

Data refinement of representation of a file

Karen Zee and Viktor Kuncak

December 12, 2009

Abstract

This document illustrates the verification of basic file operations (file creation, file read and file write) in Isabelle theorem prover [4]. We describe a file at two levels of abstraction: an abstract file represented as a resizable array, and a concrete file represented using data blocks.

Contents

1	Introduction	1
2	Arrays without bounds	2
3	Resizable arrays	3
4	Data refinement of representation of a file	4
4.1	Abstract File	4
4.2	Concrete File	5
4.2.1	Writing File	6
4.3	Reachability Invariants for Concrete File	7
4.4	Initial File Satisfies Invariants	7
4.5	Properties of Concrete File Operations	8
4.6	Concrete File Operations Preserve Invariants	9
4.7	Commuting Diagrams for Simulation Relation	12
4.7.1	Abstraction Function	12
4.7.2	Creating a File	12
4.7.3	File Size	12
4.7.4	Read Operation	12
4.7.5	Write Operation	12

1 Introduction

This document is based on [1], which explores the challenges of verifying the core operations of a Unix-like file system [5, 3]. The paper [1] formalizes

the specification of the file system as a map from file names to sequences of bytes, then formalizes an implementation that uses such standard file system data structures as i-nodes and fixed-sized disk blocks. The correctness of the implementation is verified by proving the existence of a simulation relation [2] between the specification and the implementation. The original effort of [1] started in Isabelle. The process of developing the proof in Isabelle helped to remove the initial bugs in the concrete and abstract models (though the proof has not been completed so far).

Here we present a completed proof for a simplified problem: data refinement of a single file. We present operations on both abstract and concrete files, define a function mapping concrete files to abstract files, and prove that this function is a simulation relation.

We use two libraries of arrays: arrays without bounds checks, which can be thought of as an array with an unbounded number of elements, and resizable arrays, which have a notion of the current size, but expand in response to array writes that are outside the current bounds.

2 Arrays without bounds

theory *CArrays* **imports** *Main* **begin**

For these arrays there is no built-in protection against reading or writing out-of-bounds.

types

'a cArray = nat => 'a

constdefs

makeCArray :: nat => 'a => 'a cArray
makeCArray arraySize fillValue index ==
if index < arraySize then fillValue else undefined

constdefs

readCArray :: 'a cArray => nat => 'a
readCArray array index == array index

constdefs

writeCArray :: 'a cArray => nat => 'a => 'a cArray
writeCArray array index value == array(index := value)

lemma *makeCArrayCorrect:*

index < arraySize ==>

readCArray (makeCArray arraySize fillValue) index = fillValue

by (*simp add: readCArray-def makeCArray-def*)

lemma *writeCArrayCorrect1*:
readCArray (writeCArray array index value) index = value
by (*simp add: readCArray-def writeCArray-def*)

lemma *writeCArrayCorrect2*:
index1 ~ = index2 ==>
readCArray (writeCArray array index1 value) index2 =
readCArray array index2
by (*simp add: readCArray-def writeCArray-def*)

end

3 Resizable arrays

theory *ResizableArrays* **imports** *Main* **begin**

These arrays resize themselves, padding with *fillValue*.

types

*'a rArray = nat * (nat => 'a)*

constdefs

fillAndUpdate :: nat => (nat => 'a) => nat => 'a => 'a => (nat => 'a)
fillAndUpdate len f i value fillValue j ==
if j=i then value
else if (len <= j & j < i) then fillValue
else f j

constdefs

raWrite :: 'a rArray => nat => 'a => 'a => 'a rArray
raWrite arr i value fillValue ==
(let len = fst arr;
f = snd arr
in
if i < len then (len, f(i:=value))
else (i+1, fillAndUpdate len f i value fillValue)
)

constdefs

cutoff :: 'a => 'a rArray => 'a rArray
cutoff fill arr ==
(fst arr,
% i. if i < fst arr then snd arr i else fill)

