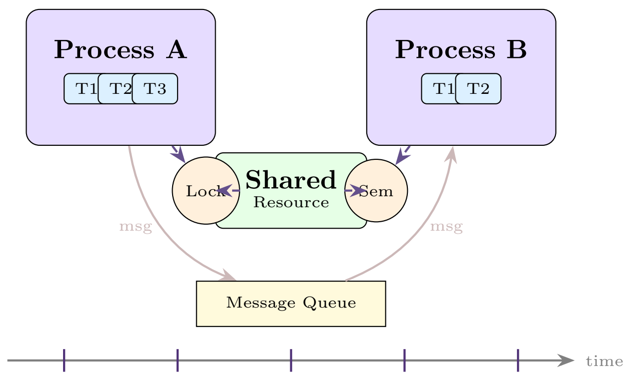
Process Viewpoint

Architecture Viewpoint Specification

Concurrency, Parallelism & Runtime Behavior



Version:	2.0
Status:	Release
Classification:	ISO/IEC/IEEE 42010 Compliant
Last Updated:	December 12, 2025

Based on the Views and Beyond approach to software architecture documentation

Contents

1 Viewpoint Name

Viewpoint Identification

Name:	Process Viewpoint
Synonyms:	Concurrency Viewpoint, Runtime Viewpoint, Thread View, Execution View, Dynamic View, Behavioral View, Parallel Processing View
Identifier:	VP-PROC-001
Version:	2.0

1.1 Viewpoint Classification

The Process Viewpoint addresses the runtime structure of a system in terms of processes, threads, and their interactions. Within the Views and Beyond approach, this corresponds to Component-and-Connector (C&C) views that emphasize runtime elements, particularly the Communicating Processes style. This viewpoint is essential for understanding how a system achieves parallelism, manages concurrency, and coordinates distributed execution.

Table 1: Viewpoint Classification Taxonomy

Attribute	Value
Style Family	Component-and-Connector (C&C)
Primary Focus	Runtime Processes, Threads, and Coordination
Abstraction Level	Runtime / Execution
Temporal Perspective	Dynamic Behavior Over Time
Related Styles	Communicating Processes, Shared-Data, Client-Server
IEEE 42010 Category	Concurrency Viewpoint
4+1 View Model	Process View

1.2 Viewpoint Scope

The Process Viewpoint encompasses multiple aspects of runtime behavior:

- **Process Structure:** Operating system processes, their boundaries, and lifecycle management.
- **Thread Architecture:** Thread organization within processes, thread pools, and worker patterns.
- **Concurrency Control:** Mechanisms for coordinating concurrent access to shared resources including locks, semaphores, monitors, and transactions.
- **Inter-Process Communication:** How processes exchange data through shared memory, message passing, pipes, sockets, and other IPC mechanisms.

- **Synchronization:** Coordination points where processes or threads must synchronize their execution.
- **Parallelism:** How work is distributed across multiple processing units for performance.
- **Scheduling:** How processing resources are allocated to competing tasks.
- **State Management:** How state is managed across concurrent execution paths.

2 Overview

The Process Viewpoint provides a comprehensive framework for documenting the runtime execution structure of a software system. It addresses how the system leverages concurrent and parallel execution, manages shared resources safely, and coordinates activities across multiple execution contexts.

2.1 Purpose and Scope

The primary purpose of this viewpoint is to establish a clear understanding of the system's runtime topology in terms of processes and threads, the mechanisms used for coordination and communication, and the strategies employed to achieve required performance and reliability characteristics in a concurrent environment.

Viewpoint Definition

The Process Viewpoint defines the runtime structure of a system in terms of processes, threads, and other units of execution. It documents how these execution units are organized, how they communicate and synchronize, and how they share resources. This viewpoint addresses concerns of concurrency, parallelism, performance, scalability, and fault tolerance at runtime.

2.2 Key Characteristics

The Process Viewpoint exhibits several distinctive characteristics:

Runtime Focus: Unlike module views that show static structure, this viewpoint shows runtime entities that exist during execution and may be created/destroyed dynamically.

Temporal Dimension: Processes and threads execute over time, making temporal ordering, synchronization points, and timing constraints essential aspects of documentation.

Non-Determinism: Concurrent systems can exhibit non-deterministic behavior due to scheduling, making analysis of possible interleavings important.

Resource Contention: Multiple execution units competing for shared resources requires explicit documentation of contention management.

Scalability Implications: Process and thread architecture directly impacts system scalability and performance under load.

2.3 Relationship to Other Viewpoints

The Process Viewpoint connects to other architectural viewpoints in significant ways:

Table 2: Relationships to Other Viewpoints

Viewpoint	**Relationship**
Development	Code modules are packaged into processes. Thread implementations reside in modules.
Deployment	Processes execute on nodes. Thread pools size based on hardware. Process distribution across nodes.
Component-and-Connector	Processes are runtime manifestations of components. Connectors implement IPC.
Information/Data	Data stores are accessed concurrently. Transactions coordinate data access.
Operational	Process monitoring, scaling policies, resource management.
Security	Process isolation, privilege separation, secure IPC.

2.4 Concurrency Architecture Overview

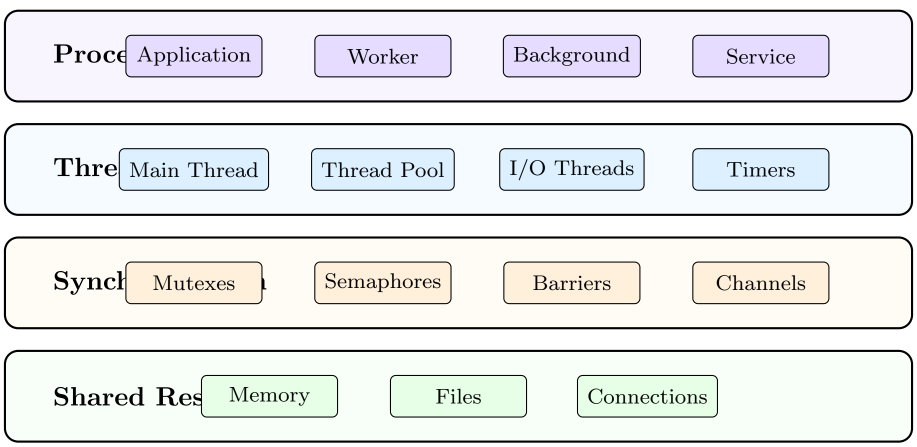


Figure 1: Concurrency Architecture Layers

3 Concerns

This section enumerates the architectural concerns that the Process Viewpoint is designed to address.

