

Augment Code Interview Preparation

Network Ports & Process Management

Alex Yang

Prepared: November 26, 2024 — Updated: December 3, 2024

Contents

1 Interview Overview	3
1.1 Problem Type	3
1.2 Reconstructed Problem (from Qasim, Dec 2024)	4
1.3 Interview Structure	4
2 Part 1: Network Ports Problem	6
2.1 Problem Statement	6
2.1.1 Data Structures	6
2.2 Requirements	7
2.3 Solution Approach	8
2.4 Implementation - Version 1 (Recursive DFS)	8
2.5 Complexity Analysis	9
2.6 Optimization Discussion	10
2.7 Implementation - Version 2 (Optimized)	10
2.8 Edge Cases	12
3 Part 2: Garbage Collection	13
3.1 Problem Statement	13
3.2 Array Analogy	13
3.3 Implementation	14
3.4 Visualization	15
3.5 Alternative: Stable Partition	16
4 Part 3: Thread Safety & Concurrency	17
4.1 Problem Context	17
4.2 Concurrency Concepts	18
4.3 Design Discussion Points	19
4.4 Common Mistakes	20
4.5 Advanced Considerations	20
5 Complete Solution Code	22

6 Interview Execution Strategy	27
6.1 Time Management (50 minutes)	27
6.2 Communication Tips	28
6.3 What to Write on Whiteboard	28
7 Key Concepts Review	30
7.1 Process Hierarchy	30
7.2 Data Structures	31
7.3 Recursion Patterns	32
7.4 Concurrency Primitives	33
7.5 Networking Basics	34
8 Common Mistakes & How to Avoid	35
9 Final Preparation Checklist	37
9.1 Day Before Interview	37
9.2 Interview Day Checklist	38
9.3 Key Reminders	39
10 Summary	40
10.1 Core Algorithms	40
10.2 Interview Mindset	40

✓ Action Item**Update Note (December 3, 2024):**

This document has been enhanced based on detailed clarification from the interviewer (Qasim). The reconstructed question provides more specific details about:

- Exact structure of process records (status field, optional port)
- Precise wording of each part's requirements
- Emphasis on avoiding unnecessary copying in Part 2
- Specific concurrent access scenarios for Part 3

All core concepts and solutions remain valid. Updates focus on alignment with exact problem statement.

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 Reconstructed Problem (from Qasim, Dec 2024)

1 Exact Problem Statement

You are given:

- A list of process records.
- Each record contains:
 - a process ID,
 - a status indicating whether the process is alive or dead, and
 - possibly a network port if the process is alive and using one.
- A function `getChildren(processId)`, which returns the list of child process IDs for a given process.

Part 1: Write a function that, given a root process ID, returns all network ports used by the alive processes in the subtree rooted at that process. This includes the root process (if alive) and all its descendants. You must work from the provided list of processes and the `getChildren` function.

Part 2: Describe and implement an efficient method to remove all dead processes from the same list of process records. This should be done in a way that avoids unnecessary copying and minimizes time complexity.

Part 3: No coding is required. Explain how you would modify your design so that the system is thread-safe and supports concurrent access, for example by multiple threads reading process information or performing garbage collection simultaneously.

1.3 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

Given:

- A list of process records
- Each record contains:
 - A process ID (PID)
 - A status indicating whether the process is alive or dead
 - Possibly a network port if the process is alive and using one
- A function `getChildren(processId)` which returns the list of child process IDs for a given process

Task: Write a function that, given a root process ID, returns all network ports used by the alive processes in the subtree rooted at that process. This includes the root process (if alive) and all its descendants.

2.1.1 Data Structures

Process Record:

```

1 class ProcessRecord:
2     def __init__(self, pid, parent_pid, ports, is_alive):
3         self.pid = pid          # Process ID (unique)
4         self.parent_pid = parent_pid # Parent PID (None for root)
5         self.ports = ports        # List of port numbers [80, 443]
6                                     # Can be empty [] if no ports
7                                     # Note: Problem states "possibly a port"
8         self.is_alive = is_alive  # Boolean: True/False (status)

```

💡 Key Insight

Implementation Note:

The problem states each record has "possibly a network port" (singular, optional). In practice, this can be represented as:

- A list that may be empty: `ports = []` (no ports) or `ports = [80]` (one port)
- A single optional value: `port = None` or `port = 80`

The list representation is more flexible (processes can listen on multiple ports) and the algorithm works the same way. The examples below use lists, but the approach applies to any representation.

