

CS-518: Assignment 2

Write Once File System

Tanya Sharma (tds104) and Sanchay Kanade (sk2656)

1. INTRODUCTION

This project aims to create a user-defined write once file system that can be implemented like the way we designed a user-defined memory allocator since it involves similar principles of data organization and management. However, due to the persistent nature of a file system, there are additional challenges and requirements that need to be addressed. A straightforward approach is to think of memory as a linked list with nodes that might be stored in order in a single space.

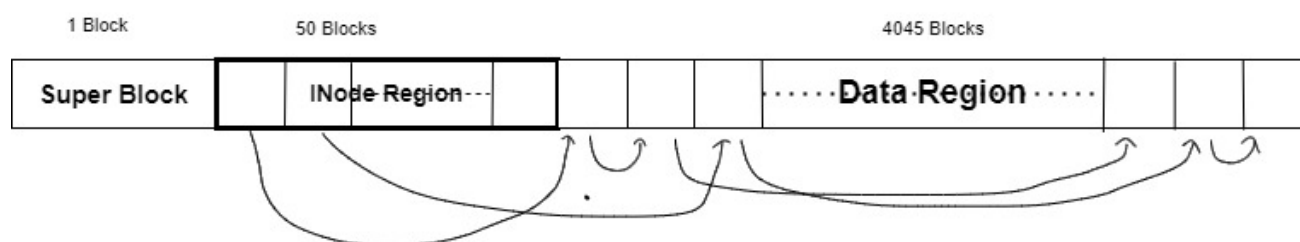
2. METHODOLOGY

Our file system is going to be divided into 3 regions.

1. **Super block:** This will be our first region of 1KB
2. **Inode region:** This will be our second region of 50 1KB block which will represent 50 inodes. For the sake of simplicity, we are assuming 50 inodes initially which can be easily extended in future. This region will be of size 50 KB (50*1024bytes).
3. **Data block region:** This region will contain all the data blocks each of size 1KB. After superblock and inode region we have 4045 blocks of 1KB left which all comprise our data block region.

All the data blocks in a particular file are going to be connected to each other in a linked list fashion. The inode corresponding to that file will contain the starting index offset of the first data block of the file and we can traverse the rest of the data blocks from the first one in that file.

Rather than storing the absolute addresses or indices, we will be storing the offset with respect to the beginning of the memory location where our file system is mounted. This is a visual representation of our file system.



We were planning on having a bit map region as well but since in the scope of this assignment we are not implementing any delete function we applied one optimization rule!!!. All the free data blocks will be after the last filled (partial or full) data block so that we always know that given any file if we want to add a new data block we can directly go to the index of the last filled data block which we are storing as count of data blocks filled in our super block. Apart from faster access speed we get more space as well. This is kind of like defragmentation concept taught in class of keeping linked list block close together for faster access.

3. DATA STRUCTURES

- **char *memory_rep**

This is a global pointer which will hold the address of the memory location where we will mount our file system. In the int main function we have malloc 4MB(4*1024*1024) of memory and passed that address to this pointer.

- **typedef enum flags**

- **WO_RDONLY:** Read-only is a file attribute that prevents writing to a file and only permits a user to see it.
- **WO_WRONLY:** Write-only is a file characteristic that only allows a user to write to a file and forbids reading from it.
- **WO_RDWR:** Users have permission to both read from a file and also write to it

- **typedef struct dataBlock**

The data block is a linked list representation of disk block quanta. The memory that represents the disk can only be read or written to or from in 1K chunks.

- **data[1020]:** stored data to be read or written as a block of data similar to a page
- **next:** an integer in which we are storing the offset of the next data block from the starting of our memory location. It is initialized as -1 so that we can know that there are no more data blocks in a file after this.

- **typedef struct superblock**

Stores information about the file system as a whole. It has a fixed location on the disk and is accessed by the file system whenever it is mounted. This allows the file system to swiftly retrieve important information about its structure and state. In our project, the superblock is the first region of the disk is of size 1024bytes that is 1 block of 1KB.

- **numberOfinodesFilled:** Stores total number of inodes occupied or number of files in our file system.
- **numberOfblocksFilled:** Stores total number of disk blocks in used by all the files in our file system. Can be used to determine the total size used in our file system (numberOfblocksFilled*1024 bytes).