3.1 Primary Concerns

C1: Process Organization

- What are the major processes in the system?
- How are processes organized and related?
- What is the lifecycle of each process?
- How are processes created, managed, and terminated?
- What isolation boundaries exist between processes?

C2: Thread Architecture

- How are threads organized within processes?
- What threading models are used (one-per-request, thread pool, event loop)?
- How are thread pools sized and managed?
- What thread-local storage is used?
- How is thread lifecycle managed?

C3: Concurrency Control

- What shared resources require protection?
- What synchronization mechanisms are used?
- How is deadlock prevented or detected?
- What locking granularity is employed?
- How is lock contention minimized?

C4: Inter-Process Communication

- How do processes communicate?
- What IPC mechanisms are used (shared memory, messages, pipes)?
- What message formats and protocols are employed?
- How is communication reliability ensured?
- What are the communication patterns (sync/async, unicast/multicast)?

C5: Parallelism and Performance

- How is work partitioned for parallel execution?
- What parallel algorithms or patterns are used?
- How does the system scale with additional processors?
- What are the performance bottlenecks?
- How is load balanced across execution units?

C6: Scheduling and Priority

- How are tasks scheduled for execution?
- What priority schemes are used?
- How is fairness ensured?
- What real-time constraints exist?
- How is scheduling latency managed?

C7: State Management

- What state is shared between execution units?
- How is state consistency maintained?
- What isolation levels are provided?
- How is distributed state coordinated?
- What happens to state during failures?

C8: Fault Tolerance

- How does the system handle process failures?
- What supervision and restart strategies exist?
- How are partial failures handled?
- What failure isolation boundaries exist?
- How is system consistency maintained after failures?

C9: Resource Management

- How are system resources (CPU, memory, connections) allocated?
- What resource limits and quotas exist?
- How is resource exhaustion prevented?
- What cleanup mechanisms ensure resource release?
- How are resources pooled and reused?

C10: Determinism and Reproducibility

- How deterministic is system behavior?
- What sources of non-determinism exist?
- How is testing of concurrent behavior performed?
- What debugging support exists for concurrency issues?
- How are race conditions detected and prevented?

3.2 Concern-Quality Attribute Mapping

Table 3: Concern to Quality Attribute Mapping

Concern	Performance	Scalability	Reliability	Availability	Security	Maintaintic.	Testability	Safety
Process Org.	○	●	●	●	●	○	○	○
Thread Arch.	●	●	○	○	○	○	○	○
Concurrency	●	○	●	○	○	○	○	●
IPC	●	●	○	○	●	○	○	–
Parallelism	●	●	○	–	–	○	○	–
Scheduling	●	○	○	○	–	–	–	●
State Mgmt	○	○	●	○	○	○	○	●
Fault Tol.	–	○	●	●	–	○	○	●
Resources	●	●	●	●	○	○	–	–
Determinism	○	–	○	–	–	○	●	●

● = Primary impact, ○ = Secondary impact, – = Minimal impact

4 Anti-Concerns

Understanding what the Process Viewpoint is *not* appropriate for helps stakeholders avoid misapplying this viewpoint.

4.1 Out of Scope Topics

AC1: Static Code Structure

- Module organization and dependencies
- Class hierarchies and inheritance
- Package structure and namespaces
- Build dependencies
- Code file organization

AC2: Physical Deployment

- Hardware specifications
- Network topology
- Container orchestration
- Cloud infrastructure
- Data center design

AC3: Data Modeling

- Entity-relationship models

- Database schema design
- Data flow diagrams (except IPC data)
- Data retention policies
- Data quality rules

AC4: User Interface

- Screen layouts and designs
- User interaction flows
- Accessibility features
- Responsive design
- UI component architecture

AC5: Business Logic Details

- Algorithmic implementations
- Business rules specifications
- Domain model details
- Validation logic
- Calculation formulas

Common Misapplications

Avoid using the Process Viewpoint for:

- Documenting code module dependencies (use Development Viewpoint)
- Specifying server topology (use Deployment Viewpoint)
- Defining data schemas (use Information Viewpoint)
- Detailing API contracts (use Interface Specifications)
- Specifying functional requirements (use Functional Viewpoint)

5 Typical Stakeholders

The Process Viewpoint serves multiple stakeholder communities with concerns about system runtime behavior.

5.1 Primary Stakeholders

Table 4: Primary Stakeholder Analysis

Stakeholder	Role Description	Primary Interests
Software Architects	Design system structure	Process topology, IPC mechanisms, concurrency strategy, scalability
Performance Engineers	Optimize system performance	Thread pools, parallelism, contention, bottleneck analysis
Senior Developers	Implement concurrent code	Synchronization primitives, thread safety, deadlock avoidance
Platform Engineers	Manage runtime platform	Process management, resource allocation, scheduling
QA/Test Engineers	Validate concurrent behavior	Race condition testing, stress testing, determinism
System Integrators	Connect system components	IPC protocols, message formats, integration patterns

5.2 Secondary Stakeholders

Table 5: Secondary Stakeholder Analysis

Stakeholder	Role Description	Primary Interests
Operations Teams	Manage production systems	Process monitoring, resource usage, failure recovery
Security Architects	Ensure system security	Process isolation, privilege separation, secure IPC
Capacity Planners	Plan system resources	Thread scaling, process distribution, resource requirements
Real-Time Engineers	Ensure timing constraints	Scheduling, latency, determinism, priority inversion
Embedded Engineers	Develop constrained systems	Resource-limited concurrency, interrupt handling
Technical Managers	Oversee development	Risk assessment, complexity management, skill requirements

5.3 Stakeholder Concern Matrix

Table 6: Stakeholder-Concern Responsibility Matrix

	Process	Thread	Concurr.	IPC	Parallel	Schedule	State	Fault	Resource	Determ.
Architect	R	R	A	R	A	C	A	A	C	C
Perf. Eng.	C	R	C	C	R	R	C	I	R	C
Developer	C	A	R	A	C	I	R	C	C	R
Platform	A	C	I	C	C	A	I	R	A	I
QA/Test	I	C	C	C	I	I	C	C	I	R
Security	C	I	C	A	I	I	C	C	C	I

R = Responsible, A = Accountable, C = Consulted, I = Informed

6 Model Types

The Process Viewpoint employs several complementary model types to capture different aspects of concurrent system structure and behavior.

