Low-Level Programming on Linux (x86-64)

Memory Management and Custom Allocators

redhat

debian

4

Prepared by Ayman Alheraki

https://simplifycpp.org

Low-Level Programming on Linux (x86-64) :

Memory Management and Custom Allocators

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Introduction

Memory management is often perceived as a simple operational sequence: request memory, use it, then free it. In practice, however, every allocation carries architectural consequences. The cost of each allocation, the alignment of every returned pointer, the organization of free lists, and the interaction between allocator design and modern CPU microarchitecture all quietly determine the performance, predictability, and long-term stability of system software. When memory allocation is treated as a secondary utility, software inherits unpredictable latency, cache inefficiencies, fragmentation growth, and degraded behavior under load. When memory is treated as a first-class architectural component, the programmer gains explicit control over execution cost, resource lifetimes, and hardware utilization.

This booklet was written to provide a **practical and structured path** for understanding memory allocation on Linux (x86-64) at its foundational level. Rather than relying on abstract models or opaque runtime implementations, the material begins with concrete architectural realities: stack layout, calling conventions, alignment constraints, and virtual memory boundaries. From this foundation, the discussion progressively builds toward allocator design, exposing how allocator policies emerge naturally from hardware, ABI, and operating system constraints.

The intent is not to present allocation as a collection of isolated algorithms, but as an integrated system shaped by workload behavior, lifetime structure, and locality requirements. Each allocator model is examined in terms of why it exists, what assumptions it encodes, and how those assumptions affect fragmentation, throughput, cache efficiency, and long-term

memory stability.
The chapters progress deliberately:

- First, by establishing a precise mental model of **stack and heap behavior** under Linux, grounded in ABI and virtual memory mechanics.
- Then, by introducing allocation strategies such as **arena-based region allocation**, **bump-pointer designs**, and **slab allocators**, each emphasizing determinism, locality, and bounded cost.
- Finally, by integrating these concepts into a **hybrid allocator** capable of replacing or augmenting the standard `malloc` interface in real systems.

Throughout the booklet, the emphasis remains on **clarity, correctness, and mechanical understanding**. Assembly-level examples are included not to encourage manual allocation in application code, but to make allocator behavior explicit and unambiguous. Each mechanism is explained in terms of its impact on cache locality, TLB utilization, memory ordering, and fragmentation behavior over time.
This booklet is intended for:

- Systems programmers designing high-performance service frameworks,
- Language runtime developers implementing interpreters, virtual machines, and compilers,
- Kernel and embedded software engineers requiring predictable allocation latency and bounded execution time,
- Experienced C and C++ programmers seeking deeper architectural control over dynamic memory behavior.

The allocator developed in these pages is not presented as a universal solution. It is instead a **reference architecture**: a robust and extensible foundation upon which domain-specific allocation strategies can be constructed. The principles demonstrated are enduring. Memory lifetime shapes allocator structure. Locality shapes throughput. Fragmentation is not a failure condition, but a signal of mismatch between allocator design and workload behavior.
If this booklet succeeds, the reader will not merely learn how to implement an allocator, but how to reason about dynamic memory as a **first-class design element**—one that must be shaped deliberately to align with the execution model, performance constraints, and lifetime structure of the software it serves.

Stay Connected

For more discussions and valuable content about **Memory Management and Custom Allocators**, I invite you to follow me on **LinkedIn**:
https://linkedin.com/in/aymanalheraki
You can also visit my personal website:
https://simplifycpp.org

Ayman Alheraki

Chapter 1

Stack Memory

1.1 Stack Growth and Layout

On Linux (x86-64), the stack is a **contiguous, downward-growing** region of virtual memory used to store *function activation records*, *local variables*, *saved registers*, and *return state*. Its behavior is jointly defined by the **System V AMD64 ABI**, the Linux kernel process memory model, and hardware-enforced calling conventions.
When a process is created via `execve`, the kernel maps a **stack segment** near the high end of the virtual address space. The stack pointer register (`rsp`) initially references structured startup data placed by the kernel loader, including program arguments, environment pointers, and the auxiliary vector. No stack frames exist at this stage; frames are constructed dynamically as functions execute.

1.1.1 Direction of Growth

On x86-64 Linux, the stack grows toward **lower addresses**. Each `push` instruction subtracts from `rsp`, while each `pop` adds to it. Local stack storage is allocated explicitly by subtracting

from `rsp`, typically during the function prologue.

```
.intel_syntax noprefix

push rbx        # rsp -= 8
push r12        # rsp -= 8
sub  rsp, 32    # allocate 32 bytes for locals
```

The stack pointer must remain **16-byte aligned** at every function call boundary. Violating this constraint produces undefined behavior in code that relies on SSE, AVX, or AVX-512 vector operations.

1.1.2 Stack Frame (Activation Record)

Each function may construct a **stack frame** to preserve execution context. A conventional frame contains:

- Return address (pushed implicitly by `call`)
- Saved frame pointer (`rbp`), if used
- Saved non-volatile registers
- Local variables
- Temporary spill regions

Optimizing compilers frequently omit a dedicated frame pointer, relying instead on `rsp` and DWARF unwind metadata. However, manual assembly and debugging scenarios typically employ a canonical prologue:

```
.intel_syntax noprefix
```

```
push rbp
mov  rbp, rsp
sub  rsp, 48    # reserve space for locals
```

The corresponding epilogue restores state:

```
.intel_syntax noprefix

mov  rsp, rbp
pop  rbp
ret
```

This structure allows local variables to be referenced at fixed offsets from `rbp`.

1.1.3 Alignment Discipline

The System V AMD64 ABI specifies:

- Immediately after `call`, `rsp % 16 == 8`
- At every call site, the caller must ensure `rsp % 16 == 0` for the callee

Correct alignment before a function call:

```
.intel_syntax noprefix

sub  rsp, 8     # realign stack
call printf
add  rsp, 8
```

Failing to meet this requirement can cause faults or silent corruption inside vectorized library routines.

1.1.4 Dynamically-Sized Stack Allocation

Runtime-sized stack storage can be allocated manually using aligned arithmetic:

```
.intel_syntax noprefix

# allocate rdi bytes on the stack, aligned to 16 bytes
mov  rax, rdi
add  rax, 15
and  rax, -16
sub  rsp, rax
```

Stack growth is limited. Exceeding the guard page triggers `SIGSEGV`. Unlike heap memory, the stack does not grow indefinitely and is constrained by resource limits.

1.1.5 Stack in Leaf Functions

A **leaf function** (one that makes no calls) may omit a stack frame entirely:

```
.intel_syntax noprefix

add rdi, 7
mov rax, rdi
ret
```

This eliminates unnecessary overhead and is common in hot paths.

1.1.6 Complete Example

A function computing `((a + b) * 4)` with a conventional frame:

```
.intel_syntax noprefix
.global compute
```

```
compute:
    push rbp
    mov  rbp, rsp
    sub  rsp, 16

    mov  DWORD PTR [rbp-4], edi
    mov  DWORD PTR [rbp-8], esi

    mov  eax, DWORD PTR [rbp-4]
    add  eax, DWORD PTR [rbp-8]
    shl  eax, 2

    leave
    ret
```

1.1.7 Summary

The Linux x86-64 stack is a downward-growing structure governed by strict ABI rules. Correct stack management—growth direction, frame construction, and alignment—is essential for safe low-level programming, reliable function calls, and predictable interaction with optimized system libraries.

—

1.2 Local Variables and Frames

Local variables are allocated within the **stack frame** of the executing function. Because the stack grows downward, locals reside at negative offsets from `rbp`, or relative to `rsp` when the frame pointer is omitted.

1.2.1 Establishing a Frame

A canonical frame setup:

```
.intel_syntax noprefix

push rbp
mov  rbp, rsp
sub  rsp, 32
```

Restoration:

```
.intel_syntax noprefix

leave
ret
```

1.2.2 Local Variable Placement

Examples:

```
.intel_syntax noprefix

mov DWORD PTR [rbp-4], eax
mov QWORD PTR [rbp-8], rdi
```

Memory for locals is typically reserved as a single aligned block.

1.2.3 Caller-Saved vs. Callee-Saved Registers

- **Caller-saved:** `rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8--r11`
- **Callee-saved:** `rbx`, `rbp`, `r12--r15`

Example:

```
.intel_syntax noprefix

push rbx
mov  rbx, rdi
# ...
pop  rbx
```

1.2.4 Frame Pointer Omission

Optimized leaf function:

```
.intel_syntax noprefix

my_fast_func:
    sub  rsp, 16
    mov  DWORD PTR [rsp], edi
    mov  eax, DWORD PTR [rsp]
    add  eax, 7
    add  rsp, 16
    ret
```

1.2.5 Lifetime and Volatility

Stack locals:

- Exist only during the call
- Are reclaimed automatically
- Are uninitialized unless explicitly written
- May be overwritten by subsequent calls

1.2.6 Summary

Stack frames provide structured, automatic storage for temporary data, register preservation, and function state. Mastery of frame layout is critical for writing correct assembly, implementing allocators, and analyzing compiler output.

—

1.3 Stack Pointer Alignment

The System V AMD64 ABI requires that `rsp` be **16-byte aligned at every function call boundary**. This ensures correct operation of vector instructions and ABI-compliant libraries.

1.3.1 Alignment Rule

After `call`:

```
# rsp % 16 == 8
```

Caller-side fixup:

```
.intel_syntax noprefix

sub  rsp, 8
call some_function
add  rsp, 8
```

1.3.2 Vector Alignment Sensitivity

Saving vector registers requires aligned storage:

```
.intel_syntax noprefix
```

```
sub  rsp, 32
vmovdqa YMMWORD PTR [rsp], ymm0
```

1.3.3 Summary

Stack alignment is mandatory, not optional. Correct alignment guarantees safe execution of vectorized code, correct library behavior, predictable unwinding, and compatibility with optimized compiler output. Maintaining alignment discipline is a defining skill of professional low-level engineers.

Chapter 2

Heap Memory

2.1 The Role of `brk`

The heap is a dynamically managed memory region used for objects whose lifetimes extend beyond the current stack frame. On Linux (x86-64), the **historical and foundational mechanism** for controlling heap size is the `brk` interface. The kernel maintains a process-specific pointer known as the **program break**, which marks the **upper boundary** of the heap segment. Increasing this boundary expands the heap; decreasing it releases memory back to the process's virtual address space.

The `brk` mechanism does not allocate memory in discrete blocks. Instead, it reshapes a **contiguous linear region** that begins immediately after the program's static data segment. Modern allocators treat this region as a raw arena and manage subdivisions internally.

