CRITICAL INTERVIEW CONCEPTS (Memorize These)

The GIL (Global Interpreter Lock) - PDF 3.1

MOST IMPORTANT: Only ONE thread executes Python bytecode at a time!

- **Threading HELPS:** I/O-bound work (network, disk, sleep)
  - GIL is released during I/O operations
  - Multiple threads can wait for I/O concurrently
- **Threading DOESN'T HELP:** CPU-bound work (math, parsing)
  - Use `multiprocessing` instead (separate processes)
- **For interview:** Assume I/O-bound request processing

Thread Count Decision (PDF 2.2.1)

Rule of Thumb:

- I/O-bound: `os.cpu_count() * 2` (workers wait on I/O)
- CPU-bound: `os.cpu_count()` (avoid context switching)
- **Trade-offs:**
  - More threads = better throughput BUT more memory/overhead
  - Each thread has stack space (8MB on Linux)
  - Context switching overhead increases
- **Interview answer:** "Start with 4-8 for I/O-bound, then profile and tune based on latency/throughput metrics"

Single Queue vs Partitioned Queues (PDF 2.4)

Aspect	Single Queue	Partitioned
Ordering	No guarantee	Per-partition
Load balancing	Excellent	Poor (hot partition)
Complexity	Simple	More complex
Use case	Unordered tasks	User requests

Common Pitfalls (PDF 3.3) - MEMORIZE!

1. **No join():** Threads keep running after main exits
2. **Shared state without locks:** Race conditions, data corruption
3. **Ignoring thread exceptions:** Silent failures (CRITICAL!)
4. **Infinite blocking:** Always use timeouts on `get()`
5. **Deadlocks:** Multiple locks acquired in different orders
6. **Increment race:** `counter += 1` is NOT atomic!

Key Queue Methods (PDF 3.2.3)

```
q = queue.Queue(maxsize=10)  # Bounded queue
q.put(item, block=True, timeout=None)  # Add item
item = q.get(block=True, timeout=None)  # Remove item
q.task_done()  # Signal one task complete
q.join()       # Wait until all tasks done
q.qsize()      # Approximate size (for monitoring)
q.empty()      # Check if empty (not reliable!)
q.full()       # Check if full (not reliable!)
```

Thread Safety Patterns

```
# Pattern 1: Lock for shared state (CRITICAL!)
lock = threading.Lock()
with lock:  # Acquires and releases automatically
    shared_counter += 1  # This operation is NOT atomic!

# Pattern 2: Event for signaling
shutdown_event = threading.Event()
shutdown_event.set()            # Signal all threads
shutdown_event.clear()          # Reset flag
if shutdown_event.is_set():     # Check flag
    break  # Worker exits

# Pattern 3: Queue for communication (already thread-safe!)
queue.Queue()  # No lock needed - built-in synchronization!

# Pattern 4: RLock for reentrant locking
rlock = threading.RLock()
with rlock:
    with rlock:  # Can acquire same lock again (Lock would deadlock!)
        shared_data.update()

# Pattern 5: Semaphore for resource pooling
sem = threading.Semaphore(3)  # Allow 3 concurrent accesses
with sem:
    limited_resource.use()  # Only 3 threads can be here

# Pattern 6: Condition for complex coordination
condition = threading.Condition()
# Producer:
with condition:
    produce_item()
    condition.notify()  # Wake one waiting consumer
# Consumer:
with condition:
    while not item_available():
        condition.wait()  # Release lock and wait for notification
    consume_item()
```

Why Operations Aren't Atomic

```
# counter += 1 is actually THREE operations:
# 1. Read counter value (e.g., 5)
# 2. Add 1 (5 + 1 = 6)
# 3. Write back (counter = 6)
```

