Operating Systems: Memory Management and Resource Allocation Journal

Course: Operating Systems **Week:** 4 **Student:** [Your Name] **Date:** October 25, 2025

Table of Contents

---

1. Memory Management and Resource Allocation

Overview

Memory management is a fundamental function of operating systems that controls and coordinates computer memory, allocating portions called blocks to various running programs to optimize overall system

performance. The operating system must efficiently manage the limited physical memory resources while satisfying the memory requirements of multiple concurrent processes.

Core Responsibilities

The memory management subsystem of an operating system has several critical responsibilities:

1. Allocation and Deallocation The OS must allocate memory to processes when they request it and reclaim memory when processes terminate or release it. This requires maintaining data structures that track which memory regions are free and which are in use.

2. Address Translation Modern operating systems provide processes with the abstraction of a continuous, private address space through address translation mechanisms. The Memory Management Unit (MMU) translates virtual addresses used by programs into physical addresses in RAM.

3. Protection Memory management ensures that processes cannot access memory regions that don't belong to them, preventing crashes and security vulnerabilities. Each process operates in its own protected address space.

4. Swapping and Paging When physical memory is exhausted, the OS can temporarily move less-used pages to disk storage (swap space), allowing more processes to run than would fit in physical RAM simultaneously.

Resource Allocation Principles

Effective resource allocation in operating systems follows several key principles:

- **Fairness:** All processes should receive a reasonable share of resources
- **Efficiency:** Maximize system throughput and minimize waste
- **Responsiveness:** Minimize delay for interactive processes
- **Predictability:** Provide consistent performance characteristics

The memory manager must balance these often-competing goals while adapting to dynamic workload patterns.

---

2. Physical vs. Virtual Memory

Physical Memory

Physical memory (RAM - Random Access Memory) refers to the actual hardware memory chips installed in a computer system. Each byte in physical memory has a unique physical address that the CPU can use to read or write data. Physical memory is:

- **Limited:** Constrained by hardware capacity (typically 4GB to 128GB in modern systems)
- **Fast:** Direct access with nanosecond latencies
- **Volatile:** Contents are lost when power is removed
- **Shared:** Must be divided among all running processes and the OS kernel

Virtual Memory

Virtual memory is an abstraction layer that provides each process with the illusion of having its own large, continuous address space, independent of the physical memory configuration. Key characteristics include:

Address Space Isolation Each process has its own virtual address space, typically: - On 32-bit systems: 4GB (2^32 bytes) - On 64-bit systems: Theoretically 16 exabytes (2^64 bytes), though practical limits are lower

Memory Mapping The operating system maintains page tables that map virtual addresses to physical addresses. The same physical memory can be mapped to different virtual addresses in different processes.

Demand Paging Not all virtual memory needs to be in physical RAM simultaneously. Pages can be loaded from disk on-demand when accessed, allowing programs to use more memory than physically available.

Advantages of Virtual Memory

1. **Process Isolation:** Programs cannot interfere with each other's memory
2. **Simplified Programming:** Programmers don't need to manage complex memory layouts
3. **Memory Overcommitment:** Total virtual memory across all processes can exceed physical RAM
4. **Memory Protection:** Hardware-enforced access controls prevent unauthorized access
5. **Shared Libraries:** Multiple processes can share a single physical copy of code in memory
6. **Simplified Loading:** Programs can be loaded at any physical address regardless of their compiled address

The Address Translation Process

When a program accesses memory at a virtual address: 1. The MMU intercepts the memory access 2. It consults the page table to find the corresponding physical address 3. If the page is in RAM (page hit), the access proceeds 4. If the page is not in RAM (page fault), the OS loads it from disk 5. The physical memory is accessed and data is returned to the program

---

3. Paging Mechanisms

Fundamental Concepts

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. The operating system divides both virtual and physical memory into fixed-size blocks:

- **Pages:** Fixed-size blocks of virtual memory (typically 4KB, 2MB, or 1GB)
- **Frames:** Fixed-size blocks of physical memory (same size as pages)
- **Page Table:** Data structure mapping virtual pages to physical frames

Page Table Structure

Each process has its own page table stored in memory. A basic page table entry (PTE) contains:

Physical Frame Number (PFN): The physical frame where the page is stored **Present Bit:** Indicates if the page is currently in physical memory **Dirty Bit:** Set when the page has been modified **Accessed Bit:** Set when the page has been read or written **Protection Bits:** Define read/write/execute permissions **Valid Bit:** Indicates if this page is part of the process's address space

Multilevel Paging

For large address spaces, single-level page tables become impractically large. Modern systems use multilevel (hierarchical) page tables:

Two-Level Paging: - Virtual address divided into: Page Directory Index | Page Table Index | Offset - The page directory points to page tables - Unused portions of the address space don't require page tables

Four-Level Paging (x86-64): - PML4 (Page Map Level 4) - PDPT (Page Directory Pointer Table) - PD (Page Directory) - PT (Page Table) - Offset within page

This hierarchical structure significantly reduces memory overhead for sparse address spaces.

