Augment Code Interview Preparation

Network Ports & Process Management

Alex Yang

Prepared: November 26, 2024

Contents

1 Interview Overview

1.1 Problem Type

- **Category:** Systems Programming / Process Management
- **Difficulty:** Medium-Hard
- **Duration:** 50 minutes
- **Company:** Augment Code (AI Coding Startup)

1.2 Interview Structure

1. **Part 1 (20-25 min):** Implement network port finding function
    - Uses recursion, hash set, array iteration
    - Process hierarchy traversal
    - Performance optimization discussion
2. **Part 2 (15-20 min):** Implement garbage collection logic
    - Similar to moving 1s in binary array
    - In-place swap approach
    - Array manipulation efficiency
3. **Part 3 (5-10 min):** Thread safety discussion
    - Read-Write (RW) locks
    - Concurrency techniques
    - Design discussion (no coding)

Critical Point

Key Interview Signals Based on Past Feedback:
What they look for:

- Quick recognition of design issues (recursion, flat-list pitfalls)
- Understanding of space vs. time complexity tradeoffs
- Clean, correct implementations (not sloppy)
- Strong communication about design decisions
- Knowledge of concurrency primitives

Common pitfalls from past candidates:

- Missing that `get_children()` returns PIDs, not full records
- Spending too long trying to avoid copying the database
- Not knowing `del` from list is $O(n)$ operation
- Sloppy implementations with edge case bugs
- Weak concurrency knowledge (thinking single lock suffices)

2 Part 1: Network Ports Problem

2.1 Problem Statement

Context: You have a database of processes where each process can have child processes forming a tree hierarchy. Some processes are listening on network ports. You need to find all ports used by a given process and all its descendants.

2.1.1 Data Structures

Process Record:

```python
class ProcessRecord:
    def __init__(self, pid, parent_pid, ports, is_alive):
        self.pid = pid              # Process ID (unique)
        self.parent_pid = parent_pid # Parent PID (None for root)
        self.ports = ports          # List of port numbers [80, 443]
        self.is_alive = is_alive    # Boolean: True/False
```

Database Interface:

```python
class ProcessDatabase:
    def __init__(self, processes: List[ProcessRecord]):
        """Store processes in flat list (database table)"""
        self.processes = processes

    def get_process(self, pid: int) -> ProcessRecord:
        """Get process by PID - O(n) lookup"""
        for proc in self.processes:
            if proc.pid == pid:
                return proc
        return None

    def get_children(self, pid: int) -> List[int]:
        """Get child PIDs (not full records!) - O(n) scan"""
        children_pids = []
        for proc in self.processes:
            if proc.parent_pid == pid:
                children_pids.append(proc.pid)
        return children_pids
```

2.2 Requirements

Implement:

```python
def find_all_ports(database: ProcessDatabase, root_pid: int) -> List[int]:
    """
    Find all network ports used by root_pid and all its descendants.

    Args:
        database: ProcessDatabase instance
        root_pid: Starting process ID

    Returns:
        List of unique port numbers (no duplicates)

    Notes:
        - Only include ALIVE processes
```

```
        - Traverse entire subtree recursively
        - Handle dead processes gracefully
    """
    pass
```

Example:

```
Process Tree:
    1 (ports=[80], alive=True)
    +-- 2 (ports=[443], alive=True)
    |   +-- 4 (ports=[8080], alive=False)  # Dead - ignore
    +-- 3 (ports=[3000, 3001], alive=True)
        +-- 5 (ports=[5000], alive=True)

find_all_ports(db, 1) -> [80, 443, 3000, 3001, 5000]
find_all_ports(db, 2) -> [443]  # Ignores dead child 4
find_all_ports(db, 3) -> [3000, 3001, 5000]
```

2.3 Solution Approach

Key Insight

Key Design Decisions:

1. **Recursion:** Natural fit for tree traversal
2. **Hash set:** Track unique ports (automatic deduplication)
3. **Filter dead processes:** Check `is_alive` before processing
4. **Understand API:** `get_children()` returns PIDs, need `get_process()` to get records

2.4 Implementation - Version 1 (Recursive DFS)

```python
def find_all_ports(database: ProcessDatabase, root_pid: int) -> List[int]:
    """
    Find all ports using recursive DFS with hash set for deduplication.

    Time: O(n) where n = number of alive processes in subtree
    Space: O(h + p) where h = height, p = unique ports
    """
    # Use set for automatic deduplication
    ports_set = set()

    def dfs(pid: int) -> None:
        """Recursive helper to traverse process tree"""
        # Get process record
        process = database.get_process(pid)

        # Handle missing or dead process
        if process is None or not process.is_alive:
            return

        # Add this process's ports
```

```
        for port in process.ports:
            ports_set.add(port)

        # Recursively process children
        # CRITICAL: get_children returns PIDs, not ProcessRecords!
        child_pids = database.get_children(pid)
        for child_pid in child_pids:
            dfs(child_pid)

    # Start DFS from root
    dfs(root_pid)

    # Convert set to list
    return list(ports_set)
```

Critical Point

Common Mistake - Past Candidate Error:
Many candidates miss that `get_children(pid)` returns **PIDs** (integers), not `ProcessRecord` objects!