Database Interface:

```

1 class ProcessDatabase:
2     def __init__(self, processes: List[ProcessRecord]):
3         """Store processes in flat list (database table)"""
4         self.processes = processes
5

```

```

6     def get_process(self, pid: int) -> ProcessRecord:
7         """Get process by PID - O(n) lookup"""
8         for proc in self.processes:
9             if proc.pid == pid:
10                 return proc
11         return None
12
13    def get_children(self, pid: int) -> List[int]:
14        """Get child PIDs (not full records!) - O(n) scan"""
15        children_pids = []
16        for proc in self.processes:
17            if proc.parent_pid == pid:
18                children_pids.append(proc.pid)
19        return children_pids

```

2.2 Requirements

⚠ Critical Point

Important Constraint:

You **must work from** the provided list of processes and the `getChildren()` function. This means:

- You cannot build your own tree structure initially
- You must use the given API to traverse the hierarchy
- Start by using what's provided, optimize later if needed

Implement:

```

1 def find_all_ports(database: ProcessDatabase, root_pid: int) -> List[int]:
2     """
3         Find all network ports used by root_pid and all its descendants.
4
5     Args:
6         database: ProcessDatabase instance
7         root_pid: Starting process ID
8
9     Returns:
10        List of unique port numbers (no duplicates)
11
12    Notes:
13        - Only include ALIVE processes
14        - Traverse entire subtree recursively
15        - Handle dead processes gracefully
16        - Must work from provided list and getChildren function
17    """
18    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)

```

1 def find_all_ports(database: ProcessDatabase, root_pid: int) -> List[int]:
2     """
3         Find all ports using recursive DFS with hash set for deduplication.
4
5         Time: O(n) where n = number of alive processes in subtree
6         Space: O(h + p) where h = height, p = unique ports
7     """
8
9     # Use set for automatic deduplication
10    ports_set = set()
11
12    def dfs(pid: int) -> None:
13        """Recursive helper to traverse process tree"""
14        # Get process record
15        process = database.get_process(pid)
16
17        # Handle missing or dead process
18        if process is None or not process.is_alive:
19            return
20
21        # Add this process's ports
22        for port in process.ports:
23            ports_set.add(port)
24
25        # Recursively process children
26        # CRITICAL: get_children returns PIDs, not ProcessRecords!
27        child_pids = database.get_children(pid)
28        for child_pid in child_pids:
29            dfs(child_pid)
30
31    # Start DFS from root
32    dfs(root_pid)

```

```
33     # Convert set to list
34     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:

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

Correct:

```
1 child_pids = database.get_children(pid)
2 for child_pid in child_pids:
3     child_process = database.get_process(child_pid)
4     if child_process and child_process.is_alive:
5         # 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)

```

1 def find_all_ports_optimized(database: ProcessDatabase, root_pid: int) -> List[int]
2     """
3         Optimized version: Build parent->children map first.
4
5         Time: O(n) total - single pass to build map + DFS
6         Space: O(n + p) - map storage + ports set
7     """
8
9     # Pass 1: Build parent->children mapping (O(n))
10    parent_to_children = {}
11    pid_to_process = {}
12
13    for process in database.processes:
14        pid_to_process[process.pid] = process
15        if process.parent_pid is not None:
16            if process.parent_pid not in parent_to_children:
17                parent_to_children[process.parent_pid] = []
18                parent_to_children[process.parent_pid].append(process.pid)
19
20    # Pass 2: DFS using precomputed map
21    ports_set = set()
22
23    def dfs(pid: int) -> None:
24        # Get process from our map
25        process = pid_to_process.get(pid)
26
27        if process is None or not process.is_alive:
28            return
29
30        # Add ports
31        ports_set.update(process.ports) # More pythonic than loop
32
33        # Get children from map (O(1) lookup)
34        child_pids = parent_to_children.get(pid, [])
        for child_pid in child_pids:

```

```
35         dfs(child_pid)
36
37     dfs(root_pid)
38     return list(ports_set)
39
40
41 def find_all_ports_space_optimized(database: ProcessDatabase, root_pid: int) ->
42     List[int]:
43     """
44     Space-Optimized Approach: "Traverse + Scan"
45
46     This addresses the interviewer's specific question about Space Complexity.
47     Instead of building a full O(N) map of the database, we only store PIDs
48     for the relevant subtree.
49
50     Strategy:
51     1. Collect target PIDs (recursively) -> Set of PIDs
52     2. Scan database once -> Filter against Set
53
54     Time: O(N + M) where N=database size, M=subtree size
55     Space: O(M) - Only store PIDs in the subtree (vs O(N) for full map)
56     """
57
58     # Pass 1: Collect all PIDs in subtree using get_children
59     subtree_pids = set()
60
61     def collect_pids(pid: int) -> None:
62         """Recursively collect all PIDs in subtree"""
63         subtree_pids.add(pid)
64         child_pids = database.get_children(pid)
65         for child_pid in child_pids:
66             collect_pids(child_pid)
67
68     collect_pids(root_pid)
69
70     # Pass 2: Single scan through database, lookup against PIDs
71     ports_set = set()
72     for process in database.processes:
73         # Check if this process is in our subtree
74         if process.pid in subtree_pids and process.is_alive:
75             ports_set.update(process.ports)
76
77     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 $O(\text{Database})$ to $O(\text{Subtree})$.
- *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 $O(\text{Height})$
- **Map Optimization:** Time $O(N)$, Space $O(N)$ (Index entire DB)
- **Traverse + Scan:** Time $O(N)$, Space $O(\text{Subtree})$ (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

