# 7. Disks and File Systems

## Motivation

The two lectures on disks and file systems are intended to show you a number of things:

Some semi-realistic examples of specs.

Many important coding techniques for file systems.

Some of the tradeoffs between a simple spec and efficient code.

Examples of abstraction functions and invariants.

Encoding: a general technique for representing arbitrary types as byte sequences.

How to model crashes.

Transactions: a general technique for making big actions atomic.

There are a lot of ideas here. After you have read this handout and listened to the lectures, it's a good idea to go back and reread the handout with this list of themes in mind.

## Outline of topics

We give the specs of disks and files in the `Disk` and `File` modules, and we discuss a variety of coding issues:

Crashes

Disks

Files

Caching and buffering of disks and files

Representing files by trees and extents

Allocation

Encoding and decoding

Directories

Transactions

Redundancy

## Crashes

The specs and code here are without concurrency. However, they do allow for crashes. A crash can happen between any two atomic commands. Thus the possibility of crashes introduces a limited kind of concurrency.





When a crash happens, the volatile global state is reset, but the stable state is normally unaffected. We express precisely what happens to the global state as well as how the module recovers by including a `Crash` procedure in the module. When a crash happens:

1. The `Crash` procedure is invoked. It need not be atomic.
2. If the `Crash` procedure does a `CRASH` command, the execution of the current invocations (if any) stop, and their local state is discarded; the same thing happens to any invocations outside the module from within it. After `CRASH`, no procedure in the module can be invoked from outside until `Crash` returns.
3. The `Crash` procedure may do other actions, and eventually it returns.
4. Normal operation of the module resumes; that is, external invocations are now possible.

You can tell which parts of the state are volatile by looking at what `Crash` does; it will reset the volatile variables.

Because crashes are possible between any two atomic commands, atomicity is important for any operation that involves a change to stable state.

The meaning of a Spec program with this limited kind of concurrency is that each atomic command corresponds to a transition. A hidden piece of state called the program counter keeps track of what transitions are enabled next: they are the atomic commands right after the program counter. There may be several if the command after the program counter has `[]` as its operator. In addition, a crash transition is always possible; it resets the program counter to a null value from which no transition is possible until some external routine is invoked and then invokes the `Crash` routine.

If there are non-atomic procedures in the spec with many atomic commands, it can be rather difficult to see the consequences of a crash. It is therefore clearer to write a spec with as much atomicity as possible, making it explicit exactly what unusual transitions are possible when there's a crash. We don't always follow this style, but we give some examples of it, notably at the end of the section on disks.

## Disks

Essential properties of a disk:

Storage is stable across crashes (we discuss error models for disks in the `Disk` spec).
It's organized in blocks, and the only atomic update is to write one block.
Random access is about 100k times slower than random access to RAM (10 ms vs. 100 ns)
Sequential access is 10-100 times slower than to RAM (40 MB/s vs. 400-6000 MB/s)
Costs 50 times less than RAM ($2/GB vs. $100/GB) in January 2002.
MTBF 1 million hours = 100 years.

Performance numbers:

Blocks of .5k - 4k bytes
40 MB/sec sequential, sustained (more with parallel disks)

3 ms average rotational delay (10000 rpm = 6 ms rotation time)
7 ms average seek time; 3 ms minimum

It takes 10 ms to get anything at all from a random place on the disk. In another 10 ms you can transfer 400 KB. Hence the cost to get 400 KB is only twice the cost to get 1 byte. By reading from several disks in parallel (called *striping* or *RAID*) you can easily increase the transfer rate by a factor of 5-10.

Performance techniques:

Avoid disk operations: use caching
Do sequential operations: allocate contiguously, prefetch, write to log
Write in background (write-behind)

*A spec for disks*

The following module describes a disk `Dsk` as a function from a `DA` to a disk block `DB`, which is just a sequence of `DBSize` bytes. The `Dsk` function can also yield `nil`, which represents a permanent read error. The module is a class, so you can instantiate as many `Disks` as needed. The state is one `Dsk` for each `Disk`. There is a `New` method for making a new disk; think of this as ordering a new disk drive and plugging it in. An extent `E` represents a set of consecutive disk addresses. The main routines are the `read` and `write` methods of `Disk`: `read`, which reads an extent, and `write`, which writes `n` disk blocks worth of data sequentially to the extent `E{da, n}`. The write is not atomic, but can be interrupted by a failure after each single block is written.

Usually a spec like this is written with a concurrent thread that introduces permanent errors in the recorded data. Since we haven't discussed concurrency yet, in this spec we introduce the errors in `reads`, using the `AddErrors` procedure. An error sets a block to `nil`, after which any read that includes that block raises the exception `error`. Strictly speaking this is illegal, since `read` is a function and therefore can't call the procedure `AddErrors`. When we learn about concurrency we can move `AddErrors` to a separate thread; in the meantime we take the liberty, since it would be a real nuisance for `read` to be a procedure rather than a function.

Since neither Spec nor our underlying model deals with probabilities, we don't have any way to say how likely an error is. We duck this problem by making `AddErrors` completely non-deterministic; it can do anything from introducing no errors (which we must hope is the usual case) to clobbering the entire disk. Characterizing errors would be quite tricky, since disks usually have at least two classes of error: failures of single blocks and failures of an entire disk. However, any user of this module must assume something about the probability and distribution of errors.

Transient errors are less interesting because they can be masked by retries. We don't model them, and we also don't model errors reported by `writes`. Finally, a realistic error model would include the possibility that a block that reports a read error might later be readable after all.

```
CLASS Disk EXPORT Byte, Data, DA, E, DBSize, read, write, size, check, Crash  =

TYPE Byte        =  IN 0 .. 255
     Data        =  SEQ Byte
     DA          =  Nat                                  % Disk block Address
     DB          =  SEQ Byte                             % Disk Block
                    SUCHTHAT (\db| db.size = DBSize)
     Blocks      =  SEQ DB
     E           =  [da, size: Nat]                      % Extent, in disk blocks
                    WITH {das:=EToDAs, "IN":=(\ e, da | da IN e.das)}
     Dsk         =  DA -> (DB + Null)                    % a DB or nil (error) for each DA

CONST DBSize     := 1024                                 % bytes in a disk block

VAR  disk        :  Dsk

APROC new(size: Int) -> Disk = <<                        % overrides StdNew
     VAR dsk | dsk.dom = size.seq.rng =>                 % size blocks, arbitrary contents
          self := StdNew(); disk := dsk; RET self >>

FUNC read(e) -> Data RAISES {notThere, error} =
     check(e); AddErrors();
     VAR dbs := e.das * disk |                           % contents of the blocks in e
          IF nil IN dbs => RAISE error [*] RET BToD(dbs) FI

PROC write(da, data) RAISES {notThere} =                 % fails if data not a multiple of DBsize
     VAR blocks := DToB(data), i := 0 |
     % Atomic by block, and in order
          check(E{da, blocks.size});
          DO blocks!i => WriteBlock(da + i, blocks(i)); i + := 1 OD

APROC WriteBlock(da, db) = << disk(da) := db >>          % the atomic update. PRE: disk!da

FUNC size() -> Int = RET disk.dom.size

APROC check(e) RAISES {notThere} =                       % every DA in e is in disk.dom
     << e.das.rng <= disk.dom => RET [*] RAISE notThere >>

PROC Crash() = CRASH                                     % no global volatile state

FUNC EToDAs(e) -> SEQ DA = RET e.da .. e.da+e.size-1     % e.das

% Internal routines

% Functions to convert between Data and Blocks.
FUNC BToD(blocks) -> Data   = RET + : blocks
FUNC DToB(data  ) -> Blocks = VAR blocks | BToD(blocks) = data => RET blocks
% Undefined if data.size is not a multiple of DBsize

APROC AddErrors() =                                      % clobber some blocks
     << DO RET [] VAR da :IN disk.dom | disk(da) := nil OD >>

END Disk
```

This module doesn't worry about the possibility that a disk may fail in such a way that the client can't tell whether a write is still in progress; this is a significant problem in fault tolerant systems that want to allow a backup processor to start running a disk as soon as possible after the primary fails.

Many disks do not guarantee the order in which blocks are written (why?) and thus do not implement this spec, but instead one with a weaker `write`:

```
PROC writeUnordered(da, data) RAISES {notThere} =
  VAR blocks := DToB(data) |
      % Atomic by block, in arbitrary order; assumes no concurrent writing.
      check(E{da, blocks.size});
      DO VAR i | blocks(i) # disk(da + i) => WriteBlock(da + i, blocks(i)) OD
```

In both specs `write` establishes `blocks = E{da, blocks.size}.das * disk`, which is the same as `data = read(E{da, blocks.size})`, and both change each disk block atomically. `writeUnordered` says nothing about the order of changes to the disk, so after a crash any subset of the blocks being written might be changed; `write` guarantees that the blocks changed are a prefix of all the blocks being written. (`writeUnordered` would have other differences from `write` if concurrent access to the disk were possible, but we have ruled that out for the moment.)

*Clarifying crashes*

In this spec, what happens when there's a crash is expressed by the fact that `write` is not atomic and changes the disk one block at a time in the atomic `WriteBlock`. We can make this more explicit by making the occurrence of a crash visible inside the spec in the value of the `crashed` variable. To do this, we modify `Crash` so that it temporarily makes `crashed` true, to give `write` a chance to see it. Then `write` can be atomic; it writes all the blocks unless `crashed` is true, in which case it writes some prefix; this will happen only if `write` is invoked between the `crashed := true` and the `CRASH` commands of `Crash`. To describe the changes to the disk neatly, we introduce an internal function `NewDisk` that maps a `dsk` value into another one in which disk blocks at `da` are replaced by corresponding blocks defined in `bs`.

Again, this wouldn't be right if there were concurrent accesses to `Disk`, since we have made all the changes atomically, but it gives the possible behavior if the only concurrency is in crashes.

```
VAR  crashed     : Bool := false

...

APROC write(da, data) RAISES {notThere} = <<          % fails if data not a multiple of DBsize
     VAR blocks := DToB(data) |
          check(E{da, blocks.size});
          IF   crashed =>                              % if crashed, write some prefix
               VAR i | i < blocks.size => blocks := blocks.sub(0, i)
          []   SKIP FI;
          disk := NewDisk(disk, da, blocks)
     >>

FUNC NewDisk(dsk, da, bs: (Int -> DB)) -> Dsk =       % result is dsk overwritten with bs at da
     RET dsk + (\ da' | da' – da) * bs

PROC Crash() = crashed := true; CRASH; crashed := false
```

For unordered writes we need only a slight change, to write an arbitrary subset of the blocks if there's a crash, rather than a prefix:

```
          IF   crashed =>                              % if crashed, write some subset
               VAR w: SET I | w <= blocks.dom => blocks := blocks.restrict(w)
```





## Specifying files

This section gives a variety of specs for files. Code follows in later sections.