Wrong:

```
children = database.get_children(pid)
for child in children:
    for port in child.ports:  # ERROR: child is int, not ProcessRecord!
        ports_set.add(port)
```

Correct:

```
child_pids = database.get_children(pid)
for child_pid in child_pids:
    child_process = database.get_process(child_pid)
    if child_process and child_process.is_alive:
        # Now we have the actual ProcessRecord
```

2.5 Complexity Analysis

Time Complexity: $O(n \times m)$ where:

- n = number of alive processes in subtree
- m = average number of children per process
- Each `get_children()` scans entire database ($O(m)$)
- Total: n calls $\times$ $O(m)$ per call = $O(n \times m)$

Space Complexity: $O(h + p)$ where:

- h = tree height (recursion stack)
- p = number of unique ports

2.6 Optimization Discussion

Key Insight

Performance Bottleneck:
The `get_children()` method scans the entire database on every call. For large databases, this becomes expensive.
Interviewer might ask: "Can you optimize this?"
Key insight from feedback: One candidate proposed "filtering out dead processes" early but communicated it oddly. The interviewer wants to hear:
Two-pass optimization:

1. **Pass 1:** Build parent→children map once ($O(n)$)
2. **Pass 2:** DFS using map for instant child lookup ($O(1)$)

This reduces time from $O(n \times m)$ to $O(n).$

2.7 Implementation - Version 2 (Optimized)

```python
def find_all_ports_optimized(database: ProcessDatabase, root_pid: int) -> List[int]:
    """
    Optimized version: Build parent->children map first.

    Time: O(n) total - single pass to build map + DFS
    Space: O(n + p) - map storage + ports set
    """
    # Pass 1: Build parent->children mapping (O(n))
    parent_to_children = {}
    pid_to_process = {}

    for process in database.processes:
        pid_to_process[process.pid] = process
        if process.parent_pid is not None:
            if process.parent_pid not in parent_to_children:
                parent_to_children[process.parent_pid] = []
            parent_to_children[process.parent_pid].append(process.pid)

    # Pass 2: DFS using precomputed map
    ports_set = set()

    def dfs(pid: int) -> None:
        # Get process from our map
        process = pid_to_process.get(pid)

        if process is None or not process.is_alive:
            return

        # Add ports
        ports_set.update(process.ports)  # More pythonic than loop

        # Get children from map (O(1) lookup)
        child_pids = parent_to_children.get(pid, [])
        for child_pid in child_pids:
```

```
            dfs(child_pid)

    dfs(root_pid)
    return list(ports_set)


def find_all_ports_space_optimized(database: ProcessDatabase, root_pid: int) ->
    List[int]:
    """
    Space-Optimized Approach: "Traverse + Scan"

    This addresses the interviewer's specific question about Space Complexity.
    Instead of building a full O(N) map of the database, we only store PIDs
    for the relevant subtree.

    Strategy:
    1. Collect target PIDs (recursively) -> Set of PIDs
    2. Scan database once -> Filter against Set

    Time: O(N + M) where N=database size, M=subtree size
    Space: O(M) - Only store PIDs in the subtree (vs O(N) for full map)
    """
    # Pass 1: Collect all PIDs in subtree using get_children
    subtree_pids = set()

    def collect_pids(pid: int) -> None:
        """Recursively collect all PIDs in subtree"""
        subtree_pids.add(pid)
        child_pids = database.get_children(pid)
        for child_pid in child_pids:
            collect_pids(child_pid)

    collect_pids(root_pid)

    # Pass 2: Single scan through database, lookup against PIDs
    ports_set = set()
    for process in database.processes:
        # Check if this process is in our subtree
        if process.pid in subtree_pids and process.is_alive:
            ports_set.update(process.ports)

    return list(ports_set)
```

Action Item

What to communicate in interview:
"The naive approach calls `get_children()` repeatedly, scanning the entire database each time. I see two optimization approaches:"
Approach 1 - Build parent→children map:

- Preprocess database to build lookup structure
- DFS with $O(1)$ child lookups
- Time: $O(m + n)$ where m=database size, n=subtree size

Approach 2 - Collect PIDs then scan (Space Optimization):

- First pass: DFS to collect only relevant PIDs in subtree
- Second pass: Single database scan, filter by PID set
- **Why this matters:** Reduces space from O(Database) to O(Subtree).
- *This matches the "aha" hint: "iterate database and lookup against pids"*

Space vs Time Tradeoff (Critical Discussion):

- **Naive DFS:** Time $O(N \times \text{Children})$, Space $O(Height)$
- **Map Optimization:** Time $O(N)$, Space $O(N)$ (Index entire DB)
- **Traverse + Scan:** Time $O(N)$, Space $O(Subtree)$ (Index only relevant items)
- *Candidate 1 missed this distinction when asked about space!*

2.8 Edge Cases

1. **Root process is dead:** Return empty list
2. **Root has no children:** Return only root's ports
3. **All descendants dead:** Return only root's ports
4. **Duplicate ports in tree:** Set handles deduplication
5. **Process with no ports:** Empty `ports` list is valid
6. **Invalid root PID:** Handle gracefully (return empty)

3 Part 2: Garbage Collection

3.1 Problem Statement

After finding ports, we need to compact the process database by removing dead processes. The interviewer describes this as "similar to moving all 1s in a binary array to the beginning."

Task: Implement garbage collection that:

- Moves all alive processes to the front of the array
- Moves all dead processes to the back
- Maintains relative order of alive processes
- Operates in-place (minimal extra space)

3.2 Array Analogy

Concept Review

Binary Array Problem (Classic Pattern):
Input: `[0, 1, 0, 1, 1, 0, 1, 0]`
Output: `[1, 1, 1, 1, 0, 0, 0, 0]`
Approach: Two-pointer swap

- Left pointer: Next position for a 1
- Right pointer: Current element being examined
- Swap when we find a 1

For processes:

- 1 = alive process
- 0 = dead process
- Goal: Move all alive to front

3.3 Implementation

```python
def garbage_collect(processes: List[ProcessRecord]) -> int:
    """
    Compact process list: move alive processes to front.

    Returns:
        Number of alive processes (boundary index)

    Time: O(n) - single pass
    Space: O(1) - in-place swaps
    """
    # Two-pointer approach
    write_idx = 0  # Next position for alive process

    # Scan through array
```

```
    for read_idx in range(len(processes)):
        if processes[read_idx].is_alive:
            # Found alive process - swap to front
            if read_idx != write_idx:
                processes[write_idx], processes[read_idx] = \
                    processes[read_idx], processes[write_idx]
            write_idx += 1

    # write_idx now points to first dead process
    # Processes[0:write_idx] are alive
    # Processes[write_idx:] are dead
    return write_idx


def garbage_collect_with_truncation(processes: List[ProcessRecord]) -> None:
    """
    Alternative: Truncate dead processes entirely.
    Modifies list in-place by removing dead entries.
    """
    alive_count = garbage_collect(processes)
    # Remove dead processes from end
    del processes[alive_count:]
```

Critical Point

Performance Trap - Past Candidate Error:
One candidate "didn't know that `del` from a list was linear time and didn't effectively internalize that after explanation."
Why this matters:

```
# WRONG: O(n^2) approach
i = 0
while i < len(processes):
    if not processes[i].is_alive:
        del processes[i]  # O(n) operation!
        # Don't increment i
    else:
        i += 1
```

Problem:

- Each `del` operation shifts all subsequent elements
- If half the list is dead: $n/2$ deletions $\times$ $O(n)$ each $= O(n^2)$

Correct approach:

- Use two-pointer swap (shown above) - $O(n)$
- Single `del` at end to truncate - $O(k)$ where k = dead count
- Total: $O(n)$

3.4 Visualization

```
Initial: [A1, D1, A2, D2, A3, D3]  (A=alive, D=dead)
```

```
         w=0  r=0

Step 1: r=0, A1 is alive
        [A1, D1, A2, D2, A3, D3]
         w$\rightarrow$r
        Swap A1 with itself, w++, r++

Step 2: r=1, D1 is dead
        [A1, D1, A2, D2, A3, D3]
             w   r
        Skip, r++

Step 3: r=2, A2 is alive
        [A1, D1, A2, D2, A3, D3]
             w       r
        Swap D1 $\leftrightarrow$ A2
        [A1, A2, D1, D2, A3, D3]
                 w
        w++, r++

Step 4: r=3, D2 is dead
        Skip, r++

Step 5: r=4, A3 is alive
        [A1, A2, D1, D2, A3, D3]
                 w           r
        Swap D1 $\leftrightarrow$ A3
        [A1, A2, A3, D2, D1, D3]
                     w
        w++, r++

Final: [A1, A2, A3, D2, D1, D3]
                     ^boundary (w=3)
       Alive: [0:3], Dead: [3:6]
```