6.1 Model Type Catalog

MT1: Process Structure Diagram

- *Purpose:* Show processes, their relationships, and communication paths
- *Primary Elements:* Processes, threads, communication channels
- *Key Relationships:* Communicates-with, spawns, supervises
- *Typical Notation:* UML component diagrams, custom process diagrams

MT2: Thread Pool Model

- *Purpose:* Document thread organization and pooling strategy
- *Primary Elements:* Thread pools, worker threads, task queues
- *Key Relationships:* Executes, queues, manages
- *Typical Notation:* Pool diagrams, queue diagrams

MT3: Synchronization Model

- *Purpose:* Show synchronization mechanisms and protected resources
- *Primary Elements:* Locks, semaphores, monitors, critical sections
- *Key Relationships:* Protects, acquires, waits-on
- *Typical Notation:* Lock diagrams, resource access matrices

MT4: Sequence Diagram (Concurrent)

- *Purpose:* Show temporal ordering of concurrent operations
- *Primary Elements:* Lifelines, messages, activation bars, parallel fragments

- *Key Relationships:* Calls, signals, synchronizes
- *Typical Notation:* UML sequence diagrams with par/alt fragments

MT5: State Machine (Concurrent)

- *Purpose:* Model states and transitions in concurrent contexts
- *Primary Elements:* States, transitions, concurrent regions
- *Key Relationships:* Transitions-to, forks, joins
- *Typical Notation:* UML statecharts with orthogonal regions

MT6: Message Flow Diagram

- *Purpose:* Document inter-process message passing
- *Primary Elements:* Processes, queues, messages, channels
- *Key Relationships:* Sends, receives, routes
- *Typical Notation:* Message sequence charts, data flow diagrams

MT7: Resource Contention Model

- *Purpose:* Analyze resource sharing and contention
- *Primary Elements:* Resources, consumers, access patterns
- *Key Relationships:* Competes-for, exclusive-access, shared-access
- *Typical Notation:* Resource graphs, Petri nets

MT8: Supervision Hierarchy

- *Purpose:* Document process supervision and failure handling
- *Primary Elements:* Supervisors, workers, restart strategies
- *Key Relationships:* Supervises, restarts, escalates
- *Typical Notation:* Supervision trees (Erlang/OTP style)

MT9: Timing Diagram

- *Purpose:* Show timing constraints and execution timelines
- *Primary Elements:* Timelines, events, durations, deadlines
- *Key Relationships:* Precedes, overlaps, deadline
- *Typical Notation:* UML timing diagrams, Gantt-style charts

6.2 Model Type Relationships

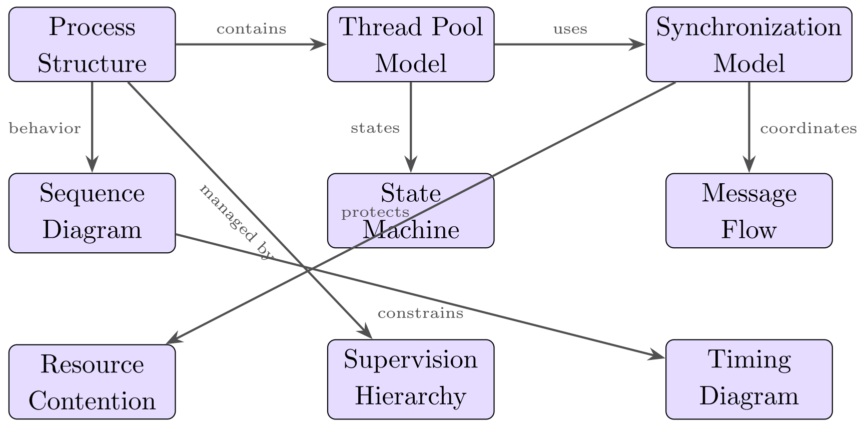


Figure 2: Model Type Dependency Relationships

7 Model Languages

For each model type, specific languages, notations, and techniques are prescribed.