We treat a file as just a sequence of bytes, ignoring permissions, modified dates and other paraphernalia. Files have names, and for now we confine ourselves to a single directory that maps names to files. We call the name a 'path name' `PN` with an eye toward later introducing multiple directories, but for now we just treat the path name as a string without any structure. We package the operations on files as methods of `PN`. The main methods are `read` and `write`; we define the latter initially as `WriteAtomic`, and later introduce less atomic variations `Write` and `WriteUnordered`. There are also boring operations that deal with the size and with file names.

```
MODULE File EXPORT PN, Byte, Data, X, F, Crash =

TYPE PN          =  String                              % Path Name
                    WITH {read:=Read, write:=WriteAtomic, size:=GetSize,
                          setSize:=SetSize, create:=Create, remove:=Remove,
                          rename:=Rename}
     I           =  Int
     Byte        =  IN 0 .. 255
     Data        =  SEQ Byte
     X           =  Nat                                 % byte-in-file indeX
     F           =  Data                                % File

     D           =  PN -> F                             % Directory

VAR  d           := D{}                                 % undefined everywhere
```

Note that the only state of the spec is `d`, since files are only reachable through `d`.

There are tiresome complications in `Write` caused by the fact that the arguments may extend beyond the end of the file. These can be handled by imposing preconditions (that is, writing the spec to do `HAVOC` when the precondition isn't satisfied), by raising exceptions, or by defining some sensible behavior. This spec takes the third approach; `NewFile` computes the desired contents of the file after the write. So that it will work for unordered writes as well, it handles sparse `data` by choosing an arbitrary `data'` that agrees with `data` where `data` is defined. Compare it with `Disk.NewDisk`.

```
FUNC Read(pn, x, i) -> Data = RET d(pn).seg(x, i)
% Returns as much data as available, up to i bytes, starting at x.

APROC WriteAtomic(pn, x, data) = << d(pn) := NewFile(d(pn), x, data) >>

FUNC NewFile(f0, x, data: Int -> Byte) -> F =
% f is the desired final file. Fill in space between f0 and x with zeros, and undefined data elements arbitrarily.
  VAR z := data.dom.max, z0 := f0.size , f, data' |
       data'.size = z /\ data'.restrict(data.dom) = data
    /\ f.size = {z0, x+z}.max
    /\ (ALL i |    ( i IN 0   .. {x, z0}.min-1 ==> f(i) = f0(i)      )
                /\ ( i IN z0  .. x-1           ==> f(i) = 0          )
                /\ ( i IN x   .. x+z-1         ==> f(i) = data'(i-x) )
                /\ ( i IN x+z .. z0-1          ==> f(i) = f0(i) )    )
    => RET f
```

```
FUNC GetSize(pn) -> X = RET d(pn).size

APROC SetSize(pn, x) = << VAR z := pn.size |
    IF   x <= z => << d(pn) := pn.read(0, z) >>        % truncate
    [*]  pn.write(z, F.fill(0, x - z + 1))             % handles crashes like write
    FI >>

APROC Create(pn) = << d(pn) := F{} >>
APROC Remove(pn) = << d := d{pn -> } >>
APROC Rename(pn1, pn2) = << d(pn2) := d(pn1); Remove(pn1) >>

PROC Crash() = SKIP                                    % no volatile state or non-atomic changes

END File
```

`WriteAtomic` changes the entire file contents at once, so that a crash can never leave the file in an intermediate state. This would be quite expensive in most code. For instance, consider what is involved in making a write of 20 megabytes to an existing file atomic; certainly you can't overwrite the existing disk blocks one by one. For this reason, real file systems don't implement `WriteAtomic`. Instead, they change the file contents a little at a time, reflecting the fact that the underlying disk writes blocks one at a time. Later we will see how an atomic `Write` could be implemented in spite of the fact that it takes several atomic disk writes. In the meantime, here is a more realistic spec for `Write` that writes the new bytes in order. It is just like `Disk.write` except for the added complication of extending the file when necessary, which is taken care of in `NewFile`.

```
APROC Write(pn, x, data) = <<
    IF   crashed =>                                    % if crashed, write some prefix
        VAR i | i < data.size => data := data.sub(0, i)
    [*] SKIP FI;
    d(pn) := NewFile(d(pn), x, data) >>

PROC Crash() = crashed := true; CRASH; crashed := false
```

This spec reflects the fact that only a single disk block can be written atomically, so there is no guarantee that all of the data makes it to the file before a crash. At the file level it isn't appropriate to deal in disk blocks, so the spec promises only bytewise atomicity. Actual code would probably make changes one page at a time, so it would not exhibit all the behavior allowed by the spec. There's nothing wrong with this, as long as the spec is restrictive enough to satisfy its clients.

`Write` does promise, however, that `f(i)` is changed no later than `f(i+1)`. Some file systems make no ordering guarantee; actually, any file system that runs on a disk without an ordering guarantee probably makes no ordering guarantee, since it requires considerable care, or considerable cost, or both to overcome the consequences of unordered disk writes. For such a file system the following `WriteUnordered` is appropriate; it is just like `Disk.writeUnordered`.

```
APROC WriteUnordered(pn, x, data) = <<
    IF   crashed =>                                    % if crashed, write some subset
        VAR w: SET I | w <= data.dom => data := data.restrict(w)
    [*] SKIP FI;
    d(pn) := NewFile(d(pn), x, data) >>
```





Notice that although writing a file is not atomic, `File`'s directory operations are atomic. This corresponds to the semantics that file systems usually attempt to provide: if there is a failure during a `Create`, `Remove`, or `Rename`, the operation is either completed or not done at all, but if there is a failure during a `Write`, any amount of the data may be written. The other reason for making this choice in the spec is simple: with the abstractions available there's no way to express any sensible intermediate state of a directory operation other than `Rename` (of course sloppy code might leave the directory scrambled, but that has to count as a bug; think what it would look like in the spec).

The spec we gave for `SetSize` made it as atomic as `write`. The following spec for `SetSize` is unconditionally atomic; this might be appropriate because an atomic `SetSize` is easier to implement than a general atomic `Write`:

```
APROC SetSize(pn, x) = << d(pn) := (d(pn) + F.fill(0, x)).seg(0, x) >>
```

Here is another version of `NewFile`, written in a more operational style just for comparison. It is a bit shorter, but less explicit about the relation between the initial and final states.

```
FUNC NewFile(f0, x, data: Int -> Byte) -> F = VAR z0 := f0.size, data' |
    data'.size = data.dom.max =>
        data' := data' + data;
        RET   (x > z0 => f0 + F.fill(0, x - z0) [*] f0.sub(0, x - 1))
            + data'
            + f0.sub(f.size, z0-1)
```

Our `File` spec is missing some things that are important in real file systems:

Access control: permissions or access control lists on files, ways of defaulting these when a file is created and of changing them, an identity for the requester that can be checked against the permissions, and a way to establish group identities.

Multiple directories. We will discuss this when we talk about naming.

Quotas, and what to do when the disk fills up.

Multiple volumes or file systems.

Backup. We will discuss this near the end of this handout when we describe the copying file system.

## Cached and buffered disks

The simplest way to decouple the file system client from the slow disk is to provide code for the `Disk` abstraction that does caching and write buffering; then the file system code need not change. The basic ideas are very similar to the ideas for cached memory, although for the disk we preserve the order of writes. We didn't do this for the memory because we didn't worry about failures.

Failures add complications; in particular, the spec must change, since buffering writes means that some writes may be lost if there is a crash. Furthermore, the client needs a way to ensure that its writes are actually stable. We therefore need a new spec `BDisk`. To get it, we add to `Disk` a

variable `oldDisks` that remembers the previous states that the disk might revert to after a crash (note that this is not necessarily all the previous states) and code to use `oldDisks` appropriately. `BDisk.write` no longer needs to test `crashed`, since it's now possible to lose writes even if the crash happens after the `write`.

```
CLASS BDisk EXPORT ..., sync  =                          % write-buffered disk

TYPE ...
CONST ...
VAR  disk         :  Dsk                                 % as in Disk
     oldDisks     :  SET Dsk := {}

...

APROC write(da, data) RAISES {notThere} = <<             % fails if data not a multiple of DBsize
  << VAR blocks := DToB(data) |
         check(E{da, blocks.size});
         disk := NewDisk(disk, da, blocks);
         oldDisks \/ := {i | i < blocks.size |
                         NewDisk(disk, da, blocks.sub(0, i))};
         Forget()
     >>

FUNC NewDisk(dsk, da, bs: (Int -> DB)) -> Dsk =          % result is dsk overwritten with bs at da
     RET dsk + (\ da' | da' – da) * bs

PROC sync() = oldDisks := {}                             % make disk stable

PROC Forget() = VAR ds: SET Dsk | oldDisks - := ds
% Discards an arbitrary subset of the remembered disk states.

PROC Crash() = CRASH; << VAR d :IN oldDisks | disk := d; sync() [*] SKIP >>

END BDisk
```

`Forget` is there so that we can write an abstraction function for code for that doesn't defer all its disk writes until they are forced by `Sync`. A write that actually changes the disk needs to change `oldDisks`, because `oldDisks` contains the old state of the disk block being overwritten, and there is nothing in the state of the code after the write from which to compute that old state. Later we will study a better way to handle this problem: history variables or multi-valued mappings. They complicate the code rather than the spec, which is preferable. Furthermore, they do not affect the performance of the code at all.

A weaker spec would revert to a state in which any subset of the writes has been done. For this, change the assignment to `oldDisks` in `write`, along the lines we have seen before.

```
         oldDisks \/ := {w: SET I | w <= blocks.dom |
                         NewDisk(disk, da, blocks.restrict(w))};
```

The module `BufferedDisk` below is code for `BDisk`. It copies newly written data into the cache and does the writes later, preserving the original order so that the state of the disk after a crash will always be the state at some time in the past. In the absence of crashes this implements `Disk` and is completely deterministic. We keep track of the order of writes with a `queue` variable, instead of keeping a `dirty` bit for each cache entry as we did for cached memory. If we didn't do





the writes in order, there would be many more possible states after a crash, and it would be much more difficult for a client to use this module. Many real disks have this unpleasant property, and many real systems deal with it by ignoring it.

A striking feature of this code is that it uses the same abstraction that it implements, namely `BDisk`. The code for `BDisk` that it uses we call `UDisk` (`U` for 'underlying'). We think of it as a 'physical' disk, and of course it is quite different from `BufferedDisk`: it contains SCSI controllers, magnetic heads, etc. A module that implements the same interface that it uses is sometimes called a *filter* or a *stackable module*. A Unix filter like `sed` is a familiar example that uses and implements the byte stream interface. We will see many other examples of this in the course.

