日曜研究室

技術的な観点から日常を綴ります

[xv6 #68] Chapter 5 – File system – Real world

テキストの76〜77ページ

本文

実際のOSにおけるバッファキャッシュは、xv6のものより複雑で興味深いが、やはりxv6と同じ2つの用途を提供する。
データのキャッシュと、ディスクアクセスの同期化である。
Unix V6のような、xv6のバッファキャッシュは、単純な「最近利用されたものがよく使われる」というポリシー（least recently used, LRU）にのっとっている。
実装可能なポリシーにはもっとたくさんの複雑なものがあり、それぞれ特定のワークドロードには有効だが、その他にたいしては効率が悪かったりする。
より効率的なLRUキャッシュは、リンクリストではなく、走査の効率を上げるためのハッシュテーブルや、LRUのためのヒープだろう。
近年のバッファキャッシュは、一般的にメモリマップドファイルをサポートするために仮想メモリに統合されている。

xv6のロギングシステムは、悲惨なほどに効率が悪い。
システムコールによる同時更新に対応しておらず、完全に異なるファイルシステム上における操作と変りない。
それは、たとえブロックの数バイトだけが変更されたとしてもブロック全体をロギングする。
ログへの書き込みは同期化されていて、一度に一つのブロックしか書き込めず、それぞれディスク全体をロックしてるようなものである。
実際のロギングのシステムは、その全ての問題に何らかの対策を行っている。

ロギングは、クラッシュリカバリを提供する唯一の方法ではない。
初期のファイルシステムは、再起動時に全てのファイル、ディレクトリ、ブロック、空きinodeリストを調査し、矛盾を解決するための「掃除人」（例えば、UNIXのfsckプログラム）を利用していた。
巨大なファイルシステム上では、その「掃除」は長い時間を要し、矛盾を解決する方法を決定出来ない場合もあった。
ログからリカバリすることは、より速く確実な方法である。

xv6は、初期のUNIXのように、inodeとディレクトリで基本的なディスク上のレイアウトを用いる。
この仕組みは、長い間非常にうまく働いてきた。
BSDのUFS/FFSと、Linuxのext2/ext3は、基本的には同じデータ構造を利用している。
ファイルシステムの、最も非効率な部分はディレクトリであり、各走査の間、全てのディスクブロックを線形走査する必要がある。
これは、ディレクトリの実装がディスクブロックを無駄に消費しないという点では理にかなっているが、ディレクトリがたくさんのファイルを保持するためにはコストが高く付く。
Microsoft WindowsのNTFSや、Mac OS XのHFSや、SolarisのZFSは、少しの名前付けを行うだけであり、ディレクトリをディスク上のブロックをバランス木として実装している。
これは複雑であるが、ディレクトリの走査を対数時間で行えることを保証する。

xv6は、ディスクの失敗には弱い。
ディスクの操作が失敗した場合、xv6はpanicを起こす。
これはハードウェアに頼るのが妥当であるかどうか。
OSが、ディスクの失敗を隠蔽するような特殊なハードウェアに載ってたら、OSはそのような失敗を稀なものとして受け止め、panicするのもやむなしだろう。
しかし実際のOSはディスクは失敗するものだと予期し、よりきちんと制御する。
あるファイルのあるブロックのロスがファイルシステムの他の部分に影響しないように。

xv6のファイルシステムは、ディスク全体を使い、サイズは変更できない。
巨大なデータベースや、マルチメディアファイルを格納するためのストレージは、より高機能のものを要求し、OS
は、「一つのディスクで一つのファイルシステム」というボトルネックを解消するために開発されている。
これを実現する基本的な手法は、複数のディスクを一つの論理ディスクにまとめる方法である。
RAIDのようなハードウェアによる手法は、未だに一番ポピュラーであるが、現在のトレンドは、この手法をできるだけソフトウェア化することである。
このようなソフトウェアの実装は、典型的に柔軟な機能をもたらし、オンザフライでディスクを追加したり削除したりすることによって領域を拡張したり、縮小したりすることが可能である。
もちろん、ストレージの層が拡張・縮小出来るということは、ファイルシステムのレベルでも同じ事が出来る必要がある。
そのような環境では、Unixのファイルシステムで使われているようなinodeブロックの固定サイズの配列は合わない。
ファイルシステムから、ディスクの管理を分離することは、よりクリーンなデザインをもたらすが、SunのZFSのように、それらを両立させるためには、その2者間に複雑なインターフェイスが必要となる。

xv6のファイルシステムは、今日の様々なファイルシステムと比べたら機能が不足している。
例えば、スナップショットや増分バックアップをサポートするための機能がない。

xv6は、2つの異なったファイルの実装を持つ。

パイプとinodeである。
近年のUnixは、パイプやネットワーク接続、そしてネットワークファイルシステムを含む様々なファイルシステムのinodeを、ファイルとして扱える。
fileread や filewrite でif文を使うことなく、それらのシステムは典型的に、各ファイルシステムの実装を呼び出すために関数ポインタを使う。
ネットワークファイルシステムやユーザレベルファイルシステムは、ネットワークRPCを呼び、レスポンスが返ってくるまで待つような関数を提供する。

感想

ファイルシステム界隈の実際の話しです。
大好きなZFSも出てきました。

ext2/3でひとつのディレクトリ内のファイル数はあまり増やさない方がいいと言われる理由が分かりました。

実は、Unix上のPythonで追加のパッケージ無しで、原始的なファイルシステムであるFAT32のメタデータを解析するプログラムは書いた事があるのですが、xv6fsはそれより多少高機能とはいえ簡単な仕組みであるということは分かりました。
ただ、その簡単な仕組を実現するためだけでも、この章を通して説明してきたコードが必要なわけであり、ZFSとかの高機能なファイルシステムなんかはどんなんなってるんだろうと想像もつきません。
まぁOSSなんで読もうと思えば読めるんですけど。

とにかく、やはり柔軟性を得ようとしたらコードは複雑にならざるを得ないんだなーと思いました。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/4/28 土曜日 [http://peta.okechan.net/blog/archives/1677] |