COMPLETE PRODUCTION IMPLEMENTATION

```python
"""
INTERVIEW-READY REQUEST EXECUTOR
================================
Time to implement: 30-40 minutes
Lines of code: ~200-250

PDF References:
- Section 2.4: Partitioning strategy
- Section 3.2: Threading fundamentals
- Section 4.1: Complete template

ARCHITECTURE DECISION TREE:
---------------------------
Q: Do requests need ordering?
  YES -> Use partitioned queues (this implementation)
  NO  -> Use single queue (simpler, better load balancing)

Q: How to handle backpressure?
  - Fast-fail (block=False): Producer gets immediate feedback
  - Block with timeout: Producer waits briefly
  - Block forever: BAD! Cascading failures

Q: How many threads?
  - I/O-bound: cpu_count() * 2
  - CPU-bound: use multiprocessing instead!
"""

import queue
import threading
import time
import os
from typing import Any, Dict, List, Optional
from dataclasses import dataclass
from collections import defaultdict

# ============================================================
# DOMAIN MODEL
# ============================================================

@dataclass
class Request:
    """
    Request with partition key for ordering.

    INTERVIEW Q: Why partition_key?
    A: Routes related requests (same user) to same worker
       for sequential processing. Hash determines worker.
    """
    request_id: str
    partition_key: str  # e.g., user_id
    payload: Any
    timestamp: float = None

    def __post_init__(self):
        if self.timestamp is None:
            self.timestamp = time.time()

# ============================================================
# MAIN EXECUTOR
# ============================================================

class PartitionedRequestExecutor:
    """
    Production-ready executor with:
    1. Request partitioning (ordering per key)
    2. Backpressure detection and fast-fail
    3. Graceful shutdown
    4. Comprehensive error handling
    5. Thread-safe metrics
    6. Worker failure detection

    PDF Section 4.1, 2.4
    """

    def __init__(
        self,
        num_workers: int = None,
        max_queue_size_per_worker: int = 100,
        process_func: Optional[callable] = None
    ):
        """
        INTERVIEW Q: Why default num_workers to None?
        A: Calculate optimal thread count based on workload.
           For I/O-bound: cpu_count() * 2

        PDF Section 2.2.1
        """
        if num_workers is None:
            num_workers = os.cpu_count() * 2
            print(f"Auto-detected {num_workers} workers "
                  f"(CPU count: {os.cpu_count()})")

        self.num_workers = num_workers
        self.max_queue_size_per_worker = max_queue_size_per_worker
        self.process_func = process_func or self._default_process

        # One queue per worker (partition)
        # CRITICAL: Bounded queues provide backpressure
        self.queues = [
            queue.Queue(maxsize=max_queue_size_per_worker)
            for _ in range(num_workers)
        ]

        # Worker thread management
        self.workers: List[threading.Thread] = []
        self.shutdown_event = threading.Event()

        # Thread-safe metrics
        # INTERVIEW Q: Why need lock for metrics?
        # A: Dict updates not atomic. Even reads need sync.
        self.metrics_lock = threading.Lock()
        self.metrics = {
            'submitted': 0,
            'processed': 0,
            'failed': 0,
            'rejected': 0,
            'per_worker': defaultdict(lambda: {
                'processed': 0,
                'failed': 0,
                'last_active': None
            })
        }

        # Tracking for correctness testing (PDF 2.2.2)
        self.submitted_ids = set()
        self.processed_ids = set()
        self.failed_ids = set()

        self._start_workers()

    def _start_workers(self):
        """
        Start worker threads.
```

```
        INTERVIEW Q: Daemon vs non-daemon?
        A: daemon=False allows graceful shutdown with join().
           Daemon threads killed abruptly on main exit.

        PDF Section 3.2.1
        """
        for worker_id in range(self.num_workers):
            t = threading.Thread(
                target=self._worker_loop,
                args=(worker_id,),
                name=f"Worker-{worker_id}",
                daemon=False  # For graceful shutdown
            )
            t.start()
            self.workers.append(t)
        print(f"Started {self.num_workers} workers")

