CS450-PA4
Jenifer Rodriguez Delgado
Yousef Suleiman

Manual for walkers

Call `inodeTBWalker` with the argument `-h` to get a display of the traversal of all the system's inodes in the form of
`type  i-number  size`

- Example: `$inodeTBWalker -h` gives the traversal of all inodes on system

Call `directoryWalker` with the arguments of paths (if no path is given "`.`" is assumed) followed by the last argument as `-h` to get a display of the file system tree traversal starting from the path in the from of
`path_name  type  i-number  size`

- Example: `$directoryWalker -h` gives the traversal starting from root
- Example: `$directoryWalker path -h` gives the traversal starting from `path`

Call `corruptor` with the arguments of directory paths to corrupt the directory specified by that path. Note that `corruptor` fails if it is not given a path to a directory.

Call `Walkers` to recover the file system. A display will be shown on how many inodes are unreachable along with their metadata in the form of
`type  i-number  size`

Design of `inodeTBWalker.c`

The user program `inodeTBWalker` is made possible by the implementation of the new system call `imeta()` which takes in an inode number and returns the type and size associated with that inode. The program `inodeTBWalker` also includes a header file called `walkerData.h` that declares and zeros out an array of `entry struct`'s called `walkerData` that saves the type and size associated with each inode (indexed by its respective i-number). The program `inodeTBWalker` linearly traverses the inode table (made up of 200 inodes) via `imeta()`. Each inode with a type greater than 0 must be in use so `inodeTBWalker` prints out that inode's metadata to console if it is flagged to be in `human_readable_mode`. To be in `human_readable_mode`, `inodeTBWalker` takes one argument (conventionally "`-h`") to set `human_readable_mode`. If no argument is passed, `inodeTBWalker` instead saves each inode that is in use as an `entry struct` from `walkerData.h`. This data gets written to file descriptor 1 (the default for standard output). The reason why `human_readable_mode` gets flagged is because `inodeTBWalker`'s output will be redirected through pipelining when being compared to `directoryWalker`'s output.

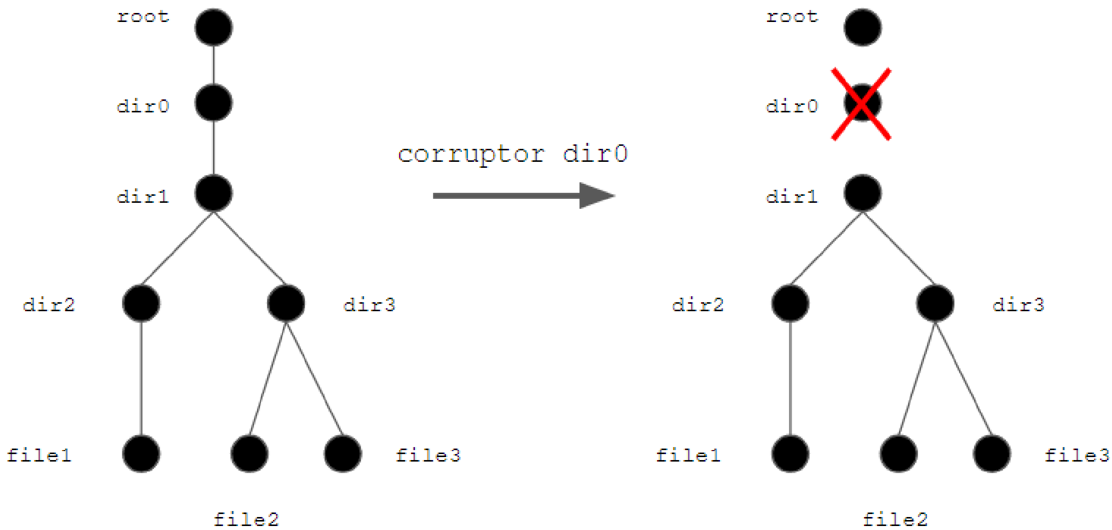
Design of `directoryWalker.c`

The user program `directoryWalker` takes in multiple paths as arguments similar to the `ls` program. However, like `inodeTBWalker`, `directoryWalker` sets `human_readable_mode` if the argument passed to it is "`-h`". If no paths are passed to `directoryWalker`, then it begins tracing the filesystem from the root directory with "`.`". The program `directoryWalker` also includes `walkerData.h`.

