Journal of Universal Computer Science, vol. 11, no. 1 (2005), 20-36
submitted: 15/7/04, accepted: 6/12/04, appeared: 28/1/05

Low-Intrusive Consistent Disk Checkpointing: A Tool for Digital Forensics[1]

Sriranjani Sitaraman
(Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083-0688.
ginss@student.utdallas.edu)

S. Venkatesan
(Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083-0688.
venky@utdallas.edu)

Abstract: An important problem in digital forensics is to record a checkpoint of a disk drive mounted as a file system on a host machine without disrupting the disk's normal operations. We present a checkpointing methodology for a disk that has a Unix-like file system. While our algorithm is built around the Unix file system, it can be used to checkpoint disks formatted for other file systems such as NTFS, etc. Our algorithm satisfies several correctness conditions.
Key Words: Checkpointing, Backup, Consistency, Snapshot, Digital Forensics, File system, Operating System.
Category: D.4.6, D.4.3

1 Introduction

Digital forensics involves collection and analysis of digital data within an investigative process [Civie and Civie 1998, Casey 2000]. An important problem is collecting evidence in the least intrusive manner. In this paper, we consider the problem of checkpointing the contents of a disk drive, which contains a Unix-like file system, without disrupting the machine's normal operations and present a low-intrusive disk checkpointing method. The checkpointing process involves recursively traversing the file system tree in the source disk starting from the root inode. Every time a file is encountered for the first time, its inode and contents are copied to the destination disk. The checkpointed disk can be mounted as a file system in a different computer system and the contents can be examined. For the checkpoint to be of use in cyber courts and to be of practical value, the checkpoint must satisfy several correctness conditions.

Assume that the checkpointing operation begins at time T_I and ends at time T_Φ. Consider a file f whose complete pathname is "/p_n/p_{n-1}/. . ./p_2/p_1/f" where p_1 is the

[1] A preliminary version of this paper appears in the Proceedings of International Conference on Information Technology: Coding and Computing (ITCC'04), Volume 1, April 5-7, 2004, Las Vegas, Nevada, USA.

parent of f, p_2 is the parent of p_1, etc. We call file f a *static_path* file with respect to the checkpointing process (that executes between T_I and T_Φ) if neither file f's path nor f's contents are modified while checkpointing is in progress. Note that any of the directories $p_n, \ldots p_1$ can be renamed (without changing the associated inode), and f is still a *static_path* file.

Our checkpointing method satisfies the following properties:

1. The duration from T_I to T_Φ can be stretched so that the disk throughput for normal system operation is not brought down drastically because of excessive disk accesses by the checkpointing operation. We allow the checkpointing algorithm to choose a parameter so that the additional load on the disk can be changed.

2. Checkpointing must be low-intrusive
   - No file in the disk can be created, modified, or deleted by the checkpointing process.
   - Access to the files by the processes of the computer system cannot be delayed indefinitely.
   - File attributes (ownership, time of modification, etc.) cannot be changed by the checkpointing algorithm.

3. If file f is a *static_path* file with respect to the duration of checkpointing, [T_I, T_Φ], then f is checkpointed.

4. Every directory d that is a *static_path* file during the time interval [T_I, T_Φ] must be checkpointed. In addition, each file f of a checkpointed directory d must also be checkpointed (even if f is deleted before time T_Φ by an application).

5. The checkpointed disk must be consistent, and it must be possible to mount it as a file system.

The paper is organized as follows: In [Section 2], we present the model under consideration. In [Section 3], we describe the checkpointing algorithm in detail. The correctness of the checkpointing algorithm is presented in [Section 4] and the properties of the checkpointing method are explained in [Section 5]. [Section 6] compares our work with related work and concludes the paper.

2 System Model

Consider a typical Unix computer system with disk partitions formatted with the second extended file system (Ext2). Each file or directory is represented by two entities: an inode and a set of data blocks [Bach 1986]. The inode contains vital information about the file such as the owner, access permissions, disk addresses of data blocks which store file contents, etc. Data blocks contain file data or block addresses. Throughout the paper, the term data block of a file system refers to either (i) a (direct) block that contains file contents or (ii) an indirect block that contains addresses of direct blocks. Starting from the inode, we can obtain the entire file contents by systematically accessing all the data blocks using the addresses found in the inode. The addresses in inode may be direct block addresses or indirect (single, double or triple) block addresses. All of these blocks are said to *belong to* the inode. The file system is organized as a tree with "/" as the root and inode number 2 is assigned to the root in most systems.

Unix file system consistency requires the following conditions:

1. Each data block must belong to an inode (thereby storing file contents) or must belong to the free list. No two inodes can address the same block, and all blocks must appear somewhere (allocated to an inode or in the free list).

2. An allocated inode must be associated with a file or directory in the file system and must not appear as free.

3. An inode's link count must be equal to the number of directory entries pointing to it.

4. The number of free data blocks and the number of free inodes in the super block must conform to the numbers that exist on disk.

