4/28/2017

File System Recovery

For MINIX File System

By PROJECT TEAM 1

Aishwarya Madbal Gowda - A20391651

Ganesh Mahesh - A20378764

Nithin Ashok - A20391878

Minix File System

In fs/minix/inode.c file, struct super_operations minix_sops is initialized. The inode->(struct inode_operations*) i_op field is set to be minix_file_inode_operations, minix_dir_inode_operations, or minix_symlink_inode_operations depending on the file referred to by the inode is a regular file, directory, or a symbolic link. The inode->i_fop field is initialized to be minix_file_operations, minix_dir_operations depending on the file referred to by the inode is a regular file or directory.

minix_dir_inode_operations: initialized in fs/minix/namei.c, Line 306.
minix_file_inode_operations: initialized in fs/minix/file.c, Line 27.
minix_symlink_inode_operations: initialized in fs/minix/inode.c, Line 346.
minix_dir_operations: initialized in fs/minix/dir.c, Line 17.
minix_file_operations: initialized in fs/minix/file.c, Line 18.
The minix filesystem module initialization is done when the minix module is inserted to the kernel. And the minix filesystem registration is done inside the init_minix_fs function. At this stage, the minix_get_sb function is available for reading the super block information from a minix filesystem.

fs/minix/inode.c:
596 module_init(init_minix_fs)

fs/filesystems.c:

```
 int register_filesystem(struct file_system_type * fs)
 {
        int res = 0;
        struct file_system_type ** p;

        if (!fs) return -EINVAL;
        if (fs->next) return -EBUSY;
        INIT_LIST_HEAD(&fs->fs_supers);
        write_lock(&file_systems_lock);
        p = find_filesystem(fs->name);
        if (*p) res = -EBUSY;
        else  *p = fs;
        write_unlock(&file_systems_lock);
        return res;
 }
```

When mounting a minix filesystem, its superblock object is prepared by the minix_get_sb function. The read_inode operation in the superblock object's s_op field can then be used to retrieve the inodes of each component of a specified pathname. The minix_read_inode function is implemented in fs/minix/inode.c (, Line 437) and registered to minix_sops in Line 96. When minix_get_sb function preparing superblock object, minix_sops, its s_op field is initialized to be minix_sops in the minix_fill_super function.

The minix_lookup function of the minix_dir_inode_operations (in the fs/minix/namei.c file, Lines 57 and 306, respectively), can be invoked to get the inode number (of type unsigned long) for each component of a given pathname. The minix_inode_by_name function implemented in fs/minix/dir.c (Line 398) is the one returning the inode number.

The i_op and i_fop fields of an (incore) inode structure are assigned to be the pair minix_file_inode_operations, minix_file_operations and minix_dir_inode_operations,

minix_dir_operations of functions in the fs/minix/inode.c file (Line 353), depending on the inode is referred to an ordinary file or a directory.

The followings are the actual functions responsible for carrying out the actual operations which we ask the system to complete. Take your time to browse through the source code. Also, confirm that the generic_read_dir function (in minix_dir_operations) is invoked to produce error message: cat: /bin: Is a directory, say, when command "$ cat /bin" is issued. You may start tracing the source code for the following set of systems calls on include/linux/syscalls.h file:

Directory related system calls

System call	Description
link, unlink	Call the (un)link function to create (delete) a link to a file
mkdir	Make directories
mknod	Make block or character special files
rename	Change the name or location of a file
rmdir	Remove empty directories
symlink	Make a new name for a file
open, creat	Open and possibly create a file or device

File related system calls

System call	Description
lseek, llseek	Reposition read/write file offset
read	Read from a file descriptor
readahead	Perform file readahead into page cache
write	Write to a file descriptor
sync	Commit buffer cache to disk

```
/* fs/minix/file.c */
 struct file_operations minix_file_operations = {
        .llseek         = generic_file_llseek,
        .read           = generic_file_read,
        .write          = generic_file_write,
        .mmap             = generic_file_mmap,
        .fsync          = minix_sync_file,
        .sendfile       = generic_file_sendfile,
 };

 struct inode_operations minix_file_inode_operations = {
        .truncate       = minix_truncate,
        .getattr        = minix_getattr,
 };

/* fs/minix/dir.c */
 struct file_operations minix_dir_operations = {
        .read           = generic_read_dir,
        .readdir        = minix_readdir,
        .fsync          = minix_sync_file,
 };

/* fs/minix/namei.c */
 /*
```

```
 * directories can handle most operations...
 */
struct inode_operations minix_dir_inode_operations = {
        .create         = minix_create,
        .lookup         = minix_lookup,
        .link           = minix_link,
        .unlink         = minix_unlink,
        .symlink        = minix_symlink,
        .mkdir          = minix_mkdir,
        .rmdir          = minix_rmdir,
        .mknod          = minix_mknod,
        .rename         = minix_rename,
        .getattr        = minix_getattr,
};
```