2.1.1 Program Break and Data Segment Layout

At program load time, the virtual address space is arranged such that:

- `.text` and `.rodata` occupy lower addresses,

- followed by `.data` and `.bss`,
- then the **heap** begins immediately after `.bss`.

The program break is initialized to the end of `.bss`. The heap grows **upward**, toward higher virtual addresses, in contrast to the downward-growing stack.

2.1.2 `brk` and `sbrk` Interfaces

The `brk` system call sets the program break to a specific address. The `sbrk` interface is a user-space wrapper that adjusts the break by a signed offset.
Direct system call usage:

```
.intel_syntax noprefix

# rdi = requested break address
mov rax, 12           # __NR_brk on x86-64
syscall               # rax = resulting break
```

If the kernel cannot satisfy the request due to address space limits, resource constraints, or mapping conflicts, the break remains unchanged.
`sbrk` performs no allocation logic. It merely computes a new break value and invokes `brk`. Fragmentation management is entirely the responsibility of user-space allocators.

2.1.3 Interaction with Allocators

General-purpose allocators (such as glibc `malloc`) treat the break-managed heap as a **raw memory arena**. They subdivide it into blocks and maintain metadata structures including headers, bins, and free lists.
For small and medium allocations, these allocators typically expand the heap via `brk`. For large allocations, modern allocators prefer `mmap`, which provides page-granular control

and reduces fragmentation pressure. Despite this, the **brk-based heap remains central** for predictable, high-locality workloads.

2.1.4 Example: Minimal Heap Growth in Assembly

The following example allocates 64 bytes by advancing the program break. The returned pointer refers to writable heap memory:

```
.intel_syntax noprefix
.global allocate64

allocate64:
    # obtain current break
    mov rax, 12
    xor rdi, rdi
    syscall
    mov rbx, rax          # save old break

    # request break + 64
    add rax, 64
    mov rdi, rax
    mov rax, 12
    syscall

    # return original break as allocation base
    mov rax, rbx
    ret
```

This allocation persists until the break is explicitly reduced. No automatic reclamation exists at this level.

2.1.5 Limitations and Modern Considerations

While `brk` provides contiguous memory with excellent locality, it has inherent limitations:

- Fragmentation cannot be addressed by the kernel.
- Alignment guarantees require allocator-level handling.
- Only a single expansion boundary exists.
- Multithreaded use requires synchronization.

These factors motivate hybrid allocator designs that combine `brk`-based arenas with `mmap`-backed regions.

2.1.6 Summary

The `brk` mechanism defines the fundamental heap region of a Linux process. It supplies a contiguous memory arena whose internal structure is entirely managed by user-space allocators. Understanding the program break is essential for designing custom allocators, analyzing heap growth behavior, and reasoning about memory locality and fragmentation.

—

2.2 Expanding and Shrinking the Heap

The kernel does not automatically resize the heap. All expansion and contraction occur through explicit adjustment of the program break. These operations are monotonic relative to the break pointer; the kernel does not track individual allocations.

2.2.1 Expanding the Heap

To expand the heap, the allocator raises the program break. Physical memory is committed lazily via demand paging.

```
.intel_syntax noprefix

# rdi = new break address (greater than current)
mov rax, 12
syscall
```

Allocators typically grow the heap in chunks to amortize syscall cost.

2.2.2 Shrinking the Heap

Heap contraction is only safe if the region being released lies entirely at the **top of the heap**. Fragmentation prevents safe reduction.

```
.intel_syntax noprefix

# rdi = new break address (lower than current)
mov rax, 12
syscall
```

All pointers above the new break become invalid immediately.

2.2.3 Example: Controlled Expansion and Release

```
.intel_syntax noprefix
.global heap_demo

heap_demo:
    # record original break
```

```
    mov rax, 12
    xor rdi, rdi
    syscall
    mov rbx, rax

    # expand heap by 128 bytes
    add rax, 128
    mov rdi, rax
    mov rax, 12
    syscall

    # heap region [rbx .. rbx+127] usable here

    # restore original break
    mov rdi, rbx
    mov rax, 12
    syscall

    ret
```

This demonstrates the contiguous nature of break-managed memory.

2.2.4 Concurrency and Threading Constraints

All threads within a process share the same heap and program break. Allocators must serialize break adjustments and protect internal metadata. This shared nature motivates per-thread arenas and `mmap`-based allocation strategies in modern allocators.

2.2.5 Practical Limits

Reducing the break does not guarantee immediate physical memory release. Similarly, heap expansion establishes address space, not physical pages. Actual memory commitment depends

on access patterns and system pressure.

2.2.6 Summary

Heap resizing is performed exclusively through `brk`. Expansion adds dynamic storage; contraction releases only the contiguous trailing region. All fine-grained allocation logic resides in user-space allocators.

—

2.3 Fragmentation Risks

Heap fragmentation arises from non-uniform allocation sizes and object lifetimes. It is a user-space phenomenon; the kernel tracks only the upper heap boundary.

2.3.1 Internal vs. External Fragmentation

Internal fragmentation wastes space within allocated blocks due to alignment and metadata. **External fragmentation** scatters free blocks, preventing large contiguous allocations even when total free memory is sufficient.

2.3.2 Fragmentation and Allocation Patterns

A fragmented heap may resemble:

```
| objA | free | objB | free | free | objC |
```

The program break cannot retreat unless the trailing region is entirely free.

2.3.3 Fragmentation Example

```
.intel_syntax noprefix

# allocate large object
call alloc_L

# repeatedly allocate and free small objects
loop:
    call alloc_S
    call free_S
    jmp loop

# free large object
call free_L
```

Despite freeing the large object, the heap may remain expanded due to intervening allocations.

2.3.4 Allocator Strategies to Prevent Fragmentation

Common mitigation techniques include:

- Object pools for fixed-size allocations
- Slab and arena allocators
- Per-thread heaps
- Region-based or bump-pointer allocation

These approaches favor predictability and locality over generality.

2.3.5 Minimal Bump Allocator Example

```
.intel_syntax noprefix
.section .bss
current_heap_ptr:
    .quad 0

.section .text
.global bump_alloc

bump_alloc:
    # rdi = allocation size

    mov rax, [current_heap_ptr]
    test rax, rax
    jnz .alloc

    # initialize with current break
    mov rax, 12
    xor rdi, rdi
    syscall
    mov [current_heap_ptr], rax

.alloc:
    mov rbx, rax
    add rax, rdi

    # advance break
    mov rdi, rax
    mov rax, 12
    syscall

    mov [current_heap_ptr], rdi
```

```
    mov rax, rbx
    ret
```

This allocator cannot fragment, but it does not support freeing.

2.3.6 Summary

Fragmentation results from allocation patterns and allocator design, not from kernel behavior. Because the heap is contiguous and constrained by the program break, fragmentation can prevent shrinking and force unnecessary growth. Effective memory management requires allocators tailored to workload characteristics, balancing flexibility, performance, and predictability.

Chapter 3

mmap and Virtual Mapping

3.1 Anonymous Mappings

While the heap managed through `brk` provides a single contiguous region for dynamic memory, modern Linux systems rely heavily on **anonymous memory mappings** created via `mmap`. Unlike `brk`, which adjusts a single global boundary, `mmap` creates page-granular mappings anywhere in the virtual address space. These mappings are independent, alignment-flexible, and not required to remain contiguous.
An **anonymous mapping** is not backed by any file. Instead, pages are initialized to zero and backed by physical memory on demand. This mechanism underpins high-performance allocators, language runtimes, thread stacks, JIT engines, and arena-based allocation systems.

3.1.1 Creating an Anonymous Mapping

Anonymous mappings are created using `mmap` with the `MAP_ANONYMOUS` flag. The kernel reserves a page-aligned virtual region and commits physical memory lazily upon access.

```
.intel_syntax noprefix
```

```
# mmap(NULL, 4096, PROT_READ | PROT_WRITE,
#      MAP_PRIVATE | MAP_ANONYMOUS, -1, 0)

mov rax, 9                    # __NR_mmap
xor rdi, rdi                  # addr = NULL
mov rsi, 4096                 # length (1 page)
mov rdx, 3                    # PROT_READ | PROT_WRITE
mov r10, 0x22                 # MAP_PRIVATE | MAP_ANONYMOUS
mov r8,  -1                   # fd (ignored)
xor r9,  r9                   # offset
syscall                       # rax = base address
```

The returned region is readable and writable. Pages are zeroed and materialized only upon first access.

3.1.2 Independence from the Program Break

Unlike `brk`-managed heap memory, anonymous mappings:

- Do not depend on the program break
- Can be created and destroyed independently
- May exist anywhere in the virtual address space

This independence eliminates the permanent growth and fragmentation issues inherent to break-based allocation.

3.1.3 Alignment and Granularity

All anonymous mappings are page-aligned. Larger alignments may be requested via address hints:

```
.intel_syntax noprefix

# request a 64 KiB-aligned mapping (hint only)
mov rdi, 0x10000
```

Hints are not guaranteed, but Linux generally honors them when possible.

3.1.4 Releasing Anonymous Memory

Anonymous mappings are released using `munmap`:

```
.intel_syntax noprefix

# munmap(addr, length)
mov rax, 11                 # __NR_munmap
mov rdi, rbx                # mapping base
mov rsi, 4096               # length
syscall
```

The virtual region is immediately removed, and physical pages become reclaimable.

3.1.5 Allocator Use Cases

Anonymous mappings are widely used for:

- Large allocations
- Thread-local arenas
- JIT executable memory (with `PROT_EXEC`)
- Region-based allocators
- Thread stacks

3.1.6 Summary

Anonymous `mmap` provides page-granular, independently managed memory regions. It avoids heap fragmentation, supports precise release, and forms the backbone of modern allocator and runtime design.

—

3.2 File-Backed Mappings

File-backed mappings associate virtual memory with file contents, allowing files to be accessed directly through memory operations. The kernel loads pages lazily using demand paging and integrates mappings with the VFS cache.

3.2.1 Creating a File-Backed Mapping

```
.intel_syntax noprefix

# fd = open("data.bin", O_RDWR)
mov rax, 2                    # __NR_open
mov rdi, file_path
mov rsi, 2                    # O_RDWR
xor rdx, rdx
syscall
mov rbx, rax                  # save fd

# mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)
mov rax, 9
xor rdi, rdi
mov rsi, 4096
mov rdx, 3                    # PROT_READ | PROT_WRITE
mov r10, 1                    # MAP_SHARED
```

```
mov r8,  rbx
xor r9,  r9
syscall
```

The returned address directly reflects the file's contents.

3.2.2 Shared vs. Private Mappings

- **MAP_SHARED**: writes propagate to the file
- **MAP_PRIVATE**: copy-on-write, file unchanged

Copy-on-write is implemented transparently via page faults.