lemma *raWriteSizeSame* [*simp*]: *i < fst arr ==> fst (raWrite arr i value fillValue) = fst arr*

by (*simp-all add: raWrite-def fillAndUpdate-def Let-def*)

lemma *raWriteSizeGrows* [*simp*]: *(fst arr <= i) ==> fst (raWrite arr i value fillValue) = i+1*

by (*simp-all add: raWrite-def fillAndUpdate-def Let-def*)

lemma *raWriteContentChanged* [*simp*]: *snd (raWrite arr i value fillValue) i = value*

by (*simp-all add: raWrite-def fillAndUpdate-def Let-def*)

lemma *raWriteContentOld* [*simp*]: $[[j < \text{fst } \text{arr}; i \sim j]] \implies$
snd (raWrite arr i value fillValue) j = snd arr j

by (*simp-all add: raWrite-def fillAndUpdate-def Let-def*)

lemma *raWriteContentFill* [*simp*]: $[[\text{fst } \text{arr} < j; j < i]] \implies$
snd (raWrite arr i value fillValue) j = fillValue

by (*simp-all add: raWrite-def fillAndUpdate-def Let-def*)

end

4 Data refinement of representation of a file

theory *FileRefinement* **imports** *Main CArrays ResizableArrays* **begin**

We describe a file at two levels of abstraction: an abstract file, represented as a resizable array, and a concrete file, represented using data blocks. We consider the following operations:

```
makeAFS      :: AFile
afsRead      :: "AFile => nat ~=> byte"
afsWrite     :: "AFile => nat => byte ~=> AFile"
afsFileSize  :: "AFile => nat"
```

typedecl

— unit of file content

byte

consts

— value used for padding

fillByte :: *byte*

consts

blockSize :: *nat* — in bytes

numBlocks :: *nat* — total number of blocks in the file system

axioms

nonZeroBlockSize: *blockSize* > 0

nonZeroNumBlocks: *numBlocks* > 0

4.1 Abstract File

types

AFile = *byte rArray* — abstract file is a resizable array of bytes

constdefs

makeAF :: *AFile*
 — initial file has size 0
makeAF == (0, % *index*. *fillByte*)

constdefs

afSize :: *AFile* => *nat*
 — file size is the length of the resizable array
afSize *afile* == *fst* *afile*

constdefs

afRead :: *AFile* => *nat* ~=> *byte*
 — reading from a file looks up the byte, reporting *None* if the index is out of file bounds

afRead *afile* *byteIndex* ==
 if *byteIndex* < *fst* *afile* then *Some* ((*snd* *afile*) *byteIndex*) else *None*

constdefs

afWrite :: *AFile* => *nat* => *byte* ~=> *AFile*
 — writing to a file updates the file content and extends the file if there is enough space

afWrite *afile* *byteIndex* *value* ==
 if *byteIndex* *div* *blockSize* < *numBlocks* then
 Some (*raWrite* *afile* *byteIndex* *value* *fillByte*)
 else *None*

4.2 Concrete File

types

Block = *byte* *cArray* — array of *blockSize* bytes

record *CFile* =

fileSize :: *nat* — in bytes
nextFreeBlock :: *nat* — next block available for allocation
data :: *Block* *cArray* — array of up to *numBlocks* blocks

constdefs

makeCF :: *CFile*
 — initial file has no allocated blocks
makeCF ==
 (| *fileSize* = 0,
 nextFreeBlock = 0,
 data = *makeCArray* *numBlocks* (*makeCArray* *blockSize* *fillByte*)
 |)

constdefs

cfSize :: *CFile* => *nat*
cfSize *cfile* == *fileSize* *cfile*

constdefs

cfRead :: *CFile* => *nat* ~=> *byte*
 — Looks up correct data block and reads its content, if *byteIndex* is within bounds, else returns *None*.
cfRead *cfile* *byteIndex* ==
 if *byteIndex* < *fileSize* *cfile* then
 (let *i* = *byteIndex* div *blockSize* in
 (let *j* = *byteIndex* mod *blockSize* in
 Some (*readCArray* (*readCArray* (*data* *cfile*) *i*) *j*)))
 else *None*