7.1 Process Diagram Notation

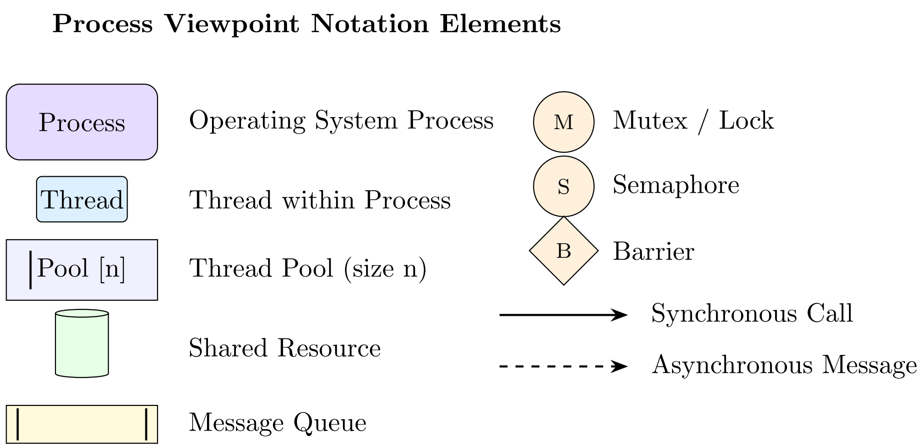


Figure 3: Process Viewpoint Notation Legend

7.2 Synchronization Primitive Summary

Table 7: Synchronization Primitive Comparison

Primitive	Purpose	Operations	Use Cases
Mutex	Mutual exclusion	lock(), unlock()	Protecting critical sections
Semaphore	Resource counting	wait(), signal()	Limiting concurrent access
Condition Variable	Wait for condition	wait(), notify(), notifyAll()	Producer-consumer
Read-Write Lock	Concurrent reads	readLock(), writeLock()	Read-heavy workloads
Barrier	Synchronization point	await()	Phased computation
Monitor	Encapsulated sync	synchronized methods	Object-level protection
Channel	Message passing	send(), receive()	CSP-style concurrency
Atomic Operations	Lock-free access	compareAndSwap()	High-performance counters

7.3 IPC Mechanism Comparison

Table 8: Inter-Process Communication Mechanisms

Mechanism	Scope	Sync/Async	Data Copy	Use Cases
Shared Memory	Same machine	Async	Zero-copy	High throughput, low latency
Pipes	Same machine	Sync/Async	Yes	Parent-child, streaming
Message Queues	Same machine	Async	Yes	Decoupled producers/consumers
Sockets	Network	Both	Yes	Distributed systems
Signals	Same machine	Async	No data	Event notification
Memory-mapped Files	Same machine	Async	Varies	Persistent shared state
RPC/gRPC	Network	Sync	Serialized	Service communication

7.4 Threading Model Comparison

Table 9: Threading Model Comparison

Model	Description	Advantages	Disadvantages
Thread-per-Request	New thread for each request	Simple model	High overhead at scale
Thread Pool	Fixed pool processes tasks	Bounded resources	Queue management
Event Loop	Single thread, async I/O	Low overhead, scalable	Blocking is problematic
Actor Model	Isolated actors with mailboxes	No shared state	Message overhead
Fork-Join	Recursive task splitting	Good for divide-conquer	Task granularity
Work Stealing	Idle threads steal work	Load balancing	Complexity

7.5 Pseudocode Conventions

```
// Process definition
process WebServer {
    thread pool workerPool[16]      // Thread pool with 16 workers
    channel<Request> requestQueue   // Bounded channel for requests
    mutex configLock                // Protects configuration state

    // Main thread - accepts connections
    while running {
        connection = accept()
        request = parseRequest(connection)
        requestQueue.send(request)  // Async send to queue
    }
}

// Worker thread definition
thread Worker {
    while running {
        request = requestQueue.receive()  // Blocks until message

        // Critical section - protected by lock
        synchronized(configLock) {
            config = readConfig()
        }

        response = processRequest(request, config)
        request.connection.send(response)
```

```
    }
}

// Parallel computation example
parallel for i in range(0, data.length) {
    results[i] = compute(data[i])
}
barrier.await()  // Wait for all iterations
aggregate(results)
```

Listing 1: Concurrency Pseudocode Example

7.6 Tabular Specifications

7.6.1 Process Catalog Table

Table 10: Example Process Catalog Format

Process	Role	Instances	Lifecycle	Communication
API Gateway	Request routing	3 (HA)	Long-running	HTTP in, gRPC out
Order Service	Order processing	5 (scaled)	Long-running	gRPC, Kafka
Worker	Background tasks	Auto-scaled	Transient	Redis queue
Scheduler	Task scheduling	1 (singleton)	Long-running	Database, Redis

7.6.2 Thread Pool Configuration Table

Table 11: Example Thread Pool Configuration

Pool Name	Core	Max	Queue	Timeout	Purpose
http-workers	10	100	Unbounded	30s	HTTP request handling
db-pool	20	20	Bounded(50)	10s	Database operations
async-io	4	4	Unbounded	–	Async I/O completion
scheduled	2	10	Delayed	–	Scheduled tasks

7.6.3 Lock Inventory Table

Table 12: Example Lock Inventory

Lock Name	**Type**	**Protects**	**Contention**	**Order**
configLock	RWLock	Configuration state	Low (mostly reads)	1
cacheLock	Mutex	Cache structure	Medium	2
sessionLock[id]	Mutex	Per-session state	Low (partitioned)	3
globalStats	Atomic	Statistics counters	High (lock-free)	–

8 Viewpoint Metamodels

This section defines the conceptual metamodel underlying the Process Viewpoint.

8.1 Core Metamodel

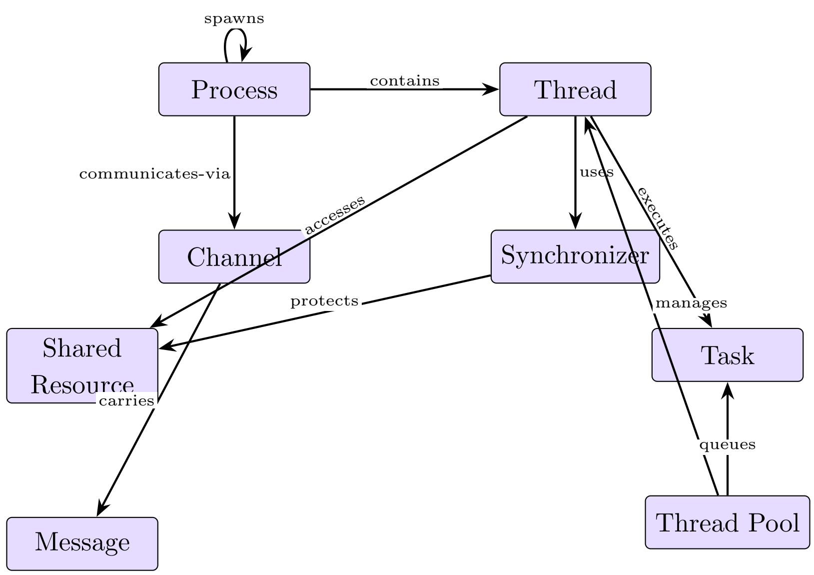


Figure 4: Process Viewpoint Core Metamodel

8.2 Entity Definitions

Entity: Process

Definition: An operating system process that provides an isolated execution environment with its own address space, resources, and one or more threads of execution.

