

メモリ管理リスト分割による大容量メモリ管理手法

Large Scale Memory Management Method Using LRU List Split

大島 訓 青木 英郎 新井 利明¹

Satoshi Oshima Hideo Aoki Toshiaki Arai

現在普及している計算機アーキテクチャは、大容量化するメモリに対して管理コストを削減するための有効な機能を提供していない。特に 32bit アーキテクチャの計算機において、仮想空間を 32bit に据え置いたまま、物理空間を拡張するものが主流となっているため、これを効率的に管理する必要がある一方で、アプリケーションの互換性や信頼性を重視する立場からメモリ管理方式の大幅な変更は受け入れにくいという側面もある。

本論文では、Linux OS を題材とし、既存のメモリ管理方式を大規模に変更することなく、大容量メモリを効率的に管理するためのメモリ管理リスト分割による大容量メモリ管理方式の提案、実装、評価を行う。

This paper describes a proposal of memory management method which split memory management list depending on the purpose, implementation and evaluation.

Current popular computer architecture doesn't provide the function that reduces management cost of large scale memory, which grows larger and larger. Especially, popular 32bit architecture supports over 32bit physical memory space. Memory manager must support such large scale of memory. On the other hand, it is very difficult to introduce completely new management method, because of application compatibility and operating system stability. Split memory management list solves a part of this problem.

¹ 日立製作所システム開発研究所

1 はじめに

計算機に搭載される物理メモリ量は、年々増加しており、1995年におけるデスクトップPCの標準的なメモリ搭載量が16MB~32MBだったものが、現在では256MB~512MBに増大している。さらに現在のPCサーバでは、12GB以上のメモリを搭載できるものが市場に出回って下り、一般的に利用されている。しかし、x86アーキテクチャでは、32bit拡張以後、メモリの管理単位が4KB(または2MB,4MB)に固定されたまま、現在に至っている。

さらに問題を複雑にする要素として、x86アーキテクチャが仮想空間を32bit(4GB)に固定したまま、物理アドレス拡張(PAE)によって物理メモリだけ36bit(64GB)空間をサポートしたことがある。オペレーティングシステムがこうしたメモリを有効利用することができれば、性能向上が期待できる。半面、効果的に実装しなければ管理に必要となるデータ量が増大した分だけ、管理オーバーヘッドが発生し性能が劣化するという問題がある。

図1は、Linux 2.4.26におけるI/O throughputを測定したものである。測定では、ワークロードジェネレータの設定を固定し、搭載メモリ量だけを変化させファイルI/Oのthroughputを測定した。メモリの搭載量が4GBを超えると、throughputが大きく低下していることがわかる。

Linux 2.4系カーネルでは、その他の不具合とあいまって、メモリ搭載量が多い計算機に大きな負荷を長時間かけた場合、システムが障害停止する可能性があることがわかっている。

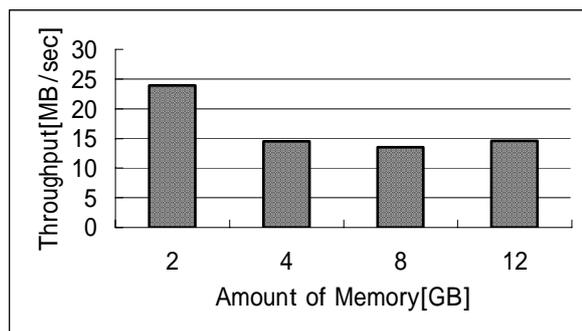


図1 Linux 2.4.26のI/O throughput

メインフレーム計算機では、こうした問題に対し、(1)プロセス空間とデータ空間を別々に管理する、(2)agingを停止する、(3)page tableやpage descriptorは物理アドレス管理とする、といった手法を用いて、管理コストの増大を抑えてきた。しかしPC向けOSでは、アーキテクチャの違いから、従来汎用機で利用されてきた手法が採用できない場合がある。そこで、本研究ではPC向けOSの代表であるLinuxを例にとりメモリ管理コスト増大の原因と、その軽減策の一つであるメモリ管理リスト分割による大容量メモリ管理手法を提案する。

2 Linux VMの現状

Linuxカーネルは各pageの状態をstruct pageで表わされるpage descriptorで管理している。Page descriptorは必ずカーネル仮想空間に常駐するよう実装されている。

Virtual Memory Manager (VM)は、pagingの対象となるpageをactive-list、inactive-list、free-listという3つのlistで世代管理する、clock algorithmを採用している。VMはmemory agingの際、active-listから参照されていないpageをinactive-listに移し、inactive-listから参照されていないpageを

free-list に移す。inactive-list 検索中に参照が確認されれば active-list に戻される。プロセスから明示的に開放された page は free-list に移される。