4.2.1 Writing File

We first present some auxiliary operations.

constdefs

cfWriteNoExtend :: *CFile* => *nat* => *byte* => *CFile*
 — Writing to a file when *byteIndex* is within bounds.
cfWriteNoExtend *cfile* *byteIndex* *value* ==
 let *i* = *byteIndex* div *blockSize* in
 let *j* = *byteIndex* mod *blockSize* in
 let *block* = *readCArray* (*data* *cfile*) *i* in
cfile(| *data* :=
writeCArray (*data* *cfile*) *i* (*writeCArray* *block* *j* *value*) |)

constdefs

cfExtendFile :: *CFile* => *nat* => *CFile*
 — Writing to a file when *byteIndex* is out of bounds. Involves allocating a new block.
cfExtendFile *cfile* *byteIndex* ==
cfile(| *fileSize* := *Suc* *byteIndex*,
nextFreeBlock := *Suc* (*byteIndex* div *blockSize*) |)

The main file write operation.

constdefs

cfWrite :: *CFile* => *nat* => *byte* ~=> *CFile*
 — Writes the file at byte location *byteIndex*, automatically extending the file to that byte location if *byteIndex* is not within bounds.
cfWrite *cfile* *byteIndex* *value* ==
 if *byteIndex* div *blockSize* < *numBlocks* then
 if *byteIndex* < *fileSize* *cfile* then
 Some (*cfWriteNoExtend* *cfile* *byteIndex* *value*)
 else
 Some (*cfWriteNoExtend* (*cfExtendFile* *cfile* *byteIndex*) *byteIndex* *value*)
 else *None*

4.3 Reachability Invariants for Concrete File

constdefs

```
nextFreeBlockInvariant :: CFile => bool
nextFreeBlockInvariant state ==
  (fileSize state + blockSize - 1) div blockSize = nextFreeBlock state
```

constdefs

```
unallocatedBlocksInvariant :: CFile => bool
— This invariant of the implementation is needed to prove writeExtendCorrect.
It says that any unallocated block contains fillByte's.
```

```
unallocatedBlocksInvariant state ==
  ALL blockNum i .
  ~ blockNum < nextFreeBlock state & blockNum < numBlocks & i < blockSize
  --> data state blockNum i = fillByte
```

constdefs

```
lastBlockInvariant :: CFile => bool
lastBlockInvariant state ==
  ALL index .
  ~ index < fileSize state & nextFreeBlock state = index div blockSize + 1
  --> data state (index div blockSize) (index mod blockSize) = fillByte
```

constdefs

```
reachabilityInvariant :: CFile => bool
reachabilityInvariant cfile ==
  nextFreeBlockInvariant cfile &
  unallocatedBlocksInvariant cfile &
  lastBlockInvariant cfile
```

4.4 Initial File Satisfies Invariants

We prove each invariant individually and then summarize.

lemma *nextFreeBlockInvariant1*:

```
nextFreeBlockInvariant makeCF
```

apply (*simp add*: nextFreeBlockInvariant-def makeCF-def)

apply (*simp add*: nonZeroBlockSize)

done

lemma *unallocatedBlocksInvariant1*:

```
unallocatedBlocksInvariant makeCF
```

apply (*simp add*: unallocatedBlocksInvariant-def makeCF-def)

apply (*simp add*: makeCArray-def)

done

lemma *lastBlockInvariant1*:

```
lastBlockInvariant makeCF
```

by (*simp add*: lastBlockInvariant-def makeCF-def)

```

lemma makeCFpreserves: reachabilityInvariant makeCF
by (simp add: reachabilityInvariant-def
      nextFreeBlockInvariant1
      unallocatedBlocksInvariant1
      lastBlockInvariant1)