Attributes:

- `processId`: Unique identifier
- `name`: Process name
- `description`: Purpose of the process
- `executable`: Binary or script that runs
- `instances`: Number of instances (1, N, auto-scaled)
- `lifecycle`: Lifecycle type (long-running, transient, scheduled)
- `priority`: Scheduling priority
- `resourceLimits`: CPU, memory, file descriptor limits
- `restartPolicy`: What happens on failure
- `dependencies`: Other processes this depends on
- `healthCheck`: How health is determined

Constraints:

- Every process must have at least one thread
- Long-running processes must have health checks
- Resource limits must be specified for production
- Restart policies must be defined

Entity: Thread

Definition: A unit of execution within a process that shares the process's address space but has its own stack, program counter, and thread-local storage.

Attributes:

- `threadId`: Unique identifier within process
- `name`: Thread name (for debugging)
- `type`: Thread type (main, worker, I/O, timer, daemon)
- `priority`: Thread priority
- `stackSize`: Stack size allocation
- `state`: Current state (new, runnable, blocked, waiting, terminated)
- `cpuAffinity`: CPU cores thread can run on
- `daemon`: Whether thread prevents process exit
- `interruptible`: Whether thread can be interrupted

Constraints:

- Daemon threads should not hold critical resources
- Thread names should be meaningful for debugging
- Stack size should be appropriate for workload

Entity: Thread Pool

Definition: A managed collection of pre-created threads that execute submitted tasks, providing efficient thread reuse and bounded concurrency.

Attributes:

- `poolId`: Unique identifier
- `name`: Pool name
- `coreSize`: Minimum number of threads maintained
- `maxSize`: Maximum number of threads allowed
- `queueType`: Type of task queue (bounded, unbounded, synchronous)
- `queueCapacity`: Maximum queue size (if bounded)
- `keepAliveTime`: How long idle threads are kept
- `rejectionPolicy`: What happens when pool is full
- `threadFactory`: How threads are created
- `metrics`: Pool utilization metrics

Constraints:

- Core size $\leq$ max size
- Rejection policy must be defined for bounded queues
- Pool sizing should be based on workload analysis
- Metrics should be exposed for monitoring

Entity: Synchronizer

Definition: A concurrency control mechanism that coordinates access to shared resources or synchronizes execution between threads.

Attributes:

- `synchronizerId`: Unique identifier
- `name`: Synchronizer name
- `type`: Type (mutex, semaphore, rwlock, barrier, condition, monitor)
- `fairness`: Whether waiting threads are served in order
- `permits`: Number of permits (for semaphores)
- `reentrant`: Whether same thread can acquire multiple times
- `timeout`: Default timeout for acquisition
- `owner`: Current owner (for mutexes)
- `waiters`: Number of waiting threads

Constraints:

- Locks must be released in reverse order of acquisition
- Timeouts should be used to prevent indefinite blocking
- Lock ordering must be documented to prevent deadlock

Entity: Channel

Definition: A communication pathway between processes or threads for exchanging messages, providing decoupled, type-safe communication.

Attributes:

- `channelId`: Unique identifier
- `name`: Channel name
- `type`: Channel type (unbuffered, buffered, broadcast)
- `capacity`: Buffer capacity (for buffered channels)
- `messageType`: Type of messages carried
- `producers`: Processes/threads that send
- `consumers`: Processes/threads that receive
- `deliveryGuarantee`: At-most-once, at-least-once, exactly-once
- `ordering`: FIFO, priority, unordered

Constraints:

- Unbuffered channels block sender until receiver ready
- Buffer capacity should be sized based on load analysis
- Message types should be well-defined

Entity: Shared Resource

Definition: A system resource (memory, file, connection, etc.) that is accessed by multiple threads or processes and requires coordination.

Attributes:

- `resourceId`: Unique identifier
- `name`: Resource name
- `type`: Resource type (memory, file, connection, device)
- `accessMode`: How resource can be accessed (read, write, exclusive)
- `capacity`: Resource capacity or limit
- `protection`: Synchronization mechanism protecting it
- `consumers`: Threads/processes that access it
- `contentionLevel`: Expected contention (low, medium, high)

Constraints:

- All shared mutable resources must have protection
- High-contention resources should use appropriate techniques
- Resource access patterns should be documented

Entity: Task

Definition: A unit of work that can be submitted for execution by a thread, representing a discrete computation or operation.

Attributes:

- `taskId`: Unique identifier
- `name`: Task name or type
- `priority`: Execution priority
- `state`: Current state (pending, running, completed, failed, cancelled)
- `timeout`: Maximum execution time
- `retryPolicy`: How failures are retried
- `dependencies`: Other tasks this depends on
- `result`: Result or error from execution
- `cancellable`: Whether task can be cancelled

Constraints:

- Tasks should be designed for cancellation
- Long-running tasks should support progress reporting
- Task dependencies must be acyclic

Entity: Message

Definition: A unit of data transmitted between processes or threads through a communication channel.

Attributes:

- `messageId`: Unique identifier
- `type`: Message type or schema
- `payload`: Message content
- `sender`: Sending process/thread
- `recipient`: Receiving process/thread (if point-to-point)
- `timestamp`: When message was created
- `correlationId`: For request-response correlation
- `priority`: Message priority
- `ttl`: Time-to-live before expiration

Constraints:

- Messages should be immutable after creation
- Message types should be versioned for compatibility
- Large messages should be chunked or use references

8.3 Relationship Definitions

Table 13: Metamodel Relationship Definitions

Relationship	**Source**	**Target**	**Description**
contains	Process	Thread	Process has this thread
spawns	Process	Process	Process creates child process
communicates-via	Process	Channel	Process uses channel for IPC
uses	Thread	Synchronizer	Thread uses this sync primitive
accesses	Thread	Resource	Thread reads/writes resource
protects	Synchronizer	Resource	Synchronizer guards resource
executes	Thread	Task	Thread runs this task
manages	Pool	Thread	Pool controls thread lifecycle
queues	Pool	Task	Pool holds pending tasks
carries	Channel	Message	Channel transmits messages

9 Conforming Notations

Several existing notations and modeling approaches align with the Process Viewpoint.