    def _worker_loop(self, worker_id: int):
        """
        Main worker loop - CRITICAL PATTERN

        MUST HAVE:
        1. Check shutdown flag in loop
        2. Timeout on get() to check shutdown periodically
        3. Try/except around ALL processing
        4. task_done() in finally block

        INTERVIEW Q: What if exception in worker?
        A: Doesn't propagate to main! Must catch here.

        PDF Section 2.2.4, 3.2.3
        """
        my_queue = self.queues[worker_id]
        print(f"Worker-{worker_id} started")

        while not self.shutdown_event.is_set():
            try:
                # CRITICAL: Use timeout to check shutdown
                # NEVER: my_queue.get() (blocks forever!)
                request = my_queue.get(timeout=0.5)

                try:
                    # Process request
                    self.process_func(worker_id, request)

                    # Success metrics (thread-safe)
                    with self.metrics_lock:
                        self.metrics['processed'] += 1
                        self.metrics['per_worker'][worker_id]['processed'] += 1
                        self.metrics['per_worker'][worker_id]['last_active'] =
                            time.time()
                        self.processed_ids.add(request.request_id)

                except Exception as e:
                    # CRITICAL: Catch ALL exceptions
                    # Don't let one bad request kill worker!
                    print(f"[Worker-{worker_id}] ERROR: "
                          f"request {request.request_id}: {e}")

                    with self.metrics_lock:
                        self.metrics['failed'] += 1
                        self.metrics['per_worker'][worker_id]['failed'] += 1
                        self.failed_ids.add(request.request_id)

                finally:
                    # ALWAYS mark task done (even on error!)
                    # Required for queue.join() to work
                    my_queue.task_done()

            except queue.Empty:
                # No work, loop back to check shutdown
                continue

        print(f"Worker-{worker_id} shutdown complete")

    def _default_process(self, worker_id: int, request: Request):
        """Default processing (override with process_func)"""
        print(f"[Worker-{worker_id}] Processing "
              f"{request.request_id} (key: {request.partition_key})")
        time.sleep(0.1)  # Simulate I/O

    def submit(self, request: Request, block: bool = False) -> bool:
        """
        Submit request with fast-fail backpressure.

        INTERVIEW Q: Why block=False?
        A: Fast-fail gives producer control to handle rejection
           (retry, return 503, etc.). Blocking causes pile-up.

        INTERVIEW Q: How does partitioning work?
        A: hash(partition_key) % num_workers
           Same key -> same hash -> same worker (ordering!)

        PDF Section 2.4.3 (Consistent Hashing)
        """
        if self.shutdown_event.is_set():
            raise RuntimeError("Executor shutting down")

        # PARTITIONING: Hash-based routing
        # CRITICAL: Same key always goes to same worker
        worker_id = hash(request.partition_key) % self.num_workers
        target_queue = self.queues[worker_id]

        try:
            target_queue.put(request, block=block)

            with self.metrics_lock:
                self.metrics['submitted'] += 1
                self.submitted_ids.add(request.request_id)

            return True

        except queue.Full:
            # BACKPRESSURE: Reject when queue full
            print(f"[BACKPRESSURE] Rejected {request.request_id} "
                  f"(Worker-{worker_id} queue full)")

            with self.metrics_lock:
                self.metrics['rejected'] += 1

            return False

    def get_queue_depths(self) -> Dict[int, int]:
        """Monitor queue depth for backpressure detection"""
        return {i: self.queues[i].qsize()
                for i in range(self.num_workers)}

    def is_backpressure(self, threshold: float = 0.8) -> bool:
        """
        Detect backpressure condition.

        INTERVIEW Q: How detect backpressure?
        A: Monitor queue depth. If > 80% full, system overwhelmed.

        INTERVIEW Q: Why check ANY worker, not average?
        A: Hot partition can bottleneck entire system.
```

```
        PDF Section 2.2.3
        """
