平成14年度学士論文

他のプロセスに
あたえる影響が少ない
実行時ミラーリングシステム

東京工業大学 理学部 情報科学科
学籍番号 99-2518-8
柳澤 佳里

指導教官
千葉 滋 助教授

平成15年2月6日

概要

近年、インターネット上での商業取り引きが増えるにつけ、そのデータの保守、管理というのは会社の信用を左右する大きな問題となっている。多くの企業ではこのデータの保守、管理をデータの複製をとるという方法で実現している。しかしながら、既存の複製方法には実行時に行うとファイルの一貫性が破壊されるという問題や他のプロセスのことを考えずに動作するため実行時に行うのが難しいという欠点などがある。

本研究では他のプロセスへの影響が少ない実行時ネットワークミラーリングシステム Tottotto を提案する。Tottotto では Progress-based Regulation を元にしたスケジューリングを行っている。Progress-based Regulation とは CPU ベースのスケジューリングでは難しかったディスクなどの CPU 以外の資源についても正しくスケジューリングを行えるようにするためのシステムである。しかし、Progress-based Regulation では検定により競合が起きているかを判定するため、検定に必要なデータが揃うまではスケジューリングができないという欠点がある。Tottotto はそれに対処するためにネットワークの流量も監視し、急激に流量が増えた場合にはサーバープロセスを邪魔しないように休止するようにしている。

また、Tottotto はミラーリング時に送信するデータから不要なものを除くことで高速化をおこなっている。Tottotto がスケジューリングによって休止している間に何度も同じファイルへの上書きが行われた場合には最後の更新結果のみを転送すれば良いので Tottotto はそれ以外の更新データを転送しない。このようにしてミラーリングのために転送するデータを削減し、高速化をはかっている。

実験により、Tottotto を用いることでディスク資源を大量に使うようなプロセスとの競合に対してきちんとスケジューリングされることを確認した。しかしながら、CPU 資源を大量に使うようなプロセスとの競合に付いては従来の CPU プライオリティーを用いたスケジューリングの方が良く働くことも確認した。さらに、Tottotto を用いることでファイル

の一貫性を破壊せずに実行時ミラーリングが可能となることも確認した。

謝辞

本研究を進めるに辺り、日頃から研究の方針や進め方について数々の有用な助言をしてくださった指導教官の千葉滋助教授に感謝致します。

また、研究の方向について助言してくださった筑波大学の横田大輔氏、東京工業大学の佐藤芳樹氏、研究をはじめるにあたり参考になる論文を教えてくださった東京工業大学の栗田亮氏に感謝致します。

さらに、研究の進み具合を気にして激励してくださった東京工業大学の西澤 無我氏、中川清志氏、宇崎 央泰氏に感謝致します。

そして、同研究室で苦楽をともにした学部生、院生のみなさんに感謝致します。

目 次

図 目 次

表 目 次

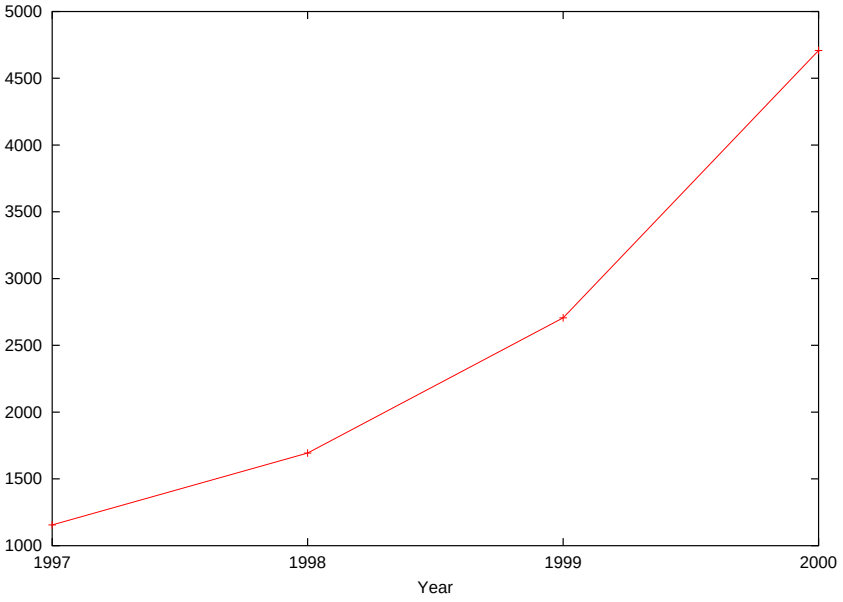
第1章　はじめに

近年、インターネット上で商取引をする機会が増えて来ている。それはインターネットを利用する人の数が増加してきていることによるだろう。(図 1.1) そして、 ADSL などの速い接続形態や携帯電話などの便利な接続形態ができたことで利用者の数はこれからもますます増え続けると見込まれている。これによりこれまで以上にインターネット上で商取引が行われることだろう。

このようなインターネット上の商取引の増加にともないそれは一般的なものとなり、インターネット上で行われない取り引きと同等の信頼性が求められるものとなって来ている。実際、インターネットを利用した商取引 (電子商取引) の数は企業から企業へ (BtoB) のものだけではなく企業から一般の利用者へ (BtoC) のものでも増加していく傾向にあり (図 1.2, 図 1.3) 、もはやインターネット上の商取引は一部の詳しい人だけではなく多くの普通の人が利用する状況になって来ていると言っても過言ではないだろう。そのように多くの人が利用する状況になってくると電子商取引ができる企業の間でも競争が起き、信頼の無い企業は淘汰されていくであろう。商売における信頼は契約を正しく履行することによって作られていくものであるが、災害などで契約を行ったというデータが消失してしまうと正しく契約を履行できなくなる。

その対策として企業ではデータの複製を行いデータの消失を防いでいる。データの複製の技術は昔から存在した。例えば、写経は昔からある複製の方法であろうし近年はほとんどのコンビニエンスストアにコピー機が置いてある。計算機の上での複製方法は簡単にはファイルのコピーでできるし、詳しい人が網羅的にやるには dump コマンドを使った磁気テープへのバックアップや RAID などさまざまな方法がある。

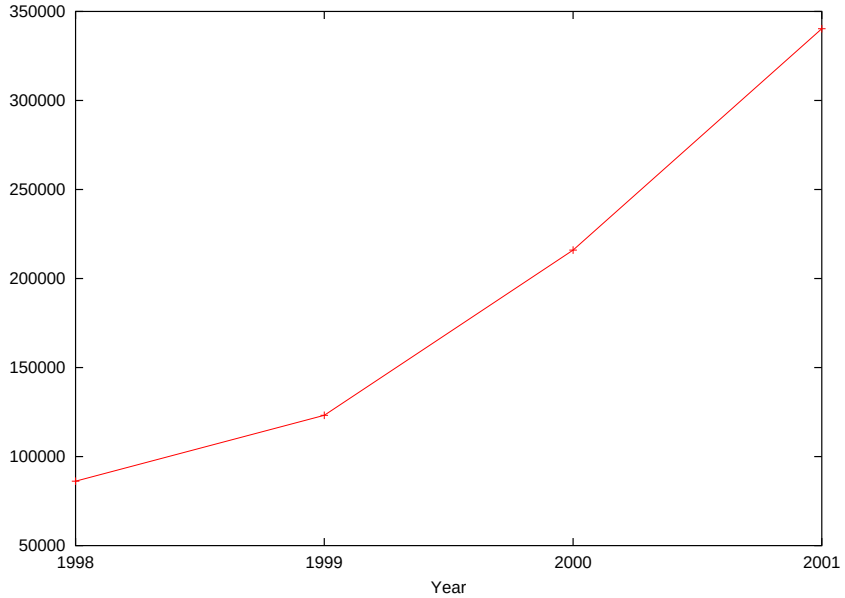
しかしながら、よく行われて来た複製方法である磁気テープへのバックアップにはさまざまな欠点がある。まず、このバックアップ方法は磁気テープに対して行うので遅く、バックアップを取った時点 (1 日数回程度) の記録しかのこらないため近年のように秒単位で取り引きが行われる状



総務省

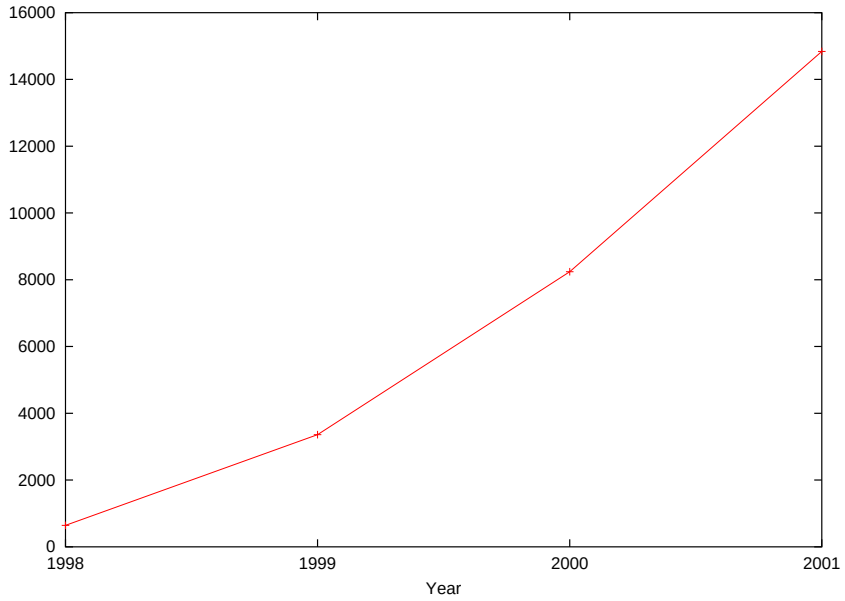
平成 13 年版 情報通信白書より

図 1.1: 日本のインターネット利用者数 (万人)



経済産業省 平成 13 年度電子商取引に関する市場規模・実態調査より

図 1.2: 日本の BtoB 市場規模の推移 (億円)



経済産業省 平成 13 年度電子商取引に関する市場規模・実態調査より

図 1.3: 日本の BtoC 市場規模の推移 (億円)

況には向かないと考えられる。さらに、磁気テープへのバックアップだとハードディスクの内容を他のプロセスのことを気にせずに逐一調べてバックアップを行うため負荷が高くバックアップ中の計算機ではとても業務などはこなせないであろう。

また、近年一般によく使われる技術である RAID でさえ災害への耐性があるとはいいがたいため最高の複製技術とは言いがたい。RAID は基本的に同一ハードウェア内部で複製を行う。よって、災害や 9.11 のような状況のときに物理的な破壊によりデータが失われてしまう恐れがあるのだ。そのため、ネットワークを介して遠く離れた場所にバックアップを取るという必要が出てくるであろう。

以上のことから、電子商取引に対応した複製システムは次のような特性を持つ必要があると考えられる。

1. 複製システムの動作が他のプロセスに影響をあたえないこと
2. 実行時に複製ができること
3. 災害が起きることを想定し離れたところにネットワークを介して複製できること

これに加えて、どんな場合でも整合性が取れたデータが複製されるという要件が必要になる。さもなければ、複製はあるけれど使えないというような何のために複製を行ったのかわからない状況になってしまうだろう。

そこで、本研究では他のプロセスにあたえる影響が少ない実行時ミラーリングシステム (Tottotto) を提案する。Tottotto は重要度の高いプロセスに CPU 資源のみではなく CPU 以外の資源も譲る機能がそなわっている。そのため、複製作業をすることで実業務にあたえる影響は少ない。また、Tottotto はミラーする際には不要なデータを減らすことで高速化をはかっている。