Translation Lookaside Buffer (TLB)

Since page table lookups require memory accesses, they can significantly slow down program execution. The TLB is a small, fast cache in the CPU that stores recent virtual-to-physical address translations:

- **Size:** Typically 64-2048 entries
- **Speed:** Access time of 1-2 CPU cycles
- **Hit Rate:** Often 95-99% in normal workloads
- **Impact:** TLB miss can cost 10-100 CPU cycles

When a TLB miss occurs, the hardware or OS performs a "page table walk" to find the translation, then caches it in the TLB.

Page Replacement Algorithms

When physical memory is full and a page fault occurs, the OS must select a victim page to evict. Common algorithms include:

FIFO (First-In-First-Out): Evicts the oldest page in memory. Simple but can remove frequently-used pages.

LRU (Least Recently Used): Evicts the page that hasn't been accessed for the longest time. Better performance but requires tracking access times.

Clock/Second Chance: A practical approximation of LRU using the accessed bit. Pages are arranged in a circular list, and the algorithm gives recently accessed pages a "second chance."

LFU (Least Frequently Used): Evicts the page with the fewest accesses. Can keep old pages that are no longer needed.

Optimal: Theoretical algorithm that evicts the page that won't be used for the longest time. Impossible to implement but serves as a benchmark.

---

4. Paging Mechanisms: Advantages and Limitations

Advantages of Paging

1. Elimination of External Fragmentation Since pages are fixed-size, any free frame can accommodate any page. This completely eliminates external fragmentation where free memory exists but is scattered in small unusable chunks.

2. Simplification of Memory Allocation Memory allocation becomes straightforward: find any free frame for each page. No need for complex algorithms to find contiguous blocks of the right size.

3. Efficient Swapping Individual pages can be swapped in and out of memory independently. The OS doesn't need to swap entire processes, only the pages currently needed (demand paging).

4. Memory Sharing Multiple processes can easily share pages by pointing their page table entries to the same physical frames. This is commonly used for: - Shared libraries (e.g., libc, system DLLs) - Shared memory segments for inter-process communication - Copy-on-write for fork() system calls

5. Protection and Isolation Page-level protection bits allow fine-grained control over memory access. Each page can be independently marked as read-only, read-write, or executable.

6. Simplified Program Loading Programs can be loaded into any available frames without relocation. The page table handles the mapping automatically.

7. Memory Overcommitment Virtual memory can exceed physical memory, allowing more processes to run concurrently through demand paging and swapping.

Limitations and Challenges

1. Internal Fragmentation The last page of a process's memory allocation is rarely completely filled. On average, half a page per region is wasted. With 4KB pages, this is usually acceptable, but it still represents overhead.

Example: A program needs 10.5KB of memory. It requires 3 pages (12KB), wasting 1.5KB (12.5% overhead).

2. Page Table Memory Overhead Page tables themselves consume memory. For a 32-bit system with 4KB pages: - Address space: 4GB = 2^32 bytes - Pages needed: 2^32 / 2^12 = 2^20 = 1 million pages - Page table size (4 bytes/entry): 4MB per process

With multilevel paging, this is reduced for sparse address spaces, but the overhead remains significant.

3. TLB Miss Overhead Each TLB miss requires a page table walk, which can take 10-100 CPU cycles. Programs with poor locality can suffer significant performance degradation from TLB thrashing.

4. Page Table Walk Latency Multi-level page tables require multiple memory accesses to translate a single address: - 4-level paging requires up to 4 memory accesses - Each access takes 50-100 nanoseconds - Total translation time can reach 400 nanoseconds without TLB

5. Page Fault Handling Cost When a required page is not in memory: - Disk I/O takes milliseconds (1,000,000+ nanoseconds) - Context switch overhead while waiting for I/O - Potential for thrashing when working set exceeds physical memory

6. Fixed Page Size Limitations Standard 4KB pages are not optimal for all workloads: - Large databases benefit from huge pages (2MB/1GB) to reduce TLB misses - Embedded systems might prefer smaller pages to reduce fragmentation - Most systems support only a few fixed page sizes

7. Memory Bloat The ease of memory allocation in paged systems can lead to inefficient memory usage by applications, as programmers may not optimize memory consumption carefully.

---

5. Segmentation Mechanisms

Segmentation Fundamentals

Segmentation is an alternative memory management technique that divides a process's address space into logical segments of variable length, each representing a logical unit of the program such as:

- **Code Segment:** Contains executable instructions
- **Data Segment:** Global and static variables
- **Stack Segment:** Function call stack and local variables
- **Heap Segment:** Dynamically allocated memory
- **Shared Library Segments:** Code and data from shared libraries

Unlike paging's fixed-size divisions, segmentation reflects the logical structure of programs.

Segment Table Structure

Each process has a segment table with entries containing:

Segment Base: Starting physical address of the segment **Segment Limit:** Length of the segment (for bounds checking) **Protection Bits:** Read/write/execute permissions **Present Bit:** Whether the segment is in memory **Growth Direction:** For stack segments that grow downward

Address Translation in Segmentation

A logical address consists of two parts: `<segment number, offset>`

The translation process: 1. Extract segment number from the logical address 2. Use segment number to index into the segment table 3. Check that offset < segment limit (bounds checking) 4. Add offset to segment base to get physical address 5. Check protection bits for access permission

Fault Conditions: - Offset exceeds segment limit ��� Segmentation Fault - Operation violates protection bits ��� Protection Fault

Advantages of Segmentation

1. Logical Organization Segments correspond to logical units in the program, making memory management more intuitive and aligned with program structure.