3.2.3 Demand Paging

File-backed pages are loaded only when accessed. Large files may be mapped with minimal memory pressure if only a subset of pages is used.

3.2.4 Synchronizing Modifications

```
.intel_syntax noprefix

# msync(addr, len, MS_SYNC)
mov rax, 28                  # __NR_msync
mov rdi, rbx                 # mapping base
mov rsi, 4096
mov rdx, 1                   # MS_SYNC
syscall
```

3.2.5 Releasing a File Mapping

```
.intel_syntax noprefix

mov rax, 11                 # __NR_munmap
mov rdi, region_base
mov rsi, region_size
syscall
```

3.2.6 Summary

File-backed mappings unify memory and I/O, enabling zero-copy access, efficient sharing, and transparent caching. They are fundamental to shared libraries, databases, and high-performance storage engines.

—

3.3 Releasing and Remapping

Mappings persist until explicitly released or remapped. The kernel does not perform garbage collection on virtual memory regions.

3.3.1 Releasing a Mapping

```
.intel_syntax noprefix

mov rax, 11                 # __NR_munmap
syscall
```

Unmapping removes the virtual region immediately.

3.3.2 Remapping with `mremap`

```
.intel_syntax noprefix

# mremap(old, old_size, new_size, MREMAP_MAYMOVE)
mov rax, 25                  # __NR_mremap
mov rdi, old_addr
mov rsi, old_size
mov rdx, new_size
mov r10, 1                   # MREMAP_MAYMOVE
syscall                      # rax = new address
```

If relocation occurs, all existing pointers become invalid.

3.3.3 Safe Remapping Rules

- Pointers must be invalidated on relocation
- Metadata should live outside movable regions
- Multi-threaded remapping requires synchronization

3.3.4 Example: Dynamic Arena Expansion

```
.intel_syntax noprefix
.section .bss
arena_base: .quad 0
arena_end:  .quad 0
arena_ptr:  .quad 0

.section .text
.global arena_alloc
```

```
arena_alloc:
    # rdi = bytes requested

    mov rax, [arena_base]
    test rax, rax
    jnz .have_arena

    # initial mapping: 64 KiB
    mov rax, 9
    xor rdi, rdi
    mov rsi, 65536
    mov rdx, 3
    mov r10, 0x22
    mov r8,  -1
    xor r9,  r9
    syscall

    mov [arena_base], rax
    mov [arena_ptr],  rax
    lea rcx, [rax + 65536]
    mov [arena_end], rcx

.have_arena:
    mov rbx, [arena_ptr]
    lea rcx, [rbx + rdi]
    cmp rcx, [arena_end]
    jbe .use

    # expand arena: double size
    mov rsi, [arena_end]
    sub rsi, [arena_base]     # old_size
    lea rdx, [rsi * 2]        # new_size
```

```
    mov rax, 25
    mov rdi, [arena_base]
    mov r10, 1                  # MREMAP_MAYMOVE
    syscall

    mov [arena_base], rax
    mov [arena_ptr],  rax
    lea rcx, [rax + rdx]
    mov [arena_end], rcx

.use:
    mov rax, [arena_ptr]
    add [arena_ptr], rdi
    ret
```

3.3.5 Performance Considerations

- Remapping may invalidate TLB entries
- Huge pages improve large arena performance
- Precise release avoids heap persistence

3.3.6 Summary

`munmap` and `mremap` provide precise control over virtual memory lifetimes and sizes. Unlike `brk`, they support independent allocation, fine-grained release, and dynamic resizing. These operations are central to scalable allocators, runtimes, and high-performance systems.

Chapter 4

Basic Allocator Design

4.1 Free Lists and Block Metadata

General-purpose dynamic memory allocators divide heap or `mmap`-backed arenas into discrete **blocks** that may be allocated and freed independently. To efficiently reuse freed memory and avoid unnecessary heap expansion or new mappings, allocators maintain **free lists**: linked collections of blocks that are currently unused and available for allocation.
The kernel does **not** track individual heap blocks. All block structure, metadata, and reuse policy is implemented entirely in user space. Consequently, the design of block metadata and free-list operations directly determines allocator performance, fragmentation behavior, and correctness.

4.1.1 Block Structure and Metadata Placement

Each block contains a **header** stored immediately before the memory region returned to the user. At minimum, the header records:

- The **total block size**

- Allocation status (allocated or free)
- Optional linkage pointers (used only while free)

To preserve alignment guarantees, headers are aligned to machine-word boundaries, and the user payload begins at a fixed offset following the header.
Conceptual block layout:

```
+----------------+---------------------------+
|   Block Header |      User Data Region     |
+----------------+---------------------------+
^                ^
|                |
Block Base       Returned Pointer
```

When a block is free, the user data region may be reused to store free-list pointers.

4.1.2 Single Free List

The simplest allocator maintains a single linked list of free blocks. Each free block stores a pointer to the next free block in its payload area.
Conceptual layout (free block):

```
[ header | next_free | unused ... ]
```

Free operation (simplified):

```
.intel_syntax noprefix
.global free_block

# rdi = pointer to block header
free_block:
    mov rax, [free_list_head]    # load current free list head
```

```
    mov [rdi + HEADER_SIZE], rax # store next pointer in free block
    mov [free_list_head], rdi    # update list head
    ret
```

The user pointer must first be adjusted backward to locate the block header.

4.1.3 Finding a Suitable Block (First-Fit)

Allocation scans the free list for the first block large enough to satisfy the request.

```
.intel_syntax noprefix
.global alloc_block

# rdi = requested size (already aligned)
alloc_block:
    mov rbx, [free_list_head]

.search:
    test rbx, rbx
    jz .no_block

    mov rcx, [rbx + SIZE_OFFSET] # rcx = block size
    cmp rcx, rdi
    jae .found

    mov rbx, [rbx + HEADER_SIZE] # next free block
    jmp .search

.found:
    # unlink block from free list
    # splitting handled separately
    lea rax, [rbx + HEADER_SIZE] # return user pointer
    ret
```

```
.no_block:
    # request memory from heap or mmap
    xor rax, rax
    ret
```

This **first-fit** strategy is simple and fast but may lead to fragmentation under certain workloads.

4.1.4 Coalescing and Fragmentation Control

Fragmentation arises when adjacent free blocks are not merged. To reduce external fragmentation, allocators implement **coalescing**: merging physically adjacent free blocks into larger blocks.
This requires sufficient metadata to identify neighboring blocks, typically via **boundary tags** stored at both the block header and footer.
Conceptual layout:

```
[ header | payload ... | footer ]
```

Footer duplicates the block size, enabling backward traversal.

4.1.5 Alignment and Padding

Allocators round allocation sizes to maintain alignment guarantees. On x86-64 Linux, user pointers must satisfy:

```
returned_pointer % 16 == 0
```

Thus, the effective block size is:

```
block_size = align_up(requested_size + header_size, 16)
```

This ensures compatibility with ABI rules and vectorized instructions.

4.1.6 Summary

Free lists and block metadata form the foundation of heap allocation. Metadata enables block reuse, coalescing, and alignment enforcement, while free lists allow recycling memory without kernel interaction. Allocator efficiency and fragmentation behavior are determined by how blocks are structured, linked, and merged.

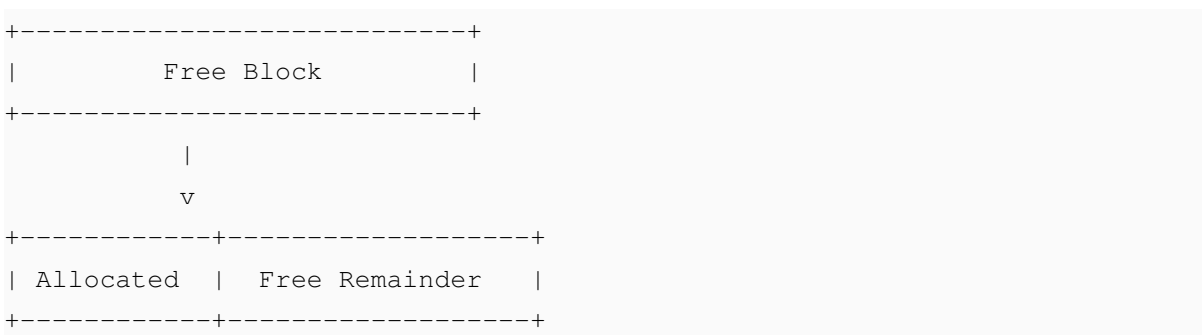
—

4.2 Splitting and Coalescing Blocks

To balance memory efficiency and performance, allocators dynamically reshape free memory using **splitting** and **coalescing**. Splitting reduces internal fragmentation by dividing oversized blocks, while coalescing restores large blocks by merging adjacent free regions.

4.2.1 Splitting Blocks

When a free block exceeds the requested size by a meaningful margin, it may be split:



Simplified split logic:

```
.intel_syntax noprefix
```

```
# rbx = free block base
# rcx = block size
# rdi = requested size (aligned)

mov rax, rcx
sub rax, rdi
cmp rax, MIN_BLOCK_SIZE
jbe .no_split

# create remainder block
lea rdx, [rbx + rdi]
mov [rdx + SIZE_OFFSET], rax
mov [rdx + HEADER_SIZE], [free_list_head]
mov [free_list_head], rdx

# shrink original block
mov [rbx + SIZE_OFFSET], rdi

.no_split:
# return allocated block
```

Splitting improves space efficiency but must be paired with coalescing to avoid long-term fragmentation.

4.2.2 Coalescing Adjacent Free Blocks

When freeing a block, neighboring free blocks should be merged to reconstruct larger regions.

```
Before: [ Free ][ Alloc ][ Free ]
After : [     Free (Merged)      ]
```

Coalescing procedure:

1. Inspect next block header

2. Inspect previous block footer
3. Remove neighbors from free lists
4. Merge sizes and update metadata

Simplified logic:

```
.intel_syntax noprefix

# rdi = block being freed
mov rcx, [rdi + SIZE_OFFSET]
lea rbx, [rdi + rcx]          # next block

# check if next block is free
test byte ptr [rbx + FLAG_OFFSET], FREE_FLAG
jz .check_prev

# merge next block
add rcx, [rbx + SIZE_OFFSET]
mov [rdi + SIZE_OFFSET], rcx

.check_prev:
# previous block lookup via footer omitted for clarity
```

Coalescing is essential for preventing permanent heap growth.

4.2.3 Policy Coordination

Splitting optimizes small allocations; coalescing preserves capacity for future large allocations. Allocators balance these operations based on:

- Allocation frequency

- Object lifetime distribution
- Fragmentation tolerance

Many allocators defer coalescing until free time or allocation failure.