3 Checkpointing Algorithm

We explain the checkpointing algorithm in steps; initially we assume that there are no modifications to the file system when checkpointing is in progress; subsequently we explain how the algorithm copes with file modification, deletion, etc.

3.1 Outline

The checkpointing process starts from the root inode of the source file system, and checkpoints all the files by traversing the file system tree recursively in a preorder manner. When a file is visited for the first time, it is checkpointed. Checkpointing a file involves copying the inode and the contents (the data blocks of the file) to the destination disk. We use a set, *SIC* (Set of Inodes to be Checkpointed), to represent the list of inodes of the files to be checkpointed. When checkpointing a directory *d*, all the inodes of the files appearing in *d* are added to *SIC*.

Consider the example in [Fig. 1] showing the state of checkpointing procedure at an intermediate point in its execution. In the figure, the labels represent inodes (and not file/directory names). The nodes shown black (*a*, *b*, *e* and *g*) represent checkpointed files (directories in this case). The others (*c*, *d*, *f*, *h* and *j*) are in *SIC*, to be checkpointed. For example, inodes *c*, *d*, and *b* are added to *SIC* at the time of checkpointing *a*.

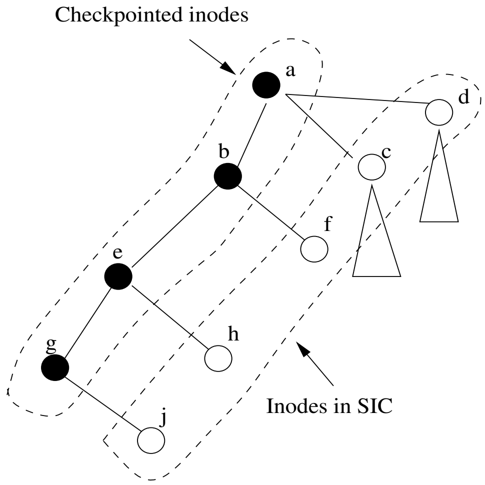

Figure 1: File system tree traversal by the Checkpointing algorithm.

When checkpointing a file, its inode is locked to prevent modification of the inode and file contents by other processes. This ensures consistency of the files at the system call level in the checkpointed copy. *SIC* is accessed by more than one entity and hence the access to *SIC* is controlled. In addition to the checkpointing process, some external entities such as user or kernel processes access *SIC*. A separate thread called Inode_List_Manager manages *SIC* and synchronizes access to *SIC* by different processes. The Inode_List_Manager presents a single interface for exclusive access to *SIC* to the various entities and does not allow more than one entity to modify *SIC* at the same time.

Note that the disk checkpointing process places an additional load on the system. The components of the system used by the checkpointing algorithm include (1) main memory, (2) disk drive, (3) system bus, and (4) CPU (for initiating DMA and executing parts of the checkpointing code). If the system is operating under a certain load such that the utilization of these four components is under 100%, then disk checkpointing can be done without any appreciable reduction in system performance as perceived by the executing applications. The disk checkpointing rate can be increased gradually (by decreasing the time between successive disk reads/writes or the time between a batch of disk operations). The appropriate value of the disk checkpointing rate so that these four components are near 100% utilization can be found by doing a bottleneck analysis with techniques similar to that of Denning and Buzen [Denning 1978].

The disk checkpointing process uses a number of data structures in order to to keep track of checkpointing information of the source disk and the destination disk. These data structures are local to the checkpointing process and are maintained till the checkpointing process terminates. These arrays are not stored on the disk. The data structures used by the checkpointing process include:

SIC: Set of Inodes to be Checkpointed
inode_copied[]: keeps track of checkpointed inodes; initially all 0
src_blk_copied[]: track checkpointed data blocks in the source disk; i^{th} element = 1 if block *i* of source disk has been copied; initially all 0
dest_blk_copied[]: track checkpointed data blocks in the destination disk; i^{th} element = 1 if block *i* of destination disk has checkpointed contents; initially all 0
src_blk_in_dest[]: array used to keep track of destination block into which the given source block is copied; has one element for each block in source disk; initially all 0
reused_inode[]: keeps track of inodes that were checkpointed and then freed; initially all 0

Procedure 1. *LOW-INTRUSIVE-CHECKPOINT*

```
/* only concurrent reads to file system are assumed; no writes */
/* choose parameters x and y */
spawn Inode_List_Manager; /* initializing SIC ← root_inode */
request Inode_List_Manager for an inode i from SIC
while (inode i is valid and not already checkpointed) {
   lock i
   copy i to destination disk
   while (there are more data blocks of file (with inode i) to be copied to destination) {
      copy x blocks (direct and indirect data blocks) using addresses found in i
      sleep for y seconds
   }
   if (i.type = directory)
      /* files in i must be checkpointed; add their inodes to SIC */
      for each child inode, c, in directory i {
         request Inode_List_Manager to add c to SIC
      }
   request Inode_List_Manager to remove i from SIC
   unlock i
   inode_copied[i] ← 1
   request Inode_List_Manager for next inode i
}
```

The algorithm in [Procedure 1] successfully checkpoints the entire file system if no other entity modifies the file system.