In the program `directoryWalker`, `main()` calls its own `ls()` with the considered path. Function `ls()` then opens the path and passes its file descriptor to system call `fstat()` saving its return to a `stat struct`. If `stat` holds a `type` of `T_FILE` or `T_DEV`, then the file's type, respective i-number, and size is printed. If `stat` holds a `type` of `T_DIR`, after printing the file's type, i-number, and size, it saves the path to a buffer, puts a '`/`' at its end, reads the directory entries of the directory file through its descriptor, then adds its child's file name after the '`/`'. This new path gets passed recursively to `ls()`. Thus, through the recursive calls of `ls()`, the remainder of the entire file system tree gets printed from the path originally given. If `human_readable_mode` is not set, then each file's inode metadata gets saved on `walkerData.h`'s `entry struct` array. Again, like `inodeTBWalker`, this data gets written to file descriptor 1.

Design of `corruptor.c`

The user program `corruptor` takes paths to *directories* as its arguments. Each directory path passed to it is corrupted. This is done by the new system call `zerout()` which takes an i-number and zeros out all of address block pointers. The inode's `size` is also set to 0 and its `nlink` is set to 1. The program `corruptor` then calls the system call unlink on the file. Thus the file has no block pointers so it looks as an empty directory so it is unlinked. Its `nlink` gets decremented to 0 and the directory inode is completely freed. Any files that were residing in the directory are left hanging and are unreachable.

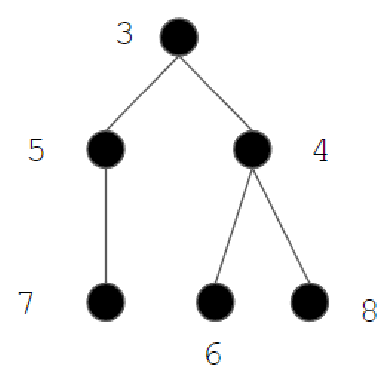


Design of `Walkers.c`

The user program `Walkers` is made possible through the 2 new system calls `scandir()` and `recover()`.

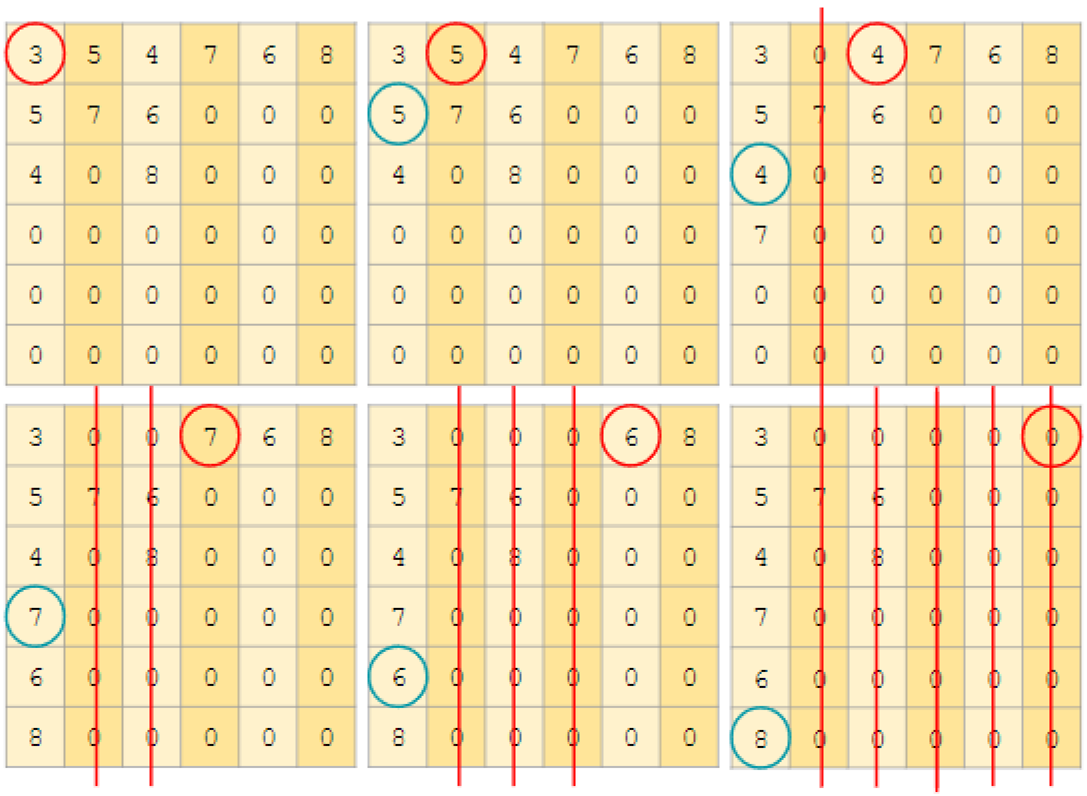
The program `Walkers` begins by using pipeline system call `pipe()` to redirect the standard output of its 2 children who call `exec()` with `inodeTBWalker` and `directoryWalker` (with `human_readable_mode` unset) in its function `runBothWalkers()`. Using the output file descriptors, `Walkers` compares the two program's outputs which are in the form of 2 `entry struct` arrays. With each inode's metadata indexable in the arrays, each entry is compared. If `Walkers` finds that `directoryWalker` could not see an inode that `inodeTBWalker` could see, that inode's metadata is saved in a third separate array of `entry struct`'s. The program `Walkers` also counts the unreachable inodes and prints out their metadata.