Tottotto は Progress-based Regulation を元にしたスケジューラーを使ってスケジューリングを行っている。Progress-based Regulation とは通常のスケジューリングと違い、重要度の低いプロセスからの進捗状況の報告に基づいて重要度の低いプロセスを休止することでスケジューリングを行う方法である。これは資源の競合が起きている場合にはプログラムの実効速度が遅くなり、報告される進捗状況も悪くなるという考えに基づいている。つまり、資源の競合が起きると重要度の低いプロセスの進捗状況が悪くなり、それをスケジューラーに報告することでスケジューラーが競合を検知し、報告したプロセスを休止するようになっている。競合の検知は競合が起きていないという帰無仮説を仮説検証することで行う。素朴な Progress-based Regulation では仮説検証を行うためのデータが揃うまでは何の制限もかけない。よって、急激に重要度の高いプロセスとの資源の競合が起きるような事態になったときにはデータが集まるまで重要度の高いプロセスの性能を下げてしまう。

Tottotto はその対策として Progress-based Regulation で検定のためのデータを集めている間もスケジューリングを行うことができるようにした。その方法として Tottotto ではネットワークの流量も参考にしてスケジューリングをするようにしている。Tottotto はサーバーでの使用を想定しているので、ネットワークの流量が急激に増えている時には、サーバープロセスという、より重要度の高いプロセスが大量に資源を消費していると考えられる。そこで、ネットワークの流量を見て急激に増えている場合にはミラーリングを休止し、より重要度の高いプロセスをスケジュールする。

さらに、Tottotto は送信するミラーすべきデータのうち不要なものを削除し、ミラーリングの高速化をはかる。Tottotto ではスケジューリングにより重要度の高いプロセスに資源を譲るために休止する期間が長く

なる。Tottotto が休止している間に、何度も同じファイルが上書きされた場合は、最後の更新結果だけ転送すれば良いので、Tottotto はそれ以外の更新データを転送しない。これによって、ミラーリングのために転送するデータの総量を削減し、高速化をはかる。

以下、2 章ではこれまでのバックアップやミラーリングを行う複製システムとその欠点について述べる。3 章では Tottotto の設計と実装について述べる。4 章では Tottotto によるミラーリングは他のプロセスへの影響がどの程度少ないかを確かめた実験について述べる。最後に、5 章で本論文をまとめる。

第2章　既存の実行時複製システムとその問題点

本研究では実行時のミラーリングに注目した。この章ではまず Tottott 以外の実行時ミラーリングシステムについて述べ、2.6 節にてまとめる。

2.1　ネットワークの監視によるバックアップシステム

ネットワークの監視によるバックアップシステム [10] は実行時バックアップの方法の一つで、NFS Server と NFS Client 間の通信を傍受、解読して NFS Server に行われた変更をバックアップシステムで再現することによりバックアップを行うシステムである。(図 2.1) このシステムは NFSv2 で接続しているサーバーをバックアップの対象とする。このシステムにはシグナルを送ることである時点でのスナップショットをとる機能もついている。

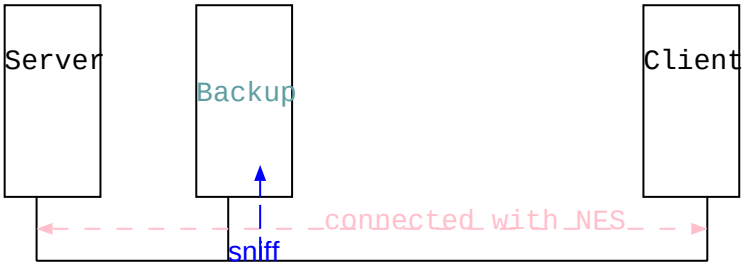


図 2.1: ネットワークの監視によるバックアップシステムの概要

2.1.1　ネットワークの監視によるバックアップシステムの利点

この方法は NFS Server,Client がバックアップのために特別な仕事をする必要が無いため、NFS Server,Client にかける負荷は少ないと思われる。よって、他のプロセスへあたえる影響の面だけを考えるとこのシステムは影響が少ない優れたシステムであると言えるだろう。

2.1.2　ネットワークの監視によるバックアップシステムの限界

NFS Server,Client はバックアップシステムの存在を知らないのでバックアップシステムが両者の通信速度についていけなかったり NFS 以外の通信により NFS パケットの傍受をしにくくされたりすることで取りこぼしが起きる可能性がある。そして、この場合には整合性の取れないデータができる。そのためこのバックアップ方法は高い信頼性が必要な場所ではとりこぼしの危険性があるため使えないといえる。

2.2　RAID

RAID[1] [8] は 1987 年にカリフォルニア大学バークレー校の David A. Patterson らによって提唱されたハードディスクなどの記憶装置を複数用いてアクセスを分散させることにより高速、大容量、高信頼性のディスクを実現する技術である。RAID では記憶すべきデータと障害回復のためのデータを複数のディスクに分散して格納することで性能と耐故障性を同時に得ることができる。この記憶すべきデータと障害回復のためのデータの格納方法の違いによって RAID はいくつかのレベルが存在するが、おもに使われるのは RAID0+1 [2]や RAID5 である。

RAID の実現方法には、OS などのソフトウェアで行う方法 (ソフトウェア RAID) と RAID 専用のハードウェアで行う方法 (ハードウェア RAID) がある。近年の OS には RAID を実現する機能が備わっているものが多く専用のカードなしに安価にソフトウェア RAID を実現できる。しかし、ディスク I/O において RAID 処理のためのオーバーヘッドがかかったり

[1] Redundant Arrays of Inexpensive(Independent) Disks
[2] ディスク 4 つでストライピングとミラーリングを組み合わせて使う

RAID を実現するために複数回同種の書き込み操作を行うことでバスの帯域を通常よりもかなり多く使ってしまったりするという欠点がある。そのため、本格的なサーバーシステムでは RAID における各種処理を専門に行うハードウェアを導入しハードウェア RAID を実現するのが普通である。

2.2.1　RAID の利点

RAID は同じ情報を複数のディスクに分散して書くためディスクの破損に対しての耐性やファイル I/O の時の高速性が見込める。例えば、RAID1 では同じ内容を 2 つのディスクに書くため片方が壊れてももう片方からデータを復元できるという故障耐性がある。また、RAID1 はディスクの読み込み時には読み込み命令を両方のディスクに分散して発行することで高速化を行える。他の RAID も分散して書き込みを行っているのでおなじことが言えるだろう。

2.2.2　RAID の限界

RAID はディスク自体の故障には耐えられるが、ディスク装置以外の故障に対応することはできない。例えば、RAID を行っている OS やハードウェアが故障した場合には RAID により保存されたデータの整合性すらも怪しくなってくる。また、事故や災害によってサーバーシステムそのものが破壊された場合には RAID を使ったところでデータそのものが破壊されてしまうため、データを破損から守るという目的を達成することはできない。そのため、ネットワークを利用して外部に複製を作っておく方法が必要となってくる。

2.3　Zebra

Zebra[4] とは Sprite OS 上で構築された、ネットワークを介して RAID4[3] を行うようなシステムである。Zebra では全ての書き込みを LFS 風のデータのつながりに変え、それをネットワーク上にある複数のストレージサー

[3] データを 3 台に分散して保存し、それらのパリティーを残りの 1 台に保存するタイプの RAID

バーに分散させて書き込む。書き込みに際してはパリティーを書き出すことで一つのディスクサーバーが壊れてもデータが復元できるようになっている。

2.3.1　LFS(Log-structred FileSystem)

LFS[9] はログ構造のファイルシステムであり、全てのファイル書き込みや変更はそのログに追記するという形で実現されている。この構造により LFS は高い耐故障性と書き込み時の高速性が期待されている。

これまでのファイルシステムの例: FFS(Fast FileSystem)

FFS[7] とは BSD 系 Unix で使われているファイルシステムでそれまでの FileSystem に比べて速く、ファイルシステムの破損に強い構造にある。FFS より前のファイルシステムでは superblock は一ヶ所に配置していたのでファイルアクセスにおいては多大な時間がかかり、superblock が失われるとファイルシステム自体が崩壊することになった。しかしながら、FFS ではシリンダをパーティションごとにシリンダグループに分け、各シリンダグループごとに superblock の複製を配置し、そのシリンダグループにあるディスクの情報を配置された superblock の複製を見て取得するようにしたのでヘッドを動かす量が減り、速いアクセスができるようになった。さらに、superblock のコピーをシリンダグループごとにおいたことで故障への耐性が向上した。

この FFS についての開発はまだ続いており、近年 soft update が実装された。soft update というのはファイル書き込み、更新時の順序を入れ換えてメタデータと実データとの一貫性が壊れないようにした非同期書き込みの方法であり、これを使うことで非同期書き込みの性能を出しつつ急なシステムダウンが起きた場合にも fsck なしで起動することができる。しかしながら、soft update を使っていると一貫性を立つ持つために行った処理のためにシステムダウンのたびに未使用であるにもかかわらず使用中とマークされた領域が出来てしまう。それを解消する機構として background fsck[6] が実装された。background fsck とはファイルシステムのスナップショットを作成し、それに対して fsck するという方法で使用中でないのに使用中であるとマークされた領域を実行時に探しだすことができる。これにより、未使用にもかかわらず使用中とされた領域を

実行時に未使用領域として OS に認識させることが出来、ディスクを無駄なく使うことができ、soft update の後処理のためだけに起動時に fsck していた煩わしさから開放された。

LFS の利点

LFS はログ形式でファイルシステムを構成しているので、どんな状態でシステムダウンしてもファイルシステムの一貫性は壊れない。システムダウンから復旧する場合にはデータが正しいかを記録されている checksum と比較することで確認していく。しかしながら、始めから逐一 checksum と比べていくと起動時に多大な時間がかかってしまうので LFS はデータの一貫性が保証できるところで checkpoint という印を入れ、システムダウンからの復旧時には最後の checkpoint までは正しいデータがあるとしてそこから先の checksum を調べていくことにしている。checkpoint は fsync(2) が発行されたときに作成されるので、通常だと 30 秒ごとに checkpoint が作られることになる。30 秒でファイルを作成できる量はたかが知れているのでダウンからの復旧時にはさほど時間がかからないであろう。

さらに、LFS ではファイル I/O をシーケンシャルに行えるので高速である。LFS では定まった位置に i-node を格納しないことにより、i-node 領域の大きさを決めることに悩んだりファイルアクセスのたびに 4 回のファイル I/O を発行したりしなければならなくなる問題を解消することが出来る。LFS 以前のファイルシステム (FFS など) は i-node を定まった位置に保存していたため、i-node 領域が多すぎればデータ領域に使える容量が減り、i-node 領域が少なすぎれば i-node が枯渇してファイルが作れなくなると言う問題があった。そのため、FileSystem を構築するときに i-node 領域の容量を適切に決めることはシステム管理者の腕の見せ所でもあった。しかしながら、LFS では i-node もデータも区別無くログとして書き込まれるので特別に i-node にしか使えない領域を取る必要が無く、この問題からは解消された。また、FFS のころにファイルをアクセスする場合には i-node 管理ブロック、i-node、データ領域管理ブロック、データと最低 4 回のアクセスが必要であったが、LFS ではデータと i-node は連続して書き込まれるため最低 1 回のアクセスですむようになる。[4]