3.2 Write operations by users while checkpointing

Here we assume that user processes can concurrently modify contents of existing files while checkpointing is in progress. Assume, for now, that files are not deleted. Locking

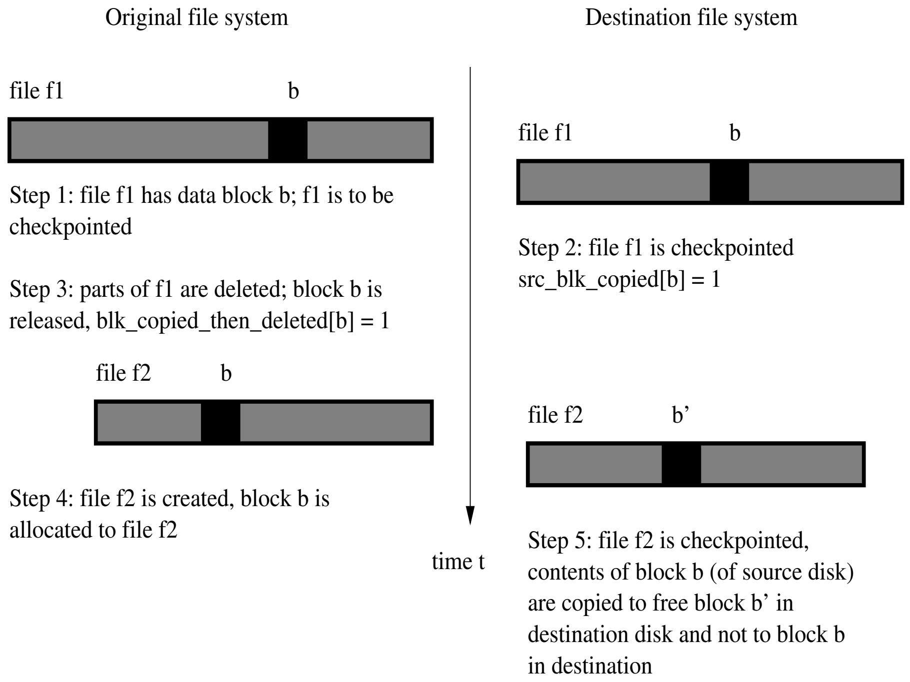

Figure 2: Checkpointing a file that shrinks in size.

an inode when the corresponding file is being checkpointed gives a consistent snapshot of the file. The data blocks must be carefully checkpointed since the data blocks of a shrinking file, which was already checkpointed, may be subsequently allocated to a growing file that is not yet checkpointed. This is illustrated in [Fig. 2]. When checkpointing file f_1 [see Step 2], the contents of data block b in the source disk are copied to block b in the destination disk. In Step 3, a process deletes parts of f_1 which results in the release of block b to the free blocks list. In Step 4, this free block b is subsequently allocated to a newly created file f_2. While checkpointing file f_2 at a later time [see Step 5], if the contents of block b of the source disk are copied to block b in the destination disk, then an inconsistency is created for file f_1 in the destination disk. In order to avoid inconsistency of the destination file system, we track the data blocks that have been checkpointed.

When we encounter a data block that needs to be checkpointed for the second or subsequent time, we copy its contents to a free block b' ($\neq b$) on the destination disk, assuming that the destination disk has larger capacity. We record the mapping of block b (in source disk) to b' (in destination disk) in array *src_blk_in_dest*[]. This mapping may be used for reconstructing the inode and disk address association during the forensic analysis of the checkpoint that is obtained.

Procedure 2. *CONCURRENT-WRITES*

```
/* executed when checkpointing contents of file whose inode is i */
for each block b of inode i {
   if (dest_blk_copied[b] = 1) {
      find a free block b′ in destination disk
      copy b in source disk to b′ in destination disk
      replace disk address b to b′ in the appropriate part of the file
            (either inode or indirect block) in the destination disk
      dest_blk_copied[b′] ← 1;
      src_blk_in_dest[b] ← b′; /* mapping */
   }
   else {
      copy b in source disk to b in destination disk
      dest_blk_copied[b] ← 1;
      src_blk_in_dest[b] ← b; /* mapping */
   }
   src_blk_copied[b] ← 1;
}
```

3.3 Hard links and Symbolic links

Hard links and symbolic links will be preserved during checkpointing. Assume there are *n* hard links to an inode *i*. Contents of inode *i* and the associated data blocks are copied to the destination disk only when we encounter *i* for the first time. For this purpose we keep track of those inodes that have already been checkpointed, using the boolean array *inode_copied*[]. Assume that directory *d* contains file *c* (file *c*'s inode is *i*) and we are checkpointing *d*. If inode *i* of *c* has already been checkpointed, there are two cases: File *c* may be a hard link or inode *i* was released to the free inode list after *i* was checkpointed and later assigned to file *c*. In the former case that *c* is a hard link, we keep the entry for *c* in its directory *i* only if *c* was created before T_I. In the latter case, inode *i* in the source disk and inode *i* in the checkpoint disk are different. In this case, the reused inode is not checkpointed and *c*'s directory entry from its parent *d* is removed (in the checkpoint of *d*) to avoid inconsistency.

