Vol.2022-OS-156 No.5
2022/7/27

ヒュージページにおけるコピーオンライトを考慮したメモリ管理機構

笠原 一真[1]　山田 浩史[1]

概要：メモリの大容量化及びアプリケーションのメモリ使用量増大化に伴い，仮想アドレスを物理アドレスへ変換するための時間の増加が大きなオーバヘッドとなっている．この問題を解決するために 1 ページ当たりのサイズを大きくした Hugepage が考案され，これを用いることでアドレス変換時間を短縮することができる．また Hugepage の活用方法も提案され，Linux では透過的に Hugepage を割り当てる THP が提供されている．しかし，従来の手法では Hugepgae の Copy-on-Write(CoW) 発生時にその Hugepgae が分解され物理的に非連続になるため，元の領域を Hugepage へ再統合することができなくなる．また，Hugepage の再割り当てを行うためにページコンパクションが必要になるという問題もある．そこで，本研究では CoW 発生時にメモリの連続領域を維持するプロセスを選択可能にする手法を提案する．これにより，Hugepage の再統合を容易に行うことが可能になる．提案手法を実現するために，本研究では CoW によりコピーされたページを，ページを共有するプロセスのページテーブルエントリに参照させる機構を作成した．提案手法を Linux 5.10.25 上に実装し，動作確認のための実験を行った．実験では，物理メモリの連続領域が維持され，Hugepage へ即座に統合されることを確認した．

キーワード：hugepage，THP，copy-on-write，page table

1. はじめに

メモリの大容量化及びアプリケーションのメモリ使用量増大化に伴い，仮想アドレスから物理アドレスへ変換するための時間が増加している [1] [2] [3] [4] [5]．メモリは基本的に 1 ページ 4KB の regular page 単位で管理されており，ページテーブルを利用してアドレスの変換や管理が行われているが，メモリの巨大化に伴いページテーブルも大きくなり，Intel® 64 アーキテクチャでは 4 段階のページテーブル構造である 4-level paging[*1] が採用されている [6]．よって，アドレスの変換には 4 段のページテーブルを経由する必要があり，またページテーブルの増加により TLB ミスが起こりやすくなることもあるためアドレス変換時間が増加する [7].

そこで，1 ページ当たりのサイズを大きくした Hugepage を利用することでアドレス変換時間を短縮することができる [8]．Hugepage を利用した場合，必要なページテーブルエントリ数が少なくなるため，TLB ヒット率，TLB ミス時のアドレス変換速度が向上する．具体例として，Intel® 64 アーキテクチャでは 2 MB の Hugepage がサポートされている．このサイズは regular page の 512 個分の大きさであり，物理的に連続した 2 MB のメモリを確保することによってページテーブルエントリの数を 512 個から 1 個にまとめられる．4-level paging で Hugepage を利用するとページテーブルが 3 段になり，TLB ミス時に経由するページテーブルの数が減ることでアドレス変換が高速化される．さらに，ページテーブルの数が減ることで TLB ミスが発生しづらくなり [9]，平均的にもアドレス変換時間が短縮できる．また， Hugepage の利用をサポートするために，Linux では透過的に Hugepage を割り当てる Transparent Huge Page (THP) [10] という機能が提供されている．

しかし， THP に対して Copy-on-write (CoW) を実行する場合には Hugepaeg の恩恵を受けられないことがある．Hugepage に対して CoW を適用する場合， Hugepage をそのままコピーするとデータサイズが大きいためレイテンシが大きくなってしまう．これを防ぐために， Linux 5.10.25 では， fork() 実行後に親子プロセス間で共有されている Hugepage に対して書き込みを行う場合，書き込みを行う側のプロセスの Hugepage を regular page に分解してからページのコピーを行う．そのため，ページコピーは regular page

[1] 東京農工大学
Tokyo University of Agriculture and Technology

[*1] さらに多くのアドレス空間を扱うことができる 5-level pageing も導入されているが，本論文ではデフォルトである 4-level pageing を扱う

単位で行われるが，書き込みを行う側のプロセスではコピー先のページを新たに参照するので物理的に連続していたコピー元のメモリ 2 MB は連続ではなくなる．したがって同じ領域を Hugepage へ再統合することができなくなるためアドレス変換時間によるレイテンシが削減されない．アプリケーションに悪影響を与える例として， Redis [11] でスナップショットを作成する場合について考える．Redis とはインメモリデータベースの一つであり，スナップショットを作成するときに fork() を実行し，子プロセスでスナップショットの作成を行う．親プロセスではそれ以降の処理が続けられるが，子プロセスが存在しているときに元々 Hugepage を利用していた領域に書き込みを行った場合に上記の問題が発生する．親プロセスのメモリの連続領域が維持されず，子プロセス終了後に同じ領域を Hugepage へ再統合することができなくなってしまう．また，Hugepage の再割り当て時にページマイグレーションが発生し，必要に応じてメモリコンパクション [12] も行われるが，これらの影響で性能が劣化する [13] [14].