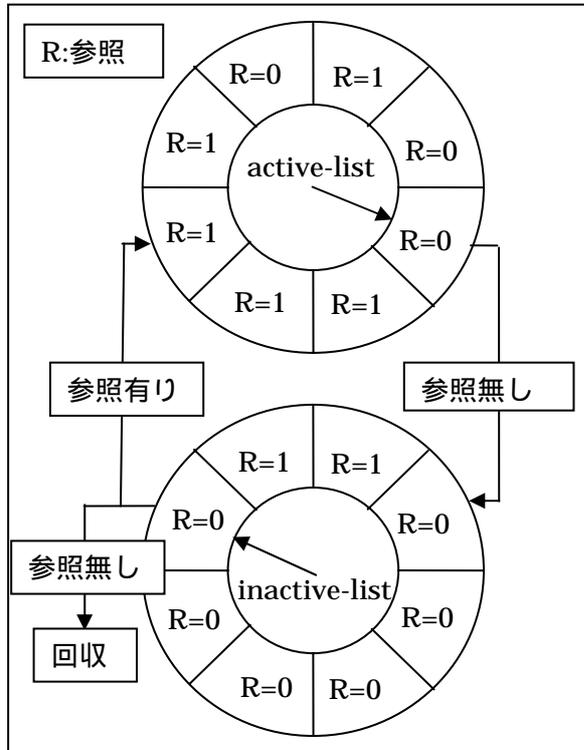


図2 Linux VM のメモリ管理リスト

このメモリ管理リストには、プロセス領域(コード、データ)、メモリマップドファイル、スラブ、ディスクキャッシュなど、様々な用途に使われるページが混在している。

3 Linux メモリ管理の課題

Linux では、2章で述べたメモリ管理方式を基本としているが、バージョンによってその実装方法が大きく異なる。ここでは現在、よく利用されているバージョンの課題をあげる。

Linux のメモリ管理実装は、大きく分けて以下の4つに分類できる。

- A) 2.4.1 ~ 2.4.9
- B) 2.4.10 ~

C) 2.6.1 ~ 2.6.6

D) 2.6.7 ~

また、カーネルは 2.4 系から 2.6 系にかけてメモリ管理以外にも大きな改善が加えられている。2.4 から 2.6 への改善項目の中でメモリ管理に影響があると考えられるものは以下の通である。

Lock の細分化

I/O throughput の向上

Pagecache と Buffercache の統合

これらを踏まえて、Linux の VM が抱える課題をあげる(括弧内はその問題を抱えるバージョン)。

Reverse mapping が無い(B)

2.4.10 以降の 2.4 系カーネルには reverse mapping table がない。つまり page の参照状態を page descriptor から page 毎に検査する手段がない。そのため VM は aging の際、すべての page table entry(PTE)を検索して、page の参照状態を調査し list 間のリバランスを行う仕組みになっている。このオーバーヘッドは、搭載メモリ量が大きくなるほど増大する。

Reverse mapping table 作成コストが大きい(A,C)

Reverse mapping が page 単位に作成されるため、搭載メモリ量が大きくなると reverse mapping 作成コストも増加する。また、reverse mapping table はカーネル仮想空間に常駐することにより、カーネル仮想空間を圧迫する。

マルチプロセッサ・スケーラビリティが低い(A,B)

lock の粒度が低いため、aging 操作の際に取得する lock の占有によって、マルチプロセッサ・スケーラビリティが低下する。特に搭載メモリ量が増加した場合、swapper が lock を取得したまま回収できる page を探して動き続けるため、他の処理が一切走らないといった障害が発生する場合がある。

カーネル仮想空間に配置されるデータ構造に

対して空間サイズが小さい(32bit アーキテクチャ全般)

32bit アーキテクチャ向け Linux では、4GByte の仮想空間を 3GByte のユーザ空間と 1GByte のカーネル空間とに分割している。カーネル仮想空間には、page descriptor、reverse mapping table、task structure 等、様々なデータが常駐する。また、現在の実装では、inode-cache、directory entry cache などの filesystem meta data も仮想空間に乗せなければならない。特にメモリに関するデータは、搭載メモリ量に依存する。カーネルのバージョンやアーキテクチャによるが、page descriptor は 32Byte ~ 128Byte 程度の大きさである。page 毎に page descriptor を作成する必要があるため、x86 アーキテクチャにおける 64GB の物理メモリに対応した page descriptor の大きさは 500MB ~ 1GB にもなる。(つまり、現在のカーネルでは、64GB のメモリを扱うことはできない)