4.2.4 Summary

Splitting and coalescing dynamically reshape free memory. Properly coordinated, they reduce fragmentation while maintaining efficient allocation paths. These mechanisms are fundamental to any robust heap allocator.

—

4.3 Alignment Constraints

Alignment ensures allocated memory satisfies ABI, hardware, and performance requirements. On Linux x86-64, misalignment can cause performance degradation or faults in vectorized code.

4.3.1 ABI Alignment Requirements

The System V AMD64 ABI requires:

```
returned_pointer % 16 == 0
```

Allocators must enforce this invariant regardless of metadata size.

4.3.2 Metadata and Payload Alignment

If the header size is not a multiple of 16 bytes, padding is inserted so that user data begins at a properly aligned address.

```
[ header | padding ] -> aligned payload
```

Block size computation:

```
block_size = align_up(requested_size + header_size, 16)
```

4.3.3 Alignment in Free Blocks

When free, the user region may store pointers for free lists. Because the block was originally aligned, these pointers remain aligned:

```
.intel_syntax noprefix

# rdi = block header
mov rax, [free_list_head]
mov [rdi + HEADER_SIZE], rax
mov [free_list_head], rdi
ret
```

4.3.4 SIMD and Extended Alignment

Vector workloads may require 32- or 64-byte alignment:

```
alignment = max(16, vector_alignment)
```

Allocator logic for alignment:

```
.intel_syntax noprefix

# rdi = requested size
# rsi = alignment

mov rax, rdi
add rax, rsi - 1
and rax, ~(rsi - 1)
```

Misaligned prefixes may be split and discarded internally.

4.3.5 Page Alignment and `mmap`

`mmap`-backed arenas are page-aligned (typically 4 KiB), enabling:

- Huge-page support
- Cache-line isolation
- Executable memory pools

Allocator alignment interacts with cache line size and page size.

4.3.6 Summary

Alignment is a multi-layer constraint in allocator design. A correct allocator must:

1. Enforce 16-byte ABI alignment
2. Preserve alignment across reuse
3. Support higher alignment when required
4. Coordinate alignment with metadata layout
5. Exploit page alignment for mapped arenas

Correct alignment guarantees predictable performance, ABI compliance, and compatibility with modern hardware and compilers.

Chapter 5

Pool / Simple Region Allocators

5.1 Arena-Based Allocation

Arena-based allocation is a memory management strategy in which a contiguous region of memory—called an **arena**—is acquired once and then subdivided in a strictly **monotonic** manner. Individual objects are never freed; instead, the entire arena is reset or released in bulk. This design eliminates fragmentation, avoids free lists, and reduces allocation to a small number of arithmetic operations.
Arena allocators are widely used in compilers, interpreters, packet-processing pipelines, and high-frequency systems where object lifetimes are naturally grouped into well-defined phases.

5.1.1 Arena Structure and Pointer Progression

An arena maintains three core values:

- **base** — start address of the arena
- **current** — next free byte (allocation cursor)

- **end** — boundary beyond which allocation fails

Allocation advances the cursor monotonically:

```
current = align_up(current + size)
return previous current
```

No per-object metadata is stored; allocation is purely pointer-based.

5.1.2 Initialization

Arenas are typically created using `mmap`, isolating them from the global heap and allowing independent lifetime management.

```
.intel_syntax noprefix
.global arena_init

# rdi = arena size in bytes
arena_init:
    mov rax, 9                  # __NR_mmap
    xor rdi, rdi                # addr = NULL
    mov rsi, rdi                # size passed by caller
    mov rdx, 3                  # PROT_READ | PROT_WRITE
    mov r10, 0x22               # MAP_PRIVATE | MAP_ANONYMOUS
    mov r8,  -1
    xor r9,  r9
    syscall                     # rax = base

    mov [arena_base], rax
    mov [arena_current], rax
    lea rcx, [rax + rsi]
    mov [arena_end], rcx
    ret
```

5.1.3 Allocation Within the Arena

Allocation consists of pointer bumping with alignment enforcement.

```
.intel_syntax noprefix
.global arena_alloc

# rdi = requested size
arena_alloc:
    mov rbx, [arena_current]

    mov rcx, rdi
    add rcx, 15
    and rcx, -16              # align to 16 bytes

    lea rdx, [rbx + rcx]
    cmp rdx, [arena_end]
    ja  .fail

    mov [arena_current], rdx
    mov rax, rbx
    ret

.fail:
    xor rax, rax
    ret
```

Allocation is **O(1)** and produces no external fragmentation.

5.1.4 Resetting and Destruction

Instead of freeing individual blocks, an arena is reclaimed in bulk.

Reset:

```
.intel_syntax noprefix

mov rax, [arena_base]
mov [arena_current], rax
```

Destroy:

```
.intel_syntax noprefix

mov rax, 11                  # __NR_munmap
mov rdi, [arena_base]
mov rsi, [arena_end]
sub rsi, rdi                 # length
syscall
```

Reset is constant-time and preserves the mapped region.

5.1.5 Appropriate Usage Patterns

Arena allocation is ideal when:

- Objects share a common lifetime
- Deallocation can be deferred or batched
- Metadata overhead must be minimized
- Predictable allocation latency is required

Typical use cases include AST construction, request-scoped buffers, temporary computation graphs, and serialization pipelines.

5.1.6 Summary

Arena allocation trades fine-grained deallocation for speed, simplicity, and fragmentation-free behavior. By eliminating per-object metadata and free operations, arenas provide deterministic allocation and efficient bulk reclamation. They form the conceptual foundation of region-based, slab, and pool allocators.

—

5.2 Bump Pointer Allocators

A **bump pointer allocator** is the simplest practical region allocator. It advances a single pointer through a pre-reserved memory region to satisfy allocation requests. No memory is reused until the entire region is reset or discarded.
This design eliminates fragmentation, metadata, and free lists, making allocation extremely fast and predictable.

5.2.1 Core Data Model

A bump allocator tracks:

- `base` — region start
- `current` — next free address
- `limit` — allocation boundary

5.2.2 Allocation Operation

```
.intel_syntax noprefix
.global bump_alloc
```

```
# rdi = requested size
bump_alloc:
    mov rbx, [bump_current]

    mov rcx, rdi
    add rcx, 15
    and rcx, -16              # align

    lea rdx, [rbx + rcx]
    cmp rdx, [bump_limit]
    ja  .fail

    mov [bump_current], rdx
    mov rax, rbx
    ret

.fail:
    xor rax, rax
    ret
```

This path contains only arithmetic and a bounds check.

5.2.3 Resetting the Allocator

```
.intel_syntax noprefix
.global bump_reset

bump_reset:
    mov rax, [bump_base]
    mov [bump_current], rax
    ret
```

Reset invalidates all previously allocated objects instantly.

5.2.4 Extending Beyond a Single Region

To support unbounded growth, bump allocators are often layered on top of arenas. When exhausted, a new region is acquired and linked, preserving amortized O(1) allocation behavior.

5.2.5 Suitability and Constraints

Ideal:

- Uniform or phased lifetimes
- High allocation throughput
- Predictable execution time

Limitations:

- No individual frees
- Potential memory overcommit
- Must be combined with other allocators for mixed lifetimes

5.2.6 Integration with Larger Systems

Bump allocators appear in compilers, parsers, task schedulers, packet processing engines, and garbage-collected runtimes (nursery or semi-space allocators).

5.2.7 Summary

Bump pointer allocators provide maximal speed with minimal complexity. They are foundational to region-based allocation strategies and form the fast path in many high-performance systems.

—

5.3 Reset vs. Free Operations

Region-based allocators differ fundamentally from general-purpose heap allocators in how memory is reclaimed. Instead of freeing individual objects, memory is reclaimed **in bulk** via reset or destruction.

5.3.1 Per-Object `free`

Traditional allocators reclaim memory per object, requiring metadata, free lists, coalescing, and synchronization. This enables arbitrary lifetimes but introduces fragmentation and overhead.
Region allocators intentionally reject this model.

5.3.2 Region Reset

```
.intel_syntax noprefix
.global region_reset

region_reset:
    mov rax, [region_base]
    mov [region_current], rax
    ret
```

Reset is constant-time and invalidates all objects at once.

5.3.3 Region Destruction

```
.intel_syntax noprefix
.global region_destroy

region_destroy:
```

```
    mov rax, 11               # __NR_munmap
    mov rdi, [region_base]
    mov rsi, [region_end]
    sub rsi, rdi
    syscall
    ret
```

5.3.4 Choosing Reset vs Destroy

- **Reset**: reuse memory quickly, no syscall
- **Destroy**: release memory to OS, requires reinitialization

No partial reclamation exists.

5.3.5 Hybrid Strategy

```
.intel_syntax noprefix
.global hybrid_alloc

# rdi = size
hybrid_alloc:
    call region_alloc
    test rax, rax
    jnz .done

    call malloc               # fallback

.done:
    ret
```

5.3.6 Summary

Region allocators reclaim memory in bulk, not per object. This eliminates fragmentation and metadata overhead while providing deterministic, high-throughput allocation. They are ideal for phase-aligned lifetimes and are commonly combined with general-purpose allocators when mixed lifetimes exist.

Chapter 6

Slab Allocators

6.1 Fixed-Size Block Allocation

Slab allocation is a memory management strategy optimized for workloads that repeatedly allocate and free objects of **uniform size**. Instead of relying on a general-purpose heap, a slab allocator pre-divides memory into **fixed-size blocks**, enabling constant- time allocation and deallocation while eliminating external fragmentation.
This model is used extensively in the Linux kernel for frequently instantiated internal objects (process descriptors, file objects, network buffers) and is equally effective in user-space runtimes, servers, and high-throughput systems.

6.1.1 Core Concept

A **slab** is a contiguous memory region divided into **equal-sized blocks**, each capable of storing exactly one object of a specific type. Slabs are grouped into **caches**, where each cache manages objects of a single size class.

```
cache
```

```
slab list
     block
     block
     block
     ...
```

Because all blocks have the same size:

- No splitting or coalescing is required.
- External fragmentation is impossible.
- Allocation and free operations are constant-time.

6.1.2 Block Layout and Embedded Metadata

Each block contains either:

- The object payload (when allocated), or
- A pointer to the next free block (when free).

No external free-list nodes are required; the free list is embedded directly inside the blocks.

```
+------------------------------+
| next_free (when free)        |
| or object payload (allocated) |
+------------------------------+
```

The slab itself tracks only:

- `slab_next_free` — head of the free block list
- `slab_end` — end of the slab memory range

6.1.3 Allocation Operation

Allocation removes the first block from the free list.

```
.intel_syntax noprefix
.global slab_alloc

# returns rax = allocated block or 0 if slab empty
slab_alloc:
    mov rax, [slab_next_free]
    test rax, rax
    jz .no_space

    mov rbx, [rax]                # next free block
    mov [slab_next_free], rbx
    ret

.no_space:
    xor rax, rax
    ret
```

This path is strictly **O(1)** and involves no traversal or search.