3.5 Alternative: Stable Partition

```python
def garbage_collect_stable(processes: List[ProcessRecord]) -> int:
    """
    Alternative using Python's stable partition.
    Maintains relative order more explicitly.
    """
    alive = [p for p in processes if p.is_alive]
    dead = [p for p in processes if not p.is_alive]

    # Rebuild list
    processes.clear()
    processes.extend(alive)
    processes.extend(dead)

```

```
14      return len(alive)
```

Key Insight

Trade-offs:
Two-pointer swap:

- Pros: True in-place ($O(1)$ extra space), fast
- Cons: Slightly more complex logic

List comprehension rebuild:

- Pros: Cleaner, more Pythonic, easier to understand
- Cons: $O(n)$ extra space for temporary lists

In interview: Mention both, explain trade-offs. Candidate who proposed "swap approach" in feedback was on the right track but implementation was "sloppy" with edge cases.

4 Part 3: Thread Safety & Concurrency

4.1 Problem Context

The interviewer asks: "How would you make this system thread-safe if multiple threads are reading/writing the process database simultaneously?"

No coding required - design discussion only.

4.2 Concurrency Concepts

Concept Review

Read-Write Lock (RWLock):
A synchronization primitive that allows:

- **Multiple readers** simultaneously (read-shared)
- **Single writer** exclusively (write-exclusive)
- Writers block all readers and other writers

When to use:

- Read-heavy workloads (common for process databases)
- Reads far outnumber writes
- Want to maximize read concurrency

Python implementation:

```python
import threading

class ProcessDatabase:
    def __init__(self):
        self.processes = []
        self.lock = threading.RLock()  # or RWLock from external lib

    def get_process(self, pid):
        with self.lock:  # Acquire for read
            # Search logic
            pass

    def update_process(self, process):
        with self.lock:  # Acquire for write
            # Update logic
            pass
```

4.3 Design Discussion Points

Action Item

Key points to articulate:

1. Identify Read vs Write Operations:

- **Reads:** `find_all_ports()`, `get_process()`, `get_children()`
- **Writes:** `garbage_collect()`, adding/updating processes

2. RW Lock Strategy:

```
class ThreadSafeProcessDatabase:
    def __init__(self):
        self.processes = []
        self.rw_lock = RWLock()

    def find_all_ports(self, root_pid):
        """Read operation - shared lock"""
        with self.rw_lock.read():
            # Multiple threads can execute concurrently
            return self._find_all_ports_impl(root_pid)

    def garbage_collect(self):
        """Write operation - exclusive lock"""
        with self.rw_lock.write():
            # Only one thread at a time
            # Blocks all readers
            return self._garbage_collect_impl()
```

3. Granularity Considerations:

- **Coarse-grained:** Lock entire database
  - Simple to reason about
  - May limit concurrency
- **Fine-grained:** Lock individual processes
  - Better concurrency
  - Risk of deadlocks
  - More complex

4. Transaction Safety:

```
def safe_update_and_gc(self):
    """Atomic operation: update + GC"""
    with self.rw_lock.write():
        # Both operations happen atomically
        self.update_processes()
        self.garbage_collect()
        # No other thread can read inconsistent state
```

4.4 Common Mistakes

Critical Point

Past candidate error: "Seemed to think he could get the benefit of locking by having a `with thread.Lock()` on `gc()` only."
Why this is wrong:

```
# INSUFFICIENT: Only locks GC, not reads
class BadDatabase:
    def __init__(self):
        self.processes = []
        self.lock = threading.Lock()

    def find_all_ports(self, root_pid):
        # NO LOCK - race condition!
        for proc in self.processes:  # Another thread might be GC'ing!
            ...

    def garbage_collect(self):
        with self.lock:  # Lock only here - not enough!
            # Swapping elements while readers access them = crash
            ...
```

Problem:

- `find_all_ports()` can run while `garbage_collect()` modifies array
- Reader might access invalid indices during swap
- Data races, undefined behavior, crashes

Correct approach:

- ALL reads must acquire read lock
- ALL writes must acquire write lock
- No exceptions