2. Flexible Segment Sizes Each segment can be exactly the size needed, growing and shrinking independently. No internal fragmentation within segments.

3. Protection and Sharing Different segments can have different protection attributes: - Code segments can be read-only and executable - Data segments can be read-write but not executable (NX bit protection) - Segments can be shared between processes (shared libraries)

4. Simplified Compilation and Linking Compilers naturally organize programs into segments. This alignment simplifies the compilation and linking process.

5. Dynamic Growth Segments like the heap and stack can grow dynamically without relocating the entire process.

Disadvantages of Segmentation

1. External Fragmentation Variable-size segments lead to external fragmentation as segments are allocated and deallocated. Over time, memory becomes fragmented with small unusable gaps.

2. Allocation Complexity Finding free space for variable-size segments requires complex algorithms: - First Fit: Use first block large enough - Best Fit: Use smallest block that fits (minimizes waste) - Worst Fit: Use largest block (leaves larger fragments)

Each has tradeoffs between speed and fragmentation.

3. Segment Table Overhead While smaller than page tables, segment tables still consume memory and require memory accesses for translation.

4. Difficulty of Swapping Variable-size segments are harder to swap to disk and back. Finding space for returning segments is non-trivial.

Segmentation with Paging

Many modern systems (including x86-64 in compatibility mode) combine segmentation and paging:

1. Logical address ��� Segmentation ��� Linear address
2. Linear address ��� Paging ��� Physical address

This hybrid approach provides: - Logical organization from segmentation - Efficient memory use and elimination of external fragmentation from paging - Protection at both segment and page levels

However, the complexity of managing both mechanisms is substantial, which is why pure 64-bit modes often minimize or eliminate segmentation.

---

6. Memory Allocation Strategies

Contiguous Allocation Strategies

When allocating memory from a pool of free blocks, various strategies determine which block to use:

First Fit

Algorithm: Allocate the first free block that is large enough.

Advantages: - Fast: stops searching after first suitable block - Simple to implement - Good performance for many workloads

Disadvantages: - Can create small fragments at the beginning of free memory - Tends to break up large blocks unnecessarily

Use Case: General-purpose allocation when speed is important.

Best Fit

Algorithm: Allocate the smallest free block that is large enough.

Advantages: - Minimizes wasted space in each allocation - Preserves larger blocks for future large allocations

Disadvantages: - Slower: must search entire free list - Creates many tiny unusable fragments - Can lead to severe fragmentation over time

Use Case: When memory is critically scarce and fragmentation of large blocks must be avoided.

Worst Fit

Algorithm: Allocate the largest available free block.

Advantages: - Leaves larger remaining fragments - Remaining fragments more likely to be useful

Disadvantages: - Breaks up large blocks immediately - Not effective at preventing fragmentation - Rarely used in practice

Use Case: Theoretical interest; rarely optimal in practice.

Next Fit

Algorithm: Like First Fit, but continues from where the last search ended.

Advantages: - Spreads allocations more evenly across memory - Slightly faster than First Fit on average

Disadvantages: - Can break up large blocks at the end of memory - More fragmentation than First Fit

Use Case: Variant of First Fit for more uniform memory utilization.

Buddy System

A specialized allocation strategy that balances speed and fragmentation:

Algorithm: 1. Memory is divided into blocks of size 2^k 2. When allocating n bytes, find smallest k where 2^k ��� n 3. If no block of size 2^k is free, split a larger block repeatedly 4. On deallocation, merge with "buddy" block if both are free

Advantages: - Fast allocation and deallocation (O(log n)) - Coalescing is simple and efficient - Reduces external fragmentation - Used in Linux kernel (slab allocator builds on buddy system)

Disadvantages: - Internal fragmentation (up to 50% worst case) - Limited block sizes (powers of 2)

Slab Allocation

Used extensively in operating system kernels for allocating kernel objects:

Concept: - Pre-allocate groups of objects of the same size (slabs) - When object needed, take from slab - When object freed, return to slab - Slabs are allocated using a lower-level allocator (e.g., buddy system)

Advantages: - Extremely fast allocation/deallocation - No fragmentation for common object sizes - Objects can remain partially initialized, improving performance - Cache-friendly: recently freed objects may still be in CPU cache

Disadvantages: - Only works for fixed-size objects - Requires advance knowledge of common sizes - Can waste memory if slabs are underutilized

Memory Compaction

A technique to address external fragmentation:

Process: 1. Move allocated blocks to one end of memory 2. Consolidate free space at the other end 3. Update all pointers to moved blocks

Advantages: - Eliminates external fragmentation - Creates large contiguous free regions

Disadvantages: - Expensive: requires copying large amounts of memory - Complex: must update all pointers to moved objects - Pauses program execution during compaction - Requires support from runtime or hardware

Use Cases: - Garbage collected languages (Java, Python) - Systems with hardware memory management support - Rarely used in OS kernel due to complexity

---

7. Garbage Collection

Introduction to Garbage Collection

Garbage collection (GC) is an automatic memory management technique that identifies and reclaims memory that is no longer accessible to a

program. Unlike manual memory management (malloc/free in C), GC relieves programmers from explicitly deallocating memory, preventing memory leaks and dangling pointer bugs.