Symbolic links are checkpointed like a regular file.

Procedure 3. *CHECKPOINT-HARD-LINKS()*

```
/* executed when checkpointing the inode i */
if (d.type = directory)
   /* add inodes of files in this directory to SIC */
   for each child file, c, with inode i_c in directory d
      if (inode_copied[i_c] = 0)
```

```
        /* c is not already checkpointed */
        request Inode_List_Manager to add i_c to SIC
    else
        if ( reused_inode[i_c] = 1 )
            remove entry for c in the checkpointed copy of d
        else
            /* c is a hard link */
            if (c was created after checkpointing started, T_I)
                remove entry for c in the checkpointed copy of d
            else
                /* c existed before T_I, so c has to be checkpointed,
                we have already preserved the directory entry for c
                in d but it is not required to update contents of c */
```

3.4 Mount points and device special files

During the traversal of the file system tree, if we visit an inode that is a mount point for another file system, then we do not traverse and checkpoint the mounted file system. Instead, we checkpoint the mount point as an empty directory. Similarly, when block and character special files are encountered, only the inodes are checkpointed and not their contents.

3.5 File Deletion

Assume that file *f* (inode i_f) in directory *d* (inode i_d) is being deleted. Two cases are possible:

Case i: Parent inode i_d is already checkpointed. In this case if i_f is not yet checkpointed, then $i_f \in$ *SIC*. Hence one occurrence of i_f is removed from *SIC* and the directory entry for *f* in the checkpointed copy of *d* is removed. Multiple entries for the same inode may be found in *SIC* because the checkpointing process may visit multiple hard links associated with the same inode before the inode is checkpointed to the destination disk. If i_f has been checkpointed, then i_f is not in *SIC* and *unlink*() can remove the link without any changes to the checkpoint.

Case ii: Parent inode i_d is not yet checkpointed. If i_f is already checkpointed, then it must have been done because of a link to i_f in another directory i'_d. Hence, no additional processing is necessary. If i_f has not been checkpointed, no additional processing is required in this case also since its parent has not been checkpointed. The *unlink*() system call removes the link from the file system.

We next show the modifications to the $unlink$() system call. Access to *SIC* by different processes and threads is synchronized by Inode_List_Manager.

Procedure 4. *UNLINK (f, d) /* f, the file that is unlinked, d, its parent */*

/ this code is to be added to the Unix system call unlink() */*
if ((*inode_copied*[i_f] = 1) && (i_f.linkcount = 1))
  /* this inode is going to be freed */
  reused_inode[i_f] $\leftarrow$ 1;
if (*inode_copied*[i_d] = 1)
  /* parent has been checkpointed */
  if (*inode_copied*[i_f] = 0) {
    / i_f is not checkpointed */*
    request Inode_List_Manager to remove $\{i_f\}$ from *SIC*
    remove entry for i_f in checkpointed copy of i_d
  }
unlink(*f*) */* execute unlink() system call on file f */*

3.6 File Creation

A small modification to the *ln*() system call is required to keep track of hard links that are created while checkpointing is in progress, i.e., after T_I. This is used by the checkpointing process when it encounters a hard link, as explained in [Section 3.3].

3.7 Preserving inconsistencies of source file system

Any inconsistencies (with respect to sharing of data blocks by two files) that exist in the source file system before checkpointing starts, must be preserved in the destination file system. For this purpose, we use boolean array *blk_copied_then_deleted*[] (initially all false) to identify the blocks that were checkpointed and then released when an entity outside the checkpointing process executes the *unlink*() system call. This information can be used later to determine if a block was reallocated to another file, or an inconsistency existed in the source file system. If the source file system has an inconsistency, it is preserved in the checkpoint.

3.8 Initializing the destination file system

After the checkpointing algorithm terminates, the super block, the block bitmap, the inode bitmap, etc. of the destination disk must be set correctly. The link count of an inode *i* in the destination file system may not properly reflect the number of hard links (directory entries) pointing to *i*.

4 Correctness

Assume that the algorithm begins at time T_I and terminates at time T_Φ. We assume that the number of files and size of each file in the source file system is finite. Consider a file f with a pathname "/p_n/p_{n-1}/. . ./p_2/p_1/f" where p_1 is the parent of f, p_2 is the parent of p_1, etc. File f is a *static_path* file with respect to the checkpointing process (that executes between T_I and T_Φ) if f satisfies the following conditions:

(i) The pathname of f is preserved between T_I and T_Φ, i.e., neither f nor any of its ancestors ($p_1, p_2, \ldots p_n$) in this file system tree have been moved or deleted.

(ii) File f or any of its ancestors (p_1, p_2, $\ldots$ p_n) in the file system tree may be renamed between T_I and T_Φ without changing the associated inode. In other words, one of f's ancestors, say p_2, that was associated with an inode number 5, may be renamed as q_2, where q_2 also refers to the same inode 5 (as it existed before the rename operation).