また、LFS ではファイルを書くときに通信帯域をフルに使うことが出

[4] データが分散している場合には FFS でも LFS でもさらに多い回数のアクセスが必要である

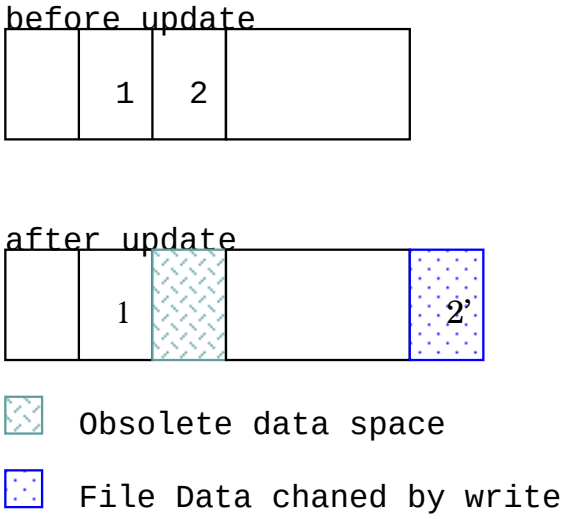


図 2.2: LFS における書き込み操作

来る。通常のファイルシステムだと管理ブロックとデータブロックが離れたところに配置されているのでファイルを書くときには両方の変更を行わなければならず通信の帯域よりもレイテンシの方が大きな問題になってくるが、LFS だとファイルの書き込みを一つのログへの追記という形で実現しているのでそのような問題は無くなる。(図 2.2:1,2 の領域を占めるファイルの 2 の部分を変更した場合のディスク領域の変化)

LFS の clean 処理

LFS は segment ごとに分けて管理していてその segment の中にデータ領域や i-node 領域が入っている。LFS を使っていると上書きされたり削除されたりした i-node やデータなどがあった領域がゴミとして累積していくが、その掃除は segment 単位で行われる。

LFS ではこの掃除処理を cleaner というデーモンプロセスが請け負っている。cleaner は $\frac{\text{掃除したことによる利益}}{\text{掃除にかかるコスト}}$ が大きい順に clean 処理を行う。掃除による利益は掃除を行うことでどれだけの領域が使えるようになるか (1-segment の使用率) とその領域の年齢 (古い程大きい) の積で算出され、コストは領域の使用率 (1+segment の使用率) で決まる。

clean 処理に際しては segment 内部に使用中の領域があれば次に書き込む segment にその内容をコピーする。そのため、コピーしなくてはなら

ない容量が少ない方がコストが安くなるようになっている。

NetBSD LFS

NetBSD の LFS は 4.4BSD の LFS に由来したファイルシステムである。NetBSD の LFS には元の LFS(Sprite LFS) には無かった Index File という機構がある。この Index File には次のような情報が含まれている。

- dirty segment の数
- clean segment の数
- 各セグメントの情報 (生きているバイト数、タイムスタンプなど)
- inode の情報 (バージョン番号、ディスクアドレスなど)

Index File の inode があるディスクアドレスは superblock に格納されている。そして、Index File は通常のファイルとして作られているので superblock のように決められた場所に配置する必要が無い。このことは次の 2 つの理由による。まず、LFS では定まった位置に i-node を格納しないからである。次に、cleaner がユーザーランドのプロセスとして実装されているからである。cleaner がユーザーランドのプロセスとして実装されているので Index File はアプリケーションプロセスからアクセスできるようになっていなくてはならない。

2.3.2　Zebra の利点

Zebra では LFS 風のデータのつながりを分割してストレージサーバーに入れ、それのパリティーを計算するので効率がよい。通常の RAID では小さいファイルを保存する場合にはさらに小さな細切れにしてそれにパリティーを計算するかファイル一つに対して一つのパリティーを作るかという方法があるが、前者ではファイル I/O 時のオーバーヘッドが大きくなり、後者では小さなファイルにすらパリティーを割り当てるのでディスクを無駄に使ってしまうと言う欠点がある。これに対して、Zebra では小さなファイルも大きなファイルも一連のストリームにしてストレージに分配し、それに対してパリティーを計算するので上記の問題が解消される。(図 2.3)

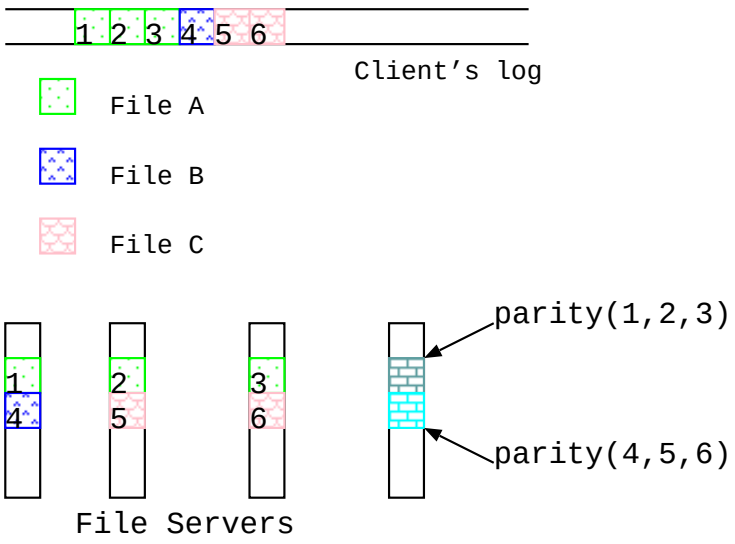


図 2.3: Zebra によるストライピング

2.3.3　Zebra の限界

Zebra は RAID で小さいファイルを書くときに効率が悪くなるという問題を解決したが、全ての I/O をネットワークを介して行うので I/O におけるオーバーヘッドははかり知れない。よって、構成する全てのサーバーをラックなどに入れて一ヶ所にまとめて光ファイバーで接続してファイルサーバーを構築し使う分には申し分なくつかえるが、インターネットを介して分散させてファイルサーバーを構築したり、同軸ケーブルによる遅いネットワークでファイルサーバーを構築したりする場合には使い物にならないほどの性能しかでないであろう。また、ファイルサーバーをつなぐ線が混んでいる場合にもファイルアクセスだけでかなりの帯域を使ってしまうので SAN[5]を特別に構築することでもしなければ使い物にならないほどの性能であろう。

他のプロセスにあたえる影響が少なく災害に強いというのがよい複製システムの条件であると考えているのでこれらのことから Zebra は不適当であると言える。

[5]ストレージエリアネットワーク

2.4 Spiralog

Spiralog[5] とは OpenVMS 上で構築されたリモートファイルシステムである。 Spiralog には一貫性を保ちつつ遅延書き込みを行うキャッシュ機能や高性能な実行時バックアップ機能がある。これらの機構によりユーザーは Files-11[6]や NTAS[7] FileSystem 上でファイルを扱うようにしてファイルを扱うことができる。

Spiralog はファイルシステムクライアント、クラーク、LFS サーバーの 3 つの要素から構成されている。ファイルシステムクライアントは Files-11 や NTAS FileSystem の操作性をユーザーに提供するための機構を提供する。そして、クラークはキャッシュを管理し、LFS サーバーに書き込む機能を持つ。LFS サーバーはログ構造でデータを保持する機構を持ち、クラークからアクセスできる cleaner を持ち合わせている。

ファイル I/O の時はこの 3 つの要素を順にたどってデータにアクセスする。まず、ファイルシステムクライアント上のユーザーがファイル I/O を行うと、それが File-11 などのファイル I/O のプロトコルから VPI という Spiralog 内部のファイル形式に変換され、クラークに送られる。クラークではキャッシュ内にファイルがあればそれを使い、ない場合には LFS サーバーに問い合わせそれを使う。

2.4.1 Spiralog による On-line バックアップ

Spiralog は実行時 (on-line) バックアップが可能である。[3] 通常のバックアップではバックアップ中にファイルが書き換えられると古いファイルと新しいファイルがごちゃまぜになった状態でバックアップされる (図 2.4)。しかしながら、Spiralog はバックエンドのファイルシステムとして LFS を使っているためファイルの一貫性が崩れることは無い。なぜなら、LFS では上書きをせずすべての変更を追記していくのでその構造を保ったままコピーすれば一貫性が崩れないからである。また、追記していくファイルシステムであるので前回バックアップをとったところを記録しておくことで差分バックアップを容易に実現することができる。

[6]OpenVMS の標準的なファイルシステム
[7]Microsoft New Technology Advanced Server

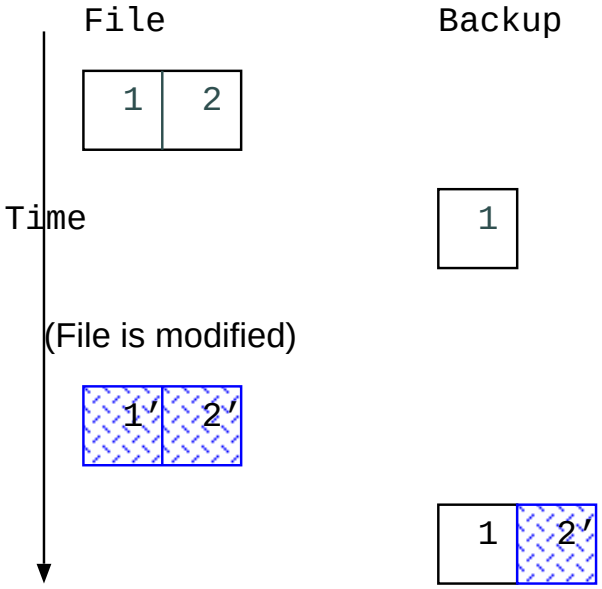


図 2.4: 実行時バックアップでファイルの一貫性が崩れる例

2.4.2　Spiralog の利点

Spiralog の On-line バックアップ機構は LFS を採用したことで一貫性を気にすること無く差分バックアップを行うことができる。一貫性を気にしないで良いことから通常のバックアップと違い、ダウンタイムなしでバックアップがとる。また、LFS サーバーをストレージとして利用していることにより差分バックアップを行うことができ、完全バックアップに比べて速いバックアップを行うことができる。

2.4.3　Spiralog の限界

Spiralog は LFS を利用しているのことで実行時に差分バックアップがとれるという利点がある。差分バックアップが取れることでフルバックアップに比べて少ない時間でバックアップが行えるようになった。

しかしながら、バックアップ中は他のプロセスについて特に気にせず動作するのでその間の負荷が大きくなる。かつては休日や深夜など使われていない時間にこのような負荷がかかる処理をすればよかったが、昨今の状況ではそのような時間は無くいつでもある程度の反応速度が要求され

る。よって、電子商取り引きが日常した近年では Spiralog による On-line バックアップは不向きと言えよう。

2.5　MS-Manner を適用したバックアップ方法

MS-Manner[2] は Progress-based Regulation の実装の一つで、CPU 以外の資源の競合に対しても kernel の改造無しにスケジューリングをできるようにする機構である。これにより従来難しかったディスク資源などの競合により重要なプロセスの性能低下がおきるような場合にもスケジューリングができるようになった。

2.5.1　Progress-based Regulation

Progress-based Regulation とは通常のスケジューリングとは違い、重要度の低いプロセスから報告される進捗状況に基づいて重要度の低いプロセスを休止させることで行われるスケジューリングである。これは重要度の低いプロセスが高いプロセスとの間で資源の競合を起こしたときに進捗状況が悪くなることを利用し資源の競合を検出する。スケジューラーは重要度が低いプロセスから進捗状況の報告を聞き、その進捗状況をもとに『現在の進捗状況は競合が起きていないときの進捗状況と等しい』という帰無仮説を検定する。検定の結果、競合が起きていると判定された場合には重要度の低いプロセスを停止することで重要度の高いプロセスに資源を譲るようになっている。なお、検定のためのデータが集まるまでは何のスケジューリングも行わない。そのため、競合が起きていると決定づける量のデータが集まるまでは重要度の低いプロセスが高いプロセスの資源を奪い、重要度の高いプロセスの性能が悪くなってしまう。