Reachability and Liveness

The fundamental principle of garbage collection:

Live Objects: Objects that are reachable from root references (global variables, stack variables, registers). These must be retained.

Garbage: Objects that are not reachable from any root. These can be safely reclaimed.

Root Set: Starting points for reachability analysis: - Global variables - Local variables and parameters on the stack - CPU registers - Static variables

Garbage Collection Algorithms

Reference Counting

Mechanism: Each object maintains a counter of how many references point to it. - When a reference is created: increment counter - When a reference is removed: decrement counter - When counter reaches zero: reclaim object

Advantages: - Simple to implement - Immediate reclamation when object becomes garbage - Predictable overhead

Disadvantages: - Cannot handle circular references (e.g., A���B���A remains uncollected) - Performance overhead on every reference assignment - Memory overhead for storing counters - Not suitable as a primary GC strategy for general-purpose systems

Use Cases: - Python (with cycle detector for circular references) - Swift (Automatic Reference Counting) - COM objects in Windows

Mark and Sweep

Mechanism: 1. **Mark Phase:** Starting from roots, traverse object graph and mark all reachable objects 2. **Sweep Phase:** Scan entire heap and reclaim unmarked objects

Advantages: - Handles circular references correctly - Simple conceptual model - Complete: finds all garbage

Disadvantages: - Requires stopping the program (stop-the-world pause) - Scan time proportional to heap size - Heap becomes fragmented over time - Poor cache locality during sweeping

Optimization - Mark and Compact: After marking, compact live objects to eliminate fragmentation, though this adds overhead.

Copying Collection

Mechanism: 1. Divide heap into two semi-spaces: From-space and To-space 2. Allocate new objects in From-space 3. When From-space is full: - Copy all live objects to To-space - Swap roles of From-space and To-space - Reclaim entire old From-space

Advantages: - Automatic compaction: no fragmentation - Fast allocation: simple pointer bump - Only traverses live objects, not garbage - Good cache locality for copied objects

Disadvantages: - Uses only half of available memory - Copying overhead proportional to live set size - Poor for long-lived objects that are copied repeatedly

Use Cases: - Young generation in generational collectors - Functional programming languages with many short-lived objects

Generational Collection

Hypothesis: Most objects die young (weak generational hypothesis).

Mechanism: - Divide heap into generations (typically 2-3): - **Young/Nursery Generation:** New objects, collected frequently - **Old/Tenured Generation:** Long-lived objects, collected rarely -

Permanent Generation (optional): Metadata, rarely collected - Objects promoted to older generation after surviving several collections

Advantages: - Most collections are fast (only young generation) - Amortizes cost of collecting long-lived objects - Significantly reduces pause times - Used by most production GC systems (JVM, .NET, V8)

Challenges: - Inter-generational references require write barriers - Promoted objects might die soon after promotion - Tuning generation sizes is application-dependent

Incremental and Concurrent Collection

Incremental GC: Interleaves small GC steps with program execution to avoid long pauses.

Concurrent GC: Runs GC concurrently with program execution on separate threads/cores.

Techniques: - **Tri-color Marking:** White (unexamined), Gray (examined but children pending), Black (examined with children) - **Write Barriers:** Track modifications to object graph during concurrent marking - **Read Barriers:** Ensure consistency when reading objects during GC

Advantages: - Reduced pause times - Better responsiveness for interactive applications - Utilizes multi-core processors

Disadvantages: - Increased complexity - Runtime overhead from barriers - Reduced throughput compared to stop-the-world - Potential for floating garbage

Garbage Collection vs. Manual Memory Management

Garbage Collection Benefits: - Eliminates entire classes of bugs (use-after-free, double-free, memory leaks) - Simplifies program development - Enables language features (closures, higher-order functions) - Can improve locality through compaction

Manual Management Benefits: - Deterministic deallocation (important for resources other than memory) - Lower runtime overhead - Smaller memory footprint - Predictable performance - No pause times

Hybrid Approaches: Modern systems often combine both: - RAII in C++ (deterministic destructors) - Rust's ownership system (compile-time memory safety) - Swift's ARC with manual weak references

---

8. Resource Isolation and Allocation

Resource Isolation Fundamentals

Resource isolation ensures that processes or containers cannot interfere with each other's resources or exceed their allocated quotas. This is essential for:

- **Security:** Preventing malicious processes from accessing sensitive data
- **Stability:** Preventing one process from crashing the entire system
- **Fairness:** Ensuring all processes get their fair share of resources
- **Performance Predictability:** Providing guaranteed resource availability

Operating System Mechanisms for Isolation

Memory Isolation

Virtual Address Spaces: Each process has a separate page table, ensuring it can only access its own memory. Attempts to access unmapped addresses result in segmentation faults.

Address Space Layout Randomization (ASLR): Randomizes the base addresses of the stack, heap, and libraries, making it harder for attackers to exploit memory corruption vulnerabilities.