4 メモリ管理リスト分割方式の提案

前記のような課題に対し、いくつかの解決策が考えられる。

例えばカーネル空間が不足する問題に対しては、page descriptor を物理メモリに配置し、仮想空間にマップすることなく直接管理する方法や必要に応じてマッピングするように変更する方法がある。この方式によれば、仮想空間の不足によって page descriptor や reverse mapping table が配置できないという問題は解決される。しかし、x86 アーキテクチャでは、物理アドレスに直接アクセスする方法がなく、また物理に配置し必要に応じてアクセスする方法は、マッピングのオーバーヘッドが大きく現実的ではない。

また、現在 3GByte:1GByte で分割されているユーザ空間:カーネル空間を、ユーザ空間=4GByte、カーネル空間=4GByte に拡張し、システムコールや割り込み処理毎に空間切り替えを行うという方法も提案されている(図3)。カーネル空間の拡大によって、page descriptor や reverse mapping table などを従来の 4 倍以上配置できるようになるため、仮想空間不足の問題を解決することはできる。しかし、この方法もほとんどの x86 プロセッサで、空間切り替えのたびに TLB がパージされる実装となっておりオーバーヘッドが無視できないほか、ライブラリやドライバの互換性の問題が大きく、決定的な解決策にはなっていない。

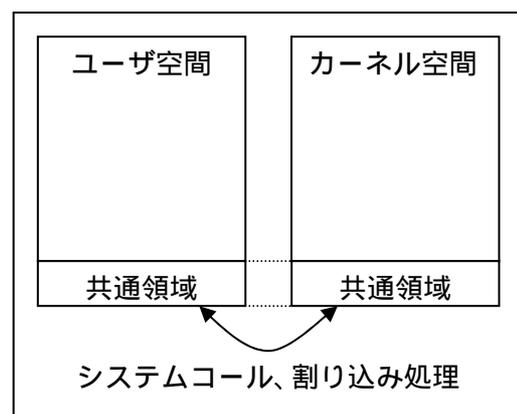


図3 4G/4G スプリット

reverse mapping table が無い問題やマルチプロセッサ・スケーラビリティが低い問題については reverse mapping table の導入や lock の細分化など、新しいバージョンのカーネルでは解決された問題であっても、API 互換性の問題から、新バージョンに移行できないアプリケーションにとっては、解決にならないという問題もある。

更には reverse mapping を単純に追加しても、アーキテクチャの制約によって、page を拡大できない場合、reverse mapping 作成のオーバーヘッドが増加し、新たな問題を引き起こし

ている。

そこで、どのようなメモリ搭載量に対しても有効なメモリ管理オーバーヘッド削減方式を検討するのではなく、メモリ搭載量が増加した場合に特化した、メモリ管理改善方式を検討する。ただし、信頼性の観点からデータ構造などに大きな変更を加えることは避ける。

そこで前記の課題や x86 アーキテクチャの制約から必要となる要件をまとめる。

- A) aging オーバヘッドの削減
- B) page descriptor に値を追加しない
- C) VM 以外の部分を変更しない

更に、メモリを多く搭載する場合、swap の必要性が低く、アプリケーションによっては不要であることを仮定する。

現在の page 管理リストには、プロセス、キャッシュ、カーネルデータなど様々な page が混在している。しかし page の使用目的によって、page が参照されるあるいは書き込まれる頻度は異なる。プロセスのコード領域などはアクセスの局所性が高く、したがって部分的な再利用率もたかい。一方、ディスクキャッシュなどは、大きなデータが一度書き込まれたら 2 度と参照されない場合がある。現在の 3 つのリストによる管理方式では、これらアクセス傾向が異なる page が一様に扱われている。もしアクセス傾向に適した管理方式を適用することができれば、不要な page の参照検査や page 管理リスト間のリバランスが削減できる。更にメモリ搭載量が十分な場合、page の使用目的によっては aging をスキップすることで、不要な aging オーバヘッドや swapin/swapout を削減できる。

そこで、アクセス傾向が異なる page をまとめて管理するために、図 4 に示すように page の使用目的毎に active-list を分割するメモリ

管理方式を提案する。分割する単位としては、プロセスのコード領域、読み込み専用データ領域、通常データ領域、プロセス間共有メモリ、バッファキャッシュなどが考えられる。