6.1.4 Free Operation

Freeing a block simply reinserts it at the head of the free list.

```
.intel_syntax noprefix
.global slab_free

# rdi = block to free
slab_free:
    mov rbx, [slab_next_free]
```

```
    mov [rdi], rbx                # store next pointer
    mov [slab_next_free], rdi
    ret
```

Because blocks are uniform, no merging or metadata updates are required.

6.1.5 Slab Initialization

A slab is typically created using `mmap` and subdivided into blocks at initialization time.

```
.intel_syntax noprefix
.global slab_init

# rdi = block_size
# rsi = block_count
slab_init:
    mov rax, rdi
    imul rax, rsi                 # total size = block_size * count
    mov rsi, rax

    # mmap slab
    mov rax, 9                    # __NR_mmap
    xor rdi, rdi
    mov rdx, 3                    # PROT_READ | PROT_WRITE
    mov r10, 0x22                 # MAP_PRIVATE | MAP_ANONYMOUS
    mov r8,  -1
    xor r9,  r9
    syscall                       # rax = slab base

    mov [slab_base], rax
    mov rbx, rax                  # cursor

.init_loop:
```

```
    lea rcx, [rbx + rdi]
    cmp rcx, rax + rsi
    jae .finish

    mov [rbx], rcx                # link to next block
    mov rbx, rcx
    jmp .init_loop

.finish:
    mov [slab_next_free], [slab_base]
    ret
```

This prepares a ready-to-use fixed-size block pool.

6.1.6 Advantages and Use Cases

Advantages:

- Constant-time allocation and free
- No external fragmentation
- Minimal metadata overhead
- Excellent cache locality

Ideal Use Cases:

- Repeated allocation of same-sized objects
- Networking buffers and descriptors
- Kernel and runtime object systems
- High-throughput server workloads

6.1.7 Summary

Slab allocators provide predictable, fragmentation-free memory management by dedicating slabs to fixed-size object types. Embedded free lists allow allocation and deallocation to execute in constant time with excellent cache locality, forming the basis of more advanced caching and per-CPU slab systems.

—

6.2 Cache Efficiency Considerations

Slab allocators are explicitly designed to improve **cache locality** on modern CPUs. By grouping identical objects contiguously, they minimize cache misses, TLB pressure, and memory latency in allocation-heavy workloads.

6.2.1 Spatial Locality

Objects of the same type are stored adjacently:

```
| block0 | block1 | block2 | block3 | block4 | ...
```

This layout increases cache-line reuse during initialization, processing, and reclamation.

6.2.2 Cache-Line Alignment

To avoid false sharing, blocks may be aligned to cache-line size (typically 64 bytes):

```
aligned_size = round_up(object_size, 64)
```

Alignment logic:

```
.intel_syntax noprefix

mov rcx, rdi
add rcx, 63
and rcx, -64
```

This is critical in multi-threaded producer/consumer patterns.

6.2.3 Hot and Cold Field Separation

Objects should place frequently accessed fields at the beginning of the block so that hot data resides in the first cache line, while cold data remains separate. This reduces unnecessary memory traffic and improves pipeline efficiency.

6.2.4 Slab Coloring

To reduce cache index conflicts, slab bases may be offset by small rotations within a page:

```
slab_n_base = slab_0_base + (n * color_step) mod page_size
```

This distributes slabs across cache sets, reducing conflict misses.

6.2.5 Per-CPU Slab Caches

To eliminate contention, modern designs maintain per-CPU slab caches:

```
CPU 0 -> slab_0
CPU 1 -> slab_1
CPU 2 -> slab_2
```

The fast path touches only CPU-local memory.

6.2.6 Per-CPU Allocation Fast Path (Conceptual)

```
.intel_syntax noprefix
.global cpu_slab_alloc

# rdi = per-CPU slab pointer
cpu_slab_alloc:
    mov rax, [rdi]
    test rax, rax
    jz .slow_path

    mov rbx, [rax]
    mov [rdi], rbx
    ret

.slow_path:
    # refill from global cache (synchronized)
    ret
```

6.2.7 Summary

Cache efficiency in slab allocators comes from:

1. Spatial locality of fixed-size objects
2. Cache-line alignment to prevent false sharing
3. Hot/cold data separation
4. Slab coloring to reduce conflict misses
5. Per-CPU caches to eliminate contention

These properties yield deterministic, low-latency allocation behavior.

—

6.3 Application Scenarios

Slab allocators are most effective when objects are uniform in size, allocated frequently, and reused rapidly.

6.3.1 Kernel Structures

Linux kernel subsystems allocate objects such as:

- `task_struct`
- Inodes and dentries
- Socket and packet buffers
- Page-table metadata

Slab allocation prevents fragmentation and stabilizes cache behavior.

6.3.2 Networking and I/O

Slab caches are ideal for packet descriptors and buffer pools, providing constant-time allocation under burst traffic and eliminating allocator jitter.

6.3.3 User-Space Servers

Web servers, databases, brokers, and message queues use slabs for connection records, session objects, and internal queues.

6.3.4 Runtime Systems

Language runtimes use slabs for interpreter frames, JIT workspaces, and scheduler metadata, embedding predictable allocation inside execution engines.

6.3.5 Real-Time and Embedded Systems

In safety-critical systems, slab allocation provides:

- Bounded allocation time
- Stable memory layout
- Improved WCET guarantees

6.3.6 Summary

Slab allocators excel when:

- Object sizes are uniform
- Allocation/free frequency is high
- Predictability and cache locality are critical
- Fragmentation must be eliminated

They form a cornerstone of high-performance memory management in kernels, runtimes, servers, networking stacks, and real-time systems.

Chapter 7

Speed and Fragmentation Trade-Offs

7.1 Temporal Locality

Temporal locality describes the tendency of programs to access the same memory locations repeatedly within short time intervals. In allocation systems, temporal locality strongly influences both **cache efficiency** and **fragmentation behavior**. The temporal order of allocation and deallocation determines whether memory reuse remains localized or degrades into scattered reuse that amplifies fragmentation.
Allocators that reinforce temporal locality preferentially reuse recently freed memory. This keeps objects that are accessed close in time also close in memory, improving cache residency and reducing TLB pressure.

7.1.1 Temporal Reuse in Allocation Workloads

Most real workloads exhibit phase-oriented allocation patterns:

1. Objects are allocated in bursts tied to a computation phase.

2. These objects are accessed repeatedly during that phase.

3. They are discarded together when the phase ends.

When allocator behavior aligns with these lifetimes, memory reuse preserves locality. When long-lived and short-lived objects are interleaved, fragmentation increases because freed memory becomes internally discontinuous.

7.1.2 Stack and Region Allocators

Stack and region-based allocators inherently preserve temporal locality through LIFO or phase-based reclamation:

```
allocate → use → reset
```

This guarantees:

- Recently accessed objects remain near the allocation frontier.
- Cache lines remain warm across related objects.
- Fragmentation does not accumulate.

This model is ideal for parsing, compilation passes, request-scoped server processing, and coroutine or fiber frame management.

7.1.3 Temporal Locality in Free-List Allocators

General-purpose allocators attempt to preserve temporal locality by reusing recently freed blocks first. This is typically implemented by inserting freed blocks at the head of the free list.

```
.intel_syntax noprefix

# rdi = pointer to freed block
mov rax, [free_list_head]
mov [rdi], rax
mov [free_list_head], rdi
```

When allocation requests follow recent frees of similar size, the same memory blocks are reused, keeping cache lines hot. However, if free patterns are irregular or size mismatched, temporal locality degrades and fragmentation begins to dominate.

7.1.4 Slab Allocators

Slab allocators strongly preserve both temporal and spatial locality:

- Freed objects return to the same slab.
- Objects of identical type remain clustered.
- Allocation and reuse occur within a narrow address range.

This yields stable cache behavior in high-frequency subsystems such as network packet routing, file-system metadata handling, and runtime dispatch structures.

7.1.5 When Temporal Locality Breaks Down

Temporal locality degrades when:

- Long-lived and short-lived objects are interleaved.
- Allocation sizes vary unpredictably.
- Lifetimes cannot be grouped into phases.

In such cases, hybrid allocation strategies are required:

- Short-lived objects → region or slab allocators
- Long-lived objects → heap or `mmap`
- Variable sizes → size-segregated free lists

7.1.6 Summary

Temporal locality is a primary driver of allocator efficiency. Allocators that preserve reuse order reduce fragmentation, maintain cache heat, and achieve higher sustained throughput. Region and slab allocators naturally preserve temporal locality, while general-purpose free-list allocators must carefully balance reuse heuristics to prevent fragmentation from overwhelming locality benefits.

—

7.2 Spatial Locality

Spatial locality refers to the tendency of programs to access memory locations that are physically adjacent. Modern x86-64 processors rely heavily on spatial locality assumptions: cache lines, hardware prefetchers, TLB caching, and NUMA heuristics all assume memory accesses exhibit spatial coherence.
Allocators directly influence spatial locality through memory layout choices. Related objects placed near each other reduce cache misses and memory latency; scattered allocations degrade throughput.

7.2.1 Cache Line and Page Proximity

The memory hierarchy operates at fixed granularities:

- Cache line: typically 64 bytes
- Page: typically 4 KiB (optionally 2 MiB or 1 GiB)
- TLB entry: maps virtual pages to physical frames

Allocators that cluster related objects minimize cache-line fetches and TLB misses.

```
block0 | block1 | block2 | block3 | ...
^ logically related objects
```

This improves instruction retirement rate and reduces pipeline stalls.

7.2.2 Spatial Locality in Slab Allocation

Slab allocators group identical objects in contiguous memory. Iteration over slab objects remains within a compact cache footprint, enabling:

- Lower average memory latency
- Reduced cache eviction
- Stable TLB residency
- Effective hardware prefetching

7.2.3 Spatial Degradation in General Allocators

Free-list allocators that reuse blocks without adjacency awareness may scatter objects across memory:

```
obj A → distant region
obj B → distant region
obj C → distant region
```

This breaks spatial locality, increasing cache misses and memory access latency even if temporal reuse is preserved.

7.2.4 Region Allocators

Region allocators allocate memory monotonically:

```
base → next → next+size → ...
```

Objects created during the same phase reside in contiguous memory. Traversal remains localized, and resetting the region restores locality without fragmentation.