そこで，本研究では CoW 発生時にメモリの連続領域を維持するプロセスを選択可能にする手法を提案する．これによって， CoW 発生時に Hugepage に対して書き込みを行ったとしても，書き込んだ側のプロセスで Hugepage への再統合を即座に行うことが可能となる．また，ユーザは従来の手法と提案手法のどちらを用いるのかを選択可能であり，自動的に選択した CoW 手法が実行される．提案手法によるメリットとして， fork() によるページの共有が解除された後に同じ領域を Hugepage を再び利用することが可能になり，アドレス変換速度が低下するのを防ぐことができる．たとえば， Redis ではスナップショット作成の終了後，親プロセスで Hugepage を即座に利用することができるためアドレス変換が高速化される．

本研究の貢献は次の通りである．

- Hugepage への書き込み命令発生時に，書き込み元の Hugepage が即座に再統合されるページテーブル管理機構を提案した．コピーされたページを書き込みが発生した側ではないプロセスで利用することによりこれを実現する．
- 提案手法を実現するために必要なメカニズムを明らかにし，設計・実装を行った．ユーザが既存手法と提案手法を選択可能であり，カーネルは自動的に選択した手法によるページコピーを行う．
- Linux 5.10.25 上に提案手法を実装した．スナップショット取得中に CoW が発生したとしても，書き込み元のプロセスで Hugepage が即座に再利用可能となる．

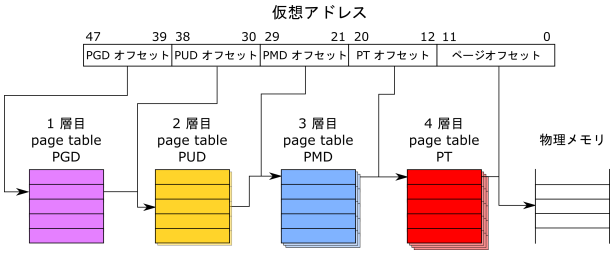

図 1: 4-level paging における regular page のテーブル構造とアドレス変換

2. 背景

2.1 アドレス変換

2.1.1 4-level paging

ページの管理や仮想アドレスから物理アドレスへの変換を行うためにページテーブルが利用されている．ページテーブルとは仮想アドレスと物理アドレスを結びつけるデータ構造であり，カーネルではこれを参照してアドレス変換を行う．x86-64 アーキテクチャでは，ページテーブルが 4 段階で構成される 4-level paging を採用している．x-86-64 の 4-level paging におけるページテーブルを図 1 に示す．x86-64 では上位のページテーブルから順に PGD, PUD， PMD， PT と名前が付けられており，それぞれ 512 エントリで構成されている．8 byte からなる各エントリは下位のページテーブルの先頭ポインタなどの情報が格納されており，最下位の PT のエントリである PTE は一つの regular page と対応している．よって， PT 一つで regular page 512 個分である 2 MB のメモリがカバーされる．また，最上位の PGD は各プロセスで一つのみであり，必要に応じてエントリが新たに作られる．この構成上，使用メモリ量が多いと最下位のページテーブルである PT の数が非常に多くなる．なお x86-64 アーキテクチャではこれに加えて p4d というテーブルが追加されているが内容は pgd と同じである．

2.1.2 アドレス変換時間によるレイテンシ

仮想アドレスから物理アドレスへの変換の流れを，図 1 を利用して説明する．仮想アドレスは各ページテーブルのオフセット 9bit が 4 つとページオフセット 12 bit の合計 48 bit から構成される [6]．まず， PGD エントリを取得するには，プロセスごとに保存されている PGD ポインタと仮想アドレスの PGD オフセットを利用する．さらに下位のテーブルのエントリを取得するには， 上位のエントリ内に格納されている下位テーブルの先頭ポインタと，仮想アドレスのオフセットを利用する．最後に PTE から物理アドレスを取得する際は， PTE 内のページフレームと仮想アドレスのページオフセットを利用する．このように，別のページテーブルのエントリを参照する度にメモリアクセスが発生する．もし対象のページの情報が TLB にキャッ

