582. Kill Process

Medium Tree Depth-First Search Breadth-First Search Array Hash Table Leetcode Link

Problem Description

In this problem, you are given the task of simulating the killing of processes in an operating system. Each process is identified by a unique ID. The relationships between processes are given in a tree-like structure, where every process is a node in the tree, and has one parent node except for the root process, which has no parent and is identified by `ppid[i] = 0`.

You are provided with two arrays: `pid`, which contains the ID of each process, and `ppid`, which contains the corresponding parent process ID for each process. The root of the tree is the process with no parent, and its ID is the one for which `ppid[i] = 0`.

Your task is to implement a function that, when given the ID of a process to kill, returns a list of all the process IDs that will be terminated. This includes the process itself and all of its descendant processes - in other words, its children, its children's children, and so on. The function should return the list of IDs in any order.

The problem statement implies that when a process is killed, it's not just that single process that stops - all processes that are 'below' it in the tree (its children) are also killed, recursively.

Intuition

To solve this problem, one intuitive approach is to map out the relationships between all processes in a way that allows you to easily identify a process's children. This can be effectively done using a graph data structure, where each process is a node and the edges represent the parent-child relationships.

First, construct a graph from the given `pid` and `ppid` arrays. In this graph, the key is the process ID of the parent, and the value is a list of IDs of its child processes. We represent this graph with a dictionary or a hashmap, where each entry corresponds to a process and its children. In Python, a `defaultdict(list)` is ideal for this because it automatically initializes a new list for each new key.

Once you have the graph, perform a Depth-First Search (DFS) starting from the process ID that needs to be killed. DFS is a suitable algorithm here because it allows us to explore all descendants of a node by going as deep as possible into the tree's branches before backtracking, thus ensuring that we find all processes that need to be terminated.

As you perform DFS, each time you visit a node (process), add its ID to the answer list. Once you exhaust all the child nodes of the process to be killed, the recursive DFS function will naturally backtrack, and the process is complete. The DFS ensures all child processes are visited and their IDs are added to the list, effectively giving us the full list of processes that will be killed.

The solution code implements this intuition, with the `dfs` function handling the actual traversal and the `killProcess` function preparing the graph and initiating the DFS call.

Solution Approach

The solution to the problem is implemented in Python using a Depth-First Search (DFS) on the graph representation of the processes. Here's an explanation of the steps taken in the given solution:

Data Structure: Graph

The graph is created as a hashmap (specifically, a `defaultdict(list)` in Python) where each key-value pair corresponds to a parent process ID (the key) and a list of its child process IDs (the value).

Graph Construction

Using the `zip` function, we iterate through both `pid` and `ppid` arrays in tandem. For each pair `(i, p)` from `pid` and `ppid`, we add process `i` to the graph under its parent `p`. The graph will store all the relations as adjacency lists.

DFS Function

The `dfs` function performs a classical depth-first search starting from the `kill` process ID. It takes an integer `i` as its argument, which is the process ID from where the search begins. The function works as follows:

1. The current node (process ID) `i` is added to the list `ans` of processes to be killed.
2. The function iterates over all the children of process `i` (if any) by looking up graph `g[i]`. For each child process `j`, the function calls itself recursively (`dfs(j)`), which leads to the child process and all its descendants being added to `ans`.

Initiating the Algorithm

We initialize an empty list `ans` which will eventually contain all process IDs to be killed.

We then call `dfs(kill)`, which begins the depth-first search starting from the process we want to kill.

After the DFS completes, 'ans' contains all the process IDs that were visited during the recursive search. Since DFS was executed starting with the process ID of `kill`, this means it includes the `kill` process and all its descendant processes, recursively.

The solution code encapsulates this approach neatly and efficiently, making sure that by the end of the call to `dfs(kill)`, the list `ans` includes every process that needs to be terminated as specified by the problem statement.

Example Walkthrough

Imagine we have a system with the following processes and parent-child relationships:

Processes: `pid = [1, 3, 10, 5]` Parent Processes:`ppid = [0, 1, 3, 3]`

The processes create a tree structure with their parent-child relationships like this:

```
      1        ppid[0]=0 (root process)
     /
    3          ppid[1]=1
   / \
 10   5        ppid[2]=3, ppid[3]=3
```

Here, process 1 is the root of the tree, and it has one child, process 3. Process 3, in turn, has two children, process 10 and process 5.

Let's walk through the solution if we want to kill process 3.

Step 1: Graph Construction First, we build the graph from `pid` and `ppid` arrays using a hashmap. Our graph would look like this:

- Key `1` will have a value `[3]`, indicating that process 1's child is process 3.
- Key `3` will have a value `[10, 5]`, indicating that process 3's children are processes 10 and 5.

This results in the graph representation being: `{1: [3], 3: [10, 5]}` with other processes linking to an empty list, since they don't have children.

Step 2: Depth-First Search (DFS) Function We want to kill process 3. So, we perform a depth-first search starting from process 3.

- We add process 3 to the `ans` list.
- We see that process 3 has two children: 10 and 5. We start a DFS for process 10.
  - We add process 10 to the `ans` list. This process has no children, so the DFS on process 10 ends.
- The DFS retraces its steps back to process 3 and starts a DFS for process 5.
  - We add process 5 to the `ans` list. This process has no children, so the DFS on process 5 ends.

The DFS has now visited all children and descendants of process 3. The process recursion completes and we have the list `ans` filled with the processes that were killed, which are `[3, 10, 5]`.

