1 Overview

This document is a record of data structure design of project 1.

Client first analyze the file descriptor input, if any, is a valid fd or not. If it's not valid, it means that this is a local file, and client send it to the original functions on client side. If not, client marshall the operation parameters as well as the size of the struct and the opration code into a char buffer, and send to server.

When server receives, it first unmarshall out the total size, and use a loop to accept all bytes sent. It then unmarshall the RPC header to get the opcode, and send the struct to different operations processors. Different processors then unmarshall the parameters and send to actual function, and marshall respond into a char buffer to client.

When client receive the respond, it first unmarshall the parameters, if any (like in read, stat, getdirentries). If the respond is a negative number, client set error node and return -1; else, it set all the local parameters, if any, and return the respond value.

2 read, unlink, stat

These operations marshall input and send to server in the same way as the operations in section 2. Server process them in the same way as in section 2, and marshall the respond and other parameters that needs to change (such as buf in read, statbuf in stat, etc) into a reply header. Then they marshall the struct into RPC header, and send back the RPC header to client.

The client unmarshall the struct from RPC header, and update their parameters accordingly.

3 getdirtree, freedirtree

freedirtree simply call orig_freedirtree.

getdirtree get the pathname of the root, marshall it to a struct, marshall the struct to a RPC header, and marshall the RPC header to a char array. Then the client send this char array to the server.

In the server side, we malloc 2 linked lists, A for the storing of inputs of getdirtree (pathnames), B for the storing of the result of getdirtree. We unmarshall the root pathname from the client and add into the first linked list. Then while the A linked list is not empty, we pop the first node in A linked list, pass the pathname into getdirtree function, and get the struct returned back. We then extract the pathname and number of children into B linked list, and keep track of the number of nodes that are added into B linked list. For the array of children, we put them into A linked list, and keep looping on them until all of them are examined with getdirtree function.

Then we marshall a respond char buffer with the size of # nodes in B linked list $\cdot$ sizeof(int) + sizeof(int), and fill it with every node in the B linked list. We send back this buffer to client.

In client side, we unmarshall the buffer, and use another linked list to build the tree. The node in this linked list has a struct with a name path, a int of number of children, and a array of this struct that represents the children nodes. We use a recursion that takes in a linked list to implement the tree: we first pop out the head of the linked list. The base case is when the child number is 0, we simply return a struct build from the pathname, child number, and an empty array; the recursive case is when the child number is not 0, we malloc an array with sizeof(struct node) * number of children, and pop out # number of children nodes from the linked list. For each node, we recursively call on the linked list with all the children popped out. When the linked list is empty, we return the root node.