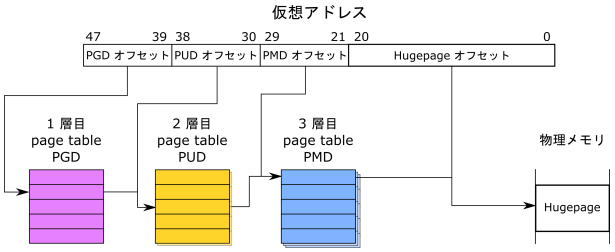

図 2: Hugepage のページテーブル構造とアドレス変換

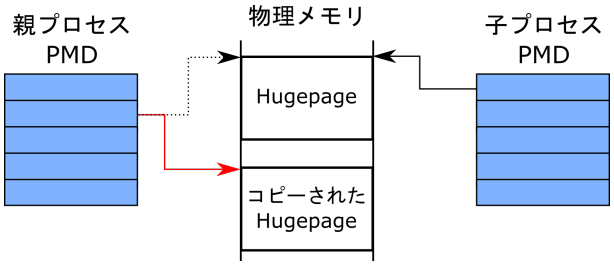

図 3: 従来の THP における Hugepage のページコピー

シュされていればアドレス変換が高速化されるが，使用メモリが大きくなるほどエントリ数が増え TLB ヒット率が低くなる．よってメモリが大きくなるほどアドレス変換によるレイテンシは無視できなくなってくる [15] [16] [17].

2.2 Hugepage

Hugepage とは 1 ページ当たりのサイズを大きくしたページのことである．これを利用することによって，アドレス変換が高速化されるというメリットがある．Intel® 64 アーキテクチャでは 2 MB の Hugepage がサポートされている．本節では 4-level paging における Hugepage の扱い方やメリットについて説明する．

2.2.1 4-level paging における Hugepage

4-level paging において Hugepage を利用する場合のページテーブル構造を図 2 で示す．regular page の場合から PT が省略されており，仮想アドレスにおける PT オフセットだった部分はページオフセットに統合される．これによって， PMD エントリ一つと 2 MB の Hugepage とが対応する．なお regular page と Hugepage は共存することができる．

2.2.2 Hugepaeg によるメリット

Hugepage を利用することによって，アドレス変換が高速化される．TLB ミス時には，参照されるテーブルの数が減るためメモリアクセス数も減り，さらに TLB エントリ 1 つにつき 2 MB のメモリをカバーすることができるため TLB ヒット率も大きくなる．4-level paging における TLB ミス時には， Hugepage の利用によりアドレス変換のために経由するページテーブルが 4 段から 3 段に減る．また， Intel® Core™ i7-6700TE Processor [18] を例に挙げると TLB は 1536 エントリあり，regular page の場合 6 MB のメモリがカバーされる．一方 Hugepage の場合， カバーされる範囲が 3 GB となりヒット率が向上することが分かる．

2.3 fork()

2.3.1 fork() によるページテーブルのコピー

fork() はプロセスを複製するシステムコールだが，はじめからメモリ全体がコピーされるわけではない．ページテーブル等については fork() 実行時にコピーが行われるが，ページに対しては CoW を適用しており，親子プロセスのどちらかで書き込みが発生した時点でページコピーが発生する [19].

ページテーブルのコピーについては regular page も Hugepage も同様に行われる．regular page の場合は PGD，PUD，PMD，PT のコピーが行われ， Hugepage の場合は PGD，PUD，PMD のコピーを行う．したがって fork() 直後は親子プロセスで異なるページテーブルが存在し，参照するページは共有されている状態となる．