MS-Manner によると検定の進捗状況が悪いのに良いとしてしまうタイプ 1 のエラーを 5%、進捗状況がいいのに悪いとしてしまうタイプ 2 のエラーを 20%として検定を行うとよいようである。MS-Manner ではこのように Progress-based Regulation を用いることで SQL サーバーへの影響を少なくして defrag [8]を行うことに成功している。

[8]ディスクアクセスを高速にできるように最適化を行うソフトウェア

2.5.2　MS-Manner を用いたバックアップの利点

MS-Manner を使ってバックアップを行えばディスク資源の競合で重要度の高いプロセスの実行を妨げるということがなくなる。これは MS-Manner が CPU ベースのスケジューリングでは対処できなかったディスク資源の競合にも対応できるためである。さらに、ネットワークを介してバックアップを行うような場合 (例:rsync) でも、変更なしでネットワーク資源の競合にも対処することができる。

このような高度なスケジューリングができる MS-Manner であるがこれは kernel の改造を必要としないため簡易に適用することができる。プログラムカウンタを見ることで進捗状況を外から測る機構を使えばソフトウェアの改造も要らないので MS-Manner の適用はすぐにでもできる容易なことと言えよう。

2.5.3　MS-Manner を用いたバックアップの限界

MS-Manner を使ったバックアップでは CPU 以外の資源の競合についてもきちんとスケジューリングしてくれるという利点があるが、検定により資源の競合を検知するまでに時間がかかってしまう事やバックアップについての配慮が不足しているという欠点がある。

資源の競合を検知するまでに時間がかかるのは MS-Manner でスケジューリングをする際に検定を行っているためである。検定の結果進捗状況が悪いと判定するには検定に足るだけのデータを集めなければならず、集まるまでのスケジューリングは全く行わない。その結果、データが集まるまでの時間は重要度の低いプロセスが重要度の高いプロセスが使う資源を横取りして重要度の高いプロセスの実行を妨げ、性能を悪くしてしまう。よって、データが集まるまでの時間も競合の可能性を予測して何らかのスケジューリングを行う機構が必要であると考えられる。

バックアップへの配慮が不足していることにより、まず、Spiralog の節で述べたようなファイルの一貫性が崩れてバックアップされるという問題が起きることが予想される。実行時バックアップに付いての何の配慮もなくバックアップを行うと大きいファイルをバックアップするときにはファイルの前方をバックアップしている間に後方を書き換えられてしまう可能性がある。その場合には前方は更新前、後方は更新後というちぐはぐなファイルがバックアップされることになり、バックアップは取ったものの使い物にならないということになりかねない。

また、Spiralog のように実行時バックアップを考慮してログ構造のファイルシステムを構築した場合は、バックアップする量が溢れる可能性がある。MS-Manner によるスケジューリングのためたびたびとまるような場合には、停止時間が長いので同一ファイルへの上書きが頻繁に行われていることが考えられる。その場合には全ての変更を逐一バックアップ先に送る必要はない。しかしながら、バックアップを素朴に実装するとログ構造を利用し、前回の続きから今までをバックアップ先に送るというのが普通であろう。だが、これでは競合を感知して一時休止すればする程バックアップをすべきデータの量が増えていってしまい、しまいにはバックアップをきちんとできなくなってしまうであろう。そのため、上書きが起きた場合には上書き前の古いデータを送らないように考慮する必要があると思われる。

2.6　関連研究のまとめ

以上の点から既存の複製機構ではどれも実行時複製システムに必要であると思われる以下の要素のすべてを満たしているとは言いがたい。

- 確実性
- 耐故障性
- 災害耐性
- 他のプロセスへの影響の少なさ

これまで見てきた複製システムについて以上の観点でまとめると表 2.1 の通りになる。以下、表 2.1 について解説する。

2.6.1　ネットワーク監視によるバックアップ

ネットワーク監視によるバックアップは NFS Server,Client で特別なプロセスは走っていないので、他のプロセスに与える影響は無いと考えられる。しかしながら、バックアップを行う機械は Server とおなじ HUB に接続している必要があるため、Server が破壊されるような災害が起きた場合には一緒に破壊されてしまう恐れがある。また、Server,Client は特に通信を傍受していることを知らないため、バックアップマシンが処理できる量の限界を越えるときちんとバックアップすることができなくなる。

	確実性	耐故障性	災害耐性	他のプロセスへの影響の少なさ
ネットワーク監視によるバックアップ	×	○	×	◎
RAID	○	○	×	○
Zebra	○	○	△	△
Spiralog	○	○	△	×
ネットワークバックアップ/MS-Manner	×	○	○	△

表 2.1: 実行時複製システム

2.6.2　RAID

RAID は I/O を並列して行うので I/O 時を高速に行えることが予想される。しかしながら、一つの計算機の中にディスクを複数入れてディスクの破損に対抗しているだけなので、ディスク以外のものが破損した場合には対応することができない。また、RAID は一台の計算機の中で構成しているため、災害が起きれば全てのミラーしたデータはともに失われることとなるだろう。

2.6.3　Zebra

Zebra において災害への耐性と高速性はトレードオフの関係にある。災害への耐性をあげるためにストレージサーバーを分散しては位置すると I/O のたびに遠くにあるストレージサーバーにアクセスする必要が出て来るため、高速性が犠牲になる。逆に、高速にしようとするとすべてのストレージサーバーを一箇所にかためて設置する必要が出て来るため災害が起きた場合には全てのデータが失われる恐れがある。

2.6.4　Spiralog

Spiralog はリモートファイルシステムを構成し、複数の LFS サーバーをおいているので災害に対する耐性はある程度あると考えられる。しかしながら、リモートファイルシステムを実用的な速度で運用するにはサーバー

を一箇所にかためなくてはならないが、そのような場合には災害が起きると同時に全てのサーバーがダメになってしまう恐れがある。また、バックアップにおいては特別なスケジューリングを行っていないため、バックアップ時間中はディスク資源を利用するプロセスでの性能低下が起きる可能性がある。

2.6.5　MS-Manner を用いたバックアップ

MS-Manner を使っただけのバックアップでは実行時バックアップという考えがないため、ファイルの一貫性が崩れたデータがバックアップされる恐れがある。ログ構造のファイルシステムを採用し、それに対してバックアップを取る場合には同一ファイルへの上書きによって不要になったデータをバックアップ対象から外すなどの工夫をしないとバックアップを取る量が溢れてしまう恐れがある。また、Progress-based Regulation によりスケジューリングを行うには、検定結果が出るだけのデータが必要となるが、データが集まるまでは何のスケジューリングも行わない。そのため、データが集まるまでの間重要度が高いプロセスの性能を下げてしまう恐れがある。

第3章　Tottottoの設計と実装

3.1　Tottottoの概要

Tottotto は Progress-based Regulation を行うことで他のプロセスに与える影響が少ないミラーリングを提供する。つまり、ミラーリングの進捗状況を逐一監視し、検定により競合が起きていないときの進捗状況と同等かを調べている。他のプロセスとの資源の競合により進捗状況が悪くなった場合には検定によりそれを検知し、他のプロセスに資源を譲るために休止するようにしている。

また、Tottotto は資源の競合を予測し Progress-based Regulation よりも早い段階でシステムの休止を行う機能がついている。Progress-based Regulation だけでは検定に必要なデータが集まるまでスケジューリングを行わない。そのため、急激に他のプロセスの動きが活性化したときに対処できず、他のプロセスに悪影響を与える恐れがある。Tottotto はこの事態に対応するためにネットワークバッファ(mbuf) の使用量を見ることで休止を行う。それは Tottotto がネットワークサーバーのバックアップに使われることを想定しているからである。つまり、ネットワークサーバー関連のプロセスが忙しくなる前にまず、サーバーへの要求が大量にやって来ることを予想されるということである。この mbuf の使用量を監視することにより Progress-based Regulation よりも早く休止し、他のプロセスに資源を譲るようにしている。

さらに、前回複製をとったところとの差分を調べやすくするために本システムは NetBSD LFSv2[1] にて構築されたファイルシステムをミラーリングの対象としている。LFS はファイルシステムに行われた変更を追記する形で表現していくファイルシステムであるので前回どこまでバックアップしたかを覚えておけば次回のバックアップはそこからはじめれば良くなる。

Tottotto はミラーリングのとき単純にミラー元からミラー先にコピーするわけではなく、不要なものを飛ばすことで高速化をはかっている。ス

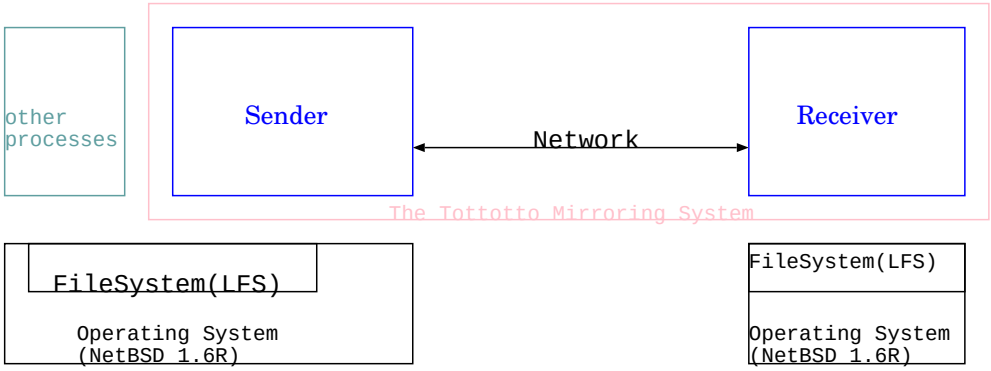


図 3.1: Tottotto の概要

ケジューラーによって重要度の高いプロセスに資源を譲るために Tottotto が長い時間停止する場合には、同一ファイルへの上書きが何度も起こっている可能性がある。そのようなファイルは最後に行われた更新を送ればよく、全ての更新を送る必要は無い。Tottotto ではミラーリングの際にそのような不要なデータを segment 単位で飛ばすことによりミラーリングのために転送するデータの量を削減し、高速化をはかっている。

3.1.1　Tottotto の構成

Tottotto はミラーする元である Sender とミラー先である Receiver の 2 つの要素で成り立っている。なお Sender と Receiver は一対一で対応する。Sender では sender というソフトウェアにて更新のチェックや更新箇所の Receiver への転送を行っていて、Receiver では receiver というソフトウェアを動かし Sender からの送信を待ち受けている。(図 3.1)

以下、3.2 節で Sender の動作について述べる。3.3 節で Receiver の動作について述べる。Sender,Receiver 間でどのようなやりとりをしているかについて 3.4 節で述べる。

3.2　Sender

Sender はミラー元となるマシンである。このミラー元となるマシンで sender というプログラムを動かすことでファイルシステムの更新のチェッ

ク、更新箇所の Receiver への転送を実現している。sender は NetBSD LFS でマウントされたデバイスを読み、前回最後に送った segment から現在書かれている segment までを Receiver に送る。この動作は次のような状態遷移を経て行われる。