```

4.5 Properties of Concrete File Operations

```

lemma cfWriteNoExtendPreservesFileSize:
  [| index < fileSize cfile1;
    cfWrite cfile1 index value = Some cfile2
  |] ==>
  fileSize cfile2 = fileSize cfile1
apply (simp add: cfWrite-def)
apply (case-tac index div blockSize < numBlocks, simp-all)
apply (simp add: cfWriteNoExtend-def Let-def)
apply force
done

```

```

lemma cfWriteExtendFileSize:
  [|  $\sim$  index < fileSize cfile1;
    cfWrite cfile1 index value = Some cfile2
  |] ==> fileSize cfile2 = Suc index
apply (simp add: cfWrite-def)
apply (case-tac index div blockSize < numBlocks, simp-all)
apply (simp add: cfWriteNoExtend-def cfExtendFile-def Let-def)
apply force
done

```

```

lemma fileSizeIncreases:
  cfWrite cfile1 index value = Some cfile2
  ==> fileSize cfile1 <= fileSize cfile2
apply (simp add: cfWrite-def)
apply (case-tac index div blockSize < numBlocks, simp-all)
apply (case-tac index < fileSize cfile1, simp-all)
apply (simp-all add: cfWriteNoExtend-def cfExtendFile-def Let-def)
apply force
apply force
done

```

```

lemma nextFreeBlockIncreases:
  [| nextFreeBlockInvariant cfile1;
    cfWrite cfile1 index value = Some cfile2
  |] ==> nextFreeBlock cfile1 <= nextFreeBlock cfile2
apply (simp add: cfWrite-def)
apply (case-tac index div blockSize < numBlocks, simp-all)
apply (case-tac index < fileSize cfile1, simp-all)
apply (simp-all add: cfWriteNoExtend-def cfExtendFile-def Let-def)
apply force

```

```

apply (simp add: nextFreeBlockInvariant-def)
apply auto
apply (subgoal-tac nextFreeBlock cfile1 =
  (fileSize cfile1 + blockSize - Suc 0) div blockSize, simp-all)
apply (subgoal-tac Suc (index div blockSize) =
  (index + blockSize) div blockSize, simp)
apply (subgoal-tac (fileSize cfile1 + blockSize - Suc 0) <=
  (index + blockSize), simp add: div-le-mono)
apply (subgoal-tac (fileSize cfile1 + blockSize - Suc 0) <
  (fileSize cfile1 + blockSize), simp)
apply (simp-all add: div-add-self2 nonZeroBlockSize)
done

```

4.6 Concrete File Operations Preserve Invariants

There is only one top-level concrete operation: write, and we show that it preserves all reachability invariants.

lemma *cfWritePreservesNextFreeBlockInvariant:*

```

  [| reachabilityInvariant cfile1;
    cfWrite cfile1 byteIndex value = Some cfile2
  |] ==> nextFreeBlockInvariant cfile2
apply (simp add: reachabilityInvariant-def
  cfWrite-def
  nextFreeBlockInvariant-def)
apply (case-tac byteIndex div blockSize < numBlocks, simp-all)
apply (case-tac byteIndex < fileSize cfile1, simp-all)
apply (simp-all add: cfWriteNoExtend-def cfExtendFile-def Let-def)
apply auto
apply (simp add: div-add-self2 nonZeroBlockSize)
done

```

lemma *modInequalityLemma:*

```

  (a::nat)  $\sim$  b & a mod c = b mod c ==> a div c  $\sim$  b div c
apply auto
apply (insert mod-div-equality [of a c])
apply (insert mod-div-equality [of b c])
apply simp
done

```

lemma *mod-round-lt:*

```

  [| 0 < (c::nat);
    a < b
  |] ==> a div c < (b + c - 1) div c
apply (subgoal-tac a <= b - 1)
apply (subgoal-tac a div c <= (b - 1) div c)
apply (insert div-add-self2 [of c b - 1])
apply (simp)
apply (simp add: div-le-mono)
apply (insert less-Suc-eq-le [of a b - 1])