File System recovery in minix:

RECOVERING LOST FILES. You can search a disk for an ASCII string, once found, the block can be written out to a file. A one line change to fs/path.c allows users to get the i-node number for a file after it has been removed from a directory. Another couple lines changed in the file system keep the i-node information available until the i-node is reused (normally this information is zeroed out when an i-node is released.) This allows a de(1) user to go to a released i-node, get all the block numbers, go to these blocks and write them back to a new file. The whole recovery process is automated by running "de -r file". So, IF a file is unlink(2)'ed (eg. "rm file"), AND IF no one allocates a new i-node or block in the mean-time, THEN you can recover the file.

Project Overview

The goal of this project is to develop the tools for the MINIX file system namely: -

1) DirectoryWalker that will traverse the directory tree of a given directory and produce all the subdirectories and the files reachable from that directory as well as the inodes and zones allocated to those subdirectories and files
2) iNodeBitMapWalker that reads the inode bit map and returns all the zones that have been allocated.
3) ZoneBitMapWalker that reads the inode bit map and returns all the inodes that have been allocated.

Scope of the project.

To develop a set of tools for the MINIX filesystem.

Decription

Manual Pages for our tools implemented.

This project aims at implementing a set of tools to manage the fie system in MINIX. Directory,inode and zones being the important component of MINIX file system.We are set to implement a set of tools to trace the directory, inode and zones . A user process calls DirectoryWalker that will traverse the directory tree for a given directory and produce all the subdirectories and files reachable from that directory as well as the inodes and zones allocated to those subdirectories and files.

The new tools implemented in this perspective include:

- **DirectoryWalker**: A **DirectoryWalker** that will traverse the directory tree of a given directory and produce all the subdirectories and files reachable from that directory as well as the inodes and zones allocated to those subdirectories and files.
- **iNodeBitMapWalker:** An **iNodeBitMapWalker** that reads the inode bit map and returns all the inodes that have been allocated.
- **ZoneBitMapWalker**: A **ZoneBitMapWalker** that reads the zone bit map and returns all the zones that have been allocated.

Structure of project implementation:

1) Declare a library function in the path **usr/src/include/unistd.h** add an entry of the function of the system call.
2) In the path usr/src/include/minix under
   - callnr.h – adding a new entry for table association, this association number should associate to the usr/src/servers/vfs/table.c entry
   - vsfif.h – adding a request number associated with usr/src/servers/mfs/table.c entry
3) Defining the system call in library routine.
   - usr/src/lib/libc/sys-minix – add a new file which is a system call for our project implementation namely inodewalker,directorywalker,zonewalker.
   - usr/src/lib/libc/sys-minix/Makefile.inc – add the newly created file in Makefile.inc. This defines the system call declaration.
4) Implementation of system call using mfs:
   - Create a new file customcall.c which is used to implement our system call wherein it defines the system call implementation.
   - Add the newly created file in usr/src/servers/mfs/Makefile.
   - In usr/src/servers/mfs/table.c add an entry associated with entry matching usr/src/servers/mfs/table.c.
   - In usr/src/servers/mfs/proto.h add a function declaration of the system call.
5) Implementation to forward the message to a particular file server which will handle the system call.
   - Create a new file customsyscall.c which will forward the message to a particular file server which will handle the system call.
   - Add the newly created file in Makefile.
   - In usr/src/servers/vfs/proto.h add a declaration for a system call.
   - In usr/src/servers/vfs/request.c declare a function which is a request function which sends a message to mfs.
   - In usr/src/servers/vfs/table.c add an entry for system call which associates with usr/src/include/minix/callnr.h.

Implementation of iNodeBitMapWalker referring the structure explained above.