3.2.1　Sender における状態遷移

sender は図 3.3 の 8 つの動作状態を遷移することで Receiver に Sender の内容の複製を送る。図 3.3 における状態の意味は次の通りである。

Init 初期状態 (sender の動作はここから始まる)

get_lastseg Receiver にどこの segment まで複製されたかを尋ねる

read_sb Sender の superblock を読み込む

make_diff read_sb で読んだ superblock の内容と get_lastseg で聞いた内容から今回 Reciver に転送する segment を決定する。

write_sb superblock の内容を Receiver に送る

write_segment 指定した segment の内容を Receiver に送る

mbuf_sleep mbuf の残量によるスケジューリング

write_commit 送った superblock の内容と segment の内容を commit するように receiver に指示

testpoint Progress-based Regulation によるスケジューリング

以下、これらの状態の特に重要ないくつかについて細かいところを説明していこう。

get_lastseg

Tottotto には sender あるいは receiver が何らかの理由で一旦休止した場合でも前回複製した segment から複製を再開できるように Receiver にどの segment まで複製したか聞くという機能が備わっている。その処理を行っている状態がこの get_lastseg である。ここで得た Receiver がどの segment まで複製を持っているかという情報は make_diff 状態のときに Receiver に送る segment を決めるところで使われる。

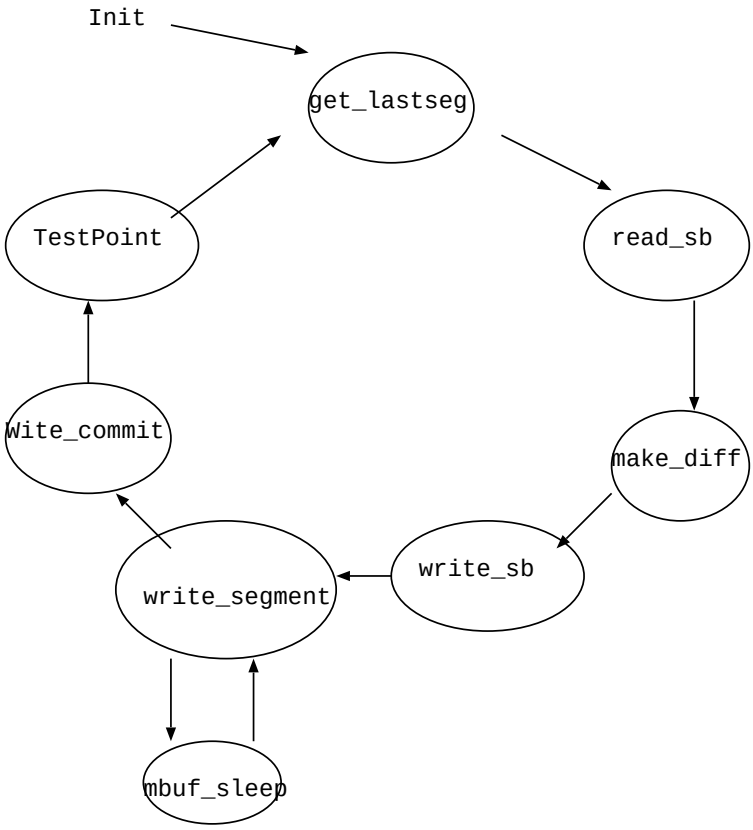


図 3.2: sender での状態遷移

make_diff

LFS の superblock には segment の大きさや segment の数、ログの末尾[1] がある segment の番号などが格納されている。このログの末尾がある segment の番号を read_sb で読むことで最先端のログのある segment 番号を特定できる。また、get_lastseg にて Receiver の最先端のログのある segment 番号を調べられる。

以上の情報を使って差分を調べる手順は次の通りである。

1. Receiver の最先端のログがある segment 番号から Sender の最先端のログのある segment 番号までをまず列挙する。

2. ファイルが削除されたり上書きされたりして使われなくなっている segment がその中に無いか調べる。使われなくなっている segment があればそれを 1 で列挙したリストから削除する。

write_segment と mbuf_sleep

write_segment 状態のときに make_diff で列挙した segment を Receiver に転送する。転送するときは急にネットワークが込んだ場合のことを考え、1 つ segment を転送するたびに mbuf_sleep 状態に移行する。mbuf_sleep 状態ではネットワークの使用量を監視し、閾値以上使用量が増加した場合には sender の動作を一時停止するようにしている。

write_commit

NetBSD LFS では Index File(ifile) にて inode 情報を管理している。よって、NetBSD LFS でファイルにアクセスする場合には ifile をまず参照してその情報を元に手繰ることで inode の位置を調べ、データにアクセスすることができる。そして、その ifile のディスクアドレスは superblock に格納されているので ifile へアクセスするには superblock を参照すれば良い。逆に、ifile を更新する場合には superblock に書かれている ifile のディスクアドレスを更新しなくては以前の ifile が参照され続けることになる。

Tottotto ではこの機構を利用し、superblock を最後に書くようにしている。つまり、write_sb で Receiver に送られた superblock は write_segment の間は保持され続けていて最後に送るべき segment を書き終った後に

[1] そのときファイルなどを作成すると書かれる segment

write_commit 状態のときに COMMIT 命令を実行することで実際にディスクに書かれるのである。このようにすることで Receiver の superblock にまだ転送されていない ifile のディスクアドレスが書かれ、ファイルシステムが壊れるという状況を防いでいる。

3.2.2　スケジューリング

Tottotto ではスケジューリングはすべて sender で行うことでスケジューリングを単純化している。また、Tottotto では Receiver は receiver を動かすための専門のマシンであり、他のプロセスとのリソースの競合は無いと想定しているため、receiver でのスケジューリングはあまり意味を持たないであろう。

Tottotto では Progress-based Regulation と流量監視して停止することの 2 つのスケジューリングを行っている。Progress-based Regulation は状態遷移を 1 周回るたびに testpoint で行われ、流量を監視して停止するというスケジューリングは segment を書くたびに mbuf_sleep で行われる。両者は Progress-based Regulation で長いスパンでのスケジューリングを行い、流量を監視して停止するスケジューリングで短いスパンでのスケジューリングを行うということで役割分担をしている。

Progress-based Regulation

Progress-based Regulation とは重要度が低いプロセスからの進捗状況の報告に基づき、スケジューリング行う方法である。報告される進捗状況を『競合が起きていない状態の進捗状況である』という帰無仮説で検定し、進捗状況の悪化を検出する。検定により重要度の低いプロセスの進捗状況が通常よりも悪化しているという結果が出た場合には監視していた重要度の低いプロセスの実行を停止することで重要度が高いプロセスに資源を譲るようになっている。これは監視している重要度の低いプロセスが他のプロセスとの間で資源の競合を起こしている場合には監視しているプロセスの進捗状況が悪くなるという考えに基づいている。なお、Progress-based Regulation は CPU 使用率に基づいてではなく、重要度の低いプロセスからの進捗状況の報告に基づいてスケジューリングをしているので、CPU 以外の資源に対してもスケジューリングが行える。

Tottotto はこの Progress-based Regulation の機構を使い、他のプロセ

スに与える影響が少ないミラーリング機構を実現している。ミラーリングではディスクを大量に読むのでディスクなど CPU 以外の部分で他のプロセスとの競合が生じる可能性がある。Progress-based Regulation は CPU 以外の資源の競合に対処できるスケジューリング機構なのでこのようなディスクの競合のような事態にも対処することができる。

Tottotto は sender から receiver に何バイトのデータをどれぐらいの時間で送ったかにより進捗状況を見る。送ったデータ量は送信した superblock のバイト数と送信した全ての segment のバイト数の和により求められる。そのため、多くの segment を送った方が進捗状況が良くなる。また、送るのにかかった時間は sender の状態遷移の testpoint 状態が終ってから次の testpoint 状態に来るまでの時間で求められる。よって、流量が多いことにより停止した場合やディスク資源の競合が起きて読みだしに時間がかかった場合などは進捗状況が悪くなる。

なお、競合が起きていない場合の進捗状況は直接求めることが難しいので次の式を計算することで漸近的に求めていくことにしている。

$$progress_{best} = 0.999 \times progress_{best} + 0.001 \times progress_{thistime} \quad (3.1)$$

この $progress_{best}$ は漸近的に求めていく競合が起きていない場合の進捗状況で、$progress_{thistime}$ は今回の進捗状況である。この式では前回の $progress_{best}$ の値と $progress_{thistime}$ を使うことで次の $progress_{best}$ を計算している。このようにして計算することで競合が起きていない場合の進捗状況に $progress_{best}$ は近付いていく。

Tottotto では進捗状況が良いのに悪いとしてしまうタイプ 1 のエラーを 5%、進捗状況が悪いのに良いとしてしまうタイプ 2 のエラーを 20% として検定を行い、その結果に基づいて動作に制限を加えるか否かを決定している。具体的には進捗状況の結果が正規分布の下位 5%に入っていた場合は進捗状況が悪いとして決められた時間動作を停止させ、上位 80%に入っていた場合には進捗状況が良いとして動作を続けるようになっている。そして、そのどちらでもない場合 (下位 5%から 20%の間) の場合には検定のためのデータが足りないとして動作を続けさせ、データを集めるようになっている。

進捗状況が悪い場合の停止時間は上限まで指数的に増加する。つまり、一度悪いと判定されれば初期値の 2 倍、次に悪いと判定されれば初期値の 4 倍の時間、その次は初期値の 8 倍の時間、... と動作を停止することになる。そして、停止時間が 1 分を越えるまで停止時間は増え続ける。しかしながら、一度進捗状況が良いと判断されると停止時間は初期値にリ

セットされるので 1 分近く停止する状態になったとしても一度良い進捗状況になればすぐに元の通りの速さでミラーリングすることができる。

流量を監視したスケジューリング

Progress-based Regulation ではどんなに混んでいたとしても検定を行えるだけのデータが集まるまで停止しない。この欠点を補うために Tottotto では流量を監視したスケジューリングを実現している。

Tottotto では流量の監視として mbuf の使用量を見ることにしている。mbuf というのは BSD 系 Unix にて実装されている OS に確保されたプロセス間通信用のバッファで通信におけるパケットを格納するのに使われる。よって、ネットワークを介してのアクセスが増えて大量のパケットが流れればそれだけ mbuf は多く使用されることになる。

Tottotto は mbuf の使用量が前回に比べて大幅に増えたときにネットワークが急激に混んで来ているとして動作を停止する。これによって、Tottotto では急激にネットワークが混むような Progress-based Regulation だけではスケジューリングしにくいところをスケジューリングできるようになっている。

mbuf の使用量を見て停止する時間は上限まで指数的に増加するようにしている。指数的に停止時間を増加させることでネットワークが急激に混んで来るという状況に対応できるようにしている。なお、一旦 mbuf の使用量がもとの状態へ戻れば停止時間を初期値にリセットするようにして、無用な停止をしないようにしている。

mbuf が急激に増えて来ているか否かは通常状態の mbuf の量から平均 mbuf を計算し、それと比較することで行う。その計算式は次の通りである。

$$mbuf_{ave} = 0.999 \times mbuf_{ave} + 0.001 \times mbuf_{current} \tag{3.2}$$

$mbuf_{ave}$ は mbuf の平均使用量で、$mbuf_{current}$ は現在の mbuf の使用量である。平均使用量 $mbuf_{ave}$ は $mbuf_{current}$ と前回の平均使用量 $mbuf_{ave}$ から計算される。このように現在の使用量をフィードバックすることで平均使用量を徐々に現状に現状に近付けていくことができ、流量が増えて戻る見込みのないときに停止し続けるという状況を避けることができる。そして、混み具合は $mbuf_{current}$ と $mbuf_{ave}$ を比較し、$mbuf_{ave}$ の 1.5 倍よりも $mbuf_{current}$ が多い場合には混んでいると判断される。停止時間は $mbuf_{ave}$ よりどれだけ多いかによって前回の何倍にするか決められ、例え