As we can see, our super block only has 2 intergers which occupy only 8 bytes in total. We can optimize our super block to occupy smaller size (like 8 Bytes) to make some more space. But for the ease of access and accessing data in chunks of 1KB we let it be of 1KB.

- **typedef struct inode**

- **id:** Identifier for each inode
- **start_index:** index of the first disk block for every file. It is also an offset from the beginning of the memory location where file system is mounted.
- **numberOfDataBlocks:** total number of disk blocks occupied by the file associated with this inode
- **name[16]:** name of the file
- **isOpen:** variable that stores the open or close condition of the file
- **rwFlag:** read-write flag that determines whether the file is open as read-only, write-only or read-write
- **bytesFilled:** stores the number of bytes filled in the last disk block associated with file, used to check whether a new disk block is required for further writes or to continue storing data in the existing disk block

4. FUNCTIONS

- **void createFS()**

- Function that creates the disk file for the file system.
- It opens a new file of size 4MB to store the superblock, inodes and disk blocks in. This function creates a file on the local filesystem called "myfile" with a size of CAPACITY * sizeof(char) - 1 bytes.
- The file is opened in write mode, and the file pointer is moved to the end of the file using the fseek function. A null character is then written to the file using the fputc function, and the file is closed.

- **int wo_mount(char* filename, void* memoryAddress)**

- This function is used to mount a file on the local filesystem as a virtual disk and initialize the file system data structures if the file is being mounted for the first time.
- The function takes in two arguments: a string containing the name of the file to be mounted and a void pointer to a memory address where a file system data structure will be stored in memory.
- The function first attempts to open the file in the file read-write mode using the fopen function. If the file cannot be opened, an error message is printed and the function returns.
- Otherwise, the file is successfully opened and the contents of the file are read into the memory address specified by the second argument using the fread function.
- The function also checks if the file is being mounted for the first time by checking the value of the "init" variable. If it has never been initialized, the function initializes the file system data structures by setting the values of the superblock and inode structures. It also sets the values of the data blocks to default values and writes the initialized data structures to the file.

- **int wo_unmount(void* memoryAddress)**

- With this function, we are unmounting a file from the virtual disk and saving any changes made to the file system data structures to the file.
- The function takes in a single argument: a void pointer to the memory address where the file system data structures are stored.
- The function tries to open the file in read-write mode using the fopen function. If the file cannot be opened, the function returns and an error message is printed.
- Otherwise, the file is successfully opened and the contents of the memory address specified by the argument are written to the file using the fwrite function.
- This function allows the changes made to the file system data structures in memory to be saved to the file on the local filesystem.

- **int wo_create(char* fileName, flags f)**

- Function that is used to create a new file on the virtual disk.
- The function takes two arguments: a string containing the name of the file to be created, and a flags variable that specifies the read-write permissions for the file.

- The function first checks if the maximum number of files that can be stored on the virtual disk has already been reached. If this is the case, the function prints an error message and returns because we cannot exceed the disk size.
- Otherwise, the function searches the inode table for an unused inode where the new file can be stored. If the file name already exists on the virtual disk, the function returns an error since two files with the same file name are not allowed to exist.
- Otherwise, the function creates a new inode for the file, sets its name and read-write permissions and increments the number of inodes in use of the virtual disk. The function returns the file descriptor of the new file.
-
- **int wo_open(char* fileName, flags f)**
 - This function is used to open an existing file on the virtual disk.
 - The function takes two arguments: a string containing the name of the file to be opened and a flags variable that specifies the read-write permissions of the file.
 - The function first searches the inode table for the file with the given name. If the file is not found, the function prints an error message and returns since we cannot open a file that cannot be found on the virtual disk.
 - Otherwise, the function checks to see if the file is already open. If the file is not open, the functions sets its read-write permissions, as long as they are valid, and marks it as open.
 - If the file is already open, the function checks if the read-write permissions match the requested permissions and the function returns the file descriptor upon success.
 - If the permissions do not match, the function returns an error. With this function, users are allowed to open a file on the virtual disk and set its read-write permissions.
- **int wo_close(int fileDescriptor)**
 - This function is used to close an open file on the virtual disk.
 - The function takes a single argument: the file descriptor of the file to be closed.
 - The function first checks if the file descriptor is valid. If the file descriptor is not valid, the function prints an error message and returns because we cannot close a file we cannot identify in the disk.
 - Otherwise, the function searches the inode table for the file with the given file descriptor. If the file is not found, the function returns an error and prints an error message.
 - If the file is found but is already closed, the function prints an error because we cannot close an already closed file.
 - Otherwise, the function marks the file as closed and returns 0 to indicate success. The function allows the user to close an open file on the virtual disk.
- **int wo_write(int fileDescriptor, void* buffer, int Bytes)**

We are assuming that if the bytes to be written are more than the capacity of the file then it will cancel the operation and return an error.