Memory Protection Keys (MPK): Hardware feature (Intel PKU, ARM Memory Domains) allowing fine-grained protection of memory regions within a single address space.

CPU Isolation

Time Slicing: The scheduler allocates CPU time slices to each process, preventing any single process from monopolizing the CPU.

CPU Affinity: Binding processes to specific CPU cores can provide performance isolation and predictability.

CPU Quotas: Cgroups (Linux) or Job Objects (Windows) can limit CPU usage to a percentage, ensuring fair sharing.

I/O Isolation

I/O Scheduling: Schedulers (CFQ, Deadline, BFQ) ensure fair access to storage devices and prevent I/O starvation.

I/O Bandwidth Limits: Cgroups blkio controller can limit read/write bandwidth per process or container.

Network Namespaces: Separate network stacks for different containers, providing network isolation.

Process Isolation Levels

Level 1: User Separation Different user accounts provide basic isolation through file permissions and process ownership.

Level 2: Process Sandboxing Techniques like seccomp, SELinux, and AppArmor restrict system call access and capabilities.

Level 3: Containers Namespaces and cgroups provide lightweight isolation similar to virtualization but with shared kernel.

Level 4: Virtual Machines Complete hardware virtualization provides the strongest isolation, with separate kernels.

Resource Allocation Policies

Fair Share Scheduling

Ensures each user or group receives a proportional share of resources based on assigned weights.

Example: If User A has weight 2 and User B has weight 1, User A receives 2/3 of CPU time.

Priority-Based Allocation

Higher-priority processes receive preferential access to resources.

Considerations: - Risk of starvation for low-priority processes - Priority inversion problems - Need for priority aging to prevent indefinite waiting

Proportional Share

Resources divided according to predefined ratios.

Example: Database container gets 40% CPU, web server gets 60% CPU.

Guaranteed Minimum with Opportunistic Maximum

Processes receive a guaranteed minimum allocation but can use more if available.

Benefits: - Ensures SLA compliance - Allows efficient use of idle resources - Common in cloud computing

Challenges in Resource Isolation

1. Shared Cache Contention Processes share CPU caches (L2, L3), leading to cache pollution and performance interference that's difficult to control.

2. NUMA Effects Non-Uniform Memory Access systems have variable memory latency. Cross-node memory access can degrade performance unpredictably.

3. Kernel Resources File descriptors, process IDs, and kernel buffers are shared system-wide and can be exhausted.

4. Side-Channel Attacks Spectre, Meltdown, and other side-channels can leak information across isolation boundaries through timing attacks.

5. Resource Accounting Overhead Tracking and enforcing resource limits adds runtime overhead.

---

9. Containerisation Concepts

Introduction to Containers

Containers are lightweight, standalone, executable packages that include everything needed to run a piece of software: code, runtime, system tools, libraries, and settings. Unlike virtual machines that virtualize hardware, containers virtualize the operating system.

Core Technologies Behind Containers

Namespaces (Linux)

Namespaces provide process-level isolation by creating separate instances of global system resources:

PID Namespace: Isolates process IDs. Processes in different PID namespaces can have the same PID. Container sees itself as PID 1.

Network Namespace: Provides separate network stack including interfaces, routing tables, and firewall rules.

Mount Namespace: Isolates mount points, allowing different filesystem views. Container has its own root filesystem.

UTS Namespace: Isolates hostname and domain name.

IPC Namespace: Isolates inter-process communication resources (message queues, semaphores, shared memory).

User Namespace: Maps user and group IDs between container and host, allowing root inside container to be non-root outside.

Cgroup Namespace: Virtualizes the view of cgroups, enhancing container isolation.

Control Groups (cgroups)

Cgroups limit, account for, and isolate resource usage:

CPU Controller: - CPU shares (proportional allocation) - CPU quota (hard limits) - CPU pinning (affinity)

Memory Controller: - Memory limits (hard cap) - Memory reservations (soft limit) - OOM (Out of Memory) control - Swap limits

Block I/O Controller: - Read/write bandwidth limits - IOPS (I/O operations per second) limits - Device weight-based proportional allocation

Network Controller: - Bandwidth limits - Priority classes

Device Controller: - Access control to devices (read/write/mknod)

Union Filesystems

Enable layered filesystem images:

Concept: - Base layer: Operating system files (read-only) - Intermediate layers: Application dependencies (read-only) - Top layer: Container-specific changes (read-write)

Benefits: - Efficient storage: shared layers used by multiple containers - Fast container startup - Easy image distribution - Copy-on-write optimization

Implementations: - OverlayFS (modern Linux default) - AUFS (older, deprecated) - Btrfs, ZFS (filesystem-native)

Container Runtime Architecture

High-Level Runtime (Docker, Podman): - Image management and distribution - Network configuration - Volume management - User-

friendly API and CLI

Low-Level Runtime (containerd, CRI-O): - Interface with kernel features (namespaces, cgroups) - Lifecycle management - Container execution

OCI Runtime (runc): - Actual container execution - Direct namespace and cgroup manipulation - Implements OCI (Open Container Initiative) specification

Containers vs. Virtual Machines

Containers: - Share host kernel - Lightweight (MBs in size) - Fast startup (milliseconds to seconds) - Higher density (100s per host) - Less isolation - Lower overhead