The hanging inode's numbers are also stored in an array of `int`'s called `hanging`. Each i-number in `hanging` is passed to the new system call `scandir()` which takes an array (`int*`), checks if the i-number belongs to a directory inode and then fills the array with all the inodes referenced by the inode directory. All the referenced inodes childed to these hanging inodes are saved in a 2d `int` array called `treeMap`. Thus if the hanging inodes look as such then the `treeMap` looks as:



3	5	4	7	6	8
5	7	6	0	0	0
4	0	8	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

The program `Walkers` then begins to coalesce arrays that are holding subdirectories by comparing each first element of each array with all the other elements. When the same i-number is found under a directory (a similar element is found under a separate array), `treeMap` gets coalesced.The algorithm looks as follows:



Thus, the first elements remaining in each array in `treeMap` holds the i-numbers whose inodes span the entire hanging section of the unreachable file tree. Arrays with first elements of 0 are ignored.

Finally, `Walkers` calls its function `gen_filename()` on only the i-numbers whose inodes span the hanging tree. Function `gen_filename()` generates a path name for the unnamed inode in the form of /`lost+found-X` where `X` is its i-number. Lastly, `Walkers` calls the new system call `recover()` on each of the spanning inodes which takes an i-number and a file path name and drops the recovered file in its root directory ('`/`'). So in the example from above, the only i-number the new system calls `recover()` on is 3 which is dropped in the root as `lost+found-3` which contains the rest of the hanging tree.

System Calls:

- `sys_imeta()`
  - System call `imeta(int inum, int *size)` takes an i-number and returns its `type` and stores its file size in `size`.
  - This is done by calling `bread` with macros `ROOTDEV` and `IBLOCK(inum, sb)` which returns the block number containing the inode with `inum`.
  - An on-disk inode pointer is made by using the locked block buffer's data from `bread` and adding to it `inum%IPB` which is the i-number modded by the inodes per block.
  - Metadata from the on-disk inode pointer can then be read such as type and size.
  - After using the locked block buffer, it must be released through `brelse()`.
  - The reason `imeta()` reads from on-disk inodes instead of in-memory inodes through `iget()` is because `iget()` can only put an inode in memory if it has a `type` > 0 (it is not free) or else it panics.
- `sys_zerout()`
  - System call `zerout(int inum)` takes an i-number and returns 0 if it is able to zero out its block address pointers, set its `size` to 0, and `nlink` to 1. Any issues that occur results in a -1 return.
  - This is done by calling `iget()` on `ROOTDEV` and the i-number so it is put in memory and returns the `inode struct`. The inode must be then locked by `ilock()`.
  - `corruptor` should only zero out directories so `zerout()` returns -1 if `type != T_DIR`.
  - The `inode struct` can then be manipulated before it is flushed to disk through `iupdate()` and unlocked through `iunlockput()`.
  - Before and after these writes are done, `begin_op()` and `end_op()` must be called to maintain the xv6 log. If this is not done, FS could panic.

- `sys_scandir()`
    - System call `zerout(int inum int *intBuf, int intBufSize)` takes an i-number with an array of integers and the array's size. It then fills the array with any other inode's i-numbers that `inum` can reach if it is a directory including itself. If `inum` is not a directory, then it is the only i-number placed in the array.
    - This is done again by using `iget()` (along with its locks) to get the `inode struct`.
    - If its `type` is `T_DIR`, all its entries (except the first 2 "`.`" and "`..`") are traversed through `readi()` incremented through the offset size of `dirent struct`.
    - If the directory entry has a `type` not equal to 0, then it is saved in the array `intBuf`.
    - If all goes well, 0 is returned.
- `sys_recover()`
    - System call `recovery(int ino, char *path)` takes an i-number and a path name. It places the inode with `ino` in the root directory. If it is an inode directory, then its "`..`" is directed to the root directory as well. If all goes well, 0 is returned or else -1.
    - This is done again by using `iget()` (along with its locks) to get the `inode struct`.
    - Then `nameiparent()` is called with `path` to get the parent directory inode (if called with `/lost+found-X` then the parent is `/`).
    - `dirlink()` is then called to add the inode in the parent directory.
    - Lastly, the inode's `type` is checked for `T_DIR` and `writei()` is used to write over the inode's second directory entry "`..`" to point back to root.