ば $mbuf_{current}$ が $mbuf_{ave}$ の 2 倍以上 3 倍未満なら 4 倍、3 倍以上 4 倍未満なら 8 倍のように決まっている。

3.2.3　流量削減

上記のようなスケジューリングを行い停止しているとミラーリングすべきデータが大量に増えてしまう可能性がある。しかしながら、頻繁に書き換えが行われるような状況の場合はそのようなミラーリングすべきデータの大半は上書きされたり削除されたことにより不要となったデータである。そのような不要となったデータをミラーリングするのは無駄であるので Tottotto にはそのようなデータを segment 単位で飛ばして送る機構が入っている。

Tottotto は Index File(ifile) にある各 segment の segment 情報を参照することで segment 中に有効なデータが何バイト含まれているかを調べ、有効なデータが無い場合 (0 バイトの時) はそれを飛ばしてミラーリングするようにしている。なお、ifile は通常のファイルとして存在しているので kernel に特別な改造を施したり ifile を参照するための特別なシステムコールを呼ぶ必要は無い。このような segment を飛ばすという処理は make_diff にて行われている。

Tottotto では segment を Sender から Receiver に送るときには segment 番号を付加して送っているので飛ばすからといって特別な処理が必要となるわけではない。つまり、飛ばす番号のものを送らなければそれが飛ばしたということになる。なお、飛ばしたとしてもその segment には有効なデータが含まれておらず、Receiver 側の ifile にはその segment には有効なデータが無いと書いてあるのでファイルシステムに矛盾が生じることは無い。

3.2.4　スケジューリングの実装

スケジューリングは Progress-based Regulation を行うための特別なライブラリと mbuf を監視して停止する特別なライブラリが担っている。そのどちらもスケジューリングが必要な箇所で呼び出すことでスケジューリングを行うようになっている。

Progress-based Regulation の実装

sender は testpoint という関数を呼び出すことで Progress-based Regulation を行うようになっている。testpoint は引数としてミラーリングを行った segment のバイト数と superblock のバイト数の和をとる。この値と testpoint が前回呼ばれて今回呼ばれるのにかかった時間から sender ではスループットを計算する。

このスループットにより Tottotto はミラーリングを継続するか停止するかを決める。競合が起きていない場合のスループットと過去数回スループットから 95%の危険率で競合が起きていない状態のスループットであるかを検定し、棄却された場合には競合が起きているとしてミラーリングを停止するために sleep(3) を発行する。sleep する時間の初期値は 15 秒としていて、検定の結果競合が検出されるたびに指数的に停止時間が延びるようになっている。

mbuf 監視によるスケジューリングの実装

sender では segment を送るたびに mbuf 監視によるスケジューリングを行っている。これにより、ネットワークの混雑をさらに大きくするようなパケット送出を避けることができる。なお、mbuf とは BSD 系 Unix にあるプロセス間通信用の構造で、1 パケットにつき 1 つ以上の mbuf が消費される。

sender では NetBSD における netstat(1) の実装と同様 kvm を読むことで mbuf の使用量を調べている。kvm からは mbuf の総使用量の他に drop 数や wait 数も調べ、drop 数や wait 数が以前に比べて増加していた場合にはネットワークの混雑により drop や wait が起きていると考え、sleep(3) にて動作を停止するようになっている。

mbuf の総使用量については大幅に増加していれば大幅な時間の停止を行うようにしている。実際、前回の平均使用量と比べてどの程度増加しているか調べ、1.5 倍なら 2 倍、2.0 倍なら 4 倍と停止時間を増やすようにしている。次の平均使用量は前回の平均使用量の 99.9%と今回の分の 0.1%を足した値で算出し、次回比べるときに用いるようにしている。なお、停止時間は平均使用量の 1 倍以下になったときに初期値 15 秒にリセットされるようになっている。

3.2.5　ファイル操作の実装

Tottotto では LFS を使うことで今更新されている segment の segment 番号や各々の segment に含まれる有効なデータのバイト数を知ることができる。そして、これらの情報を使うことで最小の量の segment を write_segmetn で送れば良いようになっている。この segment に含まれる有効なデータのバイト数は index file というファイルに記述されているため、これらの情報を知るには index file を読まなくてはならない。そして、この index file の inode がディスクのどの場所に格納されているかは superblock に格納されているので、segment に含まれる有効なデータのバイト数を調べるときはまず superblock を読むようにしている。

superblock の取得

sender の I/O 部分は cleanerd のライブラリを使っているので、ライブラリを使う準備のために fs_getmntinfo と get_fs_info で FS_INFO 構造体を埋めなくてはいけない。これらを呼び出して FS_INFO 構造体を埋めたあと、この構造体の機構を利用し LFS として mount されている raw device をしらべることができる。

superblock ははじめの segment の 0x8 ブロック目から存在するので superblock を取得するときは raw device を open して 0x8 ブロックに seek し、読み出せば良い。LFS の superblock は lfs.h にある dlfs 構造体をそのまま書いた形で格納されているので、superblock を読み出す場合には 0x8 ブロック目に seek して sizeof(struct dlfs)Bytes 読み出す。このようにして読んだ後はそのデータの中に全ての superblock がどこに位置しているかが書かれているのでその情報をもとに次の superblock を読みだし、より古いものを使うようにしている。

segment の取得

segment の取得も cleanerd のライブラリを用いて行う。sender は送る segment を決めるにあたって有効なデータが含まれている segment か調べるために index file を参照する。前回ミラーリングをした segment 番号からの差分の segment について index file を参照し、各々の segment に含まれる有効なデータのバイト数を求める。そして、有効なデータのバイト数が 0 の場合にはその部分をミラー対象のリストから省くようにして

いる。なお、これらの処理は cleanerd の benefit 関数を参考に作成した。

segment の取得にあたっては superblock を読み出すにときに取得した FS_INFO 構造体と取得したい segment 番号を cleanerd の library.c の mmap_segment という関数に渡すことで行う。この mmap_segment に segment 番号をわたすと共に渡したポインタに指定した segment 番号が書き込まれる。

3.3　Receiver

Receiver はミラー先となるマシンである。このミラー先となるマシンで receiver を動かすことで Receiver では Sender からの複製情報を受け取りディスクに保存している。receiver は NetBSD LFS にて作られたファイルシステムにファイルシステムの機構を使わず直接書く。Receiver は Sender に前回送られた最後の segment の segment 番号を教えることで前回のところから Sender のファイルシステムの複製を行うことができる。この動作は次のような状態遷移を経て行われる。

3.3.1　Receiver における状態遷移

Receiver は図 3.3 の通りの状態遷移をすることで Sender から複製を受け取り、ミラーリングを実現する。その状態の意味は次の通りである。

init 初期状態 (あるいは、待ち受け状態)

accepting Sender からの segment の送信を受け取る状態

commit wait Sender からの COMMIT 命令を処理している状態

以下、これらの状態がどのようにして次の状態に移るかなどの細かいところを詰めていく。

init

init 状態は Sender からの INQUIRE 命令を処理するための状態である。Receiver はこの状態から開始し、SUPERBLOCK 命令を受け取ることで accepting 状態に遷移する。

Receiver は init 状態のときに INQUIRE を受け取ると superblock を読むことで前回最後に複製された segment の segment 番号を調べ、それを

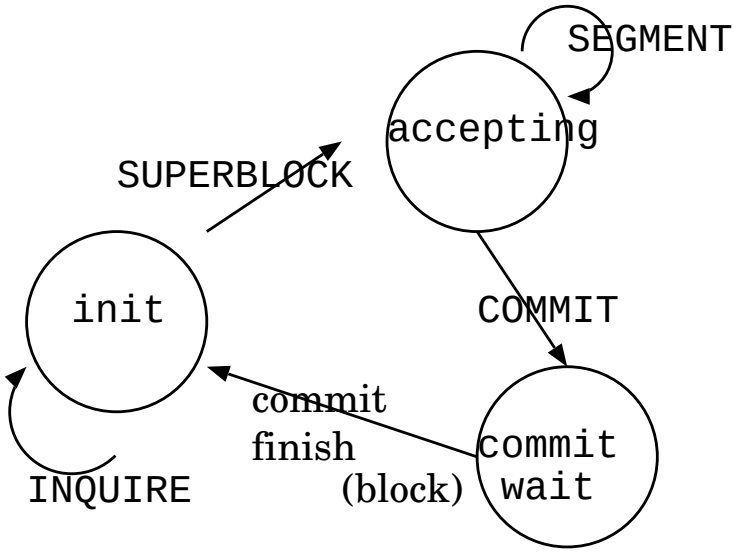


図 3.3: receiver での状態遷移

Sender に返す。Sender はこれに基づき差分を送るようにすることで差分ミラーリングを実現している。つまり、この結果を元に Sender は前回複製された segment 番号から Receiver に segment を送りはじめるのだ。

accepting

accepting 状態は Sender からの SEGMENT 命令を処理するための状態である。SUPERBLOCK 命令を受け取ることで init 状態から移行し、COMMIT 命令によって commit wait 状態に移行する。

Receiver は accepting 状態のときに SEGMENT 命令を受け取ると SEGMENT 命令と一緒に送られて来た segment を SEGMENT 命令と一緒に送られて来た segment 番号の位置に書き込む。このようにして segment 単位での複製を行っている。

commit wait

commit wait 状態はこれまで送られて来た segment を fsync(2) で同期して、SUPERBLOCK 命令の時に受け取った superblock を書き込む状態である。この状態へは COMMIT 命令を受け取ることで遷移し、superblock

の書き込みが終了した時点で Sender に終了した旨のメッセージを送り init 状態に遷移する。

commit wait 状態のときまで superblock の書き込みを遅延することで accepting 状態のときに Sender、あるいは Receiver が故障して停止した場合にファイルシステムの一貫性が崩れることを防いでいる。これは各々の inode は ifile にて管理され、superblock にてその ifile の位置を管理していることによる。つまり、何らかの故障により comit wait で superblock が更新されなければ以前のデータの一貫性が崩れていない状態の ifile を superblock は指しつづけることになるのである。よって、Tottotto は途中で停止した場合でもきちんとファイルシステムの一貫性を取れるようになっている。

3.3.2　receiver の実装

receiver は上記で述べた状態遷移で sender のファイルシステムのコピーを受け取り、それをファイルシステムに反映する。ファイルシステムには sender で使われているファイルシステムと同じく LFS を使っているため、受け取った segment を指定された segment 番号の位置に書いていくことで sender の複製をとることができる。

receiver を使うには LFS で mount されたファイルシステムが必要となる。それは次の 2 つの理由による。まず、実装を簡易にするために receiver は sender から受け取ったパケットをそのまま書いていくようになっているからである。受け取ったパケットを通常の FFS のファイルに変換していたりするとそれだけでかなりの CPU パワーを使い、ミラーリングが大量に行われた場合には処理が追い付かなくなる恐れがある。次に、最後に行われた更新でどの segment までが書かれたのかを容易に調べることができるようにするためである。LFS には最後に更新された segment 番号を保持する機構が superblock にあるため、この情報を格納するための箇所を苦心して作る必要が無くなる。