4.5 Advanced Considerations

1. **Copy-on-Write (COW):**
   - Readers use immutable snapshot
   - Writers create new version
   - Atomic pointer swap
   - Good for read-heavy workloads
2. **Lock-Free Data Structures:**
   - Use atomic operations (CAS)
   - Avoid locks entirely
   - Complex to implement correctly

- Better for high-contention scenarios

3. **Thread-Local Caching:**
   - Each thread caches recent lookups
   - Reduce lock contention
   - Trade-off: Stale data risk

Key Insight

For this interview:
Start with RW lock explanation. Only mention COW or lock-free if you have time and interviewer seems interested. Keep it practical and focused on correctness first, optimization second.
Good answer structure:

1. Identify read vs write operations
2. Explain why simple lock isn't optimal (blocks readers)
3. Propose RW lock with clear acquire points
4. Discuss granularity trade-offs
5. Mention testing strategy (race detectors, stress tests)

5 Complete Solution Code

```python
"""
Augment Code Interview - Network Ports Problem
Complete solution with all three parts
"""

import threading
from typing import List, Set, Dict
from dataclasses import dataclass


@dataclass
class ProcessRecord:
    """Process information"""
    pid: int
    parent_pid: int  # None for root
    ports: List[int]
    is_alive: bool


class ProcessDatabase:
    """Process database interface"""

    def __init__(self, processes: List[ProcessRecord]):
        self.processes = processes

    def get_process(self, pid: int) -> ProcessRecord:
        """Get process by PID - O(n) scan"""
        for proc in self.processes:
            if proc.pid == pid:
                return proc
        return None

    def get_children(self, pid: int) -> List[int]:
        """Get child PIDs - O(n) scan"""
        return [proc.pid for proc in self.processes
                if proc.parent_pid == pid]


class ThreadSafeProcessDatabase(ProcessDatabase):
    """Thread-safe version with RW lock"""

    def __init__(self, processes: List[ProcessRecord]):
        super().__init__(processes)
        # In production, use threading.RLock() or external RWLock library
        self.lock = threading.RLock()

    def find_all_ports_safe(self, root_pid: int) -> List[int]:
        """Thread-safe port finding"""
        with self.lock:  # Acquire for read
            return find_all_ports_optimized(self, root_pid)

    def garbage_collect_safe(self) -> int:
        """Thread-safe garbage collection"""
        with self.lock:  # Acquire for write
            return garbage_collect(self.processes)