Virtual Machines: - Separate kernel per VM - Heavyweight (GBs in size) - Slow startup (minutes) - Lower density (10s per host) - Strong isolation - Higher overhead

Container Orchestration

Managing containers at scale requires orchestration platforms:

Kubernetes: - Industry standard for container orchestration - Declarative configuration - Automatic scaling, load balancing, and self-healing - Multi-host cluster management

Key Concepts: - **Pods:** Group of co-located containers - **Services:** Stable network endpoints for pods - **Deployments:** Declarative updates for pods - **ConfigMaps/Secrets:** Configuration and sensitive data management - **Persistent Volumes:** Storage abstraction

Docker Swarm: - Native Docker clustering - Simpler than Kubernetes - Integrated with Docker CLI

Use Cases for Containers

1. **Microservices Architecture:** Each service runs in its own container
2. **CI/CD Pipelines:** Consistent build and test environments

3. **Development Environments:** "Works on my machine" ��➡ "Works in this container"
4. **Application Packaging:** Distribute applications with all dependencies
5. **Multi-tenancy:** Isolate customer workloads
6. **Legacy Application Modernization:** Containerize without rewriting

Security Considerations

Attack Surface: - Shared kernel increases risk compared to VMs - Kernel vulnerabilities affect all containers - Container escape vulnerabilities (rare but serious)

Best Practices: - Run containers as non-root users - Use minimal base images (distroless, Alpine) - Scan images for vulnerabilities - Apply security policies (SELinux, AppArmor, seccomp) - Keep runtime and host OS updated - Use user namespaces - Limit capabilities (drop unnecessary Linux capabilities)

---

10. Resource Limits and Quality of Service

Quality of Service (QoS) Fundamentals

Quality of Service refers to the set of techniques and mechanisms that ensure adequate performance for critical workloads even under resource contention. QoS policies define how resources are allocated when demand exceeds supply.

QoS Classes

Systems typically define multiple QoS tiers:

Guaranteed (Critical)

Characteristics: - Highest priority - Reserved resources guaranteed - Protected from eviction or throttling - Used for latency-sensitive,

business-critical applications

Example: Financial trading systems, real-time data processing

Implementation: - CPU: Reserved cores or guaranteed CPU time - Memory: Hard reservations, never swapped - I/O: Priority I/O scheduling

Burstable (Normal)

Characteristics: - Guaranteed minimum resources - Can use additional resources when available - May be throttled under contention - Most general-purpose applications

Example: Web servers, databases with variable load

Implementation: - CPU: Minimum shares with opportunistic burst - Memory: Soft limits with burst capacity - I/O: Fair-share scheduling

Best Effort (Low Priority)

Characteristics: - No resource guarantees - Uses only leftover resources - First to be throttled or evicted - Suitable for non-critical, batch processing

Example: Log processing, analytics, backups

Implementation: - CPU: Lowest priority, throttled first - Memory: Can be swapped aggressively - I/O: Background priority

Resource Limit Types

Hard Limits

Absolute maximum resources a process can use. Exceeding hard limits results in: - **Memory:** OOM killer terminates process - **CPU:** Strict throttling to limit - **I/O:** Requests queued or blocked - **File descriptors:** Open calls fail with EMFILE

Use Cases: - Prevent runaway processes from affecting others - Enforce billing/quota compliance - Security boundaries

Soft Limits

Advisory limits that can be temporarily exceeded: - Process can increase usage up to hard limit - Warnings or accounting when soft limit exceeded - Allows flexibility for bursty workloads

Use Cases: - Performance monitoring - Gradual degradation before hard enforcement - Alerts for capacity planning

Requests vs. Limits (Kubernetes Model)

Requests: - Resources guaranteed to the container - Used for scheduling decisions - Sum of requests must fit on node

Limits: - Maximum resources container can use - Prevents unlimited growth - Can exceed node capacity (overcommitment)

Example: `yaml resources: requests: memory: "256Mi" cpu: "500m" limits: memory: "512Mi" cpu: "1000m"`

Memory Limits and OOM Behavior

OOM Score

Linux assigns each process an OOM (Out-of-Memory) score based on: - Memory usage - Process age - OOM_ADJ setting (administrator preference)

When memory is exhausted, the OOM killer terminates the highest-scoring process.

OOM Control in Cgroups

OOM Killer Disable: Can disable OOM killer for a cgroup, causing memory allocation to block instead of killing processes.

OOM Notifications: Applications can receive notifications when approaching memory limits and take corrective action.

Memory Pressure: Multi-level warnings (low, medium, critical) allow proactive response.

CPU Limits Implementation

CPU Shares (Proportional)

Relative weight for CPU allocation: - Default: 1024 shares per process - Process with 2048 shares gets 2x CPU of process with 1024 shares - Only meaningful under contention

CPU Quota (Absolute)

Hard limit on CPU time: - Specified as percentage or time units - Process throttled when quota exhausted in period - Can cause latency spikes

Example: 50% CPU quota = 50ms CPU per 100ms period

CPU Sets

Restricts process to specific CPU cores: - Provides isolation from other processes - Controls NUMA placement - Useful for real-time and HPC applications

I/O Resource Limits

I/O Bandwidth Limits

Read/Write Bytes per Second: Throttles total data transfer rate.