Initial Operations

 - This function is used to write to a file on the virtual disk.
 - The function takes in three parameters: a file descriptor, a pointer to a memory location (the buffer) and the number of bytes to write.

- It first checks if the file descriptor is valid or not. A file descriptor is a non-negative integer that is used to identify a file. In our project, our file descriptor can have valid values only within the range 0 to 50. If the file descriptor is not valid, the function prints an error message indicating that no such file exists and returns.
- We calculate the number of data blocks needed to store the given data based on the number of bytes. The number of data blocks needed is determined by dividing the number of bytes by 1020, which is the maximum number of bytes that can be stored in a data block. The number of extra bytes needed is also determined. Then, the data from the buffer is copied into a new char array. We iterate through the inodes to find the inode corresponding to the given file descriptor. If the inode is found, the code checks whether the file is open and whether it is open in write mode. If both these conditions are met, the code sets a flag indicating that the file has been found and exits the loop.
- If the inode is not found or if the file is not open in write mode, the code returns and prints an error.
- We check to see if the inode of the file has 0 data blocks indicating that this is the first data block being written to the file. If this is the case, the code sets the start index of the data block using the number of blocks filled in the file system and the size of the superblock and inode regions. It then calculates the address of the data block .

If file bytes fit in one disk block

- The code checks if the number of bytes to be written is less than or equal to 1020. If this is the case, it copies the data from the buffer to the data block using the strcpy function and sets the next field of the data block to -1 (since there are no subsequent blocks).
- We also update the inode to reflect the number of bytes written and the number of data blocks in the file and increments the total number of blocks filled in the file system. It then returns 0 to indicate that the write operation was successful.

If files do not fit in one disk block

- If the number of bytes to be written is greater than 1020, we first calculate the number of data blocks needed to store the data, based on the number of bytes to be written and the maximum size of a data block.
- We then write the data in each of these data blocks, copying 1020 bytes of data from the buffer to each data block using the strncpy function. For each data block, the code sets the next field to the address of the next data block, unless this is the last block in which case it sets it to -1.
- It then updates the inode to reflect the number of bytes written and the number of data blocks in the file, and increments the total number of blocks filled in the file system. The function then returns 0 to indicate that the write operation was successful.

If file contains extra characters that do not fit in one disk block

- If there are extra characters that could not be written to a full data block, we set the next field of the current data block to the address of the next data block and updates the cursor to point to it.
- It then writes the remaining characters to this data block using the strncpy function and updates the relevant metadata. We then return a 0 to indicate that the write operation was successful.

Concatenating to a file with all disk blocks completely full

- If the file already has one or more data blocks, we check if the last data block is full as indicated by the bytesFilled field of the inode being equal to 1020. If this is the case, the code searches for the last data block in the file by starting at the address of the first data block and following the next pointers of each data block until it reached the last data block.
- Once it has found the last data block, it sets the next field of this block to the address of the new data block and updates the cursor. It then writes data to the new data blocks, copying 1020 bytes of data from the buffer to each data block using the strncpy function.
- If this is the last data block, it sets the next field of the data block to -1. We also update the relevant metadata. If there are still extra characters that could not be written to the full data block, the code sets the next field of the current data block to the address of the next and updates the cursor. It then updates the inode to reflect the changes in metadata.

Concatenating to a file with partial data in the last disk block

- We now handle the scenario where the file being written to already has some data blocks written to it and the last block is partially filled. In this scenario, the code needs to first check if there is space in the last block to write more data. If there is space, we simply concatenate the data from the buffer to the last block and update the inode to reflect the change.
- If there isn't space, it creates a new data block and writes the remaining data from the buffer to it. It then updates the inode and the superblock to reflect this change.