```

apply *simp*
done

lemma *blockNumNELemma*:

```

!!blockNum i.
  [] nextFreeBlockInvariant cfile1;
  cfile1
  (| data :=
    writeCArray (data cfile1) (byteIndex div blockSize)
    (writeCArray
      (readCArray (data cfile1) (byteIndex div blockSize))
      (byteIndex mod blockSize) value) |) =
  cfile2;
  ~ blockNum < nextFreeBlock cfile2; blockNum < numBlocks;
  i < blockSize; byteIndex div blockSize < numBlocks;
  byteIndex < fileSize cfile1 []
==> blockNum ~ = byteIndex div blockSize
apply (simp add: nextFreeBlockInvariant-def)
apply (subgoal-tac byteIndex div blockSize < nextFreeBlock cfile1)
apply force
apply (subgoal-tac nextFreeBlock cfile1 =
  (fileSize cfile1 + blockSize - Suc 0) div blockSize, simp-all)
apply (insert mod-round-lt)
apply force
done

```

lemma *cfWritePreservesUnallocatedBlocksInvariant*:

```

[] reachabilityInvariant cfile1;
  cfWrite cfile1 byteIndex value = Some cfile2
[] ==> unallocatedBlocksInvariant cfile2
apply (simp add: reachabilityInvariant-def)
apply (subgoal-tac nextFreeBlock cfile1 <= nextFreeBlock cfile2)
apply (simp add: unallocatedBlocksInvariant-def cfWrite-def)
apply auto
apply (case-tac byteIndex div blockSize < numBlocks, simp-all)
apply (case-tac byteIndex < fileSize cfile1, simp-all)
apply (simp-all add: cfWriteNoExtend-def cfExtendFile-def Let-def)
apply (simp-all add: writeCArray-def readCArray-def)
apply (subgoal-tac blockNum ~ = byteIndex div blockSize)
apply force
apply (simp add: blockNumNELemma)
apply (subgoal-tac ~ blockNum < nextFreeBlock cfile1)
apply (subgoal-tac blockNum ~ = byteIndex div blockSize)
apply auto
apply (simp add: nextFreeBlockIncreases)
done

```

lemma *cfWritePreservesLastBlockInvariant*:

```

[] reachabilityInvariant cfile1;

```

```

      cfWrite cfile1 byteIndex value = Some cfile2 || ==>
      lastBlockInvariant cfile2
apply (simp add: reachabilityInvariant-def)
apply (subgoal-tac nextFreeBlock cfile1 <= nextFreeBlock cfile2)
apply (simp add: cfWrite-def)
apply (simp (no-asm) add: lastBlockInvariant-def)
apply auto
apply (case-tac byteIndex div blockSize < numBlocks, simp-all)
apply (case-tac byteIndex < fileSize cfile1, simp-all)
apply (simp-all add: cfWriteNoExtend-def Let-def cfExtendFile-def)
apply (simp-all add: writeCArray-def readCArray-def)
apply (simp add: lastBlockInvariant-def)
apply (subgoal-tac index ~ = byteIndex)
apply (case-tac index div blockSize ~ = byteIndex div blockSize)
apply force
apply (subgoal-tac index mod blockSize ~ = byteIndex mod blockSize)
apply force
apply (insert modInequalityLemma)
apply force
apply force
apply (subgoal-tac index ~ = byteIndex)
apply (case-tac index div blockSize ~ = byteIndex div blockSize, simp-all)
apply force
apply (subgoal-tac index mod blockSize ~ = byteIndex mod blockSize)
apply (case-tac nextFreeBlock cfile1 = Suc (index div blockSize))
apply (subgoal-tac ~ index < fileSize cfile1)
apply (simp add: lastBlockInvariant-def)
apply auto
apply (simp add: unallocatedBlocksInvariant-def)
apply (erule-tac x=index div blockSize in allE)
apply (erule-tac x=index mod blockSize in allE)
apply (simp add: nonZeroBlockSize)
apply (insert modInequalityLemma)
apply auto
apply (simp add: nextFreeBlockIncreases)
done