Invocations of `UDisk` are in bold type, so you can easily see how the module depends on the lower-level code for `BDisk`.

```
CLASS BufferedDisk                                       % implements BDisk
     EXPORT Byte, Data, DA, E, DBSize, read, write, size, check, sync, Crash =

TYPE % Data, DA, DB, Blocks, E as in Disk
     I            =  Int
     J            =  Int

     Queue        =  SEQ DA                              % data is in cache

CONST
     cacheSize    := 1000
     queueSize    := 50

VAR  udisk        :  Disk
     cache        :  DA -> DB := {}
     queue        := Queue{}

% ABSTRACTION FUNCTION bdisk.disk = udisk.disk + cache
% ABSTRACTION FUNCTION bdisk.oldDisks =
     { q: Queue | q <= queue | udisk.disk + cache.restrict(q.rng) }

% INVARIANT queue.rng <= cache.dom                       % if queued then cached
% INVARIANT queue.size = queue.rng.size                  % no duplicates in queue
% INVARIANT cache.dom.size <= cacheSize                  % cache not too big
% INVARIANT queue.size <= queueSize                      % queue not too big

APROC new(size: Int) -> BDisk = <<                       % overrides StdNew
     self := StdNew(); udisk := udisk.new(size); RET self >>

PROC read(e) -> Data RAISES {notThere} =
% We could make provision for read-ahead, but do not.
     check(e);
     VAR data := Data{}, da := e.da, upTo := e.da + e.size |
          DO da < upTo =>
               IF  cache!da => data + := cache(da); da + := 1
               [*] % read as many blocks from disk as possible
                    VAR i := RunNotInCache(da, upTo),
                        buffer := udisk.read(E{da, i}),
                        k := MakeCacheSpace(i) |
                         % k blocks will fit in cache; add them.
```

```
                    DO VAR j :IN k.seq | ~ cache!(da + j) =>
                         cache(da + j) := udisk.DToB(buffer)(j)
                    OD;
                    data + := buffer; da + := i
               FI
          OD; RET data

PROC write(da, data) RAISES {notThere} =
  VAR blocks := udisk.DToB(data) |
     check(E{da, blocks.size});
     DO VAR i :IN queue.dom | queue(i) IN da .. da+size-1 => FlushQueue(i) OD;
     % Do any previously buffered writes to these addresses. Why?
     VAR j := MakeCacheSpace(blocks.size), i := 0 |
          IF  j < blocks.size => udisk.write(da, data)
               % Don't cache if the write is bigger than the cache.
          [*]  DO blocks!i =>
                    cache(da+i) := blocks(i); queue + := {da+i}; i + := 1
               OD
          FI

PROC Sync() = FlushQueue(queue.size - 1)

PROC Crash() = CRASH; cache := {}; queue := {}

FUNC RunNotInCache(da, upTo: DA) -> I =
     RET {i | da + i <= upTo /\ (ALL j :IN i.seq | ~ cache!(da + j)}.max

PROC MakeCacheSpace(i) -> Int =
% Make room for i new blocks in the cache; returning min(i, the number of blocks now available).
% May flush queue entries.
% POST: cache.dom.size + result <= cacheSize
     . . .

PROC FlushQueue(i) = VAR q := queue.sub(0, i) |
% Write queue entries 0 .. i and remove them from queue.
% Should try to combine writes into the biggest possible writes
     DO q # {} => udisk.write(q.head, 1); q := q.tail OD;
     queue := queue.sub(i + 1, queue.size - 1)

END BufferedDisk
```

This code keeps the cache as full as possible with the most recent data, except for gigantic writes. It would be easy to change it to make non-deterministic choices about which blocks to keep in the cache, or to take advice from the client about which blocks to keep. The latter would require changing the interface to accept the advice, of course.

Note that the only state of `BDisk` that this module can actually revert to after a crash is the one in which none of the queued writes has been done. You might wonder, therefore, why the body of the abstraction function for `BDisk.oldDisks` has to involve `queue`. Why can't it just be `{udisk.disk}`? The reason is that when the internal procedure `FlushQueue` does a write, it changes the state that a crash reverts to, and there's no provision in the `BDisk` spec for adding anything to `oldDisks` except during `write`. So `oldDisks` has to include all the states that the disk can reach after a sequence of 'internal' writes, that is, writes done in `FlushQueue`. And this is just what the abstraction function says.





*Building other kinds of disks*

There are other interesting and practical ways to code a disk abstraction on top of a 'base' disk. Some examples that are used in practice:

*Mirroring*: use two base disks of the same size to code a single disk of that size, but with much greater availability and twice the read bandwidth, by doing each write to both base disks.

*Striping*: use *n* base disks to code a single disk *n* times as large and with *n* times the bandwidth, by reading and writing in parallel to all the base disks

*RAID*: use *n* base disks of the same size to code a single disk *n*-1 times as large and with *n*-1 times the bandwidth, but with much greater availability, by using the *n*th disk to store the exclusive-or of the others. Then if one disk fails, you can reconstruct its contents from the others.

*Snapshots*: use 'copy-on-write' to code an ordinary disk and some number of read-only 'snapshots' of its previous state.

## Buffered files

We need to make changes to the `File` spec if we want the option to code it using buffered disks without doing too many `sync`s. One possibility is do a `bdisk.sync` at the end of each `write`. This spec is not what most systems implement, however, because it's too slow. Instead, they implement a version of `File` with the following additions. This version allows the data to revert to any previous state since the last `Sync`. The additions are very much like those we made to `Disk` to get `BDisk`. For simplicity, we don't change `oldDirs` for operations other than `write` and `setSize` (well, except for truncation); real systems differ in how much they buffer the other operations.

**`MODULE File`** `EXPORT ..., Sync =`

```
TYPE ...
VAR  d           := D{}
     oldDs     : SET D := {}
...

APROC Write(pn, x, byte) = << VAR f0 := d(pn) |
     d(pn) := NewFile(f0, x, data);
     oldDs \/ := {i | i < data.size |
                    d{pn -> NewFile(f0, x, data.sub(0, i))}} >>

APROC Sync() = << oldDirs := {} >>

PROC Crash() = CRASH; << VAR d :IN oldDirs => dir := d; Sync() [*] SKIP >>

END File
```

Henceforth we will use `File` to refer to the modified module. Since we are not giving code for, we leave out `Forget` for simplicity.

Many file systems do their own caching and buffering. They usually loosen this spec so that a crash resets each file to some previous state, but does not necessarily reset the entire system to a previous state. (Actually, of course, real file systems usually don't have a spec, and it is often very difficult to find out what they can actually do after a crash.)

```
MODULE File2 EXPORT ..., Sync =

TYPE ...
     OldFiles    =  PN -> SET F

VAR  d           := D{}
     oldFiles    := OldFiles{* -> {}}

...

APROC Write(pn, x, byte) = << VAR f0 := d(pn) |
     d(pn) := NewFile(f0, x, data);
     oldFiles(pn) \/ := {i | i < data.size | NewFile(f0, x, data.sub(0, i))} >>

APROC Sync() = << oldFiles:= OldFiles{* -> {}} >>

PROC Crash() =
     CRASH;
     << VAR d' |    d'.dom = d.dom
                 /\ (ALL pn :IN d.dom | d'(pn) IN oldFiles(pn) \/ {d(pn)})
          => d := d' >>

END File
```

A picky point about Spec: A function constructor like `(\ pn | {d(pn)})` is no good as a value for `oldFiles`, because the value of the global variable `d` in that constructor is not captured when the constructor is evaluated. Instead, this function uses the value of `d` when it is invoked. This is a little weird, but it is usually very convenient. Here it is a pain; we avoid the problem by using a *local* variable `d` whose value *is* captured when the constructor is evaluated in `SnapshotD`.

A still weaker spec allows `d` to revert to a state in which any subset of the byte writes has been done, except that the files still have to be sequences. By analogy with unordered `BDisk`, we change the assignment to `oldFiles` in `Write`.

```
     oldFiles(pn) \/ := {w: SET i | w <= data.dom |
                         NewFile(f0, x, data.restrict(w))} >>
```

## Coding files

The main issue is how to represent the bytes of the file on the disk so that large reads and writes will be fast, and so that the file will still be there after a crash. The former requires using contiguous disk blocks to represent the file as much as possible. The latter requires a representation for `D` that can be changed atomically. In other words, the file system state has type `PN -> SEQ Byte`, and we have to find a disk representation for the `SEQ Byte` that is efficient, and one for the function that is robust. This section addresses the first problem.

The simplest approach is to represent a file by a sequence of disk blocks, and to keep an *index* that is a sequence of the `DA`'s of these blocks. Just doing this naively, we have





```
TYPE F           =  [das: SEQ DA, size: N]          % Contents and size in bytes
```

The abstraction function to the spec says that the file is the first `f.size` bytes in the disk blocks pointed to by `c`. Writing this as though both `File` and its code `FImpl0` had the file `f` as the state, we get

```
File.f = (+ : (FImpl0.f.das * disk.disk)).seg(0, FImpl0.f.size)
```

or, using the `disk.read` method rather than the state of `disk` directly

```
File.f = (+ : {da :IN FImpl0.f.das | | disk.read(E{da, 1})}).seg(0, FImpl0.f.size)
```

But actually the state of `File` is `d`, so we should have the same state for `FImpl` (with the different representation for `F`, of course), and

```
File.d = (LAMBDA (pn) -> File.F =
     VAR f := FImpl0.d(pn) |                        % fails if d is undefined at pn
          RET (+ : (f.das * disk.disk)).seg(0, f.size)
```

We need an invariant that says the blocks of each file have enough space for the data.

```
% INVARIANT ( ALL f :IN d.rng | f.das.size * DBSize >= f.size )
```

Then it's easy to see how to code `read`:

```
PROC read(pn, x, i) =
     VAR f := dir(pn),
         diskData := + : (da :IN f.das | | disk.read(E{da, 1})},
         fileData := diskData.seg(0, f.size) |
          RET fileData.seg(x, i)
```

To code `write` we need a way to allocate free `DA`s; we defer this to the next section.

There are two problems with using this representation directly:

1. The index takes up quite a lot of space (with 4 byte `DA`'s and `DBSize` = 1Kbyte it takes .4% of the disk). Since RAM costs about 50 times as much as disk, keeping it all in RAM will add about 20% to the cost of the disk, which is a significant dollar cost. On the other hand, if the index is not in RAM it will take two disk accesses to read from a random file address, which is a significant performance cost.

2. The index is of variable length with no small upper bound, so representing the index on the disk is not trivial either.

To solve the first problem, store `Disk.E`'s in the index rather than `DA`'s. A single extent can represent lots of disk blocks, so the total size of the index can be much less. Following this idea, we would represent the file by a sequence of `Disk.E`'s, stored in a single disk block if it isn't too big or in a file otherwise. This recursion obviously terminates. It has the drawback that random access to the file might become slow if there are many extents, because it's necessary to search them linearly to find the extent that contains byte `x` of the file.