状態遷移中に受け取った superblock や segment は raw device を通じて書いていくので kernel を改造する必要ない。superblock を受け取った場合には write_commit 状態になるまで処理が保留されるが、write_commit 状態になったときには先頭の segment にある superblock のマスターと他の segment にある superblock のコピーを新しい superblock で上書きするようになっている。segment を書く場合は各 SEGMENT 命令にある segment

番号の箇所に seek し、write することで segment 単位での書き込みをサポートしている。なお、segment のサイズは Sender,Receiver 間で等しい大きさでなくてはいけない。

3.4　システムの動作

これまで Sender と Receiver の個々の動きについて見て来たが、ここでは両者がどのように強調して動作し、Tottotto を実現するかを示す。Tottotto は次の protocol を使って segment 単位でのバックアップを行っている。

3.4.1　Protocol

sender から receiver へは INQUIRE,SUPERBLOCK,SEGMENT,COMMIT という 4 つの命令がデータとともに送られ、命令に基づきそのデータが処理される。どの命令も正常に処理された場合には OK を返し、処理されなかった場合には NG を返すことで異常を知らせる。なお、INQUIRE のような問い合わせを行う命令に対して返事をする場合には OK につなげて要求された情報を返す。

各々の命令によって Sender と Receiver がどのような動きをするかを以下にて示す。緑の文字は Sender や Receiver の状態を表していて、Sender,Receiver 間の矢印上の文字は命令、あるいは命令に対する応答を表している。なお、Tottotto はプロトタイプであるので NG が帰って来た場合のエラー処理は特に実装しておらず、Sender が終了するようになっている。

INQUIRE

Sender が Receiver にどこまで複製を送ったかを確かめるのに INQUIRE を発行する。そして、命令に基づいて Receiver はディスクを読み、Sender にどこまで複製を送ったかを教える。

Sender が Receiver に INQUIRE 命令を発行するときは図 3.4 のような動きをし、命令の送受信によりお互いの状態が変化する。まず、get_lastlog 状態の Sender は Receiver に INQUIRE 命令を発行する。Receiver では INQUIRE を受け取ったら LFS の最後に書かれた segment(lastseg) を示す lastseg を読み込み、成功すれば OK とともに Sender に返す。失敗の

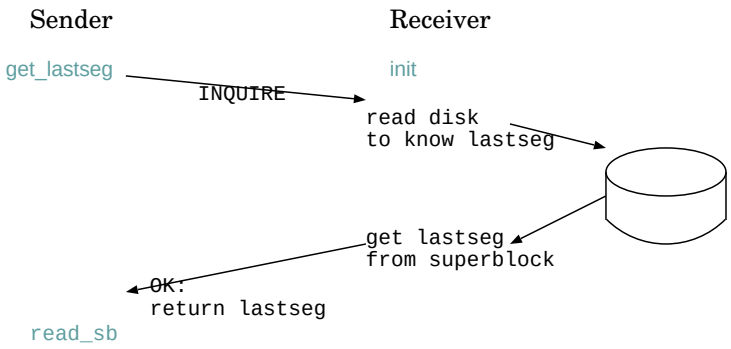


図 3.4: INQUIRE における動作

場合には NG だけを返す。Sender では OK とそれに付随する lastseg を読み、それを元に次回送り始めるセグメント番号を決め、read_sb 状態に遷移する。

SUPERBLOCK

Sender から Receiver にその時点での superblock の内容をコピーするのが SUPERBLOCK 命令である。SUPERBLOCK 命令においては Sender はディスクから superblock 情報を読みだし、それを Receiver に送る。(図 3.5) しかしながら、Receiver は受け取った superblock を書くのを COMMIT まで遅延する。このことで Receiver におけるファイルシステムの一貫性が壊れるのを防いでいる。

SUPERBLOCK 命令は Receiver にとっては Sender が segment を送信する合図となっているので、SUPERBLOCK 命令を受けると Receiver の状態は init から accepting へと遷移する。accepting へと移行した後は SEGMENT 命令により SEGMENT を書き、COMMIT で commit wait 状態に遷移する。

Sender は SUPERBLOCK 命令を発行する前に make_diff 状態となって今回送る segment を選び、その後 write_sb 状態になってから SUPERBLOCK を発行する。make_diff では使われていない segment を今回送る segment を書いたリストから外すようにすることで転送量の削減をしている。

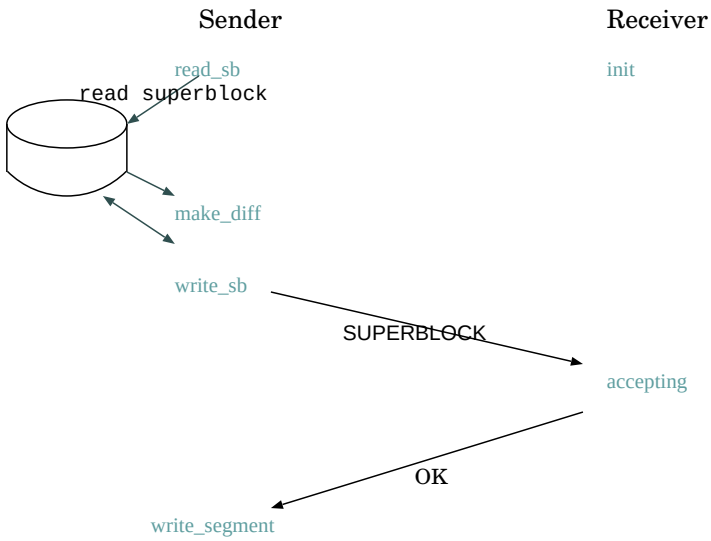


図 3.5: SUPERBLOCK における動作

SEGMENT

Sender から Receiver に segment 単位での複製を行うときに発行する命令が SEGMENT 命令である。SEGMENT 命令を発行するときは Sender は make_diff 状態の時に作成したリストを元にディスク上の segment 番号を読んで、その segment を Receiver に転送する。Receiver では転送された segment を SEGMENT 命令で指定している segment 番号に書き込み、OK を返す。(図 3.6)

COMMIT

Sender が Receiver に対して superblock の内容をハードディスクに書き込みこれまでに送った segment の内容をディスクに同期するように命令するのが COMMIT 命令である。(図 3.7) Sender は make_diff でリストアップした segment を全て送り終えると COMMIT 命令を Receiver に送り、命令を受け取った Receiver ではこれまでに受け取った内容をハードディスクに書き、fsync(2) を実行する。

COMMIT 命令まで superblock の書き込みを遅延するのは、ファイルシステムの一貫性が崩れるのを防ぐためである。LFS でファイルを読むときには superblock を見て Index File(ifile) のディスクアドレスを調べ、

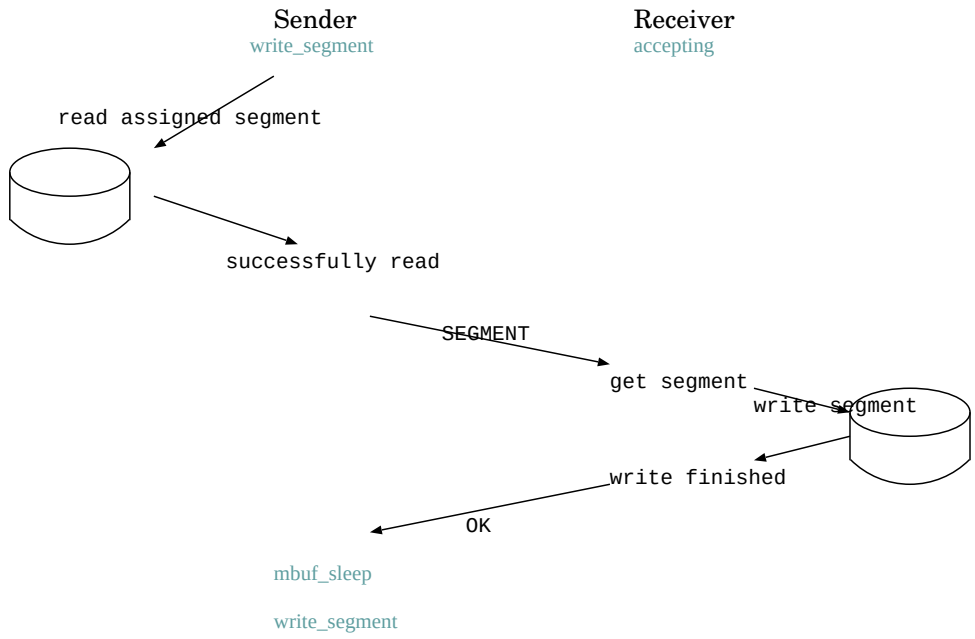


図 3.6: SEGMENT における動作

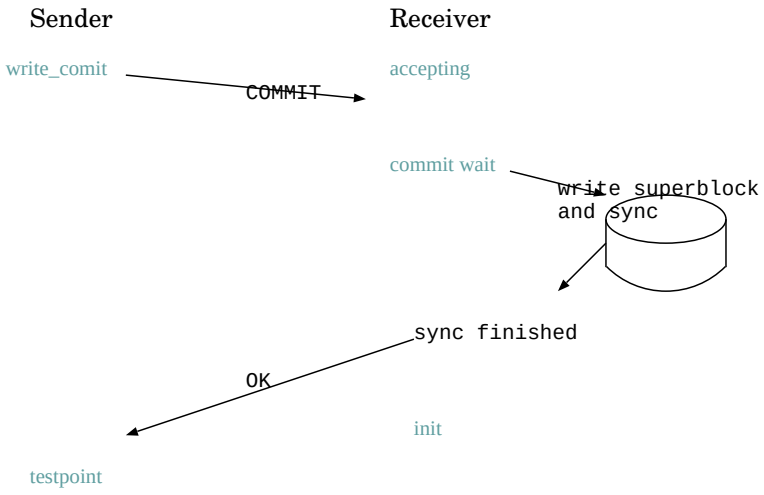


図 3.7: COMMIT における動作

ifile から inode map が書いてある木をたどっていくことで読みたいファイルの inode を見付け、その inode をもとに実際のデータが格納されているディスクを発見する。そこで、superblock が更新されなければいつまでも古い ifile を見続けることになるため、superblock を参照して ifile を読もうとしたときに ifile が無いことによって OS が panic を起こすことを防ぐことができる。Tottotto ではそのことを考え、superblock の更新を segment を書いた後に行うようにしているのである。

第4章　実験

Tottotto を使うと CPU ベースのスケジューリングでは難しかったハードディスク資源などのスケジューリングを行える。それがどの程度うまく働いているかを調べた。

4.1　実験環境

実験に用いた機材は次の通りである。

Sender AMD AthlonXP 2200+,Memory 1GB,Intel Ethernet PRO/100S,NetBSD 1.6R

Receiver AMD AthlonXP 2200+,Memory 1GB,Intel Ethernet PRO/100S,NetBSD 1.6R

http_load マシン AMD AthlonXP 2200+,Memory 1GB,Intel Ethernet PRO/1000,FreeBSD 4.7R

なお、いずれの実験も http_load というベンチマークソフトを用いた。

4.1.1　http_load

http_load というのは Web Server の耐久試験に使われるソフトウェアで、10 秒や 1 分などの決められた時間にどれだけの HTTP リクエストを処理できるかや特定の数のリクエストをどれくらいの時間で処理できるかを測ることができる。また、リクエストは指定された上限まで並列して行うこともできるので実際に複数の箇所から同時にリクエストが来る場合を想定してのテストも行うことができる。