We shall now show that the checkpointing algorithm will correctly checkpoint a *static_path* file.

Theorem 1. *If f is a static_path file during the time interval of checkpointing [T_I, T_Φ], then f is part of the final checkpoint under the same hierarchy and its contents are copied correctly.*

Proof. We prove this by induction on the height *h*, of *f*, in the file system.

Basis (*h* = 0): In this case, the root directory, "/" , is the file under consideration. Locking inodes ensures that a consistent copy of the root directory is checkpointed.

Hypothesis: Each file f_h at height *h* is checkpointed.

Induction Step: To prove: Each file f_{h+1} at height *h*+1 is checkpointed. Let f_h be an arbitrary file at height *h*. Let f_{h+1} be a file which is a child of f_h. Since f_{h+1} is not deleted or moved, the inode for f_{h+1} is added to *SIC* when f_h is checkpointed. By the induction hypothesis, f_h is checkpointed. After checkpointing a file, its inode is removed from *SIC*. Since the number of files in the file system is finite and the size of each file is finite, the algorithm terminates. Thus, f_{h+1} is guaranteed to be checkpointed and removed.

To show that the contents of a *static_path* file f (with respect to [T_I, T_Φ]) are preserved in the checkpoint, we observe that the inode of file f is locked when it is being checkpointed. We need to consider the following 2 cases:

Case 1: Each block of f in the source disk is copied to the same block in the destination disk (no translation of disk blocks is needed). In this case, all the data blocks corresponding to block addresses in inode of *f*, including those blocks pointed by addresses in indirect blocks, are copied to the destination disk and no block is missed. Also, all disk addresses are preserved. Hence the file content is preserved.

Case 2: Translation of blocks is needed. In this case, if there is translation of some block from b to b', then the corresponding disk address in the inode, or an indirect block, is also updated. Hence, file content remains intact.

The proof of [Theorem 1] follows. ■

Theorem 2. *If the source file system is consistent [see Section 2], then the destination file system is also consistent.*

Proof. For the destination file system, we prove the following assertions.

Assertion 1. *Free blocks of the destination file system do not contain file contents.*

Proof. The array, *dest_blk_copied* is initialized to 0. For each block i on the destination that has been copied from the source file system during the checkpointing, *dest_blk_copied*$[i] = 1$. Since all files are checkpointed, their contents are also copied. If *dest_blk_copied*$[i] = 0$, for some block i, then i is a free block.

Assertion 2. *If the source disk is consistent, a data block in the destination file system belongs to only one inode.*

Proof. Since the source file system is consistent, every data block in the source file system belongs to one inode only. A data block in the source file system, b, is copied to destination block b, only if *dest_blk_copied*$[b] = 0$. After copying, *dest_blk_copied*$[b]$ is set to 1. If a reused data block is encountered [see Section 3.2], then its contents are copied to another block b' that is free on the destination disk and *dest_blk_copied*$[b']$ is set to 1. Since there is no data block in the source file system such that it is referenced by two inodes, a data block can belong to one inode only.

Assertion 3. *All file attributes except the link count of the file's inode are preserved.*

Proof. Locking the contents of the entire inode while copying it ensures that all the file attributes (excepting the link count) are preserved. The link count is an inode attribute and depends on the number of files pointing to the inode. The inode link count in the destination file system is same as that in the source file system if hard links are not created or deleted. In all other cases, the link count may be different.

Assertion 4. *All symbolic links are preserved.*

Proof. A symbolic link is similar to a regular file with its own inode number having the path name of its target as its content. Each symbolic link is checkpointed like a regular file. Thus the semantics of symbolic link are preserved.

Assertion 5. *Hard links which are static_path files with respect to the checkpointing process are preserved as long as the hard links are not created or deleted between T_I and T_Φ.*

Proof. When an inode i is checkpointed, *inode_copied*$[i]$ is set to 1. If a file f, which has the same inode number i, is encountered subsequently when its parent directory d is being checkpointed, there are only two cases: f may be a hard link to inode i or the inode might have been released and subsequently reassigned to a newly created file as