IOPS Limits: Restricts number of I/O operations per second, important for random I/O workloads.

I/O Priority Classes

Real-time: Preempts other I/O **Best-effort:** Normal priority with numeric level (0-7) **Idle:** Only runs when no other I/O pending

Network QoS

Traffic Shaping

Token Bucket: Average rate with burst capacity.

Leaky Bucket: Smooth traffic to constant rate.

Priority Queuing: High-priority packets sent first.

Bandwidth Allocation

Guaranteed Bandwidth: Minimum bandwidth reserved.

Bandwidth Limits: Maximum bandwidth cap.

Fair Queuing: Equal bandwidth per flow.

Monitoring and Enforcement

Resource Accounting

Tracking actual usage: - CPU time (user, system, I/O wait) - Memory usage (RSS, cache, swap) - I/O operations and bandwidth - Network traffic

Limit Violation Handling

Throttling: Slow down resource consumption **Preemption:** Pause or interrupt execution **Termination:** Kill process (last resort) **Notification:** Alert administrators or applications

Challenges and Trade-offs

Overhead: Accounting and enforcement add CPU overhead (typically 1-5%).

Granularity: Fine-grained limits provide better control but increase complexity.

Interference: Difficult to isolate shared resources (caches, memory bandwidth).

Configuration Complexity: Setting appropriate limits requires deep understanding of application behavior.

Dynamic Workloads: Static limits may be too restrictive or too permissive for varying workloads.

---

11. Memory Monitoring Tools

System-Wide Memory Monitoring

free

Displays overall memory usage:

```
bash free -h
```

Output Interpretation: - **Total:** Total physical RAM - **Used:** Memory used by processes and kernel - **Free:** Completely unused memory - **Shared:** Memory used by tmpfs - **Buff/cache:** Kernel buffers and page cache (reclaimable) - **Available:** Memory available for new processes (without swapping)

Key Insight: Low "free" is normal; focus on "available" instead.

vmstat

Reports virtual memory statistics:

```
bash vmstat 1
```

Key Fields: - **si/so:** Swap in/out (non-zero indicates memory pressure) - **bi/bo:** Block I/O in/out - **us/sy/id/wa:** CPU time (user, system, idle, I/O wait)

Use Case: Identifying swapping and memory pressure over time.

top/htop

Interactive process monitoring:

Features: - Per-process memory usage (RES, VIRT, SHR) - Sorting by memory consumption - Real-time updates - Kill processes directly (htop)

Memory Columns: - **VIRT:** Virtual memory size (total address space) - **RES:** Resident set size (physical RAM used) - **SHR:** Shared memory with other processes - **%MEM:** Percentage of physical RAM

/proc/meminfo

Detailed memory statistics:

```
bash cat /proc/meminfo
```

Important Fields: - **MemTotal:** Total RAM - **MemFree:** Unused RAM - **MemAvailable:** Estimation of available memory - **Buffers/Cached:** Kernel caches - **SwapTotal/SwapFree:** Swap space - **Dirty:** Modified pages pending write to disk - **Slab:** Kernel slab allocator usage

Process-Specific Memory Monitoring

/proc/[pid]/status

Comprehensive process information:

```
bash cat /proc/[pid]/status
```

Memory-Related Fields: - **VmSize:** Virtual memory size - **VmRSS:** Resident set size - **VmData:** Data segment size - **VmStk:** Stack size - **VmExe:** Executable size - **VmLib:** Shared library size

/proc/[pid]/smaps

Detailed memory map analysis:

```
bash cat /proc/[pid]/smaps
```

Information Per Mapping: - Address range - Permissions (r/w/x) - Size, RSS, PSS (Proportional Set Size) - Shared vs. private memory - Referenced pages

PSS (Proportional Set Size): More accurate than RSS for shared memory. Divides shared pages proportionally among processes.

pmap

Maps process memory regions:

```
bash pmap -x [pid]
```

Output: - Address, size, permissions for each mapping - RSS and dirty pages - Mapping name (file, anonymous, stack, heap)

valgrind (memcheck)

Detects memory errors:

```
bash valgrind --leak-check=full ./program
```

Detects: - Memory leaks - Use of uninitialized memory - Invalid memory access - Double frees

Limitation: Significant slowdown (10-50x), unsuitable for production.