To solve the second problem, use some kind of tree structure to represent the index. In standard Unix file systems, for example, the index is a structure called an *inode* that contains:

a sequence of 10 DA's (enough for a 10 KB file, which is well above the median file size), followed by

the DA of an *indirect* DB that holds DBSize/4 = 250 or so DA's (enough for a 250 KB file), followed by

the DA of a second-level indirect block that holds the DA's of 250 indirect blocks and hence points to $250^2 = 62500$ DA's (enough for a 62 MB file),

and so forth. The third level can address a 16 GB file, which is enough for today's systems.

Thus the inode itself has room for 13 DA's. These systems duck the first problem; their extents are always a single disk block.

We give code for that incorporates both extents and trees, representing a file by a generalized extent that is a tree of extents. The leaves of the tree are *basic* extents Disk.E, that is, references to contiguous sequences of disk blocks, which are the units of i/o for disk.read and disk.write. The purpose of such a general extent is simply to define a sequence of disk addresses, and the E.das method computes this sequence so that we can use it in invariants and abstraction functions. The tree structure is there so that the sequence can be stored and modified more efficiently.

An extent that contains a sequence of basic extents is called a *linear* extent. To do fast i/o operations, we need a linear extent which includes just the blocks to be read or written, grouped into the largest possible basic extents so that disk.read and disk.write can work efficiently. Flatten computes such a linear extent from a general extent; the spec for Flatten given below flattens the entire extent for the file and then extracts the smallest segment that contains all the blocks that need to be touched.

Read and Write just call Flatten to get the relevant linear extent and then call disk.read and disk.write on the basic extents; Write may extend the file first, and it may have to read the first and last blocks of the linear extent if the data being written does not fill them, since the disk can only write entire blocks. Extending or truncating a file is more complex, because it requires changing the extent, and also because it requires allocation. Allocation is described in the next section. Changing the extent requires changing the tree.

The tree itself must be represented in disk blocks; methods inspired by B-trees can be used to change it while keeping it balanced. Our code shows how to extract information from the tree, but not how it is represented in disk blocks or how it is changed. In standard Unix file systems, changing the tree is fairly simple because a basic extent is always a single disk block in the multi-level indirect block scheme described above.

We give the abstraction function to the simple code above. It just says that the DAs of a file are the ones you get from Flatten.

The code below makes heavy use of function composition to apply some function to each element of a sequence: s * f is {f(s(0)), ..., f(s(s.size-1))}. If f yields an integer or a sequence, the combination + : (s * f) adds up or concatenates all the f(s(i)).





```
MODULE FSImpl =                                          % implements File

TYPE N          =  Nat
     E          =  [c: (Disk.DA + SE), size: N]          % size = # of DA's in e
                   SUCHTHAT (\e | Size(e) = e.size)
                   WITH {das:=EToDAs, le:=EToLE}
     BE         =  E SUCHTHAT (\e| e.c IS Disk.DA)       % Basic Extent
     LE         =  E SUCHTHAT (\e| e.c IS SEQ BE)        % Linear Extent: sequence of BEs
                   WITH {"+":=Cat}
     SE         =  SEQ E                                 % Sequence of Extents: may be tree

     X          =  File.X
     F          =  [e, size: X]                          % size = # of bytes

     PN         =  File.PN                               % Path Name

CONST DBSize    := 1024

VAR  d          :  File.PN -> F := {}
     disk

% ABSTRACTION FUNCTION File.d = (LAMBDA (pn) -> File.F = d!pn =>
% The file is the first f.size bytes in the disk blocks of the extent f.e
     VAR f := d(pn),
         data := + : {be :IN Flatten(f.e, 0, f.e.size).c | | disk.read(be)} |
          RET data.seg(0, f.size) )

% ABSTRACTION FUNCTION FImpl0.d = (LAMBDA (pn) -> FImpl0.F =
     VAR f := d(pn) | RET {be :IN Flatten(f.e, 0, f.e.size).c | | be.c}

FUNC Size(e) -> Int = RET ( e IS BE => e.size [*] + :(e.c * Size) )
% # of DA's reachable from e. Should be equal to e.size.

FUNC EToDAs(e) -> SEQ DA =                                % e.das
% The sequence of DA's defined by e. Just for specs.
     RET ( e IS BE => {i :IN e.size.seq | | e.c + i} [*] + :(e.c * EToDAs) )

FUNC EToLE(e) -> LE =                                     % e.le
% The sequence of BE's defined by e.
     RET ( e IS BE => LE{SE{e}, e.size}              [*] + :(e.c * EToLE ) )

FUNC Cat(le1, le2) -> LE =
% The "+" method of LE. Merge e1 and e2 if possible.
     IF   e1 = {} => RET le2
     []   e2 = {} => RET le1
     []   VAR e1 := le1.c.last, e2 := le2.c.head, se |
          IF   e1.c + e1.size = e2.c =>
               se := le1.c.reml + SE{E{e1.c, e1.size + e2.size}} + le2.c.tail
          [*]  se := le1.c + le2.c
          FI;
          RET LE{se, le1.size + le2.size}
     FI

FUNC Flatten(e, start: N, size: N) -> LE = VAR le0 := e.le, le1, le2, le3 |
% The result le is such that le.das = e.das.seg(start, size);
% This is fewer than size DA's if e gets used up.
```

```
% It's empty if start >= e.size.
% This is not practical code; see below.
        le0 = le1 + le2 + le3
     /\ le1.size = {start, e.size}.min
     /\ le2.size = {size, {e.size - start, 0}.max}.min
     => RET le2

...

END FSImpl
```

This version of `Flatten` is not very practical; in fact, it is more like a spec than code for. A practical one, given below, searches the tree of extents sequentially, taking the largest possible jumps, until it finds the extent that contains the `start`th `DA`. Then it collects extents until it has gotten `size` `DA`'s. Note that because each `e.size` gives the total number of `DA`'s in `e`, `Flatten` only needs time `log(e.size)` to find the first extent it wants, provided the tree is balanced. This is a standard trick for doing efficient operations on trees: summarize the important properties of each subtree in its root node. A further refinement (which we omit) is to store cumulative sizes in an `SE` so that we can find the point we want with a binary search rather than the linear search in the `DO` loop below; we did this in the editor buffer example of handout 3.

```
FUNC Flatten(e, start: N, size: N) -> LE =
     VAR z := {size, {e.size - start, 0}.max}.min |
          IF  z = 0   => RET E{c := SE{},        size := 0}
          [*] e IS BE => RET E{c := e.c + start, size := z}.le
          [*] VAR se := e.c AS SE, sbe : SEQ BE := {}, at := start, want := z |
               DO want > 0 =>                  % maintain at + want <= Size(se)
                    VAR e1 := se.head, e2 := Flatten(e1, at, want) |
                         sbe := sbe + e2.c; want := want - e2.size;
                         se := se.tail; at := {at - e1.size, 0}.max
               OD;
               RET E{c := sbe, size := z}
          FI
```

## Allocation

We add something to the state to keep track of which disk blocks are free:

```
VAR free: DA -> Bool
```

We want to ensure that a free block is not also part of a file. In fact, to keep from losing blocks, a block should be free iff it isn't in a file or some other data structure such as an inode:

```
PROC IsReachable(da) -> Bool =
     RET ( EXISTS f :IN d.rng | da IN f.e.das \/ ...

% INVARIANT (ALL da | IsReachable(da) = ~ free(da) )
```

This can't be coded without some sort of log-like mechanism for atomicity if we want separate representations for `free` and `f.e`, that is, if we want any code for `free` other than the brute-force search implied by `IsReachable` itself. The reason is that the only atomic operation we have on the disk is to write a single block, and we can't hope to update the representations of both `free` and `f.e` with a single block write. But `~ IsReachable` is not satisfactory code for `free`, even





though it does not require a separate data structure, because it's too expensive — it traces the entire extent structure to find out whether a block is free.

A weaker invariant allows blocks to be lost, but still ensures that the file data will be inviolate. This isn't as bad as it sounds, because blocks will only be lost if there is a crash between writing the allocation state and writing the extent. Also, it's possible to garbage-collect the lost blocks.

```
% INVARIANT (ALL da | IsReachable(da) ==> ~ free(da))
```

A weaker invariant than this would be a disaster, since it would allow blocks that are part of a file to be free and therefore to be allocated for another file.

The usual representation of `free` is a `SEQ Bool` (often called a *bit table*). It can be stored in a fixed-size file that is allocated by magic (so that the code for allocation doesn't depend on itself). To reduce the size of `free`, the physical disk blocks may be grouped into larger units (usually called 'clusters') that are allocated and deallocated together.

This is a fairly good scheme. The only problem with it is that the table size grows linearly with the size of the disk, even when there are only a few large files, and concomitantly many bits may have to be touched to allocate a single extent. This will certainly be true if the extent is large, and may be true anyway if lots of allocated blocks must be skipped to find a free one.

The alternative is a tree of free extents, usually coded as a B-tree with the extent size as the key, so that we can find an extent that exactly fits if there is one. Another possibility is to use the extent address as the key, since we also care about getting an extent close to some existing one. These goals are in conflict. Also, updating the B-tree atomically is complicated. There is no best answer.

## Encoding and decoding

To store complicated values on the disk, such as the function that constitutes a directory, we need to encode them into a byte sequence, since `Disk.Data` is `SEQ Byte`. (We also need encoding to send values in messages, an important operation later in the course.) It's convenient to do this with a pair of functions for each type, called `Encode` and `Decode`, which turn a value of the type into a byte sequence and recover the value from the sequence. We package them up into an `EncDec` pair.

```
TYPE Q          =  SEQ Byte
     EncDec     =  [enc: Any -> Q, dec: Q -> Any]     % Encode/Decode pair
                   SUCHTHAT (\ed: EncDec | ( EXISTS T: SET Any |
                                ed.enc.dom = T
                              /\ (ALL t :IN T | dec(enc(t)) = t) ))
```

Other names for 'encode' are 'serialize' (used in Java), 'pickle', and 'marshal' (used for encoding arguments and results of remote procedure calls).

A particular `EncDec` works only on values of a single type (represented by the set `T` in the `SUCHTHAT`, since you can't quantify over types in Spec). This means that `enc` is defined exactly on values of that type, and `dec` is the inverse of `enc` so that the process of encoding and then decoding does not lose information. We do *not* assume that `enc` is the inverse of `dec`, since there

may be many byte sequences that decode to the same value; for example, if the value is a set, it would be pointless and perhaps costly to insist on a canonical ordering of the encoding. In this course we will generally assume that every type has methods `enc` and `dec` that form an `EncDec` pair.

A type that has other types as its components can have its `EncDec` defined in an obvious way in terms of the `EncDec`'s of the component types. For example, a `SEQ T` can be encoded as a sequence of encoded `T`'s, provided the decoding is unambiguous. A function `T -> U` can be encoded as a set or sequence of encoded `(T, U)` pairs.