        for worker_id, depth in self.get_queue_depths().items():
            if depth > (self.max_queue_size_per_worker * threshold):
                return True
        return False

    def detect_dead_workers(self, timeout: float = 5.0) -> List[int]:
        """
        Detect workers that haven't processed in timeout seconds.

        INTERVIEW Q: How detect dead workers?
        A: Track last_active timestamp. If stale, worker may be stuck.

        PDF Section 2.2.4
        """
        dead = []
        now = time.time()

        with self.metrics_lock:
            for worker_id in range(self.num_workers):
                last = self.metrics['per_worker'][worker_id]['last_active']
                if last and (now - last) > timeout:
                    dead.append(worker_id)

        return dead

    def verify_correctness(self) -> Dict[str, Any]:
        """
        Verify no dropped/duplicate requests.

        INTERVIEW Q: How test correctness?
        A: Invariant: submitted = processed + failed + rejected

        PDF Section 2.2.2
        """
        with self.metrics_lock:
            submitted = self.metrics['submitted']
            processed = self.metrics['processed']
            failed = self.metrics['failed']
            rejected = self.metrics['rejected']

            accounted_for = processed + failed + rejected

            return {
                'submitted': submitted,
                'accounted_for': accounted_for,
                'match': submitted == accounted_for,
                'dropped': submitted - accounted_for,
                'duplicate_processed': len(self.processed_ids) != processed,
                'duplicate_failed': len(self.failed_ids) != failed
            }

    def get_metrics(self) -> Dict[str, Any]:
        """Get all metrics (thread-safe)"""
        with self.metrics_lock:
            return {
                **self.metrics,
                'queue_depths': self.get_queue_depths(),
                'backpressure': self.is_backpressure(),
                'correctness': self.verify_correctness()
            }

    def shutdown(self, wait: bool = True, timeout: float = None):
        """
        Graceful shutdown sequence.

        SHUTDOWN STEPS:
        1. Stop accepting new requests
        2. Wait for queues to drain (if wait=True)
        3. Signal workers to stop (Event.set())
        4. Wait for workers to exit (join with timeout)

        INTERVIEW Q: Why Event instead of boolean?
        A: Event is thread-safe. Boolean would need lock.
           Event.set() and is_set() are atomic.

        PDF Section 3.2.4
        """
        print("\n" + "="*50)
        print("SHUTDOWN INITIATED")
        print("="*50)

        if wait:
            print("Draining queues...")
            for i, q in enumerate(self.queues):
                q.join()  # Wait for this queue to empty
            print("All queues drained")

        # Signal all workers
        self.shutdown_event.set()

        # Wait for workers
        print("Waiting for workers...")
        for worker in self.workers:
            worker.join(timeout=timeout)

        print("="*50)
        print("SHUTDOWN COMPLETE")
        print("="*50)
        print("\nFinal Metrics:")
        for key, val in self.get_metrics().items():
            print(f"  {key}: {val}")

# ============================================================
# ALTERNATIVE: SINGLE QUEUE IMPLEMENTATION
# ============================================================

class SingleQueueExecutor:
    """
    Simpler alternative: Single shared queue.

    TRADE-OFFS vs Partitioned:
    + Better load balancing (no hot partition problem)
    + Simpler implementation
    - No ordering guarantee
    - All workers contend on one lock

    USE WHEN: Tasks are independent, no ordering needed

    PDF Section 2.4
    """

    def __init__(self, num_workers: int = None, max_queue_size: int = 1000):
        if num_workers is None:
            num_workers = os.cpu_count() * 2

        self.num_workers = num_workers
        self.queue = queue.Queue(maxsize=max_queue_size)  # Single queue!
        self.workers = []
        self.shutdown_event = threading.Event()

        self._start_workers()

    def _start_workers(self):
        for i in range(self.num_workers):
            t = threading.Thread(target=self._worker_loop, args=(i,))
            t.start()
```

```
            self.workers.append(t)

    def _worker_loop(self, worker_id: int):