Testing our iNodeBitMapwalker:

1) cd /root

cc fs_test.c

./a.out

The program will allow you to test the following: -

MENU

1)Directory Walker

2)iNodeBitMap Walker

3)ZoneBitMap Walker

4)Repair Tool

5)Damage inode bit map

6)Damage zone bit map

7)Damage directory file

8)Damage inode of a directory file

9)Exit

```
[# cc fs_test.c
[# ./a.out

================== CS551: TEST PROGRAM FOR FILE SYSTEMS ==================
This program will allow you to test the following.

MENU
1. Directory Walker
2. inodeBitMap Walker
3. ZoneBitMap Walker
4. Repair Tool
5. Damage Inode Bit Map
6. Damage Zone Bit Map
7. Damage directory file
8. Damage Inode of a Directory File
9. Exit
```

We choose 2

Then the window showing the inode of the files will be displayed.

The output also shows all the entries in other fields of the inode structure.

```
To switch to the online repository, do 'pkgin update' again. To install
all packages, do pkgin_all.

MINIX 3 supports multiple virtual terminals. Just use ALT+F1, F2, F3
and F4 to navigate among them.

For more information on how to use MINIX 3, see the wiki:
http://wiki.minix3.org.

# HEY I AM DO_INODEWALKER in VSF
=== INODEWALKER in MFS ===
Getting super node from device 17293821761648853888 ...
ninodes       = 32768
nzones        = 16384
imap_blocks   = 2
zmap_blocks   = 1
firstdatazone = 517
log_zone_size = 0
maxsize       = 2147483647
block size    = 4096
flags         = DIRTY

Loading inode bitmap from disk ...
 done.
_
```

FreeBSD 2

```
556, 557, 558, 559, 560,
561, 562, 563, 564, 565,
566, 567, 568, 569, 570,
571, 572, 573, 574, 575,
576, 583, 584, 585, 586,
587, 588, 589, 590, 591,
592, 593, 594, 595, 596,
597, 598, 599, 600, 601,
602, 603, 604, 605, 606,
607, 609, 610, 611, 612,
613, 614, 615, 616, 617,
618, 619, 623, 624, 625,
626, 627, 628, 629, 630,
631, 632, 633, 634, 635,
636, 637, 638, 639, 641,
642, 643, 645, 646, 647,
648, 649, 650, 651, 652,
653, 654, 655, 656, 657,
658, 659, 660, 661, 662,
663, 664, 665, 666, 667,
670, 671,
========================================

Used: 658 / 32768
_
```

FreeBSD 2

Implementation of ZoneBitMapWalker referring to the structure explained above.

Testing our ZoneBitMapwalker:

1) cd /root

cc fs_test.c

./a.out

The output appears stating a menu for selection.

The program will allow you to test the following:-

MENU

1)Directory Walker

2)iNodeBitMap Walker

3)ZoneBitMap Walker

4)Repair Tool

5)Damage inode bit map

6)Damage zone bit map

7)Damage directory file

8)Damage inode of a directory file

9)Exit

```
[# cc fs_test.c
[# ./a.out

================== CS551: TEST PROGRAM FOR FILE SYSTEMS ==================
This program will allow you to test the following.

MENU
1. Directory Walker
2. inodeBitMap Walker
3. ZoneBitMap Walker
4. Repair Tool
5. Damage Inode Bit Map
6. Damage Zone Bit Map
7. Damage directory file
8. Damage Inode of a Directory File
9. Exit
```

We choose 3

Then the window showing the zone of the files will be displayed.