9.1 UML Behavioral Diagrams

UML provides several diagram types suitable for modeling concurrent behavior:

Activity Diagrams: Fork/join nodes for parallelism, swimlanes for process assignment.

Sequence Diagrams: Par/alt combined fragments for concurrent messaging.

State Machine Diagrams: Orthogonal regions for concurrent states.

Conformance Level: Full support for concurrency modeling with appropriate extensions.

9.2 Petri Nets

Petri nets provide formal modeling of concurrent systems with mathematical analysis capabilities.

Elements: Places (states), transitions (events), tokens (resources/control).

Analysis: Reachability, boundedness, liveness, deadlock detection.

Conformance Level: Strong for formal analysis, less intuitive for communication.

9.3 CSP (Communicating Sequential Processes)

CSP provides algebraic notation for describing concurrent process interaction.

Operators: Sequential composition, parallel composition, choice, interleaving.

Tools: FDR model checker for refinement checking.

Conformance Level: Excellent for formal specification and verification.

9.4 Process Algebra Comparison

Table 14: Process Algebra and Formal Method Comparison

Approach	Strengths	Tool Support	Best For
CSP	Communication focus	FDR, PAT	Message-passing systems
CCS	Bisimulation theory	CWB, mCRL2	Mobile/dynamic systems
Petri Nets	Visual, analyzable	PIPE, CPN Tools	Resource/workflow modeling
TLA+	State-based, practical	TLC model checker	Distributed algorithms
Promela/SPIN	Model checking	SPIN	Protocol verification

10 Model Correspondence Rules

Model correspondence rules define how elements in process models relate to elements in other architectural views.

10.1 Deployment View Correspondence

Correspondence Rule CR-01: Process to Node Mapping

Rule: Every process must be allocated to at least one deployment node.
Formal Expression:

$$\forall p \in Processes : \exists N \subseteq Nodes : executes\_on(p, N)$$

Rationale: Ensures all processes have defined execution location.
Verification: Deployment manifest review.

Correspondence Rule CR-02: Thread Pool to Resources

Rule: Thread pool sizing must be compatible with deployment node resources.

Formal Expression:

$$\forall pool \in ThreadPools, node \in Nodes : pool.maxSize \leq node.cores \times factor$$

Rationale: Prevents over-subscription of CPU resources.

Verification: Capacity analysis.

10.2 Development View Correspondence

Correspondence Rule CR-03: Thread to Module Mapping

Rule: Thread implementations must trace to code modules.

Formal Expression:

$$\forall t \in Threads : \exists m \in Modules : implements(m, t)$$

Rationale: Ensures thread designs are implemented in code.

Verification: Code review, traceability matrix.

10.3 Component-and-Connector View Correspondence

Correspondence Rule CR-04: Channel to Connector Mapping

Rule: Inter-process channels must correspond to C&C connectors.

Formal Expression:

$$\forall ch \in Channels_{IPC} : \exists conn \in Connectors : realizes(conn, ch)$$

Rationale: Ensures IPC mechanisms are architecturally visible.

Verification: Architecture diagram comparison.

11 Operations on Views

This section defines methods for creating, interpreting, analyzing, and implementing process views.

11.1 Creation Methods

11.1.1 View Development Process

Step 1: Identify Execution Requirements

1. Gather performance and scalability requirements
2. Identify real-time or latency constraints
3. Determine throughput requirements
4. Assess resource constraints (CPU, memory)
5. Review reliability and availability requirements

Step 2: Define Process Structure

1. Identify major system processes
2. Determine process boundaries based on isolation needs
3. Define process lifecycle (long-running, transient, scheduled)
4. Specify process instances and scaling strategy
5. Document process dependencies and startup order

Step 3: Design Thread Architecture

1. Choose threading model (thread-per-request, pool, event loop)
2. Size thread pools based on workload analysis
3. Identify CPU-bound vs I/O-bound workloads
4. Define thread types and responsibilities
5. Plan thread-local storage needs

Step 4: Identify Shared Resources

1. Enumerate shared mutable state
2. Classify access patterns (read-heavy, write-heavy, balanced)
3. Assess contention levels
4. Identify immutable vs mutable data
5. Consider partitioning strategies

Step 5: Design Synchronization Strategy

1. Select appropriate synchronization primitives
2. Define lock ordering to prevent deadlock
3. Minimize lock scope and duration
4. Consider lock-free alternatives for hot paths
5. Document critical sections

Step 6: Define Communication Patterns

1. Choose IPC mechanisms based on requirements
2. Define message formats and protocols
3. Specify synchronous vs asynchronous communication
4. Plan for communication failures
5. Document channel capacities and backpressure

Step 7: Validate and Analyze

1. Review for deadlock potential
2. Analyze for race conditions
3. Verify resource bounds
4. Test under load conditions
5. Document known concurrency hazards

11.1.2 Common Concurrency Patterns

Pattern: Producer-Consumer

Context: Decouple data production from consumption with different rates.

Solution: Use bounded queue between producer and consumer threads.

Elements:

- Producer thread(s) add items to queue
- Consumer thread(s) remove items from queue
- Queue provides buffering and synchronization
- Backpressure when queue is full

Use When: Rate mismatch between stages, decoupling needed.

Pattern: Worker Pool

Context: Process many independent tasks efficiently.

Solution: Fixed pool of worker threads processing task queue.

Elements:

- Task queue holds pending work
- Worker threads pull tasks from queue
- Pool manager handles thread lifecycle
- Rejection policy for queue overflow

Use When: Many independent tasks, bounded resource usage needed.

Pattern: Read-Write Lock

Context: Resource with many readers, few writers.

Solution: Allow concurrent reads, exclusive writes.