2.3.2 ページコピー

fork() によってプロセス間で共有されているページに書き込みを行う場合，どのようにしてページコピーが発生するかを説明する．fork() 実行時にページテーブルがコピーされるが，このときエントリ内の read/write bit を 0 にして書き込みを禁止にする．この状態で書き込みが発生すると，ページフォールトが起こる．ここでページがコピーされ，書き込んだ側のプロセスのエントリの参照先をコピーしたページに変更してから書き込みを行う．regular page をコピーした場合は PTE の参照を書き換え， Hugepage をコピーした場合は PMD の参照を書き換える．read/write bit は書き込んだ側のエントリでは 1 となり，他のプロセスでは 0 のままとなる．なお各レベルのページテーブルのエントリに read/write bit が存在するため， regular page の場合は PTE， Hugepage の場合は PMD エントリまでのビットを変更する．

2.4 Copy-on-Write

Hugepage の CoW 時の動作は Linux のバージョンによって異なる．ここでは従来のバージョンと Linux 5.10.25 の動作の違いについて説明する．

2.4.1 従来の CoW

従来のバージョンの THP では，fork() で共有されている Hugepage への書き込みが発生した場合，2 MB の Hugepage をコピーしていた．具体例として，親プロセスで書き込みが発生した場合のページコピーの動作を図 3 で示す．Hugepage をコピーした後，親プロセスの PMD エントリの参照先をコピーした Hugepage に変更する．この方法の問題点として，共有された Hugepage への書き込み命令のレイテンシが大きくなる．なぜなら，書き込み命令を実行するために 2 MB という大きいサイズのページコピーが必要

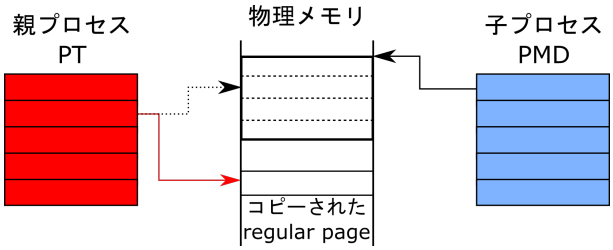

図 4: Linux 5.10.25 における Hugepage のページコピー

になるため，メモリコンパクションやページコピー自体に時間がかかるからである．

2.4.2 linux 5.10.25 における CoW

Linux 5.10.25 では，fork() で共有されている Hugepage への書き込みが発生した場合，それを regular page に分解してから regular page 単位でページコピーを行う．具体例として，親プロセスで書き込みが発生した場合のページコピーの動作を図 4 で示す．ページフォールト発生時，親プロセス側の Hugepage を参照していた PMD エントリが PT を参照するように変更することで，regular page の場合と同様にしてページのコピーが行われる．この方法によるメリットは，共有された Hugepage への書き込み命令のレイテンシが大きくならないことである．

しかしこの方法にも問題点がある．それは，親プロセスで Hugepage への書き込みを行った場合，子プロセスが終了した後に同じ領域の Hugepage への再統合が不可能になり，アドレス変換が高速化されなくなることである．Hugepage へ書き込みを行うと， CoW が発生しそのプロセスにおいて元々物理的に連続していた 2 MB のメモリが不連続になる．したがって同じ領域の regular page から Hugepage への再統合が不可能になる．Hugepage を再び割り当てるためには 2MB の連続領域を再確保し 512 regular pages をマイグレーションする必要があるが，このとき 2MB のメモリコピーによるレイテンシが発生する．

また，メモリの走査やコンパクション [20] が実行されることによって CPU リソースが圧迫され性能が劣化する場合がある [3] [21]．特に子プロセスの実行が一時的であるアプリデーションの場合， この影響が大きくなると考えられる．具体例として， Redis のスナップショット取得について考える．Redis とはインメモリデータベースの一つであり，メモリ上で動作するため高速なデータアクセスを行うことができる．Redis では特定の時点でのスナップショットをディスクに保存し永続化することができるが，このとき上記の問題が発生する場合がある．スナップショットの作成は fork() を実行後，子プロセスで行われる．親プロセスではそれ以降の処理が続けられるが，子プロセスが存在しているときに元々 Hugepage を利用していた領域に書き込みを行った場合，親プロセスのメモリの連続領域が維持されず，子プロセス終了後に元の領域を Hugepage へ再統合することができなくなってしまう．スナップショットを定期的に取得する場合，時間がたつにつれてメモリの連続領域の分裂が進み Hugepage の確保が困難になると考えられる．その結果，特にメモリサイズが大きい場合にアドレス変換によるレイテンシが無視できないほど大きくなる可能性がある．実際に， Redis などのアプリケーションでは THP の無効化が推奨されている [22] [23] [24] [25].