```

Final statement that all invariants are preserved.

```

lemma cfWritePreserves:
  || reachabilityInvariant cfile1;
  cfWrite cfile1 byteIndex value = Some cfile2 || ==>
  reachabilityInvariant cfile2
apply (simp (no-asm) add: reachabilityInvariant-def)
apply (simp add: cfWritePreservesNextFreeBlockInvariant)
apply (simp add: cfWritePreservesUnallocatedBlocksInvariant)
apply (simp add: cfWritePreservesLastBlockInvariant)
done

```

4.7 Commuting Diagrams for Simulation Relation

Here we show correctness of file system operations.

4.7.1 Abstraction Function

constdefs

```
abstFn :: CFile => AFile
abstFn cfile == (fileSize cfile,
                 % index . case cfRead cfile index of
                   None      => fillByte
                 | Some value => value)
```

consts

```
oAbstFn :: CFile option => AFile option
primrec oAbstFn None = None
        oAbstFn (Some s) = Some (abstFn s)
```

4.7.2 Creating a File

lemma *makeCFCorrect*: $abstFn\ makeCF = makeAF$
by (*simp add*: *makeCF-def makeAF-def abstFn-def cfRead-def*
split add: *bool.splits option.splits*)

4.7.3 File Size

lemma *fileSizeCorrect*:
 $cfSize\ cfile = afSize\ (abstFn\ cfile)$
by (*simp add*: *cfSize-def afSize-def abstFn-def*)

4.7.4 Read Operation

lemma *readCorrect*:
 $cfRead\ cfile = afRead\ (abstFn\ cfile)$
apply (*simp add*: *abstFn-def*)
apply (*rule ext*)
apply (*simp add*: *cfRead-def afRead-def Let-def*)
done