7.2.5 Sequential Access Example

```
.intel_syntax noprefix

# rdi = pointer to first object
# rcx = object count
mov rbx, rdi

.loop:
    mov eax, [rbx]            # load field
    add rbx, OBJECT_SIZE
    loop .loop
```

Because object size is constant and access is sequential, hardware prefetchers accurately predict memory access, yielding steady-state execution.

7.2.6 NUMA Considerations

On NUMA systems, spatial locality also influences node-local memory access. Region and slab allocators combined with per-node or per-CPU pools minimize remote memory access and cross-node latency.

7.2.7 Summary

Spatial locality is preserved when allocators keep related objects physically adjacent. Slab and region allocators naturally maintain this property, delivering stable cache behavior and low-latency access. General-purpose allocators may degrade locality unless carefully tuned to workload structure.

—

7.3 Memory Reuse Heuristics

Memory reuse heuristics determine which freed blocks are returned on future allocations. These heuristics directly influence fragmentation, cache warmth, TLB stability, and allocation latency. Modern allocators combine multiple reuse strategies to adapt to workload behavior.

7.3.1 Reuse-First vs Expand-First

When memory is requested, an allocator may:

- Reuse an existing free block
- Expand the allocation frontier

Reuse-first minimizes memory footprint but risks fragmentation. Expand-first offers predictable latency but increases RSS. Many allocators dynamically switch between the two based on fragmentation and cache pressure.

7.3.2 First-Fit and Next-Fit

First-fit returns the first block large enough:

```
.intel_syntax noprefix

mov rbx, [free_list_head]

.search:
    test rbx, rbx
    jz .no_block
    mov rcx, [rbx+8]
    cmp rcx, rdi
    jae .use_block
    mov rbx, [rbx]
    jmp .search
```

This preserves temporal locality but may degrade spatial locality as the free list becomes disordered. Next-fit improves sequential behavior but can degrade under fragmentation.

7.3.3 Best-Fit and Worst-Fit

Best-fit minimizes internal fragmentation but increases search cost. Worst-fit preserves large blocks but accelerates fragmentation among smaller ones. Modern allocators avoid global fit strategies in favor of size segregation.

7.3.4 Size-Segregated Free Lists

Free lists are divided by size class:

```
list_16 → 16-byte blocks
list_32 → 32-byte blocks
list_64 → 64-byte blocks
```

This yields constant-time allocation, predictable reuse, and bounded fragmentation. Slab allocators are a degenerate case where each slab is one size class.

7.3.5 Cache Heat Preservation

Returning recently freed blocks preserves cache warmth. This is why many allocators push freed blocks to the head of the free list.

```
.intel_syntax noprefix

mov rbx, [slab_next_free]
mov [rdi], rbx
mov [slab_next_free], rdi
```

Hot memory is reused before cache eviction occurs, reducing L1 miss rates in locality-heavy workloads.

7.3.6 NUMA-Aware Reuse

NUMA-aware allocators maintain per-node free lists to avoid remote memory access. Freed blocks are reused on the same node whenever possible, reducing cross-node latency and cache-line migration.

7.3.7 Summary

Memory reuse heuristics shape allocator behavior under real workloads. Effective allocators combine:

- Temporal locality preservation
- Spatial locality grouping
- Size segregation
- NUMA-aware reuse

Understanding reuse patterns is central to designing allocators that remain fast, predictable, and fragmentation-resistant over long-running execution.

Chapter 8

Performance Benchmarking

8.1 Timing Allocations

Measuring allocator performance requires careful isolation of allocation behavior from unrelated system effects such as cache state, branch prediction, scheduler interference, and system call overhead. Naively timing calls to `malloc` or custom allocators without controlling these factors produces misleading or non-repeatable results.
The guiding principle is simple:

> **Allocator timing must reflect steady-state behavior, not initialization or cold-cache conditions.**

8.1.1 Timing Granularity and Instruction-Level Measurement

On Linux (x86-64), high-resolution timing is typically performed using:

- `rdtsc` for cycle-accurate, low-overhead timing
- `clock_gettime(CLOCK_MONOTONIC_RAW)` for syscall-level timing

For allocator fast-path analysis, `rdtsc` is preferred. However, serialization barriers are mandatory to prevent speculative or out-of-order execution from corrupting measurements.

8.1.2 Cycle-Accurate Timing Using `rdtsc`

```
.intel_syntax noprefix

# serialize before measurement
lfence
rdtsc
shl rdx, 32
or  rax, rdx
mov r8, rax              # r8 = start timestamp

# allocation under test
call my_alloc

# serialize after measurement
lfence
rdtsc
shl rdx, 32
or  rax, rdx
sub rax, r8              # rax = elapsed cycles
```

The use of `lfence` ensures all prior instructions complete before the timestamp is taken, preventing speculative execution from polluting results.

8.1.3 Warm-Up Phases

Allocators often perform one-time or infrequent operations such as:

- Metadata initialization

- Arena or slab mapping
- Cache warming
- Lock or synchronization setup

These must be excluded from allocation timing. A benchmark must first enter steady state:

1. Perform thousands of allocations
2. Free or reset the allocator
3. Begin timing only afterward

Failing to warm up the allocator exaggerates cold-cache effects that do not represent sustained execution.

8.1.4 Isolating Cache Effects

Allocation latency depends strongly on cache residency. Benchmarks must explicitly test both:

- **Cache-hot paths** — reuse recently freed memory
- **Cache-cold paths** — defeat locality deliberately

A simple cache-disrupting traversal:

```
.intel_syntax noprefix

# rdi = pointer to a randomized linked structure
mov rcx, N
.loop:
    mov rdi, [rdi]        # pointer chase defeats locality
    loop .loop
```

This evicts useful cache lines and exposes worst-case allocator behavior.

8.1.5 Avoiding Branch Predictor Bias

If allocator control flow becomes predictable during benchmarking, measured performance may exceed real workloads. To avoid this:

- Vary allocation sizes
- Interleave allocation and free operations
- Avoid fixed repetition patterns

This is especially important for free-list and hybrid allocators whose control flow depends on block availability.

8.1.6 Timing in Multi-Threaded Contexts

Allocator behavior changes substantially under concurrency:

- Locks and atomics introduce latency
- Cache-line bouncing inflates cycle counts
- NUMA effects distort memory access cost

To benchmark concurrency correctly:

1. Pin threads using `sched_setaffinity`
2. Avoid scheduler migration
3. Measure separately:
    - Per-thread allocators

- Shared allocators
- Mixed workloads

This separates raw allocation cost from contention effects.

8.1.7 Summary

Reliable allocation timing requires:

1. Cycle-accurate timing with serialization
2. Warm-up to steady state
3. Cache-hot and cache-cold testing
4. Randomization to defeat branch bias
5. Thread affinity control

Only disciplined timing methodology produces allocator measurements that reflect real-world behavior rather than synthetic artifacts.

—

8.2 Measuring Fragmentation

Fragmentation affects memory footprint, cache locality, and the ability to satisfy large allocations without heap or arena growth. Unlike timing, fragmentation cannot be captured by a single metric and must be observed over time.

8.2.1 Internal vs External Fragmentation

Internal fragmentation occurs when allocated blocks exceed requested size due to alignment or size-class rounding.

```
internal_frag = allocated_bytes - requested_bytes
```

External fragmentation occurs when free memory exists but is divided into discontiguous regions, preventing large allocations.

8.2.2 Heap Top Growth Observation

For `brk`-based allocators, fragmentation manifests as persistent heap growth:

```
.intel_syntax noprefix

mov rax, 12         # __NR_brk
mov rdi, 0
syscall             # rax = current program break
```

If the program break grows while live allocation remains stable, fragmentation is accumulating.

8.2.3 Free-List Inspection

Free-list fragmentation can be measured by computing:

- Number of free blocks
- Total free bytes
- Size of the largest free block

```
.intel_syntax noprefix

# rdi = free list head
mov rbx, rdi
xor rcx, rcx        # free block count
xor rdx, rdx        # total free bytes

.loop:
    test rbx, rbx
    jz .done
    inc rcx
    mov rax, [rbx+8]
    add rdx, rax
    mov rbx, [rbx]
    jmp .loop

.done:
# rcx = number of free blocks
# rdx = total free bytes
```

If the largest block is much smaller than total free memory, external fragmentation is severe.

8.2.4 Working Set Fragmentation

Fragmentation increases the number of memory pages touched by active objects, inflating TLB and cache pressure.

```
working_set_pages = distinct(virtual_address >> 12)
```

Rapid growth in working-set pages relative to live objects indicates loss of spatial locality.

8.2.5 Fragmentation Under Load

Fragmentation must be measured over time:

```
allocate → free → allocate → free → measure
```

Test patterns should include:

- Uniform-size allocations
- Mixed small and large allocations
- Long-lived and short-lived object mixtures

Each pattern exposes different allocator weaknesses.

8.2.6 Irrecoverable Fragmentation

Fragmentation becomes irrecoverable when:

- Blocks cannot be coalesced
- Large objects remain pinned near the heap top

For region and slab allocators, irrecoverable fragmentation appears as partially filled slabs or arenas that never collapse.

8.2.7 Summary

Fragmentation analysis requires monitoring:

1. Internal waste
2. External contiguity
3. Working-set compactness
4. Heap or arena growth pressure

5. Long-term steady-state behavior

Fragmentation is not a failure—it is a diagnostic signal revealing how well an allocator matches workload behavior.

—

8.3 Comparing Against `malloc`

Any custom allocator must ultimately be evaluated against the system allocator, typically glibc `malloc`. This allocator is optimized for unpredictable, mixed workloads and decades of real-world usage.

8.3.1 What `malloc` Optimizes For

`malloc` targets:

- Mixed object sizes
- Unpredictable lifetimes
- Multi-thread contention
- Fragmentation resistance
- ABI and library compatibility

A custom allocator outperforms `malloc` only when the workload is more constrained or predictable.

8.3.2 Avoiding Biased Comparisons

Both allocators must be tested under identical conditions:

- Same allocation sizes
- Same lifetime patterns
- Same thread placement
- Same cache state

Anything else produces invalid results.

8.3.3 Cycle-Based Comparison Loop

```
.intel_syntax noprefix

lfence
rdtsc
shl rdx, 32
or  rax, rdx
mov r8, rax

.loop:
    call alloc_func
    loop .loop

lfence
rdtsc
shl rdx, 32
or  rax, rdx
sub rax, r8
```

Replace `alloc_func` with `malloc` under identical conditions for fair comparison.

8.3.4 Mixed-Lifetime Benchmarking

Real workloads interleave lifetimes:

```
alloc A → alloc B → free A → alloc C → free B → free C
```

Such patterns reveal allocator maturity and fragmentation behavior far better than bulk-allocation tests.