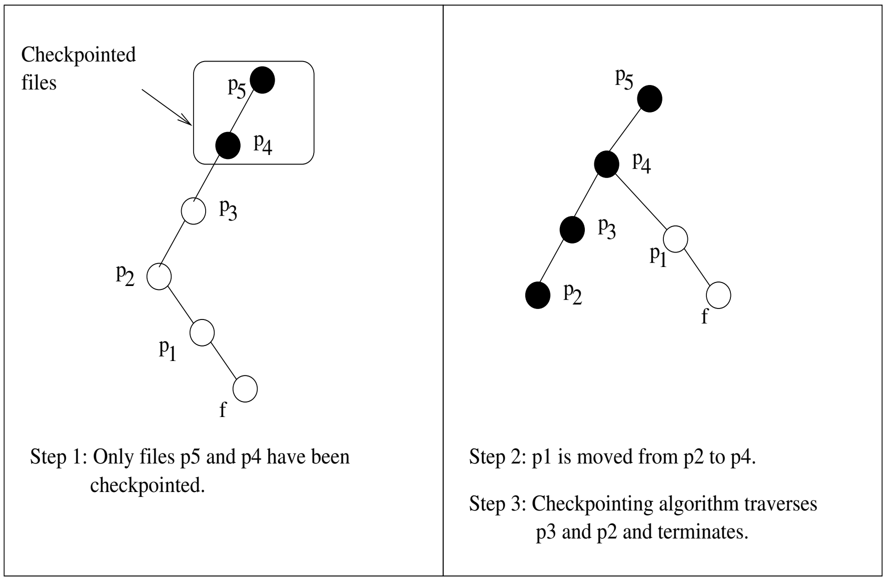

Figure 3: Files missed by the Checkpointing algorithm.

explained in [Section 3.3]. The latter case is identified by checking if *reused_inode*[i] is set to 1 (during the *unlink*() system call as shown in [Section 3.5]). If *reused_inode*[i] was 0 then the file f is a hard link to inode i. In this case, we preserve the entry for f in its directory d in the destination disk only if f was created before T_I. If f was created after T_I, then it is not necessary to checkpoint f as explained in [Property 2].

The proof of [Theorem 2] directly follows from the above assertions. ■

Theorem 3. *A data block whose contents never existed in the source file system, is never found in the checkpointed copy.*

Proof. Assume for contradiction that such a data block x exists in the checkpoint. Thus, *dest_blk_copied*[x] should be 1 because if it was 0 it would have been marked as free in the block bitmap. But by [Assertion 1], only the blocks that contain contents of a file in the source file system will have *dest_blk_copied*[] set to 1. This contradicts our claim that block x is part of the checkpoint. ■

5 Properties of the Algorithm

Consider a file f with pathname "/p_5/p_4/p_3/p_2/p_1/f" where p_1 is the parent of f, p_2 is the parent of p_1, etc. Files may be moved around in the file system tree in such a way that the checkpointing algorithm may terminate without copying the content of a file that existed from T_I to T_Φ. An example of this case is illustrated in [Fig. 3], where file *f* exists from T_I to T_Φ but is not being copied to the destination disk.

The example of [Fig. 3] shows that the checkpointing algorithm does not encounter file *f* during its execution because of the move operation that happened between T_I and T_Φ. Such a file *f* is not a *static_path* file with respect to [T_I, T_Φ]. When p_2 is checkpointed subsequent to the move operation, file p_1 is not found in p_2, and hence the inode of p_1 is not added to *SIC*. File p_1 cannot be accessed by the checkpointing algorithm through p_4 because the algorithm had already checkpointed p_4 and does not visit p_4 again.

We now describe some properties that the checkpointing algorithm satisfies.

Property 1. *For a directory d and files f_1 and f_2, if file f_1 is deleted (unlink()ed) from d, and subsequently f_2 is created in d, then both f_1 and f_2 will not appear in the checkpoint of directory d.*

Proof. Assume, for contradiction, that both f_1 and f_2 are part of the checkpoint of directory *d*. Since inode of *d* is locked when its children are added to *SIC*, only the following sequences of events may occur because of the atomic execution of the system calls.

Case 1: unlink f_1; create f_2; checkpoint *d* (add children of *d* to *SIC*)
In this case, f_1 is not in *SIC* but f_2 is in *SIC*. Hence only f_2 appears in the checkpoint.

Case 2: unlink f_1; checkpoint *d*; create f_2
Here, both f_1, f_2 are not in *SIC*. Hence both f_1, f_2 will not appear in the checkpoint.

Case 3: checkpoint *d*; unlink f_1; create f_2
In this case, f_1 is in *SIC*, f_2 is not in *SIC*. Hence only f_1 appears in the checkpoint.

This suffices to show that [Property 1] is true for the checkpointing algorithm.

Property 2. *Files created, deleted or moved between T_I and T_Φ may or may not be part of the checkpoint.*

We consider all possible cases of file deletion with respect to the checkpointing process and show that the algorithm satisfies [Property 2] in all these cases:

Case 1: Suppose a file *f* is checkpointed at T_a and then deleted at time T_b ($T_I < T_a < T_b < T_\Phi$). In this case *f* exists in the checkpoint, as the inode of *f*, *i*, will not be in *SIC*. Also, it is correct to include file *f* in the snapshot, as it existed before the checkpointing process started at time T_I, and we already have a consistent copy of the file.

Case 2: The case where *f* is deleted while it is being checkpointed is similar to Case 1, as the delete operation on *f* can be done only when the checkpointing process releases the lock on *i*.

Case 3: If *f* is deleted when its parent was already checkpointed and *f* has not yet been checkpointed, then $i \in$ *SIC*. File *f* is not included in the snapshot as the unlink causes *i* to be removed from *SIC*, and the checkpointed copy of its parent is also updated to reflect the deletion of *f*.