Elements:

- Multiple readers can hold read lock simultaneously
- Writer requires exclusive access (no readers or writers)
- Typically writer preference to prevent starvation

Use When: Read-heavy workloads, shared data structures.

Pattern: Actor Model

Context: Avoid shared mutable state entirely.

Solution: Isolated actors communicate only via messages.

Elements:

- Actors encapsulate state and behavior
- Communication via asynchronous messages
- Each actor processes one message at a time
- Supervision hierarchies for fault tolerance

Use When: Complex concurrency, fault tolerance critical.

Pattern: Future/Promise

Context: Asynchronous operations with eventual results.

Solution: Placeholder for result that will be available later.

Elements:

- Future represents pending computation result
- Promise allows setting the result
- Callbacks or blocking wait for completion
- Composition of multiple futures

Use When: Async operations, non-blocking code, composition.

Table 15: Concurrency Patterns Summary

Pattern	Description	Use When
Producer-Consumer	Queue between producer/consumer	Rate decoupling, buffering
Worker Pool	Fixed workers process task queue	Many independent tasks
Read-Write Lock	Concurrent reads, exclusive writes	Read-heavy workloads
Actor Model	Isolated actors, message passing	No shared state desired
Future/Promise	Placeholder for async result	Async composition
Barrier	Synchronization point	Phased computation
Pipeline	Stages process in sequence	Stream processing
Fork-Join	Recursive divide and conquer	Parallel algorithms
Double-Checked Locking	Lazy init optimization	Singleton, lazy loading
Thread-Local Storage	Per-thread state	Avoid sharing, context

11.2 Analysis Methods

11.2.1 Deadlock Analysis

Deadlock Detection Checklist

Four Necessary Conditions for Deadlock:

1. **Mutual Exclusion:** Resources held exclusively
2. **Hold and Wait:** Holding resources while waiting for more
3. **No Preemption:** Resources cannot be forcibly taken
4. **Circular Wait:** Circular chain of waiting

Prevention Strategies:

- Establish and enforce lock ordering
- Use timeout-based lock acquisition
- Acquire all locks atomically
- Use lock-free data structures where possible

11.2.2 Race Condition Analysis

Race Condition Identification

Check for races when:

- Multiple threads access shared mutable state
- At least one access is a write
- Accesses are not synchronized
- Order of operations affects outcome

Common Race Condition Types:

- Check-then-act (TOCTOU)
- Read-modify-write without atomicity
- Lazy initialization races
- Publication of partially constructed objects

11.2.3 Performance Analysis

Amdahl's Law

Purpose: Calculate theoretical speedup from parallelization.

Formula:

$$Speedup = \frac{1}{(1-P)+\frac{P}{N}}$$

Where:

- P = Proportion of program that can be parallelized
- N = Number of processors
- $(1 - P)$ = Sequential portion (limits speedup)

Implications:

- Even small sequential portions limit scalability
- Focus optimization on sequential bottlenecks
- Beyond certain N, adding processors has diminishing returns

Little's Law

Purpose: Relate throughput, latency, and concurrency.
Formula:

$$L = \lambda \times W$$

Where:

- L = Average number of items in system (concurrency)
- λ = Average arrival rate (throughput)
- W = Average time in system (latency)

Application:

- Size thread pools based on expected concurrency
- Predict queue lengths from arrival rates
- Balance throughput and latency goals

12 Examples

12.1 Example 1: Web Application Process Architecture

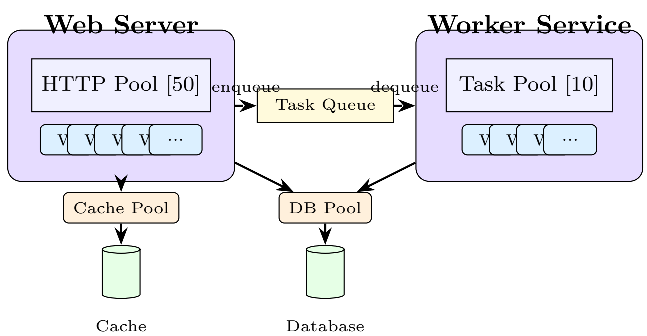


Figure 5: Web Application Process Architecture

Description: This diagram shows a typical web application with a multi-threaded web server process using a thread pool for handling HTTP requests. Background tasks are delegated via a message queue to a separate worker process with its own thread pool. Both processes share access to a database through a connection pool, and the web server uses a cache with its own pool.

12.2 Example 2: Supervision Hierarchy

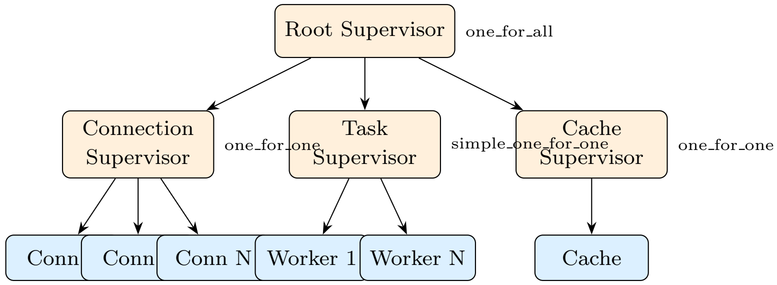


Figure 6: Erlang/OTP-Style Supervision Hierarchy

Description: This supervision tree shows how processes are organized for fault tolerance. The root supervisor manages three child supervisors with different restart strategies. If any component fails, its supervisor can restart it without affecting unrelated parts of the system.