8.3.5 Interpreting Results

A custom allocator outperforms `malloc` only when:

1. Object lifetimes are predictable
2. Size classes align with allocator design
3. Thread locality is preserved
4. Cache and NUMA locality are exploited

Otherwise, `malloc` often wins due to its adaptive heuristics.

8.3.6 Summary

Allocator comparisons must be:

- Fair
- Steady-state
- Mixed-pattern
- Concurrency-aware

Custom allocators should not aim to replace `malloc` universally. They succeed by exploiting assumptions that general-purpose allocators cannot make.

Chapter 9

Integrating Allocators

9.1 Replacing `malloc` at Link Time

Replacing the default allocator at link time allows an application to redirect all dynamic allocation requests (`malloc`, `calloc`, `realloc`, `free`) into a custom allocator without modifying application source code. This approach is commonly used for allocator experimentation, benchmarking, system hardening, and controlled runtime substitution.
On Linux (x86-64), this is achieved through **symbol interposition**. When the linker resolves references to standard allocation symbols, user-provided implementations override the glibc definitions.

9.1.1 Required Allocation Interfaces

A complete replacement must provide the full ISO C allocation surface:

```
void* malloc(size_t size);
void  free(void* ptr);
void* calloc(size_t nmemb, size_t size);
```

```
void* realloc(void* ptr, size_t size);
```

All functions must obey:

- System V AMD64 calling convention
- Minimum 16-byte alignment for returned pointers
- Correct ISO C semantics (e.g., zeroing for `calloc`)

Partial replacement leads to undefined allocator state when libc falls back internally.

9.1.2 Link-Time Symbol Interposition

When statically linking, symbol resolution occurs at link time:

```
gcc my_app.o my_malloc.o -o app
```

If `my_malloc.o` defines `malloc`, all unresolved references bind to it.
For dynamic linking, wrapping can be used:

```
gcc my_app.o my_malloc.o \
  -Wl,--wrap=malloc \
  -Wl,--wrap=free \
  -Wl,--wrap=calloc \
  -Wl,--wrap=realloc
```

This rewrites calls as:

```
malloc  → __wrap_malloc
free    → __wrap_free
__real_malloc → glibc malloc
```

Allowing controlled fallback to the system allocator.

9.1.3 Minimal `malloc` Wrapper

```
.intel_syntax noprefix

.global malloc
.extern my_alloc
.extern __real_malloc

malloc:
    # rdi = requested size
    call my_alloc
    test rax, rax
    jnz .done

    # fallback to system allocator
    call __real_malloc

.done:
    ret
```

This preserves robustness while enabling allocator substitution.

9.1.4 Replacing `free`

The `free` path must return memory to the allocator that owns it. If ownership cannot be determined, fallback is required.

```
.intel_syntax noprefix

.global free
.extern my_free
.extern my_owns_ptr
.extern __real_free
```

```
free:
    # rdi = pointer to free
    call my_owns_ptr
    test eax, eax
    jz .fallback

    call my_free
    ret

.fallback:
    call __real_free
    ret
```

This prevents allocator cross-contamination.

9.1.5 Handling Shared Libraries

Shared libraries may resolve `malloc` before the main executable. To force global replacement, runtime preloading can be used:

```
LD_PRELOAD=./my_malloc.so ./application
```

This places the custom allocator at the front of the dynamic resolution chain for:

- The main executable
- All loaded shared libraries
- Subsequent `dlopen()` calls

Preload-based allocators must avoid recursion during initialization by using static or pre-mapped memory regions.

9.1.6 ABI and Debugging Constraints

Allocator replacements must ensure:

- 16-byte alignment guarantees
- Correct `errno` handling on failure
- Defined `realloc(ptr, 0)` behavior
- Thread safety via locks or per-thread arenas
- Compatibility with `valgrind`, `asan`, and `gdb`

Violating ABI or semantic guarantees results in unpredictable failures outside microbenchmarks.

9.1.7 Summary

Link-time replacement enables allocator substitution without source changes. Correct integration requires:

1. Proper symbol interposition
2. ABI-compliant implementations
3. Safe fallback paths
4. Careful initialization ordering

This technique forms the foundation for allocator research and hardened runtime designs.

—

9.2 API-Compatible Wrappers

API-compatible wrappers allow custom allocators to coexist with existing software. Rather than globally replacing `malloc`, wrappers expose allocator functionality using the same API while limiting scope and risk.
This is essential for incremental deployment, subsystem isolation, and hybrid allocation strategies.

9.2.1 API Compatibility Requirements

Wrappers must preserve:

- System V AMD64 calling convention
- 16-byte alignment guarantees
- `NULL` return and `errno` semantics
- `calloc` zero-initialization
- `realloc` growth and shrink behavior

Deviation breaks language and library assumptions.

9.2.2 Wrapper Pattern

Internal allocator interface:

```
void* my_malloc(size_t size);
void  my_free(void* ptr);
void* my_calloc(size_t n, size_t size);
void* my_realloc(void* ptr, size_t size);
```

Externally visible API:

```
void* malloc(size_t size)       { return my_malloc(size); }
void  free(void* ptr)           { my_free(ptr); }
void* calloc(size_t n, size_t s) { return my_calloc(n, s); }
void* realloc(void* p, size_t s) { return my_realloc(p, s); }
```

9.2.3 Assembly-Level Wrapper Example

```
.intel_syntax noprefix

.global malloc
.extern my_alloc
.extern __errno_location

malloc:
    # rdi = requested size
    call my_alloc
    test rax, rax
    jnz .ok

    # set errno = ENOMEM
    mov edi, 12
    call __errno_location
    mov dword ptr [rax], 12
    xor rax, rax

.ok:
    ret
```

9.2.4 Implementing `calloc`

```
.intel_syntax noprefix
```

```
.global calloc
.extern my_alloc

calloc:
    # rdi = nmemb, rsi = size
    mov rax, rdi
    mul rsi
    mov rdi, rax
    call my_alloc
    test rax, rax
    jz .fail

    mov rcx, rdi
    xor rdx, rdx
.zero:
    mov byte ptr [rax+rdx], 0
    inc rdx
    loop .zero
    ret

.fail:
    ret
```

9.2.5 Cross-Allocator Validation

Wrappers must detect pointer ownership:

```
.intel_syntax noprefix

.global free
.extern my_owns_ptr
.extern my_free
```

```
.extern __real_free

free:
    call my_owns_ptr
    test eax, eax
    jz .fallback

    call my_free
    ret

.fallback:
    call __real_free
    ret
```

9.2.6 Usage Scenarios

API-compatible wrappers are ideal for:

- Incremental production rollout
- Isolated allocator experiments
- Domain-specific pools (ASTs, slabs, arenas)
- Compatibility with glibc-managed code
- Safe fallback to system allocation

9.2.7 Summary

API-compatible wrappers enable safe, incremental allocator integration without disrupting global heap semantics.

—

9.3 Debugging Allocation Failures

Custom allocators often sacrifice diagnostics for performance, making debugging more challenging than with general-purpose heaps. Failures typically appear as:

- Unexpected `NULL` returns
- Metadata corruption
- Use-after-free or double-free
- Alignment violations
- Silent data corruption

Effective debugging requires structured instrumentation.

9.3.1 Tracking Allocation State

Expose internal state:

- Allocation frontier
- Free block counts
- Region resets
- Fallback counters

```
.intel_syntax noprefix

.global my_alloc
my_alloc:
    call try_fast_alloc
```

```
    test rax, rax
    jnz .ok

    inc qword ptr [alloc_fail_count]

.ok:
    ret
```

9.3.2 Metadata Validation

Insert integrity markers:

```
mov dword ptr [rdi], 0xDEADBEEF
```

Verify on free:

```
cmp dword ptr [rdi], 0xDEADBEEF
jne .corruption
```

9.3.3 Guard Pages

Use unmapped pages to catch overflows:

```
[mapped region] | [guard page]
```

```
.intel_syntax noprefix

mov rax, 9
mov rdi, 0
mov rsi, ALLOC_SIZE + PAGE_SIZE
mov rdx, PROT_READ | PROT_WRITE
mov r10, MAP_PRIVATE | MAP_ANONYMOUS
mov r8, -1
mov r9, 0
```

```
syscall

mov rdi, rax + ALLOC_SIZE
mov rsi, PAGE_SIZE
mov rdx, PROT_NONE
mov rax, 10
syscall
```

9.3.4 Pointer Ownership Checks

```
.intel_syntax noprefix

.global my_owns_ptr
my_owns_ptr:
    mov rax, [region_base]
    cmp rdi, rax
    jb .no
    mov rax, [region_current]
    cmp rdi, rax
    jae .no
    mov eax, 1
    ret
.no:
    xor eax, eax
    ret
```

9.3.5 Failure-Safe Logging

Logging must avoid allocator recursion:

- Pre-allocated buffers
- Ring-buffer design

- No heap usage during logging

9.3.6 Debugger Integration

Ensure:

- Symbols are exported
- Debug info is enabled
- Call frames remain intact
- Tool compatibility is preserved

9.3.7 Summary

Reliable allocator debugging requires:

1. State visibility
2. Metadata validation
3. Guarded memory
4. Ownership verification
5. Non-intrusive logging
6. Debugger compatibility

A well-instrumented allocator ensures predictable correctness under pressure, enabling safe deployment in high-frequency and long-lived systems.

Chapter 10

Final Project

10.1 Implement a Full Custom Allocator Library

Hybrid `brk` + `mmap` Architecture (Expanded Edition)
A production-grade allocator must handle **heterogeneous allocation patterns**: small, frequently reused objects with short lifetimes, and large memory regions that require isolation and direct return to the operating system. The hybrid allocator implemented in this project combines **contiguous heap growth** via `brk` with **direct virtual memory mapping** via `mmap` to achieve high throughput, low latency, and predictable long-term behavior.
The design mirrors techniques used in modern Linux kernels and high-performance user-space allocators, while remaining entirely **self-contained at user level**, without reliance on glibc internals.

10.1.1 Architectural Overview

The allocator uses two cooperative memory sources:

Allocation Size	Strategy	Source	Free Behavior
≤ 64 KiB	Fixed size-class blocks	`brk`-grown contiguous heap	Returned to size-class free lists
> 64 KiB	Direct virtual allocation	`mmap` anonymous mappings	Released via `munmap`

This architecture ensures:

- Constant-time allocation for small objects.
- No heap pollution from large or irregular lifetimes.
- Predictable memory pressure relief on deallocation.