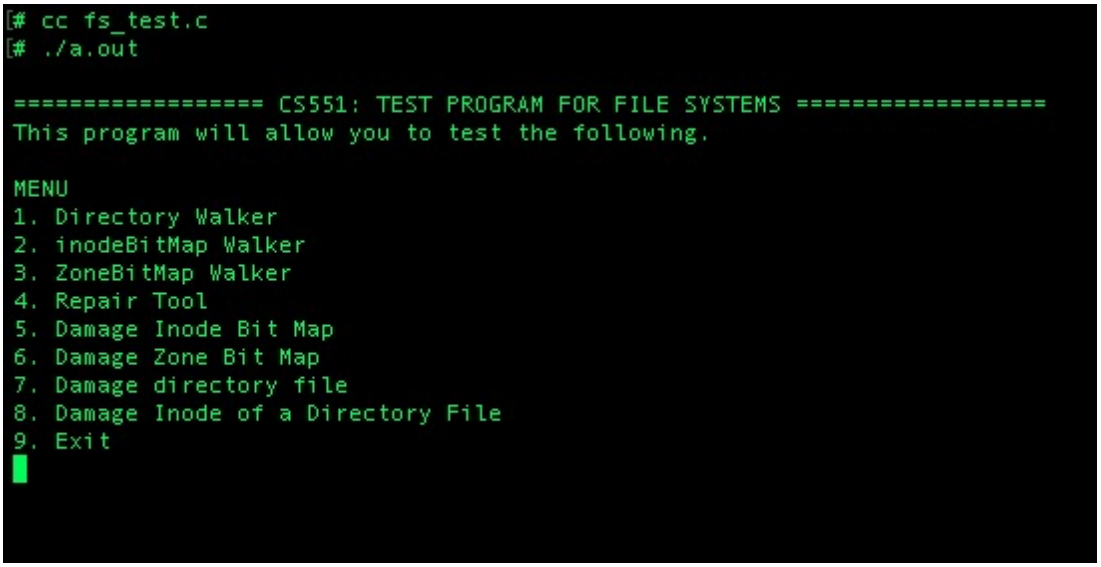
The output also shows all the entries in other fields of the zone structure.

```
# DO_ZONEMAPWALKER from VSF
=== ZONEWALKER in MFS===
Getting super node from device 896 ...
ninodes       = 32768
nzones        = 16384
imap_blocks   = 2
zmap_blocks   = 1
firstdatazone = 517
log_zone_size = 0
maxsize       = 2147483647
block size    = 4096
flags         = DIRTY

Loading zone bitmap from disk ...
 done.

============= Used blocks ==============
```

```
5905, 5906, 5907, 5908, 5909,
5910, 5911, 5912, 5913, 5914,
5915, 5916, 5917, 5918, 5919,
5920, 5921, 5922, 5923, 5924,
5925, 5926, 5927, 5928, 5929,
5930, 5931, 5932, 5933, 5934,
5935, 5936, 5937, 5938, 5939,
5940, 5941, 5942, 5943, 5944,
5945, 5946, 5947, 5948, 5949,
5950, 5951, 5975, 5976, 5977,
5978, 5979, 5980, 5981, 5982,
5983, 6496, 6497, 6498, 6499,
6500, 6501, 6502, 6503, 6504,
6528, 6529, 6530, 6531, 6532,
6533, 6534, 6535, 6536, 6537,
6538, 6539, 6540, 6541, 6542,
6543, 6544, 6545, 6546, 6547,
6548, 6549, 6550, 6551, 6552,
6553, 6554, 6555, 6556, 6557,
6558, 6559, 6588, 6589, 6590,
6591,
========================================

Used: 5677 / 16384
```