Task: Describe and implement an efficient method to remove all dead processes from the same list of process records.

Key Requirements:

- Remove dead processes from the list
- Avoid unnecessary copying
- Minimize time complexity
- Should be done in a way that's efficient and practical

Interviewer hint: This is similar to "moving all 1s in a binary array to the beginning" - a classic two-pointer partitioning problem.

What this means:

- Move all alive processes to the front of the array
- Move all dead processes to the back (or remove them entirely)
- Maintain relative order of alive processes
- Operate in-place with 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

```

1 def garbage_collect(processes: List[ProcessRecord]) -> int:
2     """
3         Compact process list: move alive processes to front.
4
5         Returns:
6             Number of alive processes (boundary index)
7
8         Time: O(n) - single pass
9         Space: O(1) - in-place swaps
10    """
11
12    # Two-pointer approach
13    write_idx = 0 # Next position for alive process
14
15    # Scan through array
16    for read_idx in range(len(processes)):
17        if processes[read_idx].is_alive:
18            # Found alive process - swap to front
19            if read_idx != write_idx:
20                processes[write_idx], processes[read_idx] = \
21                    processes[read_idx], processes[write_idx]
22            write_idx += 1
23
24    # write_idx now points to first dead process
25    # Processes[0:write_idx] are alive
26    # Processes[write_idx:] are dead
27    return write_idx
28
29 def garbage_collect_with_truncation(processes: List[ProcessRecord]) -> None:
30     """
31         Alternative: Truncate dead processes entirely.
32         Modifies list in-place by removing dead entries.
33     """
34     alive_count = garbage_collect(processes)
35     # Remove dead processes from end
36     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:

```

1 # WRONG: O(n^2) approach
2 i = 0
3 while i < len(processes):
4     if not processes[i].is_alive:
5         del processes[i] # O(n) operation!
6         # Don't increment i
7     else:
8         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)
 $w=0 \quad r=0$

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]
 $w \quad r$
 Skip, $r++$

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

```

1 def garbage_collect_stable(processes: List[ProcessRecord]) -> int:
2     """
3         Alternative using Python's stable partition.
4         Maintains relative order more explicitly.
5     """
6     alive = [p for p in processes if p.is_alive]
7     dead = [p for p in processes if not p.is_alive]
8
9     # Rebuild list
10    processes.clear()
11    processes.extend(alive)
12    processes.extend(dead)
13
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

Task: Explain how you would modify your design so that the system is thread-safe and supports concurrent access.

Scenario: Multiple threads might be:

- Reading process information (calling `find_all_ports()`, `get_process()`)
- Performing garbage collection (removing dead processes)
- Doing both simultaneously

No coding required - design discussion only.

Key question: How do you ensure correctness when multiple threads access the same process list concurrently?

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:

```
1 import threading
2
3 class ProcessDatabase:
4     def __init__(self):
5         self.processes = []
6         self.lock = threading.RLock() # or RWLock from external lib
7
8     def get_process(self, pid):
9         with self.lock: # Acquire for read
10             # Search logic
11             pass
12
13     def update_process(self, process):
14         with self.lock: # Acquire for write
15             # Update logic
16             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:

```

1 class ThreadSafeProcessDatabase:
2     def __init__(self):
3         self.processes = []
4         self.rw_lock = RWLock()
5
6     def find_all_ports(self, root_pid):
7         """Read operation - shared lock"""
8         with self.rw_lock.read():
9             # Multiple threads can execute concurrently
10            return self._find_all_ports_impl(root_pid)
11
12    def garbage_collect(self):
13        """Write operation - exclusive lock"""
14        with self.rw_lock.write():
15            # Only one thread at a time
16            # Blocks all readers
17            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:

```

1 def safe_update_and_gc(self):
2     """Atomic operation: update + GC"""
3     with self.rw_lock.write():
4         # Both operations happen atomically
5         self.update_processes()
6         self.garbage_collect()
7         # 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:

```

1 # INSUFFICIENT: Only locks GC, not reads
2 class BadDatabase:
3     def __init__(self):
4         self.processes = []
5         self.lock = threading.Lock()
6
7     def find_all_ports(self, root_pid):
8         # NO LOCK - race condition!
9         for proc in self.processes: # Another thread might be GC'ing!
10            ...
11
12     def garbage_collect(self):
13         with self.lock: # Lock only here - not enough!
14             # Swapping elements while readers access them = crash
15             ...

