CS162
Operating Systems and Systems Programming
Lecture 19

File Systems continued
Distributed Systems

April 9, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Data Durability
- Beginning of Distributed Systems Discussion
  - Lisp/ML map/fold review
  - MapReduce overview

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Important "ilities"

- **Availability:** the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- **Durability:** the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- **Reliability:** the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

How to make file system durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - use special, battery-backed RAM (called non-volatile RAM or **NVRAM**) for dirty blocks in buffer cache.
- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: **independence of failure**
    - » Could put copies on one disk, but if disk head fails…
    - » Could put copies on different disks, but if server fails…
    - » Could put copies on different servers, but if building is struck by lightning….
    - » Could put copies on servers in different continents…
- **RAID:** Redundant Arrays of Inexpensive Disks
  - Data stored on multiple disks (redundancy)
  - Either in software or hardware
    - » In hardware case, done by disk controller; file system may not even know that there is more than one disk in use

Log Structured and Journaled File Systems

- Better reliability through use of log
  - All changes are treated as *transactions*
    - » A transaction either happens *completely* or *not at all*
  - A transaction is *committed* once it is written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between "Log Structured" and "Journaled"
  - Log Structured Filesystem (LFS): data stays in log form
  - Journaled Filesystem: Log used for recovery
- For Journaled system:
  - Log used to asynchronously update filesystem
    - » Log entries removed after used
  - After crash:
    - » Remaining transactions in the log performed ("Redo")
- Examples of Journaled File Systems:
  - Ext3 (Linux), XFS (Unix), NTFS (Windows)

Functional Programming Review

- Functional operations do not modify data structures – they always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter
  - fun foo(L: int list) = sum(L) + mul(L) + length(L)
  - Order of sum(), mul(), length() does not matter, since they do not modify *L*

Functional Updates Do Not Modify Structures

- fun append(x, lst) =

```
fun append(x, lst) =
  let lst' = reverse lst in
     reverse ( x :: lst' )
```

- The append() function above reverses a list, adds a new element to the front, and returns all of that, reversed, which appends an item
- But, it *never modifies* lst!

Functions Can Be Used As Arguments

- fun DoDouble(f, x) = f (f x)
- It does not matter what f does to its argument; DoDouble() will do it twice
- *What is the type of this function?*

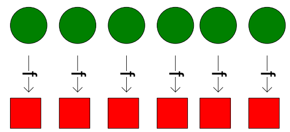
Administrivia

- Midterm #2 is next Wednesday (April 16th)
  - 6-7:30pm in 10 Evans
  - Covers projects 1-3, lectures #9 (2/25) to #19 (4/9)
    - » OS History, Services, and Structure; CPU Scheduling; Kernel and Address Spaces; Address Translation, Caching and TLBs; Demand Paging; I/O Systems; Filesystems, Disk Management, Naming, and Directories; Distributed Systems
- TA Review session TBA

Map

map f lst: ('a→'b) → ('a list) → ('b list)

Creates a new list by applying f to each element of the input list; returns output in order

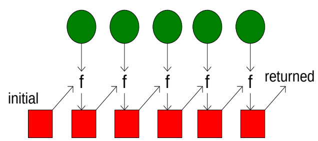


Fold

fold f x_0 lst: ('a*'b→'b)→'b→('a list)→'b

Moves across a list, applying *f* to each element plus an *accumulator*

f returns the next accumulator value, which is combined with the next element of the list



fold left vs. fold right

- Order of list elements can be significant
  - Fold left moves left-to-right across the list
  - Fold right moves from right-to-left

Standard ML Implementation:

```
      fun foldl f a []      = a
| foldl f a (x::xs) = foldl f (f(x, a)) xs

      fun foldr f a []      = a
| foldr f a (x::xs) = f(x, (foldr f a xs))
```

Example

- fun foo(l: int list) = sum(l) + mul(l) + length(l)
- How can we implement this?
- fun sum(lst) = foldl (fn (x,a)=>x+a) 0 lst
- fun mul(lst) = foldl (fn (x,a)=>x*a) 1 lst
- fun length(lst) = foldl (fn (x,a)=>1+a) 0 lst

More Complicated Problems

- More complicated fold problem
  - Given a list of numbers, how can we generate a list of partial sums?
    - » e.g.: [1, 4, 8, 3, 7, 9] → [0, 1, 5, 13, 16, 23, 32]
- More complicated map problem
  - Given a list of words, can we: reverse the letters in each word, and reverse the whole list, so it all comes out backwards?
    - » e.g.: ["my", "happy", "cat"] → ["tac", "yppah", "ym"]

map Implementation

```
fun map f []        = []
  | map f (x::xs) = (f x) :: (map f xs)
```

- This implementation moves left-to-right across the list, mapping elements one at a time
- … But does it need to?