Case 4: If f is deleted at time T_a ($T_I < T_a < T_\Phi$), when neither f nor its parent has been checkpointed, then f will not appear in the checkpoint. This is because, when the parent is checkpointed subsequently, the entry for file f will not be present and *i* is never included in *SIC*.

In a similar manner, it can be shown that a file that is created while checkpointing is in progress may or may not be checkpointed.

Property 3. *Hard links created and deleted between T_I and T_Φ may or may not be checkpointed.*

[Assertion 5] proves that *static_path* hard links will be preserved if they are not created or deleted when checkpointing is in progress. When an entity external to the checkpointing process removes a hard link f of an inode *i* which is not yet checkpointed and is in *SIC*, as explained in [Section 3.5], f is removed from *SIC* and thus is not preserved in the checkpoint. It will be preserved if f is removed after inode *i* was checkpointed. A link f to inode i that is created in a checkpointed directory d is not checkpointed and by [Property 2], it is acceptable to miss f from the checkpoint. In all these cases, [Property 3] is satisfied.

Property 4. *Let S_0 be the state of a file f just before T_I. Let seq = (e_i: $0 \leq i \leq n$) be a sequence of events that modify the contents of file f (for example system calls such as read(), write(), etc.). S_{n+1} is the state of f just after T_Φ. Then the state of the file in the checkpoint is exactly S_i for some i ($0 \leq i \leq n$).*

This property is ensured since the inode is locked while it is being checkpointed.

Property 5. *A file f existing in the destination disk has an inode i, only if that file had an inode number i in the source disk.*

When an inode *i* is checkpointed, *inode_copied*[*i*] is set to 1. If a hard link to the same inode is encountered subsequently, it is preserved in the checkpoint. But if the checkpointed inode *i* is freed because all the files pointing to it are removed from the file system and then reassigned to a new file created subsequently [see Section 3.3], then *i* is recognized as a reused inode and hence *i* is not checkpointed. In both cases no new inode number is assigned in the destination file system. Hence [Property 5] holds. The association between inode numbers on the source and the destination disks may be a valuable forensic evidence.

6 Related Work and Conclusion

Checkpointing process states has been successfully used for fault-tolerance and has been studied extensively in distributed systems [Chandy 1985][Koo 1987] and mobile systems [Prakash and Singhal 1996]. File system checkpointing approaches presented in [Pei et al. 2000] try to minimize the overhead of the checkpointing process, but do not address the issue of reduction in system performance.

Traditional backup systems are built to recover files that are accidentally deleted. Several projects (Coda at CMU and Echo at DEC/SRC) explored backup by replication [Hisgen 1990, Satyanarayanan et al. 1990]. The Unix file system does provide the capability to take periodic backups [Preston 1999]. However, these are only taken with the aim of protecting data and they require the entire file system to be quiescent.

File systems such as the Elephant file system [Santry 1999], Plan-9 [Presotto 1992], AFS [Howard et al. 1988] and WAFL [Brown et al., Hitz et al. 1994] provide the capability to take periodic file system checkpoints of online file systems. WAFL is a file system similar to Unix and has inherent snapshot capability allowing snapshots of the file when users are modifying files. WAFL needs sizable free space for copying blocks and locks the entire file system. Valuable forensic evidence may be lost as disk and inode number associations between the source and destination disks are not maintained even if only small parts of the files are changed, and file system inconsistencies of the source are not maintained in the checkpoint.

In comparison, our work presents a low-intrusive disk checkpointing algorithm that can be used in the digital forensic analysis of a hard disk. In addition, our algorithm can be used to take periodic backups of the hard disk without locking the file system. The algorithm satisfies several correctness conditions and its speed can be fine-tuned so that other processes can access the disk without long waits. The algorithm attempts to preserve disk address associations (when files are not modified) and some file system inconsistencies that may be useful in digital forensic analysis. Our results can also be integrated into a network operating system.

Usually remnants of deleted files are found in unallocated sections (inodes and data blocks) of the storage medium. Our disk checkpointing algorithm, as described in this paper, obtains a snapshot of only those data blocks or inodes that are associated with the file system tree. In order to maximize the amount of digital evidence collected, we can augment disk checkpointing with functionality that copies the contents of all unallocated data blocks into a log file in a manner that can be easily analyzed by available utilities.

Currently, we are implementing the checkpointing method as a tool built in the Linux kernel (version 2.4). The details of the implementation, when complete, along with performance measures will appear elsewhere.

Acknowledgments

This work is supported in part by a grant from the Department of Navy.

References

[Alagar et al. 1997] Alagar, S., Rajagopalan, R., Venkatesan, S.: "Integrating files and processes: A comprehensive approach to checkpointing"; Proceedings of Fifth International Conference on Advanced Computing, pages 453–458, December 1997.

[Bach 1986] Bach, Maurice J.:"The Design of the Unix Operating System"; Prentice Hall PTR, ISBN: 0132017997.