        while not self.shutdown_event.is_set():
            try:
                # All workers pull from SAME queue
                # Whoever is free gets next task (good load balancing!)
                request = self.queue.get(timeout=0.5)
                self._process(worker_id, request)
                self.queue.task_done()
            except queue.Empty:
                continue

    def _process(self, worker_id: int, request: Request):
        print(f"Worker-{worker_id}: {request.request_id}")
        time.sleep(0.1)

    def submit(self, request: Request, block: bool = False) -> bool:
        try:
            self.queue.put(request, block=block)
            return True
        except queue.Full:
            return False

    def shutdown(self):
        self.queue.join()
        self.shutdown_event.set()
        for w in self.workers:
            w.join()

# ============================================================
# TESTING & USAGE EXAMPLES
# ============================================================

def test_correctness():
    """
    Test that no requests are dropped or duplicated.

    INTERVIEW Q: How do we test this?
    A: Submit N requests, verify N processed/failed/rejected.

    PDF Section 2.2.2
    """
    print("\n" + "="*50)
    print("CORRECTNESS TEST")
    print("="*50)

    executor = PartitionedRequestExecutor(
        num_workers=3,
        max_queue_size_per_worker=5
    )

    # Submit 20 requests
    num_requests = 20
    for i in range(num_requests):
        req = Request(
            request_id=f"req-{i}",
            partition_key=f"user-{i % 3}",
            payload={}
        )
        executor.submit(req, block=False)

    # Wait and check
    executor.shutdown(wait=True)

    correctness = executor.verify_correctness()
    print("\nCorrectness Check:")
    print(f"  Submitted: {correctness['submitted']}")
    print(f"  Accounted: {correctness['accounted_for']}")
    print(f"  Match: {correctness['match']}")

    if correctness['match']:
        print("  PASS: No dropped requests!")
    else:
        print(f"  FAIL: {correctness['dropped']} requests dropped!")


def test_backpressure():
    """
    Test backpressure handling.

    INTERVIEW Q: What happens when overwhelmed?
    A: Bounded queue rejects, producer gets immediate feedback.
    """
    print("\n" + "="*50)
    print("BACKPRESSURE TEST")
    print("="*50)

    executor = PartitionedRequestExecutor(
        num_workers=2,
        max_queue_size_per_worker=3  # Small to trigger backpressure
    )

    # Submit many requests quickly
    rejected = 0
    for i in range(20):
        req = Request(
            request_id=f"req-{i}",
            partition_key="user-1",  # All to same worker!
            payload={}
        )

        if not executor.submit(req, block=False):
            rejected += 1

    print(f"\nRejected due to backpressure: {rejected}")
    executor.shutdown(wait=True)


def test_partitioning():
    """
    Test that same partition_key goes to same worker.

    INTERVIEW Q: How verify partitioning works?
    A: Track which worker processes each key. Same key should
       always go to same worker.
    """
    print("\n" + "="*50)
    print("PARTITIONING TEST")
    print("="*50)

    key_to_worker = {}

    def track_worker(worker_id: int, request: Request):
        key = request.partition_key
        if key in key_to_worker:
            if key_to_worker[key] != worker_id:
                print(f"ERROR: {key} went to multiple workers!")
        else:
            key_to_worker[key] = worker_id

        print(f"Worker-{worker_id}: {request.request_id} "
              f"(key: {key})")
        time.sleep(0.05)

    executor = PartitionedRequestExecutor(
        num_workers=3,
        process_func=track_worker
```

```
    )

    # Submit requests for 3 users
    users = ["alice", "bob", "charlie"]
    for i in range(15):
        req = Request(
            request_id=f"req-{i}",
            partition_key=users[i % 3],
            payload={}
        )
        executor.submit(req)

    executor.shutdown(wait=True)

    print("\nPartitioning Results:")
    for key, worker_id in key_to_worker.items():
        print(f"  {key} -> Worker-{worker_id}")


if __name__ == "__main__":
    # Run all tests
    test_correctness()
    test_backpressure()
    test_partitioning()

    print("\n" + "="*50)
    print("ALL TESTS COMPLETE")
    print("="*50)
```