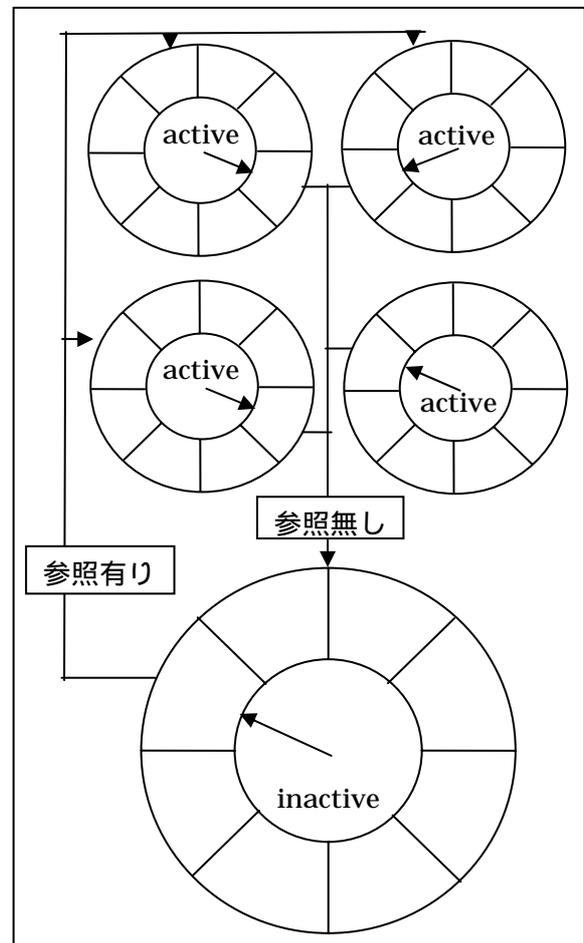


図 4 active-list 分割によるメモリ管理

aging の際には、これら active-list の使用目的ごとに適した割合、頻度で、適したアルゴリズムで inactive-list へのリバランスを実施する。

active-list 分割を導入することによって、例えばプロセス間共有メモリは参照される頻度が低く、利用される容量が大きい場合であっても、pageout された場合のペナルティが大きい場合 aging の頻度を減らし、利用目的によっては aging 対象外とする、また、バックアップサーバでは、ディスクキャッシュは再利用され

る可能性が非常に低いため、優先的に aging し、積極的に inactive-list へ page を移すようにするといったように、システムの利用形態に併せて、aging 方法を柔軟に変更することが可能となる。

Linux でも 2.6 系カーネルでは、active-list に含まれるプロセスに利用されている page は、aging の際一定の割合で無条件に active-list に戻すような実装になっている。しかし、aging の際 active-list から page を取り出し、その page の使用目的を検査して、プロセスであることを確認してから、active-list に戻すという手続きを踏んでいるため、無駄な処理が発生している。さらに、ロック取得時間や aging の時間を大きくしすぎないために、メモリ要求フラグによっては、回収可能ページの探索を一定量、active-list 探索を終えた段階であきらめて、しばらく待つといった処理も行われている。この方法は、システムの安定性には大きく貢献するが、active-list 内の参照/書き込みページの位置的なバランスが崩れた場合、メモリ回収速度がばらつき、プロセスの実行や I/O throughput のばらつきにつながる。

一方、提案の方式によれば、page は確保時や inactive-list から戻される際に、使用目的を検査され、属する active-list に入れられることになる。このため、aging のコストは平均して低くなることが予想される。

また、active-list の分割方式は、前記 A ~ D いずれのバージョンへも実装することが可能で、大容量メモリ搭載時の aging コストを一定割合で削減することが期待できることも利点のひとつである。さらにページの使用目的がわかっている場合、不要な lock の取得が回避できるため、マルチプロセッサ・スケーラビリティの向上が期待できる場合がある。そして、カーネル空間に追加する必要があるデータ量は、

ゾーン毎に細分化した list の LIST_HEAD 構造体のみであるため、数十 ~ 数百バイトであると考えられる。

5 active-list 分割によるメモリ管理方式の実装

本論文で提案した、active-list 分割によるメモリ管理方式の実装を行った。

今回の実装では、考えられる細分化をすべて実装するのではなく、実現可能性調査を兼ねて、

- (1) プロセス空間にマップされた page のリスト(active-mapped-list)
- (2) プロセス空間にマップされない page のリスト(active-unmapped-list)