12.3 Example 3: Lock Ordering Diagram

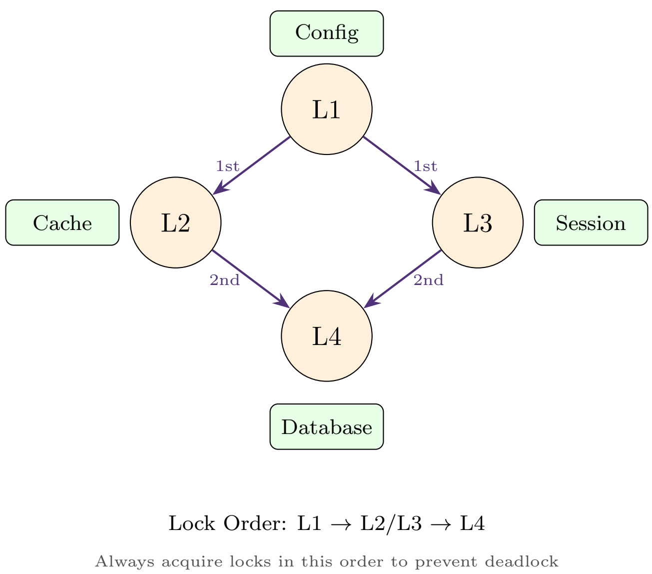


Figure 7: Lock Ordering to Prevent Deadlock

Description: This diagram shows the required lock acquisition order to prevent deadlock. Locks must always be acquired from top to bottom (L1 before L2/L3, L2/L3 before L4). The partial ordering allows L2 and L3 to be acquired in either order since they never need to be held simultaneously.

13 Notes

13.1 Concurrency Hazards

Common Concurrency Bugs

1. **Deadlock:** Circular wait for resources; system hangs
2. **Livelock:** Threads actively changing state but making no progress
3. **Starvation:** Thread never gets resources due to priority/fairness
4. **Race Condition:** Outcome depends on timing of operations
5. **Data Race:** Unsynchronized concurrent access to shared data
6. **Priority Inversion:** High-priority thread blocked by low-priority
7. **Memory Visibility:** Changes not visible to other threads
8. **Atomicity Violation:** Compound operation interrupted

13.2 Thread Pool Sizing Guidelines

Thread Pool Sizing Heuristics

CPU-Bound Tasks:

$$threads = N_{cpu} + 1$$

One extra thread to utilize CPU when a thread is briefly blocked.

I/O-Bound Tasks:

$$threads = N_{cpu} \times \frac{1+W/C}{1}$$

Where W = wait time, C = compute time.

Mixed Workloads:

- Separate pools for CPU-bound and I/O-bound work
- Monitor and adjust based on actual utilization
- Consider using async I/O to reduce thread count

13.3 Common Pitfalls

Common Mistakes to Avoid

1. **Insufficient Synchronization:** Assuming operations are atomic
2. **Over-Synchronization:** Excessive locking reducing concurrency
3. **Holding Locks During I/O:** Blocking other threads during slow ops
4. **Nested Lock Acquisition:** Creating deadlock potential
5. **Ignoring InterruptedException:** Breaking cancellation contracts
6. **Spawning Unbounded Threads:** Exhausting system resources
7. **Shared Mutable Collections:** Using non-thread-safe collections
8. **Double-Checked Locking Bugs:** Incorrect lazy initialization

14 Sources

14.1 Primary References

1. Clements, P., et al. (2010). *Documenting Software Architectures: Views and Beyond* (2nd ed.). Addison-Wesley Professional.
2. Goetz, B., et al. (2006). *Java Concurrency in Practice*. Addison-Wesley Professional.
3. Herlihy, M., & Shavit, N. (2012). *The Art of Multiprocessor Programming* (Revised ed.). Morgan Kaufmann.
4. Lea, D. (1999). *Concurrent Programming in Java* (2nd ed.). Addison-Wesley Professional.
5. Armstrong, J. (2013). *Programming Erlang* (2nd ed.). Pragmatic Bookshelf.

14.2 Supplementary References

6. Ben-Ari, M. (2006). *Principles of Concurrent and Distributed Programming* (2nd ed.). Addison-Wesley.
7. Butcher, P. (2014). *Seven Concurrency Models in Seven Weeks*. Pragmatic Bookshelf.
8. Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.
9. Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools*. Addison-Wesley.
10. Hoare, C.A.R. (1985). *Communicating Sequential Processes*. Prentice Hall.

14.3 Online Resources

- The Little Book of Semaphores: `https://greenteapress.com/wp/semaphores/`
- Java Concurrency Tutorial: `https://docs.oracle.com/javase/tutorial/essential/concurrency/`
- Go Concurrency Patterns: `https://go.dev/blog/pipelines`
- Erlang/OTP Documentation: `https://www.erlang.org/doc/`
- TLA+ Resources: `https://lamport.azurewebsites.net/tla/tla.html`

A Process View Checklist

Item	Complete?
Process Structure	
Major processes identified and documented	☐
Process lifecycle defined	☐
Process dependencies documented	☐
Scaling strategy specified	☐
Resource limits defined	☐
Thread Architecture	
Threading model selected and justified	☐
Thread pools sized appropriately	☐
Thread types documented	☐
Thread-local storage identified	☐
Synchronization	
Shared resources identified	☐
Synchronization mechanisms documented	☐
Lock ordering established	☐
Deadlock prevention verified	☐
Critical sections minimized	☐
Communication	
IPC mechanisms documented	☐
Message formats specified	☐
Channel capacities defined	☐
Error handling documented	☐
Analysis	
Race condition analysis performed	☐
Performance analysis completed	☐
Scalability assessed	☐
Failure modes documented	☐

B Glossary

Atomic Operation
An operation that completes entirely or not at all, with no visible intermediate state.

Barrier A synchronization point where threads wait until all have arrived.

Channel A communication primitive for passing messages between threads or processes.

Critical Section
Code that accesses shared resources and must execute atomically.

Deadlock A state where threads are permanently blocked waiting for each other.

Lock A synchronization primitive providing mutual exclusion.

Monitor An object combining mutual exclusion with condition variables.

Mutex A mutual exclusion lock allowing only one thread access.

Process An operating system execution unit with isolated address space.

Race Condition A bug where outcome depends on timing of operations.

Semaphore A synchronization primitive with a counter for limiting access.

Thread A unit of execution within a process sharing its address space.

Thread Pool A collection of pre-created threads for executing tasks.

Thread Safety Property of code that functions correctly under concurrent access.