[Brown et al.] Brown, K., Katcher, J., Walters, R., Watson, A.: "SnapMirror and SnapRestore: Advances in Snapshot Technology"; Network Appliance, Inc.

[Bandel 1997] Bandel, David A.:"Disk Maintenance under Linux (Disk Recovery)"; Linux Journal, January, 1997.

[Bhargava and Lian 1988] Bhargava, B., Lian, S.: "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems"; Proceedings Seventh Symposium Reliable Distributed Systems, 1988, pp. 3-12.

[Casey 2000] Casey, Eoghan: "Digital Evidence and Computer Crime: Forensic Science, Computers and the Internet"; Academic Press, London, 2000.

[Chandy 1985] Chandy, Mani K., Lamport, Leslie: "Distributed snapshots: determining global states of distributed systems"; ACM Transactions on Computer Systems (TOCS), Vol. 3, Issue 1, February, 1985.

[Civie and Civie 1998] Civie, V., Civie, R.: "Future technologies from trends in computer forensic science"; Proceedings of the Information Technology Conference, pp. 105-108, September, 1998.

[Denning 1978] Denning, Peter: "The Operational Analysis of Queuing Network Models"; ACM Computing Surveys (CSUR), pp. 226-261, Vol. 10, Issue 3, September 1978.

[Hisgen 1990] Hisgen, A., Birrell, A., Jerian, C., Mann, T., Schroeder, M., Swart, G.: "Granularity and Semantic Level of Replication in the Echo Distributed File System"; Proceedings of the Workshop on Management of Replicated Data, IEEE, pp. 2-4, Houston, Texas, November, 1990.

[Hitz et al. 1994] Hitz, D., Lau, J., Malcolm, M.: "File system design for a file server appliance"; Proceedings of the 1994 Winter USENIX Technical Conference, pp. 235-245, San Francisco, CA, January, 1994.

[Howard et al. 1988] Howard, John H., Kazar, Michael L., Menees, Sherri G., Nichols, David A., Satyanarayanan, M., Sidebotham, Robert N., West, Michael J.: "Scale and performance in a distributed file system"; ACM Transactions on Computer Systems, pp. 51-81, Vol. 6, Issue 1, February 1988.

[Koo 1987] Koo, R., Toueg, S.: "Checkpointing and Rollback-Recovery for Distributed Systems"; IEEE Transactions on Software Engineering, pp. 23-31, Vol. 13, Issue 1, January, 1987.

[Open File Manager] "Open File Manager: Preventing Data Loss During Backups Due to Open Files"; St Bernard Software Inc.

[Patel and Ciardhuain 2002] Patel, A. and Ciardhuain, S.: "The impact of forensic computing on telecommunications"; IEEE Communications Magazine, pp. 64-67, Vol. 38, Issue 11, ISSN: 0163-6804, November, 2002.

[Pei et al. 2000] Pei, D., Wang, D., Shen, M., Zheng, W.: "Design and implementation of a low-overhead file checkpointing approach"; Proceedings of The Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, pp. 439-441, Vol. 1(14-17), May, 2000.

[Prakash and Singhal 1996] Prakash, R., Singhal, M.: "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems"; IEEE Transactions on Parallel and Distributed Systems, pp. 1035-1048, October 1996.

[Presotto 1992] Presotto, David: "Plan 9"; Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, USENIX Association, pp. 31-38, Seattle, WA, USA, April, 1992.

[Preston 1999] Preston, Curtis W.: "UNIX Backup and Recovery"; O'Reilly and Associates.

[Santry 1999] Santry, Douglas S., Feeley, Michael J., Hutchinson, Norman C., Veitch, Alistair C., Carton, Ross W., Ofir, Jacob: "Deciding when to forget in the Elephant file system"; ACM

SIGOPS Operating Systems Review, Proceedings of the 17th ACM symposium on Operating systems principles, Vol. 3, Issue 5, December, 1999.

[Satyanarayanan et al. 1990] Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: "Coda: a highly available file system for a distributed workstation environment"; IEEE Transactions on Computers, pp. 447-459, Vol. 39, Issue 4, April, 1990.

[Singhal and Shivaratri 1994] Singhal, Mukesh, Shivaratri, Niranjan G.: "Advanced Concepts in Operating Systems"; McGraw-Hill, Inc. and The MIT Press, ISBN: 0-07-057572-X, 1994.

[Spezialetti and Kearns 1986] Spezialetti, M., Kearns, P.: "Efficient Distributed Snapshots"; Proceedings of Sixth International Conference on Distributed Computing Systems, pp. 382-388, 1986.

[Stringer-Calvert 2002] Stringer-Calvert, David WJ.: "Digital Evidence"; ACM Press, ISSN: 0001-0782, New York, NY, USA, 2002.

[Venkatesan et al. 1997] Venkatesan, S., Juang, T.T.-Y., Alagar, S.: "Optimistic crash recovery without changing application messages"; IEEE Transactions on Parallel and Distributed Systems, pp. 263-271, Vol. 8, Issue 3, March, 1997.