```

```python

# ============================================================
# Part 1: Find All Ports
# ============================================================

def find_all_ports(database: ProcessDatabase, root_pid: int) -> List[int]:
    """
    Basic recursive solution.

    Time: O(n * m) where n=processes, m=avg children
    Space: O(h + p) where h=height, p=unique ports
    """
    ports_set = set()

    def dfs(pid: int) -> None:
        process = database.get_process(pid)

        if process is None or not process.is_alive:
            return

        # Add this process's ports
        ports_set.update(process.ports)

        # Traverse children (remember: get_children returns PIDs!)
        child_pids = database.get_children(pid)
        for child_pid in child_pids:
            dfs(child_pid)

    dfs(root_pid)
    return list(ports_set)


def find_all_ports_optimized(database: ProcessDatabase,
                             root_pid: int) -> List[int]:
    """
    Optimized with preprocessing.

    Time: O(n) - single pass to build map + DFS
    Space: O(n + p) - map storage + ports set
    """
    # Preprocessing: Build maps
    parent_to_children: Dict[int, List[int]] = {}
    pid_to_process: Dict[int, ProcessRecord] = {}

    for process in database.processes:
        pid_to_process[process.pid] = process
        if process.parent_pid is not None:
            parent_to_children.setdefault(
                process.parent_pid, []
            ).append(process.pid)

    # DFS with O(1) lookups
    ports_set = set()

    def dfs(pid: int) -> None:
        process = pid_to_process.get(pid)

        if process is None or not process.is_alive:
            return
```

```
        ports_set.update(process.ports)

        # O(1) lookup from map
        for child_pid in parent_to_children.get(pid, []):
            dfs(child_pid)

    dfs(root_pid)
    return list(ports_set)


# ============================================================
# Part 2: Garbage Collection
# ============================================================

def garbage_collect(processes: List[ProcessRecord]) -> int:
    """
    Move alive processes to front using two-pointer swap.

    Time: O(n) - single pass
    Space: O(1) - in-place swaps

    Returns:
        Index of first dead process (= count of alive)
    """
    write_idx = 0

    for read_idx in range(len(processes)):
        if processes[read_idx].is_alive:
            # Swap alive process to front
            if read_idx != write_idx:
                processes[write_idx], processes[read_idx] = \
                    processes[read_idx], processes[write_idx]
            write_idx += 1

    return write_idx


def garbage_collect_with_removal(processes: List[ProcessRecord]) -> None:
    """
    Variant: Actually remove dead processes from list.
    """
    alive_count = garbage_collect(processes)
    # Single O(k) deletion at end, not O(n^2) loop
    del processes[alive_count:]


# ============================================================
# Testing
# ============================================================

def test_basic_tree():
    """Test basic process tree traversal"""
    processes = [
        ProcessRecord(1, None, [80], True),
        ProcessRecord(2, 1, [443], True),
        ProcessRecord(3, 1, [3000, 3001], True),
        ProcessRecord(4, 2, [8080], False),  # Dead
        ProcessRecord(5, 3, [5000], True),
```

```
    ]

    db = ProcessDatabase(processes)

    # Test from root
    ports = find_all_ports(db, 1)
    assert set(ports) == {80, 443, 3000, 3001, 5000}

    # Test from subtree
    ports = find_all_ports(db, 2)
    assert set(ports) == {443}  # Ignores dead child

    # Test optimized version
    ports = find_all_ports_optimized(db, 1)
    assert set(ports) == {80, 443, 3000, 3001, 5000}

    print("[PASS] Basic tree test passed")


def test_garbage_collection():
    """Test GC compaction"""
    processes = [
        ProcessRecord(1, None, [80], True),
        ProcessRecord(2, 1, [], False),
        ProcessRecord(3, 1, [443], True),
        ProcessRecord(4, 1, [], False),
        ProcessRecord(5, 3, [8080], True),
    ]

    alive_count = garbage_collect(processes)

    assert alive_count == 3
    # First 3 should be alive
    for i in range(3):
        assert processes[i].is_alive
    # Rest should be dead
    for i in range(3, 5):
        assert not processes[i].is_alive

    print("[PASS] Garbage collection test passed")


def test_edge_cases():
    """Test edge cases"""
    # Dead root
    processes = [ProcessRecord(1, None, [80], False)]
    db = ProcessDatabase(processes)
    ports = find_all_ports(db, 1)
    assert ports == []

    # Empty ports
    processes = [ProcessRecord(1, None, [], True)]
    db = ProcessDatabase(processes)
    ports = find_all_ports(db, 1)
    assert ports == []

    # Single process
    processes = [ProcessRecord(1, None, [80, 443], True)]
    db = ProcessDatabase(processes)
```

```
    ports = find_all_ports(db, 1)
    assert set(ports) == {80, 443}

    print("[PASS] Edge cases test passed")


if __name__ == "__main__":
    test_basic_tree()
    test_garbage_collection()
    test_edge_cases()
    print("\n[SUCCESS] All tests passed!")
```

6 Interview Execution Strategy

6.1 Time Management (50 minutes)

Action Item

Recommended breakdown:

1. **Minutes 0-5:** Understand problem, ask clarifications
   - What does `get_children()` return? (PIDs!)
   - Should we filter dead processes? (Yes)
   - Can ports be duplicated? (Yes - use set)
2. **Minutes 5-10:** Explain approach
   - "I'll use recursive DFS with a hash set"
   - "Key insight: `get_children()` returns PIDs, need `get_process()`"
   - Draw small tree example
3. **Minutes 10-25:** Implement Part 1
   - Write basic recursive solution first
   - Test with simple example
   - Discuss optimization if interviewer asks
4. **Minutes 25-30:** Complexity analysis
   - Time: $O(n \times m)$ basic, $O(n)$ optimized
   - Space: $O(h)$ stack, $O(p)$ ports set
   - Explain trade-offs
5. **Minutes 30-40:** Implement Part 2 (GC)
   - Explain two-pointer approach
   - "Like partitioning 1s and 0s in array"
   - Mention `del` is $O(n)$ pitfall
6. **Minutes 40-50:** Part 3 discussion
   - Identify reads vs writes
   - Explain RW lock benefit
   - Discuss where to acquire locks
   - Mention testing strategy

6.2 Communication Tips

Key Insight

Based on feedback - What interviewers want to hear:
Design recognition (early in interview):

- "I notice this is a tree traversal - recursion makes sense"
- "The flat list means lookups are $O(n)$ - we could optimize with a map"
- "Need to be careful: `get_children()` returns IDs, not objects"

During implementation:

- "I'm using a set here for automatic deduplication"
- "Need to check `is_alive` before processing"
- "This recurses down the tree, collecting ports at each level"

Performance discussion:

- "Current approach calls `get_children()` n times, each scanning the database"
- "Trade-off: We could precompute a map for $O(1)$ lookups using $O(n)$ space"
- "For production, the optimization is worth it"

Don't say (from feedback):

- Vague statements about "filtering" without clear explanation
- "I think this works" without walking through logic
- Proposing approach you don't understand

6.3 What to Write on Whiteboard

1. **Small tree example:**

```
    1 [80]
   / \
  2   3 [443]
 /     \
4(D)    5 [8080]

Output: {80, 443, 8080}
```

2. **Two-pointer GC visualization:**

```
[A, D, A, D, A] $\rightarrow$ [A, A, A, D, D]
```

```
 w  r              w=3
```

3. **Lock hierarchy:**

```
Read operations     Write operations
----------------    ----------------
find_all_ports  $\leftrightarrow$  garbage_collect
get_process     $\leftrightarrow$  update_process
(multiple concurrent) (exclusive)
```

7 Key Concepts Review

7.1 Process Hierarchy

Concept Review

Process Tree Concepts:

- **PID (Process ID):** Unique identifier for each process
- **Parent PID (PPID):** ID of parent process
- **Root process:** Has `parent_pid = None` (e.g., init/systemd)
- **Child processes:** Created via fork/spawn
- **Orphan processes:** Parent died, re-parented to init
- **Zombie processes:** Dead but not yet reaped

Real-world examples:

- Web server (nginx) spawns worker processes
- Each worker inherits parent's listening ports
- `ps auxf` shows tree hierarchy
- `pstree` visualizes process relationships

7.2 Data Structures

Concept Review

Hash Set Properties:

- **Insertion:** $O(1)$ average
- **Lookup:** $O(1)$ average
- **Automatic deduplication:** Adding same element is no-op
- **No ordering:** Iteration order undefined
- **Python:** `set()` built-in type

When to use:

- Need unique elements
- Fast membership testing
- Collecting distinct values

Array vs Hash Set:

- Array: Order preserved, duplicates allowed, $O(n)$ search
- Set: No order, no duplicates, $O(1)$ search

7.3 Recursion Patterns

Concept Review

Tree Recursion Template:

```python
def traverse_tree(node):
    # Base case
    if node is None:
        return

    # Process current node
    process(node)

    # Recursive case: visit children
    for child in node.children:
        traverse_tree(child)
```

Key considerations:

- **Base case:** When to stop recursing (null node, leaf)
- **Stack depth:** Recursion uses call stack - $O(h)$ space
- **Tail recursion:** Can be optimized by compiler (not Python)
- **Alternative:** Iterative with explicit stack

7.4 Concurrency Primitives

Concept Review

Lock Types:

1. **Mutex (Mutual Exclusion):**
   - One thread at a time (read or write)
   - Simple but restrictive
   - Python: `threading.Lock()`
2. **RWLock (Read-Write Lock):**
   - Multiple readers OR single writer
   - Better for read-heavy workloads
   - Python: Need external library (e.g., `readerwriterlock`)
3. **Semaphore:**
   - N threads at a time
   - Rate limiting, connection pools
   - Python: `threading.Semaphore(n)`
4. **Condition Variable:**
   - Wait for specific condition
   - Producer-consumer patterns
   - Python: `threading.Condition()`

Common patterns:

```
# Context manager (automatic release)
with lock:
    # Critical section
    access_shared_data()

# Manual (error-prone)
lock.acquire()
try:
    access_shared_data()
finally:
    lock.release()
```

7.5 Networking Basics

Concept Review

Network Ports:

- **Port number:** 16-bit integer (0-65535)
- **Well-known:** 0-1023 (HTTP=80, HTTPS=443, SSH=22)
- **Registered:** 1024-49151 (Application-specific)
- **Dynamic:** 49152-65535 (Ephemeral)

Process port binding:

- Process calls `bind(port)` to listen
- Only one process per port (usually)
- `SO_REUSEADDR` allows exceptions
- Child processes can inherit listening sockets

Checking ports:

- Linux: `lsof -i :port` or `netstat -tulpn`
- macOS: `lsof -i :port`
- Windows: `netstat -ano`

8 Common Mistakes & How to Avoid

Critical Point

Mistake 1: Misunderstanding `get_children()` return type
Error:

```
children = database.get_children(pid)
for child in children:
    ports.extend(child.ports)  # child is int, not ProcessRecord!
```

Fix:

```
child_pids = database.get_children(pid)
for child_pid in child_pids:
    child = database.get_process(child_pid)
    if child and child.is_alive:
        ports.extend(child.ports)
```

Prevention: Read method signatures carefully!