の 2 つの active-list に分割する方式を採用したパッチ[III]をベースとした。

現在の実装では、active-mapped-list には、プロセスのコードセクション、データセクション、メモリマップドファイル、カーネルのコードセクション、データセクション等が含まれる。一方、active-unmapped-list には、ページキャッシュが含まれる。

active-mapped-list と active-unmapped-list の aging 割合はチューニングパラメータ(mapped_page_cost)によって、決定される。

さらにメモリが十分に搭載されている場合などには、mapped_page_cost を 100(最大値)とすることで、active-mapped-list の aging を停止する機能を追加し、プロセスの paging が不要な場合、プロセス page の aging オーバヘッドを削減できるようにした。

6 active-list 分割によるメモリ管理

方式の評価

active-list 分割の効果を検証するために、Linux 2.6.6-mm3 と、Linux 2.6.6-mm3 に active-list を分割したカーネルを作成し、I/O throughput の比較を行った。測定条件は以下の通りである。

測定環境

CPU : Xeon 1.6Ghz * 4CPU(HT 無効)

メモリ : 12GB

HDD : 36GB*6 RAID 5

負荷生成プログラム

プロセス数 : 1024

ファイル I/O : 100KB ~ 5MB 書き込み

プログラム実行時間 : 5 時間

カーネルバージョン

(1) 2.6.6-mm3

(2) 2.6.6-mm3-split-stop-aging

mapped-page-cost の設定

(2)の場合のみ 100

この条件の下で、Linux に認識させるメモリ量を 2GB ~ 12GB の間で変化させながら、I/O throughput の測定を行った。測定結果は図 5 に示すとおりである。

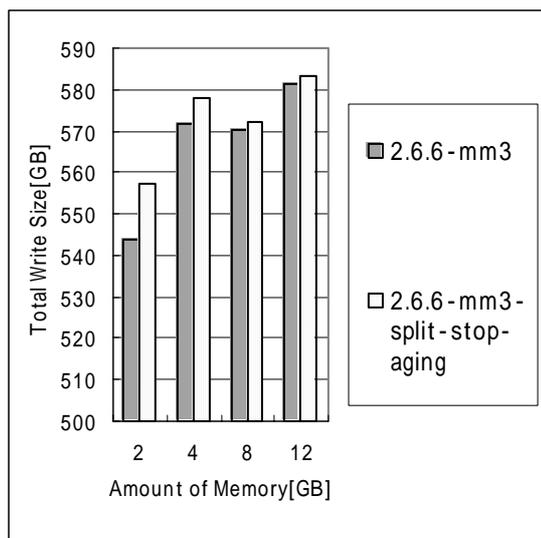


図 5 I/O throughput 測定結果

測定結果から明らかなように、active-list 分割前と比較して、分割後は性能が向上している。今回は、測定したワークロードが単純な I/O throughput の測定であったため、aging 停止の差が出にくいですが、DBMS や Java など多数のプロセスが大きなメモリを確保するようなアプリケーションに対しては、この性能向上が顕著になることが予想される。

7 今後の取り組み

今回は、active-list を 2 つに分割する実装としたが、より細かい分類を実現することで、さらに低いオーバーヘッドときめ細かい制御を実現したい。特に、アクティブリストにプロセスやカーネルの領域とメモリマップドファイルの領域が混在していることは、大きな問題であり、この分割から続けたいと考えている。

8 終わりに

計算機に搭載されるメモリ量の増加に対し、管理単位であるページが増えないという問題は、Linux にとどまらず多くの OS に影響を与

える問題と考えられる。今回提案した方式は、その解決策の一端を担う可能性はあるものの、この対策だけで問題をすべて解決できるとは考えられない。今回は、特に active-list に着目し、性能改善を提案したが、[V]に紹介されているように inactive-list を分割し、性能向上を図る方式や free-list 管理方式の見直しなど、様々な性能改善を組み合わせることで、オペレーティングシステム全体のスケーラビリティを向上させていく必要がある。

Linux は、Linus Torvalds 氏、米国およびその他の国における登録商標 あるいは商標です。

インテル®は、米国およびその他の国におけるインテル コーポレーションまたはその子会社の商標または登録商標です。

参考文献

- [I] 「詳解 Linux カーネル 第2版」, Daniel P. Bovet 著, 高橋 浩和監訳, オライリー・ジャパン
- [II] 「Operating System Fourth Edition」, William Stallings 著, Prentice Hall
- [III] 「VM split active lists」, Nick Piggin, <http://seclists.org/lists/linux-kernel/2004/Mar/1798.html>
- [IV] 「IA-32 インテル®アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル」, Intel Corporation
- [V] 「Towards an O(1) VM」, Rik van Riel, Proceedings of the Linux Symposium