Secure Mode (`-DMI_SECURE=ON`): Performance penalty: ~3% average (surprisingly low!)

1. **Guard Pages**
   - OS guard pages between every mimalloc page
   - Heap overflow limited to single 64KiB page
   - Metadata protected by guard pages
2. **Randomized Allocation**
   - Initial free list randomized per page
   - Defeats heap feng shui attacks
   - Sometimes extends instead of using local_free
3. **Encrypted Free Lists**
   - XOR-encoded with per-page keys
   - Prevents overwriting with known values
   - Detects heap corruption
4. **First-Class Heaps**
   - Create isolated heaps for sensitive data
   - VTables in separate heap from user data
   - `mi_heap_destroy()` frees all at once

- Double-free detection
- Invalid pointer detection
- Use-after-free detection (some forms)

Guarded Mode

Guarded Mode (`-DMI_GUARDED=ON`)

Places OS guard pages **behind** certain allocations to catch buffer overflows.

Configuration:

MIMALLOC_GUARDED_SAMPLE_RATE=N   # Guard every Nth allocation (default: 4000)
MIMALLOC_GUARDED_SAMPLE_SEED=S   # Reproducible sampling

Trade-offs:

- Each guarded allocation: minimum 8KiB (4KiB alignment + 4KiB guard)
- High memory overhead if sampling too aggressively
- Excellent for finding buffer overflow bugs in large programs

Use Case:

- Development/testing builds
- Debugging memory corruption
- Not for production (memory overhead)

API and Integration (4 ways to use)

```c
#include <mimalloc.h>

void example() {
    // Standard allocation
    void* p = mi_malloc(1024);

    // Zero-initialized (replaces calloc)
    void* zeroed = mi_zalloc(1024);

    // Aligned allocation (e.g., for SIMD or AVX)
    void* aligned = mi_malloc_aligned(1024, 64);

    // Cleanup
    mi_free(p);
    mi_free(zeroed);
    mi_free(aligned);
}
```

```bash
# Set the environment variable to point to the shared library
LD_PRELOAD=/usr/local/lib/libmimalloc.so ./my_application
```

```cmake
# In your CMakeLists.txt
find_package(mimalloc REQUIRED)

add_executable(myapp main.cpp)

# Link mimalloc to override system allocation at the symbol level
target_link_libraries(myapp PRIVATE mimalloc)
```

```cpp
#include <mimalloc-new-delete.h>

int main() {
    // All calls to 'new' and 'delete' now go through mimalloc
    int* myArray = new int[1000];

    delete[] myArray;
    return 0;
}
```

Key Takeaways

1. Locality Matters More Than You Think

- Per-page free lists gave Lean 25%+ speedup
- Objects allocated together should be stored together
- Cache misses dominate modern performance

2. Minimize Fast Path Conditionals

- mimalloc: single `if (block == NULL)` check
- Every branch is a potential misprediction
- Push complexity to slow path

3. Shard Everything

- Free lists per page, not per size class
- Thread-free lists per page, not per heap
- Contention distributed → probability of collision low

4. Batch Expensive Operations

- Cross-thread frees collected in bulk
- Temporal cadence guarantees maintenance runs
- Amortize cost over many allocations

5. Keep It Simple

- 10k LOC vs 25k for glibc
- Uniform data structures reduce special cases
- Simpler = fewer bugs = easier to optimize

Comparison Summary Table

Feature	glibc (ptmalloc2)	mimalloc
Code Size	~25k LOC	~10k LOC
Free Lists	Per size class	Per page (sharded)
Thread Safety	Arena mutex	Lock-free (atomics only)
Cross-thread Free	Return to arena (lock)	Atomic push to page
Fast Path	Multiple conditionals	Single conditional
Page Lookup	Chunk header traversal	Pointer masking O(1)
Metadata Overhead	Variable	~0.2%
Security Mode	MALLOC_CHECK_, safe-linking	Guard pages, encrypted lists, randomization
Deferred Free Support	No	Yes (callback hook)
Heartbeat Support	No	Yes (deterministic)