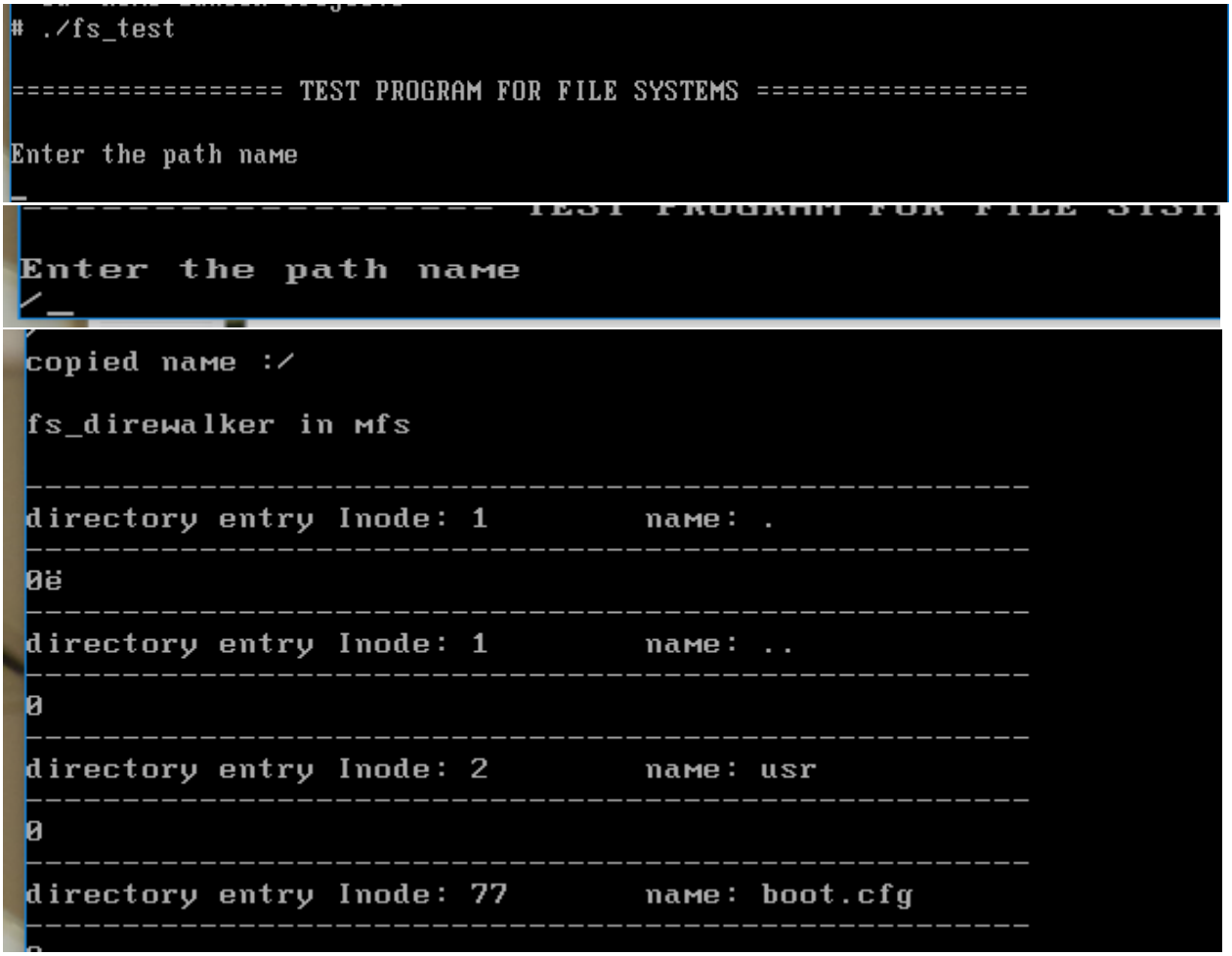
Implementation of DirectoryBitMapWalker referring to the structure explained above.

- **Testing our DirectoryWalker**
- **cd /root**
- **cc fs_test.c**
- **./a.out**
- **The output appears stating a menu for selection.**
- **The program will allow you to test the following: -**
- **MENU**
- **1)Directory Walker**
- **2)iNodeBitMap Walker**
- **3)ZoneBitMap Walker**
- **4)Repair Tool**
- **5)Damage inode bit map**
- **6)Damage zone bit map**
- **7)Damage directory file**
- **8)Damage inode of a directory file**
- **9)Exit**
- **We choose 1**



  - **Then the window showing all the directory.**

– **The output also shows all the directories and inodes number in other fields of the directory structure.**



Implementation of a set of utilities that will repair the file system when:

1) **The inode bit map is damaged.**

- ✓ The mapping from blocks to inode is many-to-one, organized such that each consecutive set of blocks, belong to consecutive inodes. Say each inode contains 8 blocks. Under this arrangement, the first 8 blocks belong to inode 1, the next 8 blocks belong to inode 2 and so on. To obtain the first block in an inode, we simply have to left shift the inode number using the scaling factor
- ✓ We directly map the inode for which we are looking for.
- ✓ In mfs we going to loop until we find the particular inode we are looking for.
- ✓ starting from the root node we get the blocks of the root inode and traverse the inode structure which is in the form of Inode number and Name.
- ✓ if the searched name is not found then indirect inodes are also checked.
- ✓ Once it is found that inode number is used to fetch the inode itself.
- ✓ Now we print all the elements present in the data block of that particular inode which is again of the form.

- ✓ inode number and name which shows all the inodes and normal files present in that particular inode.