Implicit Parallelism In map

- In a purely functional setting, elements of a list being computed by map cannot see the effects of the computations on other elements
- If order of application of f to elements in list is commutative, we can reorder or parallelize execution
- *This is the "secret" that MapReduce exploits!*

BREAK

MapReduce

- Motivation: Large Scale Data Processing
  - Want to process lots of data (> 1 TB)
  - Want to parallelize across hundreds/thousands of CPUs
  - Want to make this easy…
- Features:
  - Automatic parallelization & distribution
  - Fault-tolerant
  - Provides status and monitoring tools
  - Clean abstraction for programmers
- Hadoop:
  - Open-source version of Google's MapReduce

Programming Model

- Borrows from functional programming
- Users implement interface of two functions:

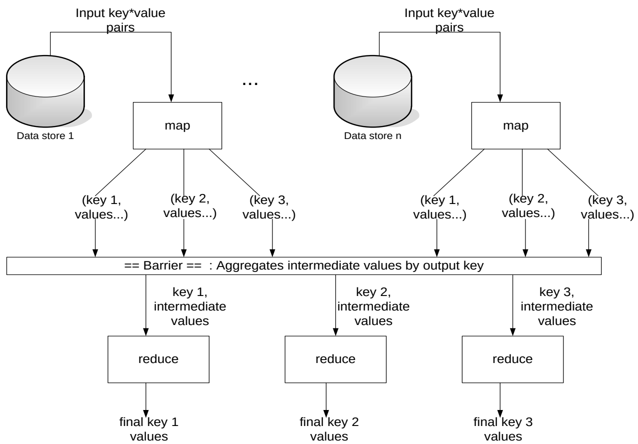
```
– map  (in_key, in_value) ->
     (out_key, intermediate_value) list

– reduce (out_key, intermediate_value list) ->
     out_value list
```


Functions

- Map:
  - Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as (key, value) pairs
    - » e.g., (filename, line)
  - map() produces one or more intermediate values along with an output key from the input
- Reduce:
  - After the map phase is over, all the intermediate values for a given output key are combined together into a list
  - reduce() combines those intermediate values into one or more *final values* for that same output key
    - » (in practice, usually only one final value per key)

MapReduce



Parallelism

- map() functions:
  - Run in parallel, creating different intermediate values from different input data sets
- reduce() functions:
  - Also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can''t start until map phase is completely finished

Example: Count word occurrences

```
map(String input_key, String input_value):
     // input_key: document name
     // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```


Example vs. Actual Source Code

- Example is written in pseudo-code
  - Actual Google implementation is a C++ library with Python/Java interfaces
  - Hadoop implementation is in Java
- True code is somewhat more involved
  - Defines how input key/values are divided up, accessed, …
- Locality:
  - Master program divvies up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

MapReduce Techniques

- Locality
  - Master pgm places map() tasks based on data location
  - map() task inputs are divided into 64 MB blocks
- Fault-Tolerance
  - Master detects worker failures
    - » Re-executes completed & in-progress map() tasks
    - » Re-executes in-progress reduce() tasks
  - Master notices particular input key/values cause crashes in map(), and skips those values on re-exec
- Optimization – speculative execution
  - No reduce can start until map is complete:
    - » A single slow machine rate-limits the whole process
  - Master redundantly executes "slow" map tasks
    - » Uses results of first copy to finish – *why is this safe?*

Summary

- Important system properties
  - Availability: how often is the resource available?
  - Durability: how well is data preserved against faults?
  - Reliability: how often is resource performing correctly?
- MapReduce has proven to be a useful abstraction
  - Functional programming paradigm can be applied to large-scale applications
  - Greatly simplifies large-scale computations at Google, Yahoo!, Facebook, and others
  - Fun to use: focus on problem, let library deal with messy details