3. 関連研究

3.1 On-demand-fork()

3.1.1 概要

On-demand-fork() [19] とは，使用メモリ量が多い場合に fork() のレイテンシが無視できないほど大きくなる問題を解決するための手法である．fork() は，Redis によるスナップショットの作成，ベンチマークテスト時の入力の共有，VM クローニングといったアプリケーションに利用されるが，これらはメモリ負荷が高くなる場合がある．しかし，fork() の実行時間は主にページテーブルのコピーが大半を占めるため，使用メモリ量が多くページテーブル量も多い場合には実行時のレイテンシが大きくなるという問題がある．この問題に対応するための既存手法としては Hugepage の利用が挙げられる．ページテーブルの中で最も数が多い PT を省略することで大幅に fork() の実行時間を削減することが可能である．しかし問題点があり， Hugepage を利用することによって内部フラグメンテーションの増加，CoW による書き込み時のレイテンシの急上昇が発生する．On-demand-fork() ではこれらを解決するため， fork() 実行時にページだけではなく PT も親子プロセス間で共有する．つまり， Hugepage と同じく fork() 実行時に PGD，PUD，PMD のみをコピーする．PT のコピーは 所属する PTE のうちどれか一つが更新された場合に実行される．

Hugepage を利用せずに PT のコピーを省略することが可能であるため， fork() 実行時間の短縮と書き込み時のレイテンシの抑制を両立することが可能となる．

3.1.2 問題点

この手法の問題点は， アドレス変換によるレイテンシが改善できないことである．メモリ負荷が高いアプリケーションでは，ページテーブルの増加によりアドレス変換と fork() のレイテンシが大きくなる．On-demand-fork() では fork() によるレイテンシを小さくすることが可能だが，Hugepage を利用しないためアドレス変換自体は高速化されない．また，この手法を Hugepage に対して適用した場合は， Hugepage の利用による問題が発生してしまう．具体的には内部フラグメンテーションの増加によるメモリ圧迫や Hugepage の CoW によるメモリの連続領域の分散である．

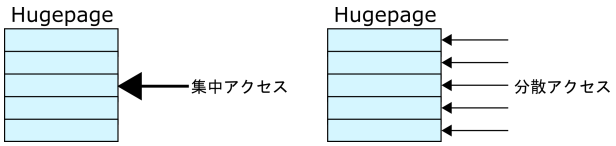

図 5: Hugepage へのアクセスパターンの違い

3.2 HawkEye

3.2.1 概要

HawkEye [1] は， Hugepage を内部フラグメンテーションの発生を抑制したまま利用し，メモリ圧迫の抑制とアドレス変換の高速化を両立するための手法である．Hugepage を割り当てると内部フラグメンテーションが発生するが，アドレス変換速度は向上する．一方 Hugepage を分解すると，内部フラグメンテーションは解消されるがアドレス変換速度は低下する．このようにメモリ使用効率とメモリアドレス変換速度はトレードオフの関係になっており，既存の Hugepage を活用するシステムではトレードオフに関連する問題がある．たとえば FreeBSD [26] では，内部フラグメンテーションの増加を抑えているがアドレス変換速度の向上が小さい．ingenes [21] では，メモリの圧迫具合に応じて Hugepage の利用を制限することでトレードオフの改善を行っているが， Hugepage の割り当て方式が最適ではないという問題がある [1]．一方 HawkEye では，Hugepage 内の regular page 領域それぞれのアクセス頻度の違いに注目することでトレードオフを改善している．図 5 右のように， regular page 領域に満遍なくアクセスが発生している場合は， Hugepage に統合すると TLB ヒット率が良くなるため Hugepage を積極的に割り当てる．メモリの圧迫が進んだら，図 5 左のように，一部の領域にアクセスが集中しており分解しても TLB ヒット率の減少が小さい部分から順に Hugepage を分解しメモリ回復を行う．

HawkEye によって Hugepage の効率的な割り当てと分解が行われることで，アドレス変換の高速化と内部フラグメンテーションによるメモリ圧迫からの回復を両立することが可能となる．