4.7.5 Write Operation

lemma *writeNoExtendCorrect*:
[[*index* < *fileSize* *cfile1*;
 Some *cfile2* = *cfWrite* *cfile1* *index* *value*
]] ==> $Some\ (abstFn\ cfile2) = afWrite\ (abstFn\ cfile1)\ index\ value$
apply (*simp add*: *abstFn-def afWrite-def raWrite-def Let-def cfWrite-def*)
apply (*case-tac index div blockSize < numBlocks, simp-all*)
apply (*simp-all add*: *cfWriteNoExtend-def Let-def*)
apply (*rule ext*)
apply (*simp add*: *cfRead-def writeCArray-def readCArray-def Let-def*)

```

apply (case-tac  $indexa < fileSize\ cfile1$ , simp-all)
apply (case-tac  $indexa = index$ , simp-all)
apply (case-tac  $indexa \bmod blockSize = index \bmod blockSize$ , simp-all)
apply (subgoal-tac  $indexa \operatorname{div} blockSize \sim = index \operatorname{div} blockSize$ , simp-all)
apply (simp-all add: modInequalityLemma)
done

lemma writeExtendCorrect:
  [| nextFreeBlockInvariant cfile1;
    unallocatedBlocksInvariant cfile1;
    lastBlockInvariant cfile1;
     $\sim index < fileSize\ cfile1$ ;
    Some cfile2 = cfWrite cfile1 index value
  |] ==> Some (abstFn cfile2) = afWrite (abstFn cfile1) index value
apply (insert nextFreeBlockIncreases [of cfile1 index value cfile2])
apply (simp add: abstFn-def afWrite-def raWrite-def Let-def)
apply (case-tac  $\sim index \operatorname{div} blockSize < numBlocks$ ,
  simp-all add: cfWrite-def cfWriteNoExtend-def cfExtendFile-def Let-def)
apply (rule ext)
apply (simp add: cfRead-def fillAndUpdate-def Let-def writeCArrayCorrect1)
apply (case-tac  $indexa < fileSize\ cfile1$ , simp-all)
apply (subgoal-tac  $indexa \sim = index$ , simp-all)
apply (case-tac  $indexa \operatorname{div} blockSize = index \operatorname{div} blockSize$ )
apply (case-tac  $indexa \bmod blockSize = index \bmod blockSize$ ,
  simp add: modInequalityLemma)
apply (simp-all add: writeCArrayCorrect1 writeCArrayCorrect2)
apply (case-tac  $indexa < index$ , simp-all)
apply (case-tac  $indexa \operatorname{div} blockSize = index \operatorname{div} blockSize$ )
apply (case-tac  $indexa \bmod blockSize = index \bmod blockSize$ ,
  simp add: modInequalityLemma)
apply (simp-all add: readCArray-def writeCArray-def lastBlockInvariant-def)
apply (erule-tac  $x=indexa$  in alle, simp-all)
apply (case-tac nextFreeBlock cfile1 = nextFreeBlock cfile2, simp-all)
apply (simp add: unallocatedBlocksInvariant-def)
apply (subgoal-tac  $\sim indexa \operatorname{div} blockSize < nextFreeBlock\ cfile1$ , simp-all)
apply (subgoal-tac  $indexa \bmod blockSize < blockSize$ , simp-all)
apply (insert nonZeroBlockSize)
apply force
apply (simp add: unallocatedBlocksInvariant-def)
apply (case-tac  $\sim indexa \operatorname{div} blockSize < nextFreeBlock\ cfile1$ , simp-all)
apply (subgoal-tac  $indexa \operatorname{div} blockSize < numBlocks$ , simp-all)

apply (subgoal-tac  $indexa \operatorname{div} blockSize \leq index \operatorname{div} blockSize$ , simp-all)
apply (simp add: div-le-mono)
apply (subgoal-tac nextFreeBlock cfile1 = Suc (indexa div blockSize), simp)
apply (simp add: nextFreeBlockInvariant-def)
apply (subgoal-tac nextFreeBlock cfile1 =
  (fileSize cfile1 + blockSize - Suc 0) div blockSize, simp-all)
apply (subgoal-tac (fileSize cfile1 + blockSize - Suc 0) div blockSize <=

```

```

    Suc (indexa div blockSize), simp-all)
apply (subgoal-tac Suc (indexa div blockSize) =
  (indexa + blockSize) div blockSize)
apply (simp only:)
apply (rule div-le-mono)
apply auto
done

lemma writeSucceedCorrect:
  [| nextFreeBlockInvariant cfile1;
    unallocatedBlocksInvariant cfile1;
    lastBlockInvariant cfile1;
    Some cfile2 = cfWrite cfile1 index value
  |] ==> Some (abstFn cfile2) = afWrite (abstFn cfile1) index value
apply (case-tac index < fileSize cfile1)
apply (simp-all add: writeExtendCorrect writeNoExtendCorrect)
done

lemma writeFailCorrect:
  cfWrite cfile1 index value = None ==>
  afWrite (abstFn cfile1) index value = None
apply (simp add: abstFn-def cfWrite-def afWrite-def)
apply (case-tac index div blockSize < numBlocks, simp-all)
apply (case-tac index < fileSize cfile1, simp-all)
done

lemma writeCorrect:
  reachabilityInvariant cfile1 ==>
  oAbstFn (cfWrite cfile1 index value) = afWrite (abstFn cfile1) index value
apply (simp add: reachabilityInvariant-def)
apply (case-tac cfWrite cfile1 index value)
apply (simp add: writeFailCorrect)
apply (simp add: writeSucceedCorrect)
done

end

```

References

- [1] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a file system implementation. In *Sixth International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, Seattle, Nov 8-12, 2004 2004.
- [2] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. 1998.

- [3] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.
- [4] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. <http://www.in.tum.de/~nipkow/LNCS2283/>.
- [5] K. Thompson. UNIX implementation. *The Bell System Technical Journal*, 57(6 (part 2)), 1978.