A directory is one example of a situation in which we need to encode a sequence of values into a sequence of bytes. A log is another example of this, discussed below, and a stream of messages is a third. It's necessary to be able to parse the encoded byte sequence unambiguously and recover the original values. We can express this idea precisely by saying that a parse is an `EncDec` sequence, a language is a set of parses, and the language is unambiguous if for every byte sequence `q` the language has at most one parse that can completely decode `q`.

```
TYPE M          =  SEQ Q                                % for segmenting a Q
     P          =  SEQ EncDec                           % Parse
     % A sequence of decoders that parses a Q, as defined by IsParse below
     Language   =  SET P

FUNC IsParse(p, q) -> Bool = RET ( EXISTS m |
        + :m = q                                        % m segments q
     /\ m.size = p.size                                 % m is the right size
     /\ (ALL i :IN p.dom | (p(i).dec)!m(i)] )           % each p decodes its m

FUNC IsUnambiguous(l: Language) -> Bool = RET (ALL q, p1, p2|
     p1 IN l /\ p2 IN l /\ IsParse(p1, q) /\ IsParse(p2, q) ==> p1 = p2)
```

Of course ambiguity is not decidable in general. The standard way to get an unambiguous language for encodings is to use type-length-value (TLV) encoding, in which the result `q` of `enc(x)` starts with some sort of encoding of `x`'s type, followed by an encoding of `q`'s own length, followed by a `Q` that contains the rest of the information the decoder needs to recover `x`.

```
FUNC IsTLV(ed: EncDec) -> Bool =
     RET (ALL x :IN ed.enc.dom | (EXISTS d1, d2, d3 |
             ed.enc(x) = d1 + d2 + d3 /\ EncodeType(x) = d1
          /\ (ed.enc(x).size).enc = d2 ))
```

In many applications there is a grammar that determines each type unambiguously from the preceding values, and in this case the types can be omitted. For instance, if the sequence is the encoding of a `SEQ T`, then it's known that all the types are `T`. If the length is determined from the type it can be omitted too, but this is done less often, since keeping the length means that the decoder can reliably skip over parts of the encoded sequence that it doesn't understand. If desired, the encodings of different types can make different choices about what to omit.

There is an international standard called ASN-1 (for Abstract Syntax Notation) that defines a way of writing a grammar for a language and deriving the `EncDec` pairs automatically from the grammar. Like most such standards, it is rather complicated and often yields somewhat inefficient encodings. It's not as popular as it used to be, but you might stumble across it.





Another standard way to get an unambiguous language is to encode into S-expressions, in which the encoding of each value is delimited by parentheses, and the type, unless it can be omitted, is given by the first symbol in the S-expression. A variation on this scheme which is popular for Internet Email and Web protocols, is to have a 'header' of the form

```
attribute1: value1
attribute2: value2
...
```

with various fairly ad-hoc rules for delimiting the values that are derived from early conventions for the human-readable headers of Email messages.

The trendy modern version serialization languae is called XML (eXtensible Markup Language). It generalizes S-expressions by having labeled parentheses, which you write `<foo>` and `</foo>`.

In both TLV and S-expression encodings, decoding depends on knowing exactly where the byte sequence starts. This is not a problem for `Q`'s coming from a file system, but it is a serious problem for `Q`'s coming from a wire or byte stream, since the wire produces a continuous stream of voltages, bits, bytes, or whatever. The process of delimiting a stream of symbols into `Q`'s that can be decoded is called *framing*; we will discuss it later in connection with networks.

## Directories

Recall that a `D` is just a `PN -> F`. We have seen various ways to represent `F`. The simplest code relies on an `EncDec` for an entire `D`. It represents a `D` as a file containing `enc` of the `PN -> F` map as a set of ordered pairs.

There are two problems with this scheme:

- Lookup in a large `D` will be slow, since it requires decoding the whole `D`. This can be fixed by using a hash table or B-tree. Updating the `D` can still be done as in the simple scheme, but this will also be slow. Incremental update is possible, if more complex; it also has atomicity issues.
- If we can't do an atomic file write, then when updating a directory we are in danger of scrambling it if there is a crash during the write. There are various ways to solve this problem. The most general and practical way is to use the transactions explained in the next section.

It is very common to code directories with an extra level of indirection called an 'inode', so that we have

```
TYPE INo        =  Int                                  % Inode Number
     D          =  PN -> INo
     INoMap     =  INo -> F

VAR  d          :  D := {}
     inodes     :  INoMap := {}
```

You can see that `inodes` is just like a directory except that the names are `INo`'s instead of `PN`'s. There are three advantages:

Because INo's are integers, they are cheaper to store and manipulate. It's customary to provide an Open operation to turn a PN into an INo (usually through yet another level of indirection called a 'file descriptor'), and then use the INo as the argument of Read and Write.

Because INo's are integers, if F is fixed-size (as in the Unix example discussed earlier, for instance) then inodes can be represented as an array on the disk that is just indexed by the INo.

The enforced level of indirection means that file names automatically get the semantics of pointers or memory addresses: two of them can point to the same file variable.

The third advantage can be extended by extending the definition of D so that the value of a PN can be another PN, usually called a "symbolic link".

```
TYPE D          =  PN -> (INo + PN)
```

## Transactions

We have seen several examples of a general problem: to give a spec for what happens after a crash that is acceptable to the client, and code for that satisfies the spec even though it has only small atomic actions at its disposal. In writing to a file, in maintaining allocation information, and in updating a directory, we wanted to make a possibly large state change atomic in the face of crashes during its execution, even though we can only write a single disk block atomically.

The general technique for dealing with this problem is called *transactions*. General transactions make large state changes atomic in the face of arbitrary concurrency as well as crashes; we will discuss this later. For now we confine ourselves to 'sequential transactions', which only take care of crashes. The idea is to conceal the effects of a crash entirely within the transaction abstraction, so that its clients can program in a crash-free world.

The code for sequential transactions is based on the very general idea of a *deterministic state machine* that has inputs called *actions* and makes a deterministic transition for every input it sees. The essential observation is that:

> If two instances of a deterministic state machine start in the same state and see the same inputs, they will make the same transitions and end up in the same state.

This means that if we record the sequence of inputs, we can replay it after a crash and get to the same state that we reached before the crash. Of course this only works if we start in the same state, or if the state machine has an 'idempotency' property that allows us to repeat the inputs. More on this below.

Here is the spec for sequential transactions. There's a state that is queried and updated (read and written) by actions. We keep a stable version ss and a volatile version vs. Updates act on the volatile version, which is reset to the stable version after a crash. A 'commit' action atomically sets the stable state to the current volatile state.





```
MODULE SeqTr [                                  % Sequential Transaction
    V,                                          % Value of an action
    S WITH { s0: ()-> S }                       % State; s0 initially
    ] EXPORT Do, Commit, Crash =

TYPE A          =  S->(V, S)                    % Action

VAR  ss         := S.s0()                       % Stable State
     vs         := S.s0()                       % Volatile State

APROC Do(a) -> V = << VAR v | (v, vs) := a(vs); RET v >>
APROC Commit() = << ss := vs >>
APROC Crash () = << vs := ss >>                 % Abort is the same

END SeqTr
```

In other words, you can do a whole series of actions to the volatile state vs, followed by a Commit. Think of the actions as reads and writes, or queries and updates. If there's a crash before the Commit, the state reverts to what it was initially. If there's a crash after the Commit, the state reverts to what it was at the time of the commit. An action is just a function from an initial state to a final state and a result value.

There are many coding techniques for transactions. Here is the simplest. It breaks each action down into a sequence of *updates*, each one of which can be done atomically; the most common example of an atomic update is a write of a single disk block. The updates also must have an 'idempotency' property discussed later. Given a sequence of Do's, each applying an action, the code concatenates the update sequences for the actions in a volatile *log* that is a representation of the actions. Commit writes this log atomically to a stable log. Once the stable log is written, Redo applies the volatile log to the stable state and erases both logs. Crash resets the volatile to the stable log and then applies the log to the stable state to recover the volatile state. It then uses Redo to update the stable state and erase the logs. Note that we give S a "+" method s + l that applies a log to a state.

This scheme reduces the problem of implementing arbitrary changes atomically to the problem of atomically writing an arbitrary amount of stuff to a log. This is easier but still not trivial to do efficiently; we discuss it at the end of the section.

```
MODULE LogRecovery [                            % implements SeqTr
    V,                                          % Value of an action
    S0 WITH { s0: () -> S0 }                    % State
    ] EXPORT Do, Commit, Crash =

TYPE A          =  S->(V, S)                    % Action
     U          =  S -> S                       % atomic Update
```

```
    L           =  SEQ U                                % Log
    S           =  S0 WITH { "+":=DoLog }               % State; s+l applies l to s

VAR ss          := S.s0()                               % Stable State
    vs          := S.s0()                               % Volatile State
    sl          := L{}                                  % Stable Log
    vl          := L{}                                  % Volatile Log

% ABSTRACTION to SeqTr
    SeqTr.ss = ss + sl
    SeqTr.vs = vs

% INVARIANT vs = ss + vl

FUNC DoLog(s, l) -> S =                                 % s+l = DoLog(s, l)
% Apply the updates in l to the state s.
    l={} => RET s [*] RET DoLog((l.head)(s),l.tail))

APROC Do(a) -> V =
% Find an l (a sequence of updates) that has the same effect as a on the current state.
    << VAR v, l | (v, vs + l) = a(vs) =>
          vl := vl + l; vs := vs + l; RET v >>

PROC Commit() = << sl := vl >>; Redo()

PROC Redo() =                                           % replay vl, then clear sl
    DO vl # {} => << ss := ss + vl.head; vl := vl.tail >> OD; << sl := {} >>

PROC Crash() =
    CRASH;
    << vl := {}; vs := S.s0()  >>;                      % crash erases vs, vl
    << vl := sl; vs := ss + vl >>;                      % recovery restores them
    Redo()                                              % and repeats the Redo; this is optional

END LogRecovery
```

For this *redo* crash recovery to work, `l` must have the property that repeatedly applying prefixes of it, followed by the whole thing, has the same effect as applying the whole thing. For example, suppose `l = L{a,b,c,d,e}`. Then `L{a,b,c, a, a, a,b,c,d, a,b, a,b,c,d,e, a, a,b,c,d,e}` must have the same effect as `l` itself; here we have grouped the prefixes together for clarity. We need this property because a crash can happen while `Redo` is running; the crash reapplies the whole log and runs `Redo` again. Another crash can happen while the second `Redo` is running, and so forth.