- **int wo_read(int fileDescriptor, void* buffer, int Bytes)**

- This function is used to read a file on the virtual disk.
- The function has three parameters: a file descriptor, a pointer to a memory location (the buffer) and number of bytes to be read.
- It checks if the file is open and if it can be read. It starts off by checking if the file descriptor is valid (within the range of valid file descriptor). If it is valid, the code then searches through the list of inodes in memory, looking for an inode with an ID that matches the file descriptor.
- If it finds a matching inode, it checks if the inode indicates that the file is open and if it has valid read permissions. If all the conditions are met, then we set a flag to indicate that the corresponding inode has been found and the file is open and can be read.
- If any of these conditions are not met, we print an error message and return.
- If the inode has been found, we calculate the number of data blocks that need to be read to satisfy the read request as well as the number of extra characters.
- If the number of blocks to be read is less than or equal to the number of data blocks in the file, the code sets a cursor to the start of the first data block of the file and reads each of the, into the buffer, concatenating data from each block as it goes.
- If there are extra characters needed to be read, it then reads the minimum of the number of bytes remaining in the file and the number of extra characters needed to be read, and appends them to the buffer.
- If the number of blocks needed to be read is greater than the number of data blocks in the file, the code instead reads all of the data blocks in the file, into the buffer. Finally, 0 is returned and the file is read successfully.

5. ERROR HANDLING

When any part of the function does not behave as expected, we use linux error codes to communicate this error. Error management is not directly supported in C and we must identify the error and deal with it. Return values of a method signify a program's success or failure. Errno is a name for an external variable in the C programming language which can be used to utilize error handling routines. A few error codes used in our project are:

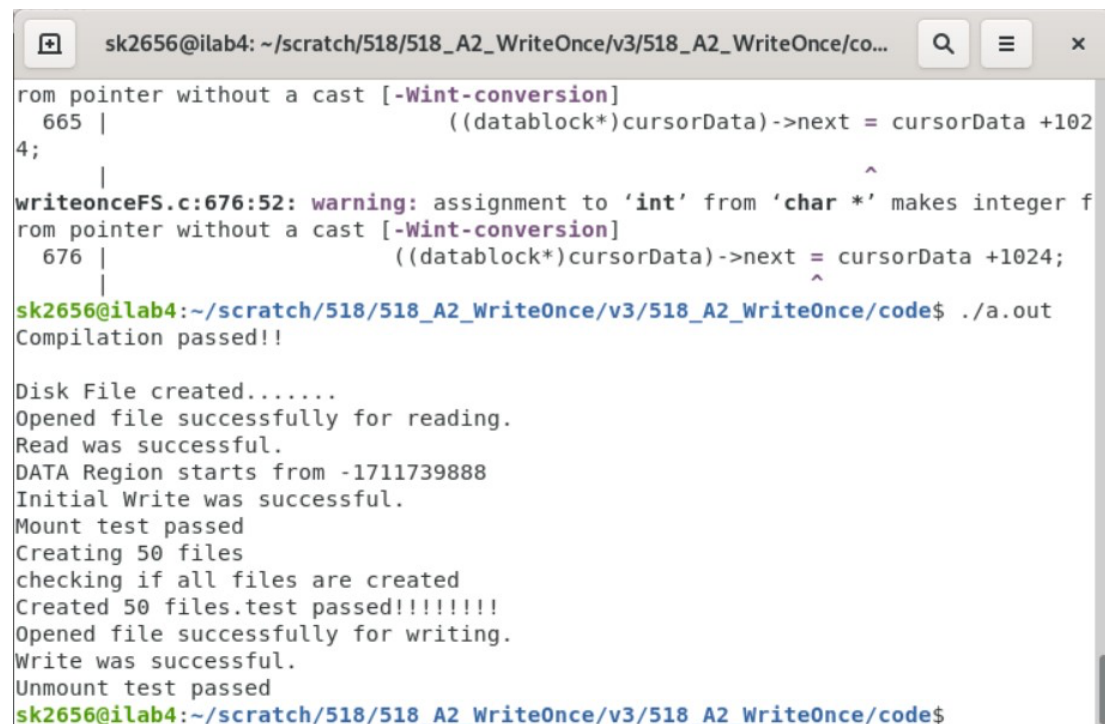
- **ENOENT(2)** : no such file or directory
- **ENFILE(23)**: File table overflow
- **EEXIST(17)**: File already exists
- **EBADF(9)**: Bad file number
- **ENOSPC(28)**: no space left on device
- **EROFS(30)**: Read-only file system

A successful operation always has a function that returns 0. Otherwise, upon failure or error, the function prints the corresponding error code and returns a negative value (-1).