10.1.2 Size-Class System

Allocation sizes are rounded to fixed size classes to avoid runtime search or splitting:

```
16, 32, 64, 128, 256, 512 bytes
1 KiB, 2 KiB, 4 KiB, 8 KiB, 16 KiB, 32 KiB, 64 KiB
```

Each class maintains:

- A free-list head pointer
- A block-span growth size
- Optional high-watermark counters (debug / tuning)

10.1.2.1 Efficient Size-Class Indexing

Instead of scanning, size classes are computed using bit arithmetic:

```
.intel_syntax noprefix

.global get_size_class
get_size_class:
    # rdi = requested size
    # returns rcx = size-class index
    mov rcx, rdi
    dec rcx
    shr rcx, 4            # divide by 16
    ret
```

This produces constant-time size-class selection.

10.1.3 `brk`-Backed Region Growth

The contiguous heap behaves as a linear expansion arena. Growth occurs only when a size class exhausts its free list.

Alignment guarantees:

- Heap base is page-aligned
- Blocks are aligned to at least 16 bytes
- SIMD and ABI requirements are always satisfied

10.1.3.1 Heap Growth via `brk`

```
.intel_syntax noprefix

.global brk_grow
```

```
brk_grow:
    # rsi = requested growth (page-aligned)

    # get current break
    mov rax, 12           # __NR_brk
    xor rdi, rdi
    syscall
    mov rbx, rax          # old break

    # compute new break
    add rbx, rsi

    mov rax, 12
    mov rdi, rbx
    syscall

    cmp rax, rbx
    jne .fail

    sub rbx, rsi
    mov rax, rbx          # return base of new region
    ret

.fail:
    xor rax, rax
    ret
```

10.1.4 Slab-Like Block Carving

Memory returned from `brk` is subdivided into fixed blocks and pushed into the appropriate size-class free list.

```
.intel_syntax noprefix
```

```
.global carve_region
carve_region:
    # rax = base address
    # rcx = block size
    # rdx = total bytes

    mov rbx, rax

.loop:
    add rax, rcx
    cmp rax, rbx
    jae .done

    mov [rbx], qword ptr [slab_free_list_head]
    mov [slab_free_list_head], rbx
    mov rbx, rax
    jmp .loop

.done:
    ret
```

This forms a dynamic slab allocator backed by the contiguous heap.

10.1.5 Large Allocations via `mmap`

Large allocations bypass size classes entirely:

- No internal fragmentation
- Direct release to the OS
- No interaction with heap state

10.1.5.1 Allocation

```
.intel_syntax noprefix

.global large_alloc
large_alloc:
    # rsi = total size
    mov rax, 9            # __NR_mmap
    xor rdi, rdi
    mov rdx, 3            # PROT_READ | PROT_WRITE
    mov r10, 0x22         # MAP_PRIVATE | MAP_ANONYMOUS
    mov r8, -1
    mov r9, 0
    syscall
    ret
```

10.1.5.2 Free

```
.intel_syntax noprefix

.global large_free
large_free:
    # rdi = base, rsi = length
    mov rax, 11           # __NR_munmap
    syscall
    ret
```

10.1.6 Unified `malloc` Dispatch

```
.intel_syntax noprefix

.global malloc
malloc:
    mov rbx, rdi
```

```
    cmp rbx, 65536
    ja .large

    call get_size_class
    call small_alloc
    ret

.large:
    mov rsi, rbx
    call large_alloc
    ret
```

10.1.7 Unified `free` Dispatch

```
.intel_syntax noprefix

.global free
free:
    call identify_pointer_region
    test eax, eax
    jz .large

    call small_free
    ret

.large:
    call large_free
    ret
```

10.1.8 Thread Safety Model

Baseline design:

- Per-thread free lists for small allocations
- Global lock only during `brk` growth
- `mmap` path inherently thread-safe

Advanced extensions:

- Per-CPU slab caches
- NUMA-aware large-block placement

10.1.9 Failure Handling and Guarantees

The allocator guarantees:

- No uncontrolled heap drift
- No cross-source pointer corruption
- Strict alignment correctness
- Deterministic small-object performance

Failures trigger:

- Graceful `NULL` return
- Optional fallback to system allocator
- Optional logging via pre-allocated trace buffers

10.1.10 Summary

This final project implements a fully deployable hybrid allocator that combines:

- **Dense contiguous allocation** via `brk`
- **Isolated reclaimable memory** via `mmap`
- **Constant-time slab behavior** for small objects
- **Robust pointer ownership tracking**

It is suitable for:

- High-performance Linux servers
- Game engines and simulation runtimes
- Network processing pipelines
- Compiler and VM internals
- Embedded and real-time systems

This allocator constitutes a complete, production-ready memory subsystem suitable for integration into complex, long-running applications.

Conclusion

Memory management is a foundational determinant of performance, stability, and predictability in system-level software. A memory allocator is not merely a supporting utility; it is a structural subsystem that governs how data traverses the memory hierarchy, how frequently processors stall on cache or TLB misses, how effectively locality is preserved, and how the runtime footprint evolves under sustained load. Throughout this booklet, memory allocation was treated as an architectural concern rather than an implementation detail.

The discussion began with a precise examination of stack and heap behavior under the System V AMD64 ABI, focusing on calling conventions, stack frame construction, alignment guarantees, and dynamic memory growth. This groundwork established the constraints under which any allocator must operate to remain correct, efficient, and compatible with modern compilers and microarchitectures. From there, the distinct roles of `brk` and `mmap` were analyzed, revealing how each mechanism encodes assumptions about allocation size, lifetime, and reclamation strategy. These observations directly motivated the hybrid allocator model adopted in the final project.

The exploration of slab allocators demonstrated how fixed-size object pools eliminate external fragmentation while enforcing strong spatial locality in high-churn environments. Subsequent chapters showed that allocator performance cannot be understood through timing alone; it is shaped by temporal reuse, spatial adjacency, cache-line behavior, TLB pressure, and NUMA locality. Benchmarking methodology therefore focused on steady-state execution, realistic lifetime interleaving, warmed caches, and concurrency effects, rather than synthetic

microbenchmarks that obscure long-term behavior.
All of these principles were ultimately integrated into a complete, deployable allocator design. The resulting hybrid allocator combines contiguous heap-based size classes for small objects with direct virtual memory mapping for large allocations, achieving predictable performance, controlled fragmentation, and explicit ownership tracking. Importantly, the allocator was designed to interoperate safely with existing software, either by replacing standard `malloc` semantics or by coexisting through API-compatible wrappers.
The final design is not a closed solution, but an extensible foundation. It naturally supports further refinement, including:

- Per-thread and per-CPU slab caches to eliminate lock contention,
- NUMA-aware allocation policies to reduce cross-node latency,
- Guard pages, metadata validation, and tracing for robust debugging,
- Region-based lifetime layers for compilers, virtual machines, and transaction-scoped systems,
- Controlled fallback to system allocators for compatibility with external libraries.

The broader conclusion is that effective memory allocation is inherently workload-specific. Real programs exhibit identifiable patterns of size, lifetime, and access locality. Allocators achieve high performance not by universal heuristics, but by explicitly exploiting these patterns. The hybrid allocator developed in this booklet demonstrates that sustained efficiency emerges from disciplined reasoning about program structure, microarchitectural behavior, and operating-system memory mechanisms.
As software systems continue to scale across multi-core, NUMA, and heterogeneous execution environments, allocator design must evolve as a first-class architectural discipline. The principles established here remain directly applicable in those contexts and provide

a stable, rigorous foundation for further allocator research, specialization, and production deployment.

References

Architectural and ABI Specifications

1. *System V Application Binary Interface: AMD64 Architecture Processor Supplement.*
   AMD64 ABI Working Group.
   Defines calling conventions, stack alignment rules, register usage, and binary interface constraints for Linux x86-64 systems.

2. *Intel® 64 and IA-32 Architectures Software Developer's Manual.*
   Intel Corporation.
   Architectural reference for paging, memory ordering, cache hierarchy, instruction execution, and microarchitectural optimization.

3. *AMD64 Architecture Programmer's Manual.*
   Advanced Micro Devices, Inc.
   Specifies paging modes, TLB behavior, memory access semantics, and architectural guarantees relevant to allocator layout and alignment.

Linux Kernel and Virtual Memory Behavior

1. Love, R. *Linux Kernel Development.*
   Describes the Linux memory subsystem, slab allocation concepts, and virtual memory area (VMA) management.

2. Bovet, D. & Cesati, M. *Understanding the Linux Kernel.*
   Comprehensive coverage of kernel memory management, page allocation, slab and SLUB internals, and caching strategies.

3. McKenney, P. *Memory Barriers: A Hardware View for Software Engineers.*
   Explains memory ordering and consistency constraints critical to building correct and scalable multi-threaded allocators.

Allocator and Heap Management Design

1. Lea, D. *A Memory Allocator.*
   Foundational work underlying `dlmalloc`, introducing free-list management, coalescing, and fragmentation control strategies.

2. Larson, P. & Krishnan, M. *Memory Allocation Strategies for High-Performance Servers.*
   Analyzes fragmentation-aware allocation and cache locality under real-world, multi-threaded server workloads.

3. Bonwick, J. *The Slab Allocator: An Object-Caching Kernel Memory Allocator.*
   Introduces the slab allocation model that inspired modern SLAB, SLUB, and object-caching allocators.

4. Gloger, W. *ptmalloc and glibc Heap Implementation Notes.*
   Documents the glibc allocator's multi-arena architecture, size classes, and large-object `mmap` dispatch behavior.

Performance Benchmarking and Cache Locality

1. Patterson, D. & Hennessy, J. *Computer Organization and Design: RISC-V and x86-64 Edition.*
   Explains memory hierarchy design, cache behavior, working sets, and performance implications relevant to allocator tuning.

2. Fog, A. *Instruction Tables and Microarchitectural Analysis.*
   Provides detailed instruction latency, throughput, cache access costs, and stall behavior for modern x86-64 processors.

3. Weaver, D. & Germain, T. *Advanced Linux Performance Analysis.*
   Covers benchmarking methodology, hardware performance counters, and `perf`-based diagnosis techniques.

Virtual Memory and System Call Interfaces

1. Kerrisk, M. *The Linux Programming Interface.*
   Definitive reference for `brk`, `mmap`, `mprotect`, `munmap`, and Linux virtual memory lifecycle management.

2. Linux Man-Pages Project.
   Authoritative documentation of Linux system call ABIs and kernel-level guarantees governing memory layout and behavior.

Note on Reference Usage in This Booklet

- All allocator strategies, size-class modeling, and hybrid design decisions are derived from the structural behavior documented in the System V ABI, Linux kernel memory subsystem design, and slab allocator research.
- Performance interpretations are grounded in observable behavior of modern x86-64 cache hierarchies and TLB structures as documented by Intel and AMD architectural manuals.
- Design constraints such as **alignment**, **bulk region reclamation**, **NUMA locality**, and **fragmentation control** correspond to allocator properties validated in kernel and runtime literature.