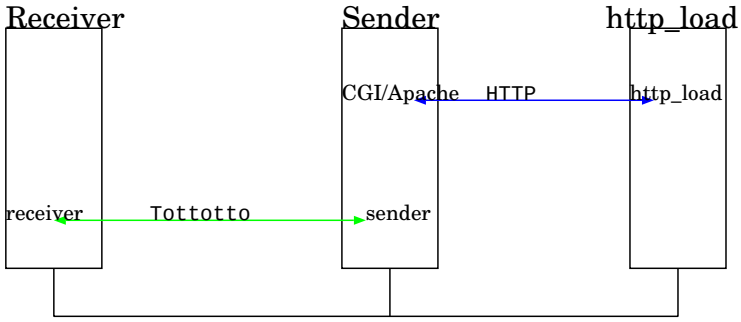


図 4.1: 実験システムの構成

4.2　ディスクへの負荷が大きい実験

4.2.1　実験手順

Tottotto の稼働が CGI の動作に与える影響を見ることで実験を行う。まず、Sender と Receiver にて Tottotto システムを構成してミラーリングを行う。そして、その最中にに別のマシンから Sender で動いている Apache にある CGI を呼び出す。(図 4.1)

実験に際しては http_load を用いて 8 並列で 60 秒[1] リクエスト行い、この CGI の 1 秒あたりの実行回数を比べて、この実行回数が Tottotto を使っていない場合に比べてどの程度違うのかを調べた。これは他のプロセスに与える影響が大きければ他のプロセスの実行が通常時よりも遅く実行されるという考えに基づいているので、実行回数が多ければ多い程影響を受けていないということなので良い。

使った CGI は find(1) を用いてディスクへの負荷が大きくなるようにしている。それは find(1) で/usr/src 以下の QUERY_STRING にて与えた文字が含まれているファイルを探し、表示するというものである。/usr/src 以下を全て探索するとキャッシュできる量を使い果たし、キャッシュに乗らないようになると考えられるので、CGI を実行するたびに頻繁にディスクアクセスを行うことが想定されている。

[1] http_load -parallel 8 -seconds 60 test.url

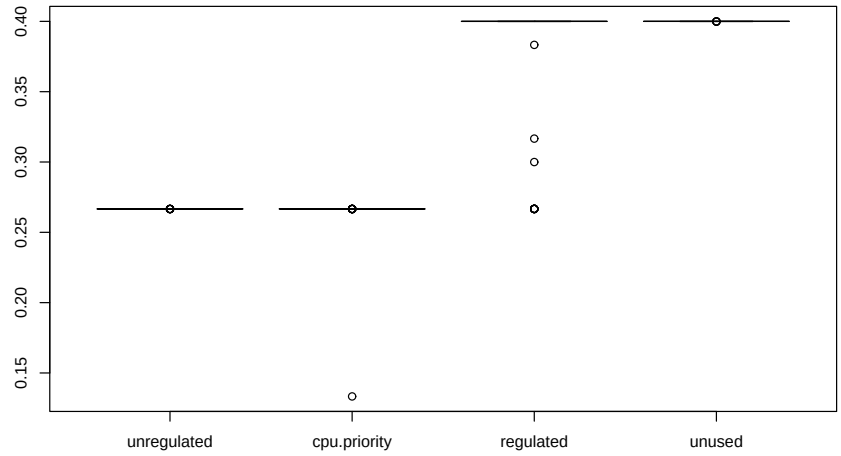


図 4.2: ディスクアクセスを頻繁に行う CGI の場合 (fetchs/sec)

4.2.2　実験結果

find で QUERY_STRING にある文字列を探す CGI の 1 秒あたりの処理時間は図 4.2 の通りである。この結果の unregulated がスケジューリングをしなかった場合、cpu.priority が nice 値を下げることで CPU ベースのスケジューリングをした場合、regulated がきちんとスケジューリングした Tottotto を用いた場合、unused がまったくミラーリングを行わなかった場合の結果である。

4.2.3　考察

この結果から CPU ベースのスケジューリングでは特にスケジューリングをしなかった場合と同じである一方、Tottotto を用いれば使わない場合 (unused) と同等の性能が出るということが読み取れる。

4.3　CPU 負荷がある程度ある実験

4.3.1　実験手順

CPU の負荷が大きい実験は節と同等の環境 (図 4.1) で行った。つまり、Sender から Receiver へ Tottotto にてミラーリングを行っている状態で Sender に対しての HTTP 要求が 1 秒に何回処理されているかを調べた。

実験のために作成した CGI はファイルの内容を 100 文字程度の QUERY_STRING で与えられた文字に書き換えるというものである。これによりある程度のファイル I/O が行われるが、LFS の仕組みではその書き込みのほとんどはキャッシュされて非同期に書き込まれることになると思われる。

4.3.2　実験結果

QUERY_STRING で与えられた文字列をファイルに write(2) するような CGI と競合が起きる CGI であったためかディスク負荷が大きい実験の結果とうって変わって CPU ベースのスケジューリングの方が良いという結果になってしまった。(図 4.3) なお、図 4.3 の見方は**??**と同じで、左から順にスケジューリングが無い場合、CPU ベースのスケジューリングをする場合、スケジューリングがある場合、Tottotto を使わない場合である。

4.3.3　考察

実験結果のグラフより CPU プライオリティーベースのスケジューリングの方が Tottotto でスケジューリングをした場合よりも良いと考えられる。この原因は CPU 負荷がある程度ある環境の場合は検定の結果に大きな変化を与える程の性能低下が無いからであると思われる。また、実験の結果 Tottotto が CPU プライオリティーベースのスケジューリングに劣っていたとはいえ、スケジューリングをしない場合とは優位な差があるということが言えるであろう。

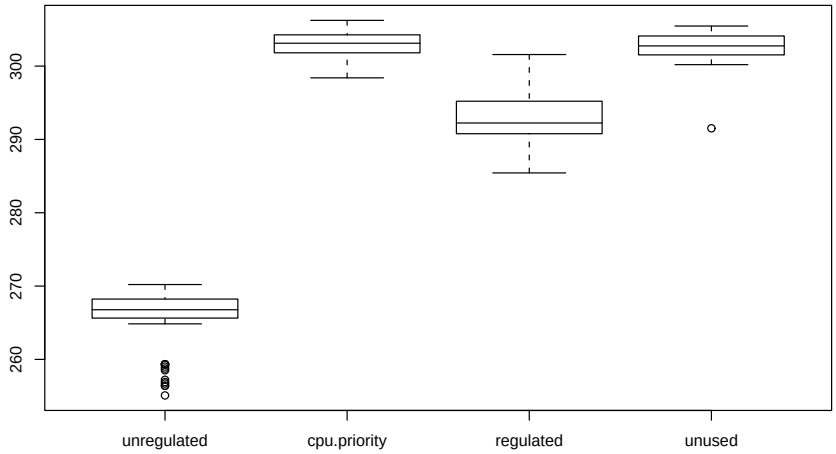


図 4.3: CPU をある程度使う CGI の場合 (fetchs/sec)

第5章　まとめ

本研究ではCPU資源以外の資源の競合が起きても他のプロセスに与える影響の少ないミラーリングシステムTottottoを提案した。Tottottoはネットワーク越しのミラーリングに特化してProgress-based Regulationを改良したシステムであり、NetBSD LFSv2にて構築されたファイルシステムをミラーリングの対象とする。Tottottoではmbufの使用量を見ることでネットワークの混み具合を推し量り、混んでいる場合に動作を停止することでProgress-based Regulationが働きはじめるまでの他のプロセスに与える影響を少なくする。Tottottoを用いることでディスク資源というCPUではない資源の競合が起きた場合にもスケジューリングが働くことを確認した。しかしながら、CPU資源をある程度使うような場合にはnice値をあげるというCPUベースのスケジューリングの方が良い結果が出るというのも確認した。

以下、今後の課題を列挙する。

1. mbufの残量によるスケジューリングの効果の測定
   この論文ではmbufの残量を見ることによりどの程度スケジューリングがうまく働くかを調べていない。今後はmbufの残量を見る場合と見ない場合でどれくらいスケジューリングが働くまでの時間に差があるのかや、その差がどのような影響をもたらすかについて調べたい。

2. 使用されていないsegmentを飛ばすことの効果の測定
   この論文では使用されていないsegmentを飛ばす方法については論じているものの、それを適用した結果どの程度性能が向上したかについては調べていない。今後は使用されていないsegmentを飛ばす場合と飛ばさない場合を比較して、どの程度効率が良くなったかを調べたい。

3. CPU資源の競合への対応
   CPU資源についてのスケジューリングがうまくいっていないので

その原因を調べ、CPU 資源が競合した場合でもきちんとスケジューリングされるようにしたい。

4. スケジューリング間隔の狭小化
   現在のシステムでは最小でも 1segment を送るごとにしかスケジューリングをしておらず、重要度の高いプロセスが活発に動き出してからの対応はまだまだ速いとは言えない。今後は 1segment を送る間でも mbuf を監視するなど、スケジューリングの間隔を小さくして急激な変化により対応できるようにしたい。

5. パケットレベルでの調整
   現在は TCP を使っているので混んでいる場合に通信に失敗すると重要度の低いものであるにもかかわらず何度も再送しようとして重要度が高いプロセスが出したパケットを潰しかねない。混んで来た場合には Tottotto によるミラーリングのためのパケットを再送しないように調整することで優先度が高いプロセスへネットワーク資源を開放するようにしたい。

6. LFS 依存のシステムからの脱却
   現在の Tottotto は差分をすぐに見付けられるということで LFS でしか使えないシステムとなっているが、Background fsck で使われている snapshot 技術を参考にしながら FFS など他の一般の Unix ファイルシステムでも使えるようにしたい。

参考文献

[1] : README, `ftp://ftp.netbsd.org/pub/NetBSD/NetBSD-release-1-6/src/sys/ufs/lfs/READ%ME`.

[2] Douceur, J. R. and Bolosky, W. J.: Progress-based regulation of low-importance processes, *Proceedings of the seventeenth ACM symposium on Operating systems principles*, ACM Press, pp. 247–260 (1999).

[3] Green, R., Baird, A. and Davies, J.: Designing a Fast, On-line Backup System for a Log-structured File System, *Digital Technical Journal*, Vol. 8, No. 2, pp. 32–45 (1996).

[4] Hartman, J. H. and Ousterhout, J. K.: The Zebra striped network file system, *ACM Transactions on Computer Systems (TOCS)*, Vol. 13, No. 3, pp. 274–310 (1995).

[5] Johnson, J. E. and Laing, W. A.: Overview of the Spiralog file system, *Digital Technical Journal*, Vol. 8, No. 2, pp. 5–14 (1996).

[6] McKusick, M. K.: Running "fsck" in the Background, *Proceedings of the BSDCon 2002 Conference*, The USENIX Association (2002).

[7] McKusick, M. K., Bostic, K., Karels, M. J. and Quarterman, J. S.: *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Longman,Inc. (1996).

[8] Patterson, D. A., Gibson, G. and Katz, R. H.: A case for redundant arrays of inexpensive disks (RAID), *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, ACM Press, pp. 109–116 (1988).

[9] Rosenblum, M. and Ousterhout, J. K.: The design and implementation of a log-structured file system, *ACM Transactions on Computer Systems (TOCS)*, Vol. 10, No. 1, pp. 26–52 (1992).

[10] 種村昌之, 新城靖, 千葉滋, 板野肯三: An Implementation of the Backup System Using Network Monitoring Technology, 情報処理学会コンピュータシステム・シンポジウム論文集, Vol. 2001, No. 16, pp. 57–64 (2001).