This 'hiccup' property follows from 'log idempotence':

```
    s + l + l = s + l                                   (1)
```

From this we get (recall that < is the 'prefix' predicate for sequences).

```
    k < l ==> (s + k + l = s + l)                       (2)
```

because `k < l` implies there is a `l'` such that `k + l' = l`, and hence

```
      s + k + l = s + k + (k + l') = (s + k + k) + l'
    = (s + k) + l' = s + (k + l') = s + l
```

From (2) we get the property we want:

```
    IsHiccups(k, l) ==> (s + k + l = s + l)             (3)
```

where





```
FUNC IsHiccups(k, l) -> Bool =
% k is a sequence of attempts to complete l
    RET    k = {}
        \/ (EXISTS k', l'|    k = k' + l' /\ l' # {} /\ l' <= l
                           /\ IsHiccups(k', l) )
```

because we can keep absorbing the last hiccup `l'` into the final complete `l`. For example, taking some liberties with the notation for sequences:

```
      abcaaabcdababcdeaabcde
    = abcaaabcdababcde + (a + abcde)
    = abcaaabcdababcde + abcde                          by (2)
    = abcaaabcdab + (abcde + abcde)
    = abcaaabcdab + abcde                               by (2)
    = abcaaabcd + (ab + abcde)
    = abcaaabcd + abcde                                 by (2)
```

and so forth.

To prove (3), observe that

```
    IsHiccups(k, l) /\ k # {} ==> k = k' + l' /\ l' <= l /\ IsHiccups(k', l).
```

Hence

```
    s+k+l = (s+k')+l'+l = s+k'+l                        by (2)
```

and `k' < k`. But we have `IsHiccups(k', l)`, so we can proceed by induction until `k' = {}` and we have the desired result.

We can get log idempotence if the `U`'s commute and are idempotent (that is, `u * u = u`), or if they are all writes. More generally, for arbitrary `U`'s we can attach a `UID` to each `U` and record it in `S` when the `U` is applied, so we can tell that it shouldn't be applied again. Calling the original state `SS`, and defining a `meaning` method that turns a `U` record into a function, we have

```
TYPE
    S           =  [ss, tags: SET UID]
    U           =  [uu: SS->SS, tag: UID] WITH { meaning:=Meaning }

FUNC Meaning(u, s)->S =
        u.tag IN s.tags => RET s                        % u already done
    [*] RET S{ (u.uu)(s.ss), s.tags + {u.tag} }
```

If all the `U`'s in `l` have different tags, we get log idempotence. The tags make `U`'s 'testable' in the jargon of transaction processing; after a crash we can test to find out whether a `U` has been done or not. In the standard database code each `U` works on one disk page, the tag is the 'log sequence number', the index of the update in the log, and the update writes the tag on the disk page.

*Writing the log atomically*

There is still an atomicity problem in this code: `Commit` atomically does `<< sl := vl >>`, and the logs can be large. A simple way to use a disk to code a log that requires this assignment of arbitrary-sized sequences is to keep the size of `sl` in a separate disk block, and to write all the data first, then do a `Sync` if necessary, and finally write the new size. Since `sl` is always empty before this assignment, in this representation it will remain empty until the single `Disk.write` that sets its size. This is rather wasteful code, since it does an extra disk write.

More efficient code writes a 'commit record' at the end of the log, and treats the log as empty unless the commit record is present. Now it's only necessary to ensure that the log can never be mis-parsed if a crash happens while it's being written. An easy way to accomplish this is to write a distinctive 'erased value into each disk block that may become part of the log, but this means that for every disk write to a log block, there will be another write to erase it. To avoid this cost we can use a ring buffer of disk blocks for the log and a sequence number that increments each time the ring buffer wraps around; then a block is 'erased' if its sequence number is not the current one. There's still a cost to initialize the sequence numbers, but it's only paid once. With careful code, a single bit of sequence number is enough.

In some applications it's inconvenient to make room in the data stream for a sequence number every `DBsize` bytes. To get around this, use a 'displaced' representation for the log, in which the first data bit of each block is removed from its normal position to make room for the one bit sequence number. The displaced bits are written into their own disk blocks at convenient intervals.

Another approach is to compute a strong checksum for the log contents, write it at the end after all the other blocks are known to be on the disk, and treat the log as empty unless a correct checksum is present. With a good *n*-bit checksum, the probability of mis-parsing is $2^{-n}$.

## Redundancy

A disk has many blocks. We would like some assurance that the failure of a single block will not damage a large part of the file system. To get such assurance we must record some critical parts of the representation redundantly, so that they can be recovered even after a failure.

The simplest way to get this effect is to record *everything* redundantly. This gives us more: a single failure won't damage *any* part of the file system. Unfortunately, it is expensive. In current systems this is usually done at the disk abstraction, and is called *mirroring* or *shadowing* the disk.

The alternative is to record redundantly only the information whose loss can damage more than one file: extent, allocation, and directory information.

Another approach is to

do all writes to a log,

keep a copy of the log for a long time (by writing it to tape, usually), and

checkpoint the state of the file system occasionally.

Then the current state can be recovered by restoring the checkpoint and replaying the log from the moment of the checkpoint. This method is usually used in large database systems, but not in any file systems that I know of.

We will discuss these methods in more detail near the end of the course.





## Copying File Systems

The file system described in `FSImpl` above separates the process of adding `DB`'s to the representation of a file from the process of writing data into the file. A *copying* file system (CFS) combines these two processes into one. It is called a 'log-structured' file system in the literature[1], but as we shall see, the log is not the main idea. A CFS is based on three ideas:

- Use a generational copying garbage collector (called a *cleaner*) to reclaim `DB`'s that are no longer reachable and keep all the free space in a single (logically) contiguous region, so that there is no need for a bit table or free list to keep track of free space.
- Do *all* writes sequentially at one end of this region, so that existing data is never overwritten and new data is sequential.
- Log and cache updates to metadata (the index and directory) so that the metadata doesn't have to be rewritten too often.

A CFS is a very interesting example of the subtle interplay among the ideas of sequential writing, copying garbage collection, and logging. This section describes the essentials of a CFS in detail and discusses more briefly a number of refinements and practical considerations. It will repay careful study.

Here is a picture of a disk organized for a CFS:

```
abc==defgh====ijkl=m=nopqrs----------------
```

In this picture letters denote reachable blocks, `=`'s denote unreachable blocks that are not part of the free space, and `-`'s denote free blocks (contiguous on the disk viewed as a ring buffer). After the cleaner copies blocks `a-e` the picture is

```
-------fgh====ijkl=m=nopqrsabcde------------
```

because the data `a-e` has been copied to free space and the blocks that used to hold `a-e` are free, together with the two unreachable blocks which were not copied. Then after blocks `g` and `j` are overwritten with new values `G` and `J`, the picture is

```
-------f=h====i=kl=m=nopqrsabcdeGJ----------
```

The new data `G` and `J` has been written into free space, and the blocks that used to hold `g` and `j` are now unreachable. After the cleaner runs to completion the picture is

```
--------------------nopqrsabcdeGJfhiklm----
```

*Pros and cons*

A CFS has two main advantages:

- All writing is done sequentially; as we know, sequential writes are much faster than random writes. We have a good technique for making disk reads faster: caching. As main memory caches get bigger, more reads hit in the cache and disks spend more of their time writing, so we need a technique to make writes faster.

[1] M. Rosenblum and J. Osterhout, The design and implementation of a log-structured file system, *ACM Transactions on Computer Systems*, **10**, 1, Feb. 1992, pp 26-52.

- The cleaner can copy reachable blocks to anywhere, not just to the standard free space region, and can do so without interfering with normal operation of the system. In particular, it can copy reachable blocks to tape for backup, or to a different disk drive that is faster, cheaper, less full, or otherwise more suitable as a home for the data.

There are some secondary advantages. Since the writes are sequential, they are not tied to disk blocks, so it's easy to write items of various different sizes without worrying about how they are packed into `DB`'s. Furthermore, it's easy to compress the sequential stream as it's being written[2], and if the disk is a RAID you never have to read any blocks to recompute the parity. Finally, there is no bit table or free list of disk blocks to maintain.

There is also one major drawback: unless large amounts of data in the same file are written sequentially, a file will tend to have lots of small extents, which can cause the problems discussed on page 13. In Unix file systems most files are written all at once, but this is certainly not true for databases. Ways of alleviating this drawback are the subject of current research. The cost of the cleaner is also a potential problem, but in practice the cost of the cleaner seems to be small compared to the time saved by sequential writes.

*Updating metadata*

For the CFS to work, it must update the index that points to the `DB`'s containing the file data on every write and every copy done by the cleaner, not just when the file is extended. And in order to keep the writing sequential, we must handle the new index information just like the file data, writing it into the free space instead of overwriting it. This means that the directory too must be updated, since it points to the index; we write it into free space as well. Only the *root* of the entire file system is written in a fixed location; this root says where to find the directory.

You might think that all this rewriting of the metadata is too expensive, since a single write to a file block, whether existing or new, now triggers three additional writes of metadata: for the index (if it doesn't fit in the directory), the directory, and the root. Previously none of these writes was needed for an existing block, and only the index write for a new block. However, the scheme for logging updates that we introduced to code transactions can also handle this problem. The idea is to write the *changes* to the index into a log, and cache the updated index (or just the updates) only in main memory. An example of a logged change is "block 43 of file 'alpha' now has disk address 385672". Later (with any luck, after several changes to the same piece of the index) we write the index itself and log the consequent changes to the directory; again, we cache the updated directory. Still later we write the directory and log the changes to the root. We only write a piece of metadata when:

We run out of main memory space to cache changed metadata, or

The log gets so big (because of many writes) that recovery takes too long.

To recover we replay the *active tail* of the log, starting before the oldest logged change whose metadata hasn't been rewritten. This means that we must be able to read the log sequentially

[2] M. Burrows et al., On-line compression in a log-structured file system, *Proc. 5th Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992, pp 2-9. This does require some blocking so that the decompressor can obtain the initial state it needs.





from that point. It's natural to write the log to free space along with everything else. While we are at it, we can also log other changes like renames.

Note that a CFS can use exactly the same directory and index data as an ordinary file system, and in fact exactly the same code for `Read`. To do this we must give up the added flexibility we can get from sequential writing, and write each `DB` of data into a `DB` on the disk. Several codes have done this (but the simple code below does not).

The logged changes serve another purpose. Because a file can only be reached from a single directory entry (or inode), the cleaner need not trace the directory structure in order to find the reachable blocks. Instead, if the block at `da` was written as block `b` of file `f`, it's sufficient to look at the file index and find out whether block `b` of file `f` is still at `da`. But the triple `(b, f, da)` is exactly the logged change. To take advantage of this we must keep the logged change as long as `da` remains reachable since the cleaner needs it (it's called 'segment summary' information in the literature). We don't need to replay it on recovery once its metadata is written out, however, and hence we need the sequential structure of the log only for the active tail.

Existing CFS's use the extra level of naming called inodes that is described on page 19. The inode numbers don't change during writing or copying, so the `PN -> INo` directory doesn't change. The root points to index information for the inodes (called the 'inode map'), which points to inodes, which point to data blocks or, for large files, to indirect blocks which point to data blocks.