INTERVIEW WALKTHROUGH (30 MIN CODING)

Step 1: Basic Structure (5 min)

```
import queue, threading, time

class RequestExecutor:
    def __init__(self, num_workers):
        self.queues = [queue.Queue()
                       for _ in range(num_workers)]
        self.workers = []
        self.shutdown = threading.Event()
        self._start_workers()
```

Step 2: Worker Loop (10 min)

```
def _worker_loop(self, worker_id):
    my_queue = self.queues[worker_id]

    while not self.shutdown.is_set():
        try:
            req = my_queue.get(timeout=0.5)
            # Process request
            print(f"Worker-{worker_id}: {req}")
            my_queue.task_done()
        except queue.Empty:
            continue
```

Step 3: Submit with Partitioning (5 min)

```
def submit(self, request, block=False):
    worker_id = hash(request.key) % len(self.queues)
    try:
        self.queues[worker_id].put(request, block=block)
        return True
    except queue.Full:
        return False  # Backpressure
```

Step 4: Shutdown (5 min)

```
def shutdown(self, wait=True):
    if wait:
        for q in self.queues:
            q.join()

    self.shutdown.set()
    for w in self.workers:
        w.join()
```

Step 5: Add Metrics (5 min)

```
def __init__(self, ...):
    # ... existing code ...
    self.lock = threading.Lock()
    self.processed = 0

def _worker_loop(self, worker_id):
    # ... after processing ...
```

```
    with self.lock:
        self.processed += 1
```

FOLLOW-UP QUESTIONS - PREPARE ANSWERS

Q1: Thread Count (PDF 2.2.1)

Q: How many threads should we use?
A: Start with `cpu_count() * 2` for I/O-bound. Trade-offs:

- More threads = higher throughput but more memory
- Each thread 8MB stack space
- Too many = context switching overhead
- Profile latency p50/p99 and tune

Q2: Testing Correctness (PDF 2.2.2)

Q: How ensure no dropped/duplicate requests?
A: Track invariants:

```
submitted_count == processed + failed + rejected
len(processed_ids) == processed_count  # No duplicates
```

Q3: Worker Failures (PDF 2.2.4)

Q: What if a worker thread dies?
A:

- Try/except in worker loop catches exceptions
- Track last_active timestamp per worker
- Monitor: if stale > 5s, worker may be stuck
- Recovery: restart worker thread or requeue task

Q4: Backpressure (PDF 2.2.3, 2.3.2)

Q: When signal backpressure to upstream?
A: Signal when:

- Queue depth > 80% capacity
- Request latency p99 > threshold (e.g., 5s)
- Worker saturation > 90%
- Monitor over time window (avoid false alarms)

Q5: Hot Partition (PDF 2.4.4)

Q: What about hot partition problem?
A:

- Trade-off: Ordering vs load balancing
- One user sends 1000 req = 1 worker overloaded
- Solutions:
  - Sub-partitioning (split hot keys)
  - Dynamic rebalancing (complex!)
  - Monitor per-worker queue depth
- Accept limitation for most use cases

Q6: Single Queue Alternative

Q: Why not use single queue?
A: Trade-offs:

- Single queue: Better load balancing, no ordering
- Partitioned: Ordering guarantee, hot partition issue
- Choice depends on requirements

KEY TALKING POINTS (MEMORIZE)

Why Threading for This Problem?

"This is an I/O-bound workload - processing requests involves network calls, database queries, or external API calls. During I/O operations, Python releases the GIL, allowing other threads to execute. This means we can handle multiple requests concurrently even with the GIL."

Why Partitioned Queues?

"We need to guarantee that requests from the same user are processed in order - for example, login before update profile. By hashing the user_id and routing to the same worker, we ensure sequential processing per user while still processing different users in parallel."

Why Fast-Fail Backpressure?

"When a queue is full, blocking would cause requests to pile up at the producer, potentially exhausting resources. Fast-fail with block=False gives the producer immediate feedback to handle it appropriately - retry with exponential backoff, return HTTP 503, or route to another instance."

Why Bounded Queues?

"Unbounded queues can lead to out-of-memory errors if requests arrive faster than we can process them. Bounded queues provide a natural backpressure mechanism - when full, we reject and let upstream handle it."

QUICK REFERENCE CARD

Essential Threading APIs

```python
# Thread creation
t = threading.Thread(target=func, args=(a,))
t.start()  # Begin execution
t.join()   # Wait for completion

# Synchronization
lock = threading.Lock()
with lock:  # Acquire/release
    shared_state += 1

event = threading.Event()
event.set()        # Signal
event.is_set()     # Check

# Queue operations
q = queue.Queue(maxsize=10)
q.put(item, block=False)    # Add
item = q.get(timeout=1)     # Remove
q.task_done()               # Mark complete
q.join()                    # Wait all done
```

Common Mistakes to Avoid

1. Forgetting `task_done()` after `get()`
2. Using `get()` without timeout (blocks forever!)
3. Shared state without locks (race conditions)
4. Assuming exceptions propagate (they don't!)
5. Using daemon=True when need graceful shutdown
6. Not checking `shutdown_event` in worker loop

Interview Time Management

- 0-5 min: Clarify requirements, draw architecture
- 5-20 min: Core implementation (basic executor)
- 20-30 min: Add partitioning + backpressure
- 30-40 min: Add error handling + metrics
- 40-50 min: Testing + discussion
- 50-60 min: Follow-up questions