Container-Specific Monitoring

docker stats

Real-time container resource usage:

```
bash docker stats [container]
```

Metrics: - Memory usage and limit - Memory percentage - CPU usage - Network I/O - Block I/O

cgroup memory statistics

Direct cgroup interface:

```
bash cat
/sys/fs/cgroup/memory/docker/[container_id]/memory.stat
```

Detailed Metrics: - cache, rss, mapped_file - pgpgin, pgpgout (page in/out) - pgfault, pgmajfault - inactive/active anonymous and file pages

Kubernetes Metrics

kubectl top: `bash kubectl top pod [pod-name] kubectl top node`

Metrics Server: Cluster-wide resource usage **Prometheus:** Time-series monitoring with alerting

Advanced Memory Profiling

perf

Linux performance profiler:

```
bash perf record -g ./program perf report
```

Capabilities: - Memory access patterns - Cache misses - TLB misses - Memory bandwidth utilization

BCC/bpftrace

eBPF-based dynamic tracing:

```bash

# Track page faults by process
```

bpftrace -e 'software:page-fault:1 { @[comm] = count(); }' ```

Use Cases: - Page fault analysis - Memory allocation patterns - Slab allocator monitoring

strace

System call tracing:

```
bash strace -e trace=memory ./program
```

Tracks: - brk (heap expansion) - mmap/munmap (memory mapping) - mprotect (permission changes)

Memory Leak Detection

AddressSanitizer (ASan)

Compiler-based memory error detector:

```
bash gcc -fsanitize=address program.c ./a.out
```

Detects: - Heap buffer overflow - Stack buffer overflow - Use-after-free - Memory leaks

Advantage: Lower overhead than valgrind (2x slowdown).

LeakSanitizer (LSan)

Standalone leak detector, part of ASan:

```
bash gcc -fsanitize=leak program.c
```

Heap Profilers

massif (part of valgrind): `bash valgrind --tool=massif ./program ms_print massif.out.[pid]`

Generates heap usage over time graph.

Google's tcmalloc: Built-in heap profiler with minimal overhead.

Best Practices for Memory Monitoring

1. **Establish Baselines:** Know normal memory usage patterns
2. **Monitor Trends:** Watch for gradual increases (leaks)
3. **Set Alerts:** Notify when thresholds exceeded
4. **Correlate Metrics:** Memory + I/O + CPU for complete picture
5. **Use Appropriate Tools:** System-wide vs. process-specific vs. profiling
6. **Consider Overhead:** Profiling tools add overhead
7. **Automate Collection:** Use monitoring systems (Prometheus, Grafana)
8. **Archive Data:** Historical data aids capacity planning

Conclusion

This journal has explored the fundamental concepts of memory management and resource allocation in modern operating systems. These topics form the backbone of system performance, security, and reliability.

Key Takeaways

Memory Management Evolution: From simple contiguous allocation to sophisticated virtual memory with paging and segmentation, operating systems have evolved to provide efficient, secure, and flexible memory abstractions.

Trade-offs Are Inherent: Every memory management technique involves trade-offs between speed, memory efficiency, fragmentation, complexity, and overhead. Understanding these trade-offs is essential for system design.

Abstraction Enables Progress: Virtual memory, garbage collection, and containerization are all abstractions that simplify programming while enabling powerful capabilities like process isolation, memory overcommitment, and application portability.

Resource Management Is Critical: As systems become more complex and multi-tenant, effective resource isolation, allocation, and quality of service mechanisms become increasingly important for security, fairness, and predictability.

Monitoring Informs Optimization: A rich ecosystem of monitoring tools enables understanding system behavior, diagnosing problems, and optimizing performance. Effective use of these tools is essential for production systems.

Practical Applications

The concepts covered in this journal are directly applicable to:

- **System Programming:** Understanding memory management is crucial for writing efficient and correct systems software
- **Application Development:** Knowledge of garbage collection and memory allocation helps in optimizing application performance
- **DevOps and Cloud Computing:** Containerization and resource limits are fundamental to modern deployment practices
- **Performance Tuning:** Memory monitoring tools and understanding of virtual memory enable effective performance optimization
- **Security:** Memory protection, isolation, and resource limits are key security mechanisms

Future Directions

Memory management continues to evolve:

- **Persistent Memory:** NVDIMMs blur the line between memory and storage
- **Hardware Innovations:** Intel MPK, ARM MTE provide new security and isolation capabilities
- **Compiler Advances:** Rust's ownership system provides memory safety without garbage collection
- **Heterogeneous Memory:** Managing multiple memory tiers (DRAM, HBM, NVM) efficiently
- **Security Focus:** Mitigating side-channel attacks and providing stronger isolation

Understanding the foundational concepts covered in this journal provides the basis for engaging with these emerging technologies and contributing to the next generation of operating system innovations.

References

Textbooks

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
2. Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.
3. Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books.

Research Papers

1. Denning, P. J. (1970). Virtual Memory. *ACM Computing Surveys*, 2(3), 153-189.
2. Jones, R., & Lins, R. (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley.

Technical Documentation

1. Linux Kernel Documentation - Memory Management: https://www.kernel.org/doc/html/latest/admin-guide/mm/
2. Docker Documentation: https://docs.docker.com/
3. Kubernetes Documentation: https://kubernetes.io/docs/

Online Resources

1. OSDev Wiki - Memory Management: https://wiki.osdev.org/Memory_Management
2. Brendan Gregg's Performance Tools: http://www.brendangregg.com/linuxperf.html

Standards

1. Open Container Initiative (OCI) Specifications: https://opencontainers.org/
2. POSIX.1-2017: https://pubs.opengroup.org/onlinepubs/9699919799/

End of Journal

This journal provides a comprehensive overview of memory management and resource allocation concepts essential for understanding modern operating systems. The explanations combine theoretical foundations with practical applications, preparing students for both academic examinations and real-world system design challenges.

Table of Contents