1. **Corruption** -> we traverse the bitmap and find the particular node which user has entered we are going to flip the bit which was previously 1.
2. **Repair**-> we will traverse the list of inode which are present in the inode.h file maintaining all the inodes which are open.
   - And we will also consider unused_inodes que which is also maintained in indoe.h considering both we rectify the corrupted inode map

2) The zone bit map is damaged.

- ✓ The mapping from blocks to znode is many-to-one, organized such that each consecutive set of blocks, belong to consecutive zones. Say each inode contains 8 blocks. Under this arrangement, the first 8 blocks belong to inode 1, the next 8 blocks belong to inode 2 and so on. To obtain the first block in an inode, we simply have to left shift the inode number using the scaling factor
- ✓ We directly map the zone for which we are looking for.
- ✓ In mfs we going to loop until we find the particular zone we are looking for.
- ✓ starting from the root node we get the blocks of the root zone and traverse the zone structure which is in the form of zone number and Name.
- ✓ if the searched name is not found then indirect zones are also checked.
- ✓ Once it is found that zone number is used to fetch the zone itself.
- ✓ Now we print all the elements present in the data block of that particular zone which is again of the form.
- ✓ zone number and name which shows all the zones and normal files present in that particular zone.

Corruption: We traverse the bitmap and find the particular node which user has entered we are going to flip the bit which was previously 1.

Repair: We will traverse the list of zones which are present in the inode.h file maintaining all the zones which are open.

- And we will also consider unused zones que which is also maintained in indoe.h considering both we rectify the corrupted zone map.

3) **A directory file is completely corrupted**.

- ✓ We start our search from the root inode which in vfs
- ✓ In mfs we going to loop until we find the particular directory we are looking for.
- ✓ Starting from the root node we get the blocks of the root directory node and traverse the directory structure which is in the form of Inode number and Name.
- ✓ If the searched name is not found then indirect zones are also checked once it is found.
- ✓ That inode number is used to fetch the inode itself.

- ✓ Now we print all the elements present in the data block of that particular inode which is again of the form.
- ✓ inode number and name which shows all the directory and normal files present in that particular directory.

Corruption: We traverse the bitmap and find the particular node which user has entered we are going to flip the bit which was previously 1.

Repair: We will traverse the list of inode which are present in the inode.h file maintaining all the inodes which are open.

- And we will also consider unused inodes queue which is also maintained in indoe.h considering both we rectify the corrupted inode map

4) **The inode of a directory file is completely corrupted.**

   Although the inofe of a file is corrupted the inode of a directory file is completely corrupted, the data of that would still persist in the data blocks of the file system

5) **How we could recover from the following situations: -**

Part of the superblock is damaged.

If the superblock of a file system is damaged, the file system cannot be accessed. You can fix a corrupted magic number in the file system superblock.

Most damage to the superblock cannot be repaired. The following procedure describes how to repair a superblock in a JFS file system when the problem is caused by a corrupted magic number. If the primary superblock is corrupted in a JFS2 file system, use the fsck command to automatically copy the secondary superblock and repair the primary superblock.

A block (or blocks) allocated to a file is damaged.

All read-write file systems must deal with the problem of assigning blocks to files. There are actually two problems that may be solved separately: finding a free block (which requires remembering which blocks are free and which are not), and remembering the blocks that store the data of a certain file.

The first distinguishing feature is the use of *extents*. An extent specifies a range of blocks by giving the first block's number and the length of the range in blocks. The alternative is to store each block's number separately, which usually takes more space, but is slightly simpler to handle.

Many file systems use an *allocation bitmap* to store which blocks are used. That bitmap requires one bit per useable block. Searching such a bitmap can be quite slow. Some file systems cache a short list of free blocks, which can be updated in the background.

The traditional Minix way to store the blocks that make up a file doesn't use extents, but a block number list. The first few numbers are stored in the inode structure; these are called *direct blocks*. If that space is exhausted, one block is allocated to store further block numbers. Only the block number of this *indirect block* is stored in the inode structure. If that space is also exhausted, the method is applied recursively, leading to double-indirect and sometimes triple-indirect blocks.

Exceptional cases:

1) **Problem**: Path mentioned for the directory walker is incorrect.

Solution: The path is checked component by component, if any of the component is invalid then a prompt saying invalid path is displayed.

2) **Problem**: unable to obtain the starting start node when inodebitmap walker is executed.

   Solution: The message will be prompted saying access to inode map denied.