6. TESTING & ANALYSIS

Test 1: Creating 50 files

In this test we created a function called saturatingInodes() which successfully create 50 files showing that put file system is capable of storing atleast 50 files. This function can be directly called after creating and mounting file system. We have commented this function in our int main function where we used it.



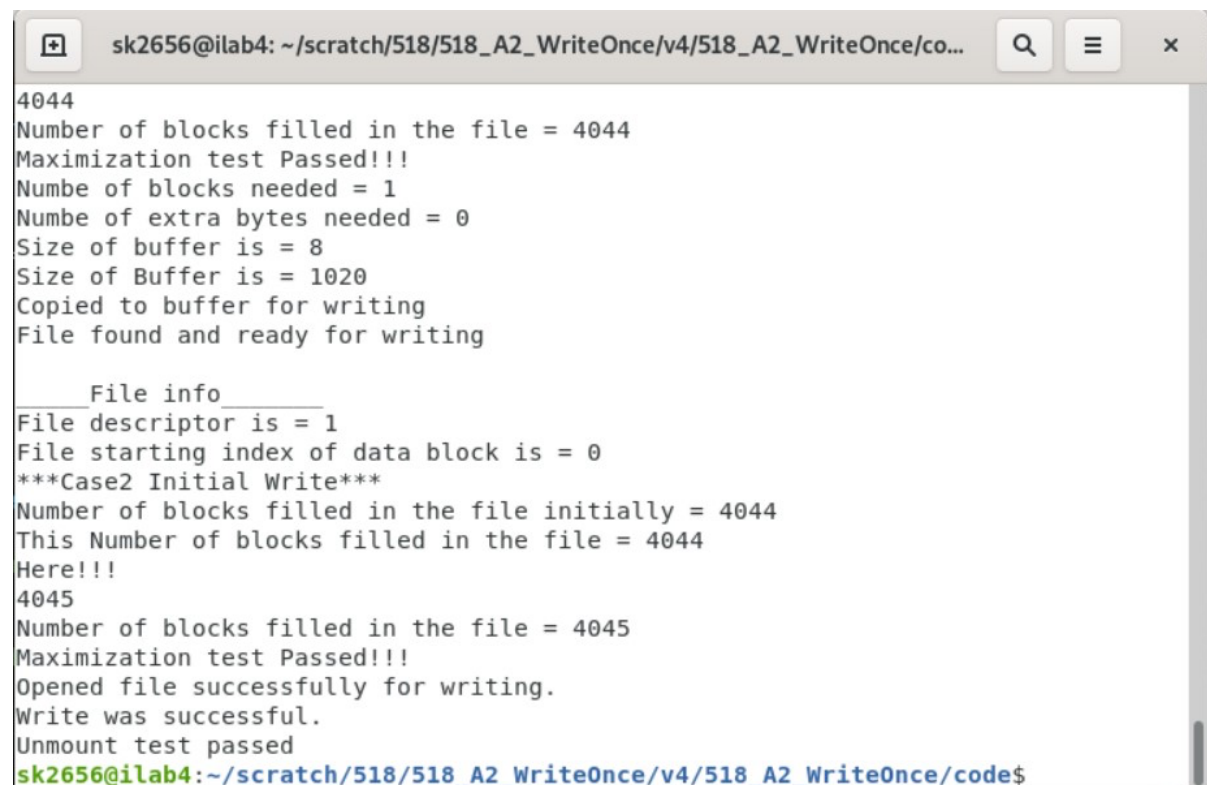
```
sk2656@ilab4: ~/scratch/518/518_A2_WriteOnce/v3/518_A2_WriteOnce/co...
rom pointer without a cast [-Wint-conversion]
665 | ((datablock*)cursorData)->next = cursorData +102
4;
|
writeonceFS.c:676:52: warning: assignment to 'int' from 'char *' makes integer f
rom pointer without a cast [-Wint-conversion]
676 | ((datablock*)cursorData)->next = cursorData +1024;
|
sk2656@ilab4:~/scratch/518/518_A2_WriteOnce/v3/518_A2_WriteOnce/code$ ./a.out
Compilation passed!!

Disk File created.....
Opened file successfully for reading.
Read was successful.
DATA Region starts from -1711739888
Initial Write was successful.
Mount test passed
Creating 50 files
checking if all files are created
Created 50 files.test passed!!!!!!!
Opened file successfully for writing.
Write was successful.
Unmount test passed
sk2656@ilab4:~/scratch/518/518_A2_WriteOnce/v3/518_A2_WriteOnce/code$
```