Critical Point

Mistake 2: Premature optimization
Error: Spending 10+ minutes trying to avoid copying database
Interviewer feedback: "Candidate spent 10min spinning wheels trying linear-time approach avoiding copying"
Better approach:

1. Write correct basic solution first (5 min)
2. Explain optimization opportunity (2 min)
3. Implement optimization if interviewer interested (5 min)

Remember: Correct ¿ Fast

Critical Point

Mistake 3: Using `del` in loop
Error:

```
i = 0
while i < len(processes):
    if not processes[i].is_alive:
        del processes[i]  # O(n) operation in loop!
    else:
        i += 1
# Total: O(n^2)
```

Fix: Use two-pointer approach ($O(n)$) or single deletion at end
When interviewer explains: Acknowledge, internalize, and use knowledge

Critical Point

Mistake 4: Insufficient locking
Error:

```
def garbage_collect(self):
    with self.lock:  # Only lock writes
        # Swap elements

def find_ports(self, pid):
    # NO LOCK - race condition!
    for proc in self.processes:  # Being modified by GC!
        ...
```

Fix: ALL access to shared data must be synchronized
Key principle: Locks protect data, not functions

Critical Point

Mistake 5: Sloppy implementation
Feedback: "Part 2 implementation showed right ideas but needlessly complex and generated edge-case problems"
How to avoid:

- Write pseudocode first
- Test with simple example before finalizing
- Check boundary conditions (empty list, single element)
- Walk through code line-by-line with example

9 Final Preparation Checklist

9.1 Day Before Interview

Action Item

Practice drill (do this 2-3 times):

1. **Part 1 (20 min):**
   - Implement `find_all_ports()` from scratch
   - Write 2-3 test cases
   - Explain optimization
2. **Part 2 (15 min):**
   - Implement `garbage_collect()` with two-pointer
   - Test with example array
   - Explain why `del` in loop is bad
3. **Part 3 (10 min):**
   - Explain RW lock concept
   - Sketch where locks go in code
   - Discuss granularity trade-offs

Total practice time: 45 minutes per iteration

9.2 Interview Day Checklist

Action Item

Before interview:

- [x] Review process tree concepts
- [x] Review hash set properties
- [x] Review recursion patterns
- [x] Review RW lock behavior
- [x] Practice explaining your thinking out loud

During interview:

- [x] Ask clarifying questions upfront
- [x] Explain approach before coding
- [x] Communicate assumptions
- [x] Test code with examples
- [x] Discuss complexity analysis
- [x] Be receptive to hints

Red flags to avoid:

- Silent coding without explanation
- Not testing your code
- Ignoring interviewer hints
- Defensive about mistakes
- Giving up when stuck

9.3 Key Reminders

Key Insight

Success factors from past feedback:
What got candidates hired:

- Quick recognition of recursion and flat-list issues
- Clear communication of design decisions
- Clean, correct implementations
- Strong understanding of complexity trade-offs
- Immediately picking up on swapping approach for GC

What led to rejection:

- Consistently substantial gaps in implementation
- Not understanding performance implications
- Sloppy code with edge case bugs
- Weak concurrency knowledge
- Poor communication about design choices

Bottom line:

- Correctness first, optimization second
- Communicate clearly throughout
- Be receptive to feedback and hints
- Test your code with examples
- Show systems thinking (not just coding)

10 Summary

10.1 Core Algorithms

1. **Part 1 - Find All Ports:**
   - DFS recursion through process tree
   - Hash set for deduplication
   - Filter dead processes
   - Optional: Precompute parent→children map
   - Time: $O(n)$ optimized, Space: $O(n)$
2. **Part 2 - Garbage Collection:**
   - Two-pointer in-place swap
   - Move alive to front, dead to back
   - Single pass, $O(n)$ time, $O(1)$ space
   - Avoid `del` in loop ($O(n^2)$)
3. **Part 3 - Thread Safety:**
   - RW lock for read-heavy workloads
   - Multiple readers, single writer
   - All access must be synchronized
   - Discuss granularity trade-offs

10.2 Interview Mindset

Technical competence:

- Solid understanding of data structures
- Clean implementation skills
- Performance awareness

Communication:

- Explain approach before coding
- Think out loud
- Acknowledge and learn from hints

Problem-solving:

- Ask clarifying questions
- Start with correct solution
- Optimize after basics work

You've got this!