```

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

```

1 """
2 Augment Code Interview - Network Ports Problem
3 Complete solution with all three parts
4 """
5
6 import threading
7 from typing import List, Set, Dict
8 from dataclasses import dataclass
9
10
11 @dataclass
12 class ProcessRecord:
13     """Process information"""
14     pid: int
15     parent_pid: int # None for root
16     ports: List[int]
17     is_alive: bool
18
19
20 class ProcessDatabase:
21     """Process database interface"""
22
23     def __init__(self, processes: List[ProcessRecord]):
24         self.processes = processes
25
26     def get_process(self, pid: int) -> ProcessRecord:
27         """Get process by PID - O(n) scan"""
28         for proc in self.processes:
29             if proc.pid == pid:
30                 return proc
31         return None
32
33     def get_children(self, pid: int) -> List[int]:
34         """Get child PIDs - O(n) scan"""
35         return [proc.pid for proc in self.processes
36                 if proc.parent_pid == pid]
37
38
39 class ThreadSafeProcessDatabase(ProcessDatabase):
40     """Thread-safe version with RW lock"""
41
42     def __init__(self, processes: List[ProcessRecord]):
43         super().__init__(processes)
44         # In production, use threading.RLock() or external RLock library
45         self.lock = threading.RLock()
46
47     def find_all_ports_safe(self, root_pid: int) -> List[int]:
48         """Thread-safe port finding"""
49         with self.lock: # Acquire for read
50             return find_all_ports_optimized(self, root_pid)
51
52     def garbage_collect_safe(self) -> int:
53         """Thread-safe garbage collection"""
54         with self.lock: # Acquire for write
55             return garbage_collect(self.processes)
56

```

```

57
58 # =====
59 # Part 1: Find All Ports
60 # =====
61
62 def find_all_ports(database: ProcessDatabase, root_pid: int) -> List[int]:
63     """
64     Basic recursive solution.
65
66     Time: O(n * m) where n=processes, m=avg children
67     Space: O(h + p) where h=height, p=unique ports
68     """
69     ports_set = set()
70
71     def dfs(pid: int) -> None:
72         process = database.get_process(pid)
73
74         if process is None or not process.is_alive:
75             return
76
77         # Add this process's ports
78         ports_set.update(process.ports)
79
80         # Traverse children (remember: get_children returns PIDs!)
81         child_pids = database.get_children(pid)
82         for child_pid in child_pids:
83             dfs(child_pid)
84
85     dfs(root_pid)
86     return list(ports_set)
87
88
89 def find_all_ports_optimized(database: ProcessDatabase,
90                             root_pid: int) -> List[int]:
91     """
92     Optimized with preprocessing.
93
94     Time: O(n) - single pass to build map + DFS
95     Space: O(n + p) - map storage + ports set
96     """
97
98     # Preprocessing: Build maps
99     parent_to_children: Dict[int, List[int]] = {}
100    pid_to_process: Dict[int, ProcessRecord] = {}
101
102    for process in database.processes:
103        pid_to_process[process.pid] = process
104        if process.parent_pid is not None:
105            parent_to_children.setdefault(
106                process.parent_pid, []
107            ).append(process.pid)
108
109    # DFS with O(1) lookups
110    ports_set = set()
111
112    def dfs(pid: int) -> None:
113        process = pid_to_process.get(pid)
114
115        if process is None or not process.is_alive:

```

```
116     ports_set.update(process.ports)
117
118     # O(1) lookup from map
119     for child_pid in parent_to_children.get(pid, []):
120         dfs(child_pid)
121
122     dfs(root_pid)
123     return list(ports_set)
124
125
126
127 # =====
128 # Part 2: Garbage Collection
129 # =====
130
131 def garbage_collect(processes: List[ProcessRecord]) -> int:
132     """
133     Move alive processes to front using two-pointer swap.
134
135     Time: O(n) - single pass
136     Space: O(1) - in-place swaps
137
138     Returns:
139         Index of first dead process (= count of alive)
140     """
141     write_idx = 0
142
143     for read_idx in range(len(processes)):
144         if processes[read_idx].is_alive:
145             # Swap alive process to front
146             if read_idx != write_idx:
147                 processes[write_idx], processes[read_idx] = \
148                     processes[read_idx], processes[write_idx]
149             write_idx += 1
150
151     return write_idx
152
153
154 def garbage_collect_with_removal(processes: List[ProcessRecord]) -> None:
155     """
156     Variant: Actually remove dead processes from list.
157     """
158     alive_count = garbage_collect(processes)
159     # Single O(k) deletion at end, not O(n^2) loop
160     del processes[alive_count:]
161
162
163 # =====
164 # Testing
165 # =====
166
167 def test_basic_tree():
168     """Test basic process tree traversal"""
169     processes = [
170         ProcessRecord(1, None, [80], True),
171         ProcessRecord(2, 1, [443], True),
172         ProcessRecord(3, 1, [3000, 3001], True),
173         ProcessRecord(4, 2, [8080], False), # Dead
174         ProcessRecord(5, 3, [5000], True),
```

```
175     ]
176
177     db = ProcessDatabase(processes)
178
179     # Test from root
180     ports = find_all_ports(db, 1)
181     assert set(ports) == {80, 443, 3000, 3001, 5000}
182
183     # Test from subtree
184     ports = find_all_ports(db, 2)
185     assert set(ports) == {443} # Ignores dead child
186
187     # Test optimized version
188     ports = find_all_ports_optimized(db, 1)
189     assert set(ports) == {80, 443, 3000, 3001, 5000}
190
191     print("[PASS] Basic tree test passed")
192
193
194 def test_garbage_collection():
195     """Test GC compaction"""
196     processes = [
197         ProcessRecord(1, None, [80], True),
198         ProcessRecord(2, 1, [], False),
199         ProcessRecord(3, 1, [443], True),
200         ProcessRecord(4, 1, [], False),
201         ProcessRecord(5, 3, [8080], True),
202     ]
203
204     alive_count = garbage_collect(processes)
205
206     assert alive_count == 3
207     # First 3 should be alive
208     for i in range(3):
209         assert processes[i].is_alive
210     # Rest should be dead
211     for i in range(3, 5):
212         assert not processes[i].is_alive
213
214     print("[PASS] Garbage collection test passed")
215
216
217 def test_edge_cases():
218     """Test edge cases"""
219     # Dead root
220     processes = [ProcessRecord(1, None, [80], False)]
221     db = ProcessDatabase(processes)
222     ports = find_all_ports(db, 1)
223     assert ports == []
224
225     # Empty ports
226     processes = [ProcessRecord(1, None, [], True)]
227     db = ProcessDatabase(processes)
228     ports = find_all_ports(db, 1)
229     assert ports == []
230
231     # Single process
232     processes = [ProcessRecord(1, None, [80, 443], True)]
233     db = ProcessDatabase(processes)
```

```
234     ports = find_all_ports(db, 1)
235     assert set(ports) == {80, 443}
236
237     print("[PASS] Edge cases test passed")
238
239
240 if __name__ == "__main__":
241     test_basic_tree()
242     test_garbage_collection()
243     test_edge_cases()
244     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	\$\rightarrow\$ garbage_collect
get_process	\$\rightarrow\$ 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:

```
1 def traverse_tree(node):
2     # Base case
3     if node is None:
4         return
5
6     # Process current node
7     process(node)
8
9     # Recursive case: visit children
10    for child in node.children:
11        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:

```
1 # Context manager (automatic release)
2 with lock:
3     # Critical section
4     access_shared_data()
5
6 # Manual (error-prone)
7 lock.acquire()
8 try:
9     access_shared_data()
10 finally:
11     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:

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

Fix:

```
1 child_pids = database.get_children(pid)
2 for child_pid in child_pids:
3     child = database.get_process(child_pid)
4     if child and child.is_alive:
5         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:

```
1 i = 0
2 while i < len(processes):
3     if not processes[i].is_alive:
4         del processes[i] # O(n) operation in loop!
5     else:
6         i += 1
7 # 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:

```
1 def garbage_collect(self):
2     with self.lock: # Only lock writes
3         # Swap elements
4
5 def find_ports(self, pid):
6     # NO LOCK - race condition!
7     for proc in self.processes: # Being modified by GC!
8         ...
```

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:

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

During interview:

- ✓ Ask clarifying questions upfront
- ✓ Explain approach before coding
- ✓ Communicate assumptions
- ✓ Test code with examples
- ✓ Discuss complexity analysis
- ✓ 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!