*Segments*

Running the cleaner is fairly expensive, since it has to read and write the disk. It's therefore important to get as much value out of it as possible, by cleaning lots of unreachable data instead of copying lots of data that is still reachable. To accomplish this, divide the disk into *segments*, large enough (say 1 MB or 10 MB) that the time to seek to a new segment is much smaller than the time to read or write a whole segment. Clean each segment separately. Keep track of the amount of unreachable space in each segment, and clean a segment when (unreachable space) * (age of data) exceeds a threshold. Rosenblum and Osterhout explain this rule, which is similar in spirit to what a generational garbage collector[3] does; the goal is to recover as much free space as possible, without allowing too much unreachable space to pile up in old segments.

Now the free space isn't physically contiguous, so we must somehow link the segments in the active tail together. We also need a table that keeps track for each segment of whether it is free, and if not, what its unreachable space and age are; this is cheap because segments are so large.

*Backup*

As we mentioned earlier, one of the major advantages of a CFS is that it is easier to back up. There are several reasons for this.

[3] H. Lieberman and C. Hewitt, A real-time garbage collector based on the lifetimes of objects, *Comm. ACM* **26**, 6, June 1983, pp 419-429.

1. You can take a snapshot just by stopping the cleaner from freeing cleaned segments, and then copy the root information and the log to the backup medium, recording the logged data backward from the end of the log.
2. This backup data structure allows a single file (or a small set of files) to be restored in one pass.
3. It's only necessary to copy the log back to the point at which the previous backup started.
4. The disks reads done by backup are sequential and therefore fast. This is an important issue when the file system occupies many terabytes. At the 10 MB/s peak transfer rate of the disk, it takes $10^5$ seconds, or a bit more than one day, to copy a terabyte. This means that a small number of disks and tapes running in parallel can do it in a fraction of a day. If the transfer rate is reduced to 1 MB/s by lots of seeks (which is what you get with random seeks if the average block size is 10 KB), the copying time becomes 10 days, which is impractical.
5. If a large file is partially updated, only the updates will be logged and hence appear in the backup.
6. It's easy to merge several incremental backups to make a full backup.

To get these advantages, we have to retain the ordering of segments in the log even after recovery no longer needs it.

There have been several research implementations of CFS's, and at least one commercial one called Spiralog in Digital Equipment Corporation's (now Compaq's) VMS system. You can read a good deal about it at http://www.digital.com/info/DTJM00/.

## A simple CFS code

We give code for `CopyingFS` of a CFS that contains all the essential ideas (except for segments, and the rule for choosing which segment to clean), but simplifies the data structures for the sake of clarity. `CopyingFS` treats the disk as a root `DB` plus a ring buffer of bytes. Since writing is sequential this is practical; the only cost is that we may have to pad to the end of a `DB` occasionally in order to do a `Sync`. A `DA` is therefore a byte address on the disk. We could dispense with the structure of disk blocks entirely in the representation of files, just write the `data` of each `File.Write` to the disk, and make a `FSImpl.BE` point directly to the resulting byte sequence on the disk. Instead, however, we will stick with tradition, take `BE = DA`, and represent a file as a `SEQ DA` plus its size.

So the disk consists of a root page, a busy region, and a free region (as we have seen, in a real system both busy and free regions would be divided into segments); see the figure below. The busy region is a sequence of encoded `Item`'s, where an `Item` is either a `D` or a `Change` to a `DB` in a file or to the `D`. The busy region starts at `busy` and ends just before `free`, which always points to the start of a disk block. We could write `free` into the root, but then making anything stable would require a (non-sequential) write of the root. Instead, the busy region ends with a recognizable `endDB`, put there by `Sync`, so that recovery can find the end of the busy region.





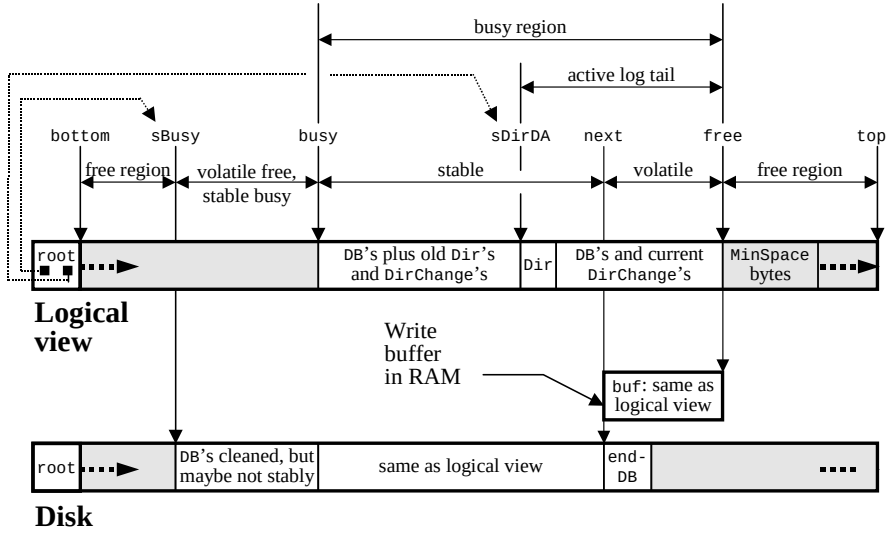
`dDA` is the address of the latest directory on the disk. The part of the busy region after `dDA` is the active tail of the log and contains the changes that need to be replayed during recovery to reconstruct the current directory; this arrangement ensures that we start the replay with a `d` to which it makes sense to apply the changes that follow.

This code does bytewise writes that are buffered in `buf` and flushed to the disk only by `Sync`. Hence after a crash the state reverts to the state at the last `Sync`. Without the replay done during recovery by `ApplyLog`, it would revert to the state the last time the root was written; be sure you understand why this is true.

We assume that a sequence of encoded `Item`'s followed by an `endDB` can be decoded unambiguously. See the earlier discussion of writing logs atomically.

Other simplifications:

1. We store the `SEQ DA` that points to the file `DB`'s right in the directory. In real life it would be a tree, along one of the lines discussed in `FSImpl`, so that it can be searched and updated efficiently even when it is large. Only the top levels of the tree would be in the directory.
2. We keep the entire directory in main memory and write it all out as a single `Item`. In real life we would cache parts of it in memory and write out only the parts that are dirty (in other words, that contain changes).
3. We write a data block as part of the log entry for the change to the block, and make the `DA`'s in the file representation point to these log entries. In real life the logged change information would be batched together (as 'segment summary information') and the data written



separately, so that recovery and cleaning can read the changes efficiently without having to read the file data as well, and so that contiguous data blocks can be read with a single disk operation and no extra memory-to-memory copying.

4. We allocate space for data in `Write`, though we buffer the data in `buf` rather than writing it immediately. In real life we might cache newly written data in the hope that another adjacent write will come along so that we can allocate contiguous space for both writes, thus reducing the number of extents and making a later sequential read faster.

5. Because we don't have segments, the cleaner always copies items starting at `busy`. In real life it would figure out which segments are most profitable to clean.

6. We run the cleaner only when we need space. In real life, it would run in the background to take advantage of times when the disk is idle, and to maintain a healthy amount of free space so that writes don't have to wait for the cleaner to run.

7. We treat `WriteData` and `WriteRoot` as atomic. In real life we would use one of the techniques for making log writes atomic that are described on page 23.

8. We treat `Init` and `Crash` as atomic, mainly for convenience in writing invariants and abstraction functions.In real life they do several disk operations, so we have to lock out external invocations while they are running.

9. We ignore the possibility of errors.

```
MODULE CopyingFS EXPORTS PN, Sync =                  % implements File, uses Disk

TYPE DA          =  Nat                              % Disk Address in bytes
                    WITH "+":=DAAdd, "-":=DASub}
     LE          =  SEQ DA                           % Linear Extent
     Data        =  File.Data

     X           =  File.X
     F           =  [le, size: X]                    % size = # of bytes

     PN          =  String WITH [...]                % Path Name
     D           =  PN -> F

     Item        =  (DBChange + DChange + D + Pad)   % item on the disk
     DBChange    =  [pn, x, db]                      % db is data at x in file pn
     DChange     =  [pn, dOp, x]                     % x only for SetSize
     DOp         =  ENUM[create, delete, setSize]
     Pad         =  [size: X]                        % For filling up a DB;
                                                     % Pad{x}.enc.size = x.

     IDA         =  [item, da]
     SI          =  SEQ IDA                          % for parsing the busy region

     Root        =  [dDA: DA, busy: DA]              % assume encoding < DBSize

CONST
     DBSize      := Disk.DBSize
     diskSize    := 1000000
```





```
     rootDA      := 0
     bottom      := rootDA + DBSize                  % smallest DA outside root
     top         := (DBSize * diskSize) AS DA
     ringSize    := top - bottom
     endDB       := DB{...}                          % starts unlike any Item

VAR                                                  % All volatile; stable data is on disk.
     d           :  D    := {}
     sDDA        :  DA   := bottom                   % = ReadRoot().dDA
     sBusy       :  DA   := Bottom                   % = ReadRoot().busy
     busy        :  DA   := bottom
     free        :  DA   := bottom
     next        :  DA   := bottom                   % DA to write buf at
     buf         :  Data := {}                       % waiting to be written
     disk                                            % the disk

ABSTRACTION FUNCTION File.d = ( LAMBDA (pn) -> File.F =
% The file is the data pointed to by the DA's in its F.
     VAR f := d(pn), diskData := + :(f.le * ReadOneDB) |
         RET diskData.seg(0, f.size) )

ABSTRACTION FUNCTION File.oldDs = { SD(), d }

INVARIANT 1: ( ALL f :IN d.rng | f.le.size * DBSize >= f.size )
% The blocks of a file have enough space for the data. From FSImpl.
```

The reason that `oldDs` doesn't contain any intermediate states is that the stable state changes only in a `Sync`, which shrinks `oldDs` to just `d`.

During normal operation we need to have the variables that keep track of the region boundaries and the stable directory arranged in order around the disk ring, and we need to maintain this condition after a crash. Here are the relevant current and post-crash variables, in order (see below for `MinSpace`). The 'post-crash' column gives the value that the 'current' expression will have after a crash.

| *Current* | *Post-crash* | |
|---|---|---|
| `busy` | `sBusy` | start of busy region |
| `sDDA` | `sDDA` | most recent stable `d` |
| `next` | | end of stable busy region |
| `free` | `next` | end of busy region |
| `free + minSpace()` | `next + minSpace()` | end of cushion for writes |

In addition, the stable busy region should start and end before or at the start and end of the volatile busy region, and the stable directory should be contained in both. Also, the global variables that are supposed to equal various stable variables (their names start with 's') should in fact do so. The analysis that leads to this invariant is somewhat tricky; I hope it's right.

```
INVARIANT 2:
       IsOrdered((SEQ DA){next + MinSpace(), sBusy, busy, sDDA, next, free,
                          free + MinSpace(), busy})
    /\ EndDA() = next /\ next//DBSize = 0 /\ Root{sDDA, sBusy} = ReadRoot()
```

Finally,

The busy region should contain all the items pointed to from DA's in d or in global variables.