Our super block and inode region combined only of size 52224bytes(~52KB) which leaves us with 4045 1KB blocks for the data region which is slightly less than 4MB. We can easily extend our inode region upto 2MB as per assignment instructions to accommodate a greater number of files(inodes). But then maximum allocation size will also reduce to 2MB.

Test 2: Maximum allocation size for a single file

In this test we will check how many data blocks we were able to fill in one single file. For this we created a function called maximization which can be called after `wo_mount()`. It will create a single file in write mode and will write maximum amount of data blocks that we can which is 4045. It is commented in the main code.

A terminal window screenshot showing the output of a file maximization test. The window title is "sk2656@ilab4: ~/scratch/518/518_A2_WriteOnce/v4/518_A2_WriteOnce/co...". The output text is as follows:

```
4044
Number of blocks filled in the file = 4044
Maximization test Passed!!!
Numbe of blocks needed = 1
Numbe of extra bytes needed = 0
Size of buffer is = 8
Size of Buffer is = 1020
Copied to buffer for writing
File found and ready for writing

    File info
File descriptor is = 1
File starting index of data block is = 0
***Case2 Initial Write***
Number of blocks filled in the file initially = 4044
This Number of blocks filled in the file = 4044
Here!!!
4045
Number of blocks filled in the file = 4045
Maximization test Passed!!!
Opened file successfully for writing.
Write was successful.
Unmount test passed
sk2656@ilab4:~/scratch/518/518_A2_WriteOnce/v4/518_A2_WriteOnce/code$
```

As we can see from the results, we are successfully able to allocated maximum number of data blocks 4045 to a single file which is around $4045 \times 1024 \times 1024$ bytes which is more than 2MB. This satisfies the 2MB data region constraint for the assignment.

Test 3 : Mount, Create, write, close, unmount, mount, read unmount

In this test we first mount our file system which is opened for the first time so all the data structures are written successfully. Next, we create a file in write mode and write few data blocks in it and close it and unmount our file system.

Then we mout out file system again to a new memory location which will now do a sanity check for disk failures and if passed, mount it to a memory location. Now we open the same file we wrote to earlier and try reading the data.


```
sk2656@ilab4: ~/scratch/518/518_A2_WriteOnce/v3/518_A2_WriteOnce/co...
File opened !!
File created and opened!! File Handle is = 1
File found and ready for reading
We are at = -1822206960
Address of block ysqaX w

0 th block printed
String Copied to buffer
__Printing buffer__

ergqemx bkiogtrkca uxlypfdobw yelzyofmxl vysndgjwlb prpylsbxgr lvpcbildvp pmjhupx
mqq nydkmaysbb wiwiefbjth lkuhjahvas xxbxivogen pdsdqvrtz vkgfxepbnl cfucbwrrvp
zinujpwxvr wvjwmjegws euiigjklxx kbneondgfl idvlkliagn efqjdqkrrh ranhfsmqzq ez
pxfnddpt wadqvltrls sijoajindx zkodcryyi iqufdzxmlic sqcmkbtvej tvyngzfmfr asnfd
biimk ajinaavraw gprbanubxy emcjlwufgw ydpsypklui hbildwxrzs phmejtcyxc gnlxedgd
fu iuytmyirkx dowdueensn hjfxdawzom zwleshnivb jymjqtqqcu yesastidhj tsdkatdnll
puogcvawvs qugaoyysqaX w
Read test Passed!!!
File closed!!!
close test passed
Opened file successfully for writing.
Write was successful.
Unmount test passed
sk2656@ilab4:~/scratch/518/518_A2_WriteOnce/v3/518_A2_WriteOnce/code$
```

As we can see from the results, the our file system is free from disk failures and we are able to retrieve or read the store data properly without any data loss.