Step 3: Return the Result The list `ans` is returned from the function, which contains all the processes that were killed when process 3 was terminated. In this case, the final output would be `[3, 10, 5]`. And so, the algorithm successfully solves the problem by identifying the process to kill along with all of its descendants using a DFS approach on the process tree.

Python Solution

```python
from collections import defaultdict
from typing import List

class Solution:
    def killProcess(self, pid: List[int], ppid: List[int], kill: int) -> List[int]:
        # Recursive depth-first search (DFS) function to kill a process and its subprocesses.
        def kill_all_subprocesses(process_id: int):
            # Add the current process to the list of killed processes.
            killed_processes.append(process_id)
            # Kill all the subprocesses of the current process.
            for subprocess_id in process_graph[process_id]:
                kill_all_subprocesses(subprocess_id)

        # Create a graph using a dictionary where each key is a parent process ID
        # and the value is a list of its child process IDs.
        process_graph = defaultdict(list)
        for child_pid, parent_pid in zip(pid, ppid):
            process_graph[parent_pid].append(child_pid)

        # Initialize the list of killed processes.
        killed_processes = []
        # Start the killing process from the process with ID kill.
        kill_all_subprocesses(kill)
        # Return the list of all processes that were killed.
        return killed_processes

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

class Solution {
    private Map<Integer, List<Integer>> processGraph = new HashMap<>();
    private List<Integer> terminatedProcesses = new ArrayList<>();

    // This method takes in the process id list, parent process id list, and a process id to kill.
    // It returns the list of process ids that will be terminated.
    public List<Integer> killProcess(List<Integer> pid, List<Integer> ppid, int kill) {
        int numberOfProcesses = pid.size();

        // Build a process graph where the key is the parent process id and the value
        // is a list of direct child process ids.
        for (int i = 0; i < numberOfProcesses; ++i) {
            processGraph.computeIfAbsent(ppid.get(i), k -> new ArrayList<>()).add(pid.get(i));
        }

        // Perform a depth-first search starting from the kill process id to find
        // all processes that will be terminated.
        depthFirstSearch(kill);
        return terminatedProcesses;
    }

    // This helper method performs a depth-first search on the process graph.
    private void depthFirstSearch(int processId) {
        // Add the current process id to the list of terminated processes.
        terminatedProcesses.add(processId);

        // Recursively apply depth-first search on all child processes.
        for (int childProcessId : processGraph.getOrDefault(processId, Collections.emptyList())) {
            depthFirstSearch(childProcessId);
        }
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <functional> // For std::function
using namespace std;

class Solution {
public:
    // Function to get all the processes that will be killed if process "kill" is terminated.
    vector<int> killProcess(vector<int>& pid, vector<int>& ppid, int kill) {
        // Create a graph that represents the parent-child relationship between processes.
        unordered_map<int, vector<int>> processGraph;
        int numOfProcesses = pid.size();
        for (int i = 0; i < numOfProcesses; ++i) {
            processGraph[ppid[i]].push_back(pid[i]);
        }

        // List that will hold all the processes to kill.
        vector<int> processesToKill;

        // Depth-First Search (DFS) function to traverse the graph and add process IDs to the list.
        /*
        @param currentProcess The process ID of the current process in DFS.
        */
        function<void(int)> dfs = [&](int currentProcess) {
            // Add current process to the list of processes to kill.
            processesToKill.push_back(currentProcess);
            // Recur for all the processes that the current process is a parent of.
            for (int childProcess : processGraph[currentProcess]) {
                dfs(childProcess);
            }
        };

        // Kick-start DFS from the process we want to kill.
        dfs(kill);

        // Return the final list of processes to be killed.
        return processesToKill;
    }
};

```

Typescript Solution

```typescript
function killProcess(pid: number[], ppid: number[], kill: number): number[] {
    // Create a map to represent the process tree, where the key is parent process id,
    // and the value is an array of child process ids
    const processTree: Map<number, number[]> = new Map();

    // Populate the process tree map with child processes
    for (let i = 0; i < pid.length; ++i) {
        if (!processTree.has(ppid[i])) {
            processTree.set(ppid[i], []);
        }
        processTree.get(ppid[i])?.push(pid[i]);
    }

    // List to record all processes to be killed
    const processesToKill: number[] = [];

    // Helper function to perform depth-first search on the process tree
    // to find all child processes that must be killed
    const depthFirstSearch = (currentId: number) => {
        // Add the current process to the kill list
        processesToKill.push(currentId);

        // Iterate over all child processes of the current process
        for (const childId of processTree.get(currentId) ?? []) {
            // Recursively apply the depth-first search to the children
            depthFirstSearch(childId);
        }
    };

    // Start the depth-first search with the process to be killed
    depthFirstSearch(kill);

    // Return the list of all processes to be killed
    return processesToKill;
}

```

Time and Space Complexity

The input lists `pid` and `ppid` are traversed once to build the graph `g`, contributing `O(n)` to the time complexity, where `n` is the number of processes. The depth-first search (DFS) `dfs` is called once for each node in the graph within its own subtree. Since each node will be visited exactly once, the total time for all DFS calls is also `O(n)`. Thus, the total time complexity of the algorithm is `O(n)`.

The space complexity is `O(n)` as well, primarily due to two factors: the space taken by the graph `g`, which can have at most `n` edges, and the space taken by the `ans` list. There is also the implicit stack space used by the DFS calls, which in the worst case (when the graph is a straight line), could be `O(n)`. However, since this stack space does not exceed the size of the input, the overall space complexity remains `O(n)`.