The directory on disk at sDDA plus the changes between there and free should agree with d.

This condition should still hold after a crash.

```
INVARIANT 3:
        IsAllGood(ParseLog(busy, buf), d)
     /\ IsAllGood(ParseLog(sBusy, {}), SD())
```

The following functions are mainly for the invariants, though they are also used in crash recovery. ParseLog expects that the disk from da to the next DB with contents endDB, plus data, is the encoding of a sequence of Item's, and it returns the sequence SI, each Item paired with its DA. ApplyLog takes an SI that starts with a D and returns the result of applying all the changes in the sequence to that D.

```
FUNC ParseLog(da, data) -> SI = VAR si, end: DA |
% Parse the log from da to the next endDB block, and continue with data.
        + :(si * (\ ida | ida.item.enc) = ReadData(da, end - da) + data
     /\ (ALL n :IN si.dom - {0} |
           si(n).da = si(n-1).da + si(n-1).item.enc.size)
     /\ si.head.da = da
     /\ ReadOneDB(end) = endDB => RET si

FUNC ApplyLog(si) -> D = VAR d' := si.head.item AS D |
% si must start with a D. Apply all the changes to this D.
     DO VAR item := si.head.item |
         IF   item IS DBChange  => d'(item.pn).le(item.x/DBSize) := si.head.da
         []   item IS DChange => d' := ...              % details omitted
         [*]  SKIP                                     % ignore D and Pad
         FI; si := si.tail
     OD; RET d'

FUNC IsAllGood(si, d') -> Bool = RET
% All d' entries point to DBChange's and si agrees with d'
        (ALL da, pn, item | d'!pn /\ da IN d'(pn).le /\ IDA{item, da} IN si
                         ==> item IS DBChange)
     /\ ApplyLog(si) = d'

FUNC SD() -> D = RET ApplyLog(ParseLog(sDDA), {})
% The D encoded by the Item at sDDA plus the following DChange's

FUNC EndDA() -> DA = VAR ida := ParseLog(sDDA).last |
% Return the DA of the first endDB after sDDA, assuming a parsable log.
     RET ida.da + ida.item.enc.size
```

The minimum free space we need is room for writing out d when we are about to overwrite the last previous copy on the disk, plus the wasted space in a disk block that might have only one byte of data, plus the endDB.

```
FUNC MinSpace() -> Int = RET d.enc.size + (DBSize-1) + DBsize
```

The following Read and Write procedures are much the same as they would be in FSImpl, where we omitted them. They are full of boring details about fitting things into disk blocks; we include them here for completeness, and because the way Write handles allocation is an important part





of CopyingFS. We continue to omit the other File procedures like SetSize, as well as the handling in ApplyLog of the DChange items that they create.

```
PROC Read(pn, x, size: X) -> Data =
     VAR f     := d(pn),
         size  := {{size, f.size - x}.min, 0}.max,     % the available bytes
         n     := x/DBSize,                            % first block number
         nSize := NumDBs(x, size),                     % number of blocks
         blocks:= n .. n + nSize -1,                   % blocks we need in f.le
         data  := + :(blocks * f.le * ReadItem *       % all data in these blocks
                      (\ item | (item AS DBChange).db)) |
             RET data.seg(x//DBSize, size)             % the data requested

PROC Write(pn, x, data) = VAR f := d(pn) |
% First expand data to contain all the DB's that need to be written
     data := Data.fill(0, x - f.size) + data;          % add 0's to extend f to x
     x := {x, f.size}.min;                             % and adjust x to match
     IF   VAR y := x//DBSize | y # 0 =>                % fill to a DB in front
          x := x - y; data := Read(pn, x, y) + data
     [*]  SKIP FI;
     IF   VAR y := data.size//DBSize | y # 0 =>        % fill to a DB in back
          data + := Read(pn, x + data.size, DBSize - y)
     [*]  SKIP FI;
     % Convert data into DB's, write it, and compute the new f.le
     VAR blocks := Disk.DToB(data), n := x/DBSize,
         % Extend f.le with 0's to the right length.
         le := f.le + LE.fill(0, x + blocks.size - le.size),
         i := 0 |
             DO blocks!i =>
                 le(n + i) := WriteData(DBChange{pn, x, blocks(i)}.enc);
                 x + := DBSize; i + := 1
             OD; d(pn).le := le
```

These procedures initialize the system and handle crashes. Crash is somewhat idealized; more realistic code would read the log and apply the changes to d as it reads them, but the logic would be the same.

```
PROC Init() = disk := disk.new(diskSize); WriteD()     % initially d is empty

PROC Crash() = <<                                      % atomic for simplicity
     CRASH;
     sDDA   := ReadRoot().sDDA; d := SD();
     sBusy    := ReadRoot().busy; busy := sBusy;
     free := EndDA(); next := free; buf := {} >>
```

These functions read an item, some data, or a single DB from the disk. They are boring. ReadItem is somewhat unrealistic, since it just chooses a suitable size for the item at da so that Item.dec works. In real life it would read a few blocks at DA, determine the length of the item from the header, and then go back for more blocks if necessary. It reads either from buf or from the disk, depending on whether da is in the write buffer, that is, between next and free.

```
FUNC ReadItem(da) -> Item = VAR size: X |
     RET Item.dec( (   DABetween(da, next, free) => buf.seg(da - next, size)
     [*]  ReadData(da, size) ) )
```

```
FUNC ReadData(da, size: X) -> Data =                    % 1 or 2 disk.read's
     IF   size + da <= top =>                           % Int."+", not DA."+"
          % Read the necessary disk blocks, then pick out the bytes requested.
          VAR data := disk.read(LE{da/DBSize, NumDBs(da, size)}) |
               RET data.seg(da//DBSize, size)
     [*]  RET ReadData(da, top - da) + ReadData(bottom, size - (top - da))

PROC ReadOneDB(da) = RET disk.read(LE{da/DBSize, 1}))
```

`WriteData` writes some data to the disk. It is not boring, since it includes the write buffering, the cleaning, and the space bookkeeping. The writes are buffered in `buf`, and `Sync` does the actual disk write. In this module `Sync` is only called by `WriteD`, but since it's a procedure in `File` it can also be called by the client. When `WriteData` needs space it calls `Clean`, which does the basic cleaning step of copying a single item. There should be a check for a full disk, but we omit it. This check can be done by observing that the loop in `WriteData` advances `free` all the way around the ring, or by keeping track of the available free space. The latter is fairly easy, but `Crash` would have to restore the information as part of its replay of the log.

These write procedures are the only ones that actually write into `buf`. `Sync` and `WriteRoot` below are the only procedures that write the underlying disk.

```
PROC WriteData(data) -> DA =                            % just to buf, not disk
     DO IsFull(data.size) => Clean() OD;
     buf + := data; VAR da := free | free + := data.size; RET da

PROC WriteItem(item) = VAR q := item.enc | buf + := q; free + := q.size
% No check for space because this is only called by Clean, WriteD.

PROC Sync() =
% Actually write to disk, in 1 or 2 disk.write's (2 if wrapping).
% If we will write past sBusy, we have to update the root.
     IF (sBusy - next) + (free - next) <= MinSpace() => WriteRoot()[*]   SKIP FI;
     % Pad buf to even DB's. A loop because one Pad might overflow current DB.
     DO VAR z := buf.size//DBSize | z # 0 => buf := buf + Pad{DBSize-z}.enc OD;.
     buf := buf + endDB;                                % add the end marker DB
     <<   % atomic for simplicity
     IF   buf.size + next < top => disk.write(next/DBSize, buf)
     [*]  disk.write(next  /DBSize, buf.seg(0       , top-next  ));
          disk.write(bottom/DBSize, buf.sub(top-next, buf.size-1))
     FI;
     >>; free := next + buf.size - DBSize; next := free; buf := {}
```

The constraints on using free space are that `Clean` must not cause writes beyond the stable `sBusy` or into a disk block containing `Item`'s that haven't yet been copied. (If `sBusy` is equal to `busy` and in the middle of a disk block, the second condition might be stronger. It's necessary because a write will clobber the whole block.) Furthermore, there must be room to write an `Item` containing `d`. Invariant 2 expresses all this precisely. In real life, of course, `Clean` would be called in the background, the system would try to maintain a fairly large amount of free space, and only small parts of `d` would be dirty. `Clean` drops `DChange`'s because they are recorded in the `D` item that must appear later in the busy region.

```
FUNC IsFull(size: X) -> Bool = RET busy - free < MinSpace() + size
```





```
PROC Clean() = VAR item := ReadItem(busy) |              % copy the next item
     IF   item IS DBChange /\ d(item.pn).le(item.x/DBSize) = busy =>
          d(item.pn).le(item.x/DBSize) := free; WriteItem(item)
     []   item IS D /\ da = sDDA => WriteD()             % the latest D
     [*]  SKIP                                          % drop DChange, Pad
     FI; busy := busy + item.enc.size

PROC WriteD() =
% Called only from Clean and Init. Could call it more often to speed up recovery
%, after DO busy - free < MinSpace() => Clean() OD to get space.
     sDDA := free; WriteItem(d); Sync(); WriteRoot()
```

The remaining utility functions read and write the root, convert byte sizes to `DB` counts, and provide arithmetic on `DA`'s that wraps around from the top to the bottom of the disk. In real life we don't need the arithmetic because the disk is divided into segments and items don't cross segment boundaries; if they did the cleaner would have to do something quite special for a segment that starts with the tail of an item.

```
FUNC ReadRoot() -> Root = VAR root, pad |
     ReadOneDB(rootDA) = root.enc + pad.enc => RET root

PROC WriteRoot() = << VAR pad, db | db = Root{sDDA, busy}.enc + pad.enc =>
     disk.write(rootDA, db); sBusy := busy >>

FUNC NumDBs(da, size: X) -> Int = RET (size + da//DBSize + DBSize-1)/DBSize
% The number of DB's needed to hold size bytes starting at da.

FUNC DAAdd(da, i: Int) -> DA = RET ((da - bottom + i) // ringSize) + bottom

FUNC DASub(da, i: Int) -> DA = RET ((da - bottom - i) // ringSize) + bottom
% Arithmetic modulo the data region. abs(i) should be < ringSize.

FUNC DABetween(da, da1, da2) -> Bool = RET da = da1 \/ (da2 - da1) < (da1 - da)

FUNC IsOrdered(s: SEQ DA) -> Bool =
     RET (ALL i :IN s.dom - {0, 1} | DABetween(s(i-1), s(i-2), s(i)))

END CopyingFS
```