3.2.2 問題点

この手法の問題点は， Hugepage を fork() した場合に生じる問題に対応していないことである．HawkEye は効率的な Hugepage の割り当てを行うが， fork() 後の CoW による Hugepage の分解が発生すると書き込みを行ったプロセスで元の領域の Hugepage への再割り当てが行えなくなる．前述したように元の Hugepage として確保されていたメモリが物理的に連続しなくなるからである．特に子プロセスの実行が一時的である Redis のスナップショット作成のような場合，親子プロセス間でページを共有している間に親プロセスで書き込みが多発すると子プロセス終了後の親プロセスでは Hugepage への統合がほとんど行われないことも考えられる．

3.3 Quicksilver

3.3.1 概要

Quicksilver [13] とは，既存の Hugepage 利用手法を分析しその結果を基に提案された手法である．Hugepage を透過的に管理する手法は異なる OS において複数提案されているが，それら既存手法を比較検討するための枠組みが存在しない．そのため，OS に依存せずに既存手法を比較することができる機構を実装し，それから得られた知見によって提案されたのが Quicksilver である．Llinux の THP，Ingens，HawkEye と FreeBSD を分析・比較した結果，Hugepage においては FreeBSD が採用する予約ベースの割り当て [27]，THP が採用する積極的な割り当て，実使用量の少ないページ [28] のディアロケートが有効であることが分かった [29].

Quicksilver ではそれらを FreeBSD 上に実装することで，アドレス変換速度，断片化への耐性の向上，メモリ断片化の抑制を達成した．

3.3.2 問題点

この手法の問題点は，Hugepage に対する CoW 発生時，1 ページでも CoW が発生すると連続領域が物理的に分解されるため Hugepage の恩恵が受けられなくなることである．また，再び Hugepage として利用するためにはマイグレーションのコストが発生する．頻繁に書き込みが行われる Hugepage 領域がある場合，fork() を実行すると高確率でページの分解が行われる．そのような領域に Hugepage を割り当てるのが効率的だが，本手法ではそのためのコストが発生する．

4. 提案

本研究では fork() 後の共有されたページに書き込みを行う場合， CoW 発生時にメモリの連続領域を維持するプロセスを選択することができるメモリ管理機構を提案する．これにより，ページの共有終了後に Hugepage を利用したいプロセスで再統合を行うことができる．

既存手法では，親子プロセスのうち書き込みを行った側がコピー先ページを参照するように実装されておりメモリの連続領域を維持するプロセスを選択することができない．提案手法では，書き込みを行った側ではないプロセスへのページコピーを行う機構をカーネルに追加し，メモリの連続領域を自動的に維持する．ユーザ空間から従来の手法と提案手法のどちらを利用するのか選択可能にすることで選択的なページコピーを実現する．

4.1 想定する CoW 発生状況

本研究では fork() によって共有された Hugepage へ書き込みを行う状況を想定する．このとき Hugepage は regular page に分解されてからページコピーが行われるが，書き込みを行った側でないプロセスにコピーされたページを参照

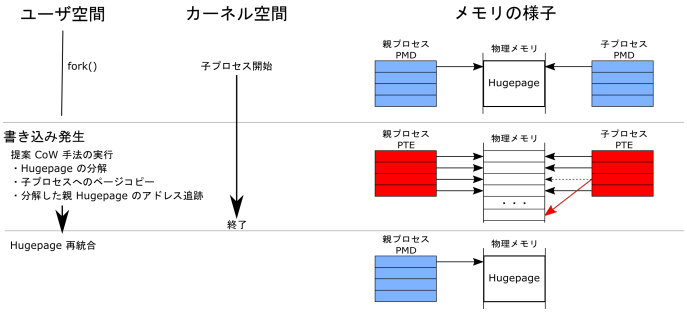

図 6: 提案方式の処理の流れとメモリの様子

させる機構を作成する．なお子プロセスが二つ以上発生する場合は想定しない．

4.2 アプローチ

以上を実現するために，ページを共有している他のプロセスにページコピーを行う CoW 機構を用意し， Hugepage への書き込みを行う場合に自動的に実行する方式を提案する．提案方式を用いる例として，親プロセスの Hugepege へ書き込みを行うときの処理の流れとメモリの様子を図 6 で示す．親プロセスで書き込みを行う場合，提案した CoW 機構が実行されコピーしたページは子プロセスのページテーブルで参照される．子プロセスで書き込みを行う場合は従来の CoW 方式を利用する．これによって親プロセスでは元の Hugepage 領域が連続に保たれるため，ページ共有終了後に同じ領域を Hugepage へ再統合するができる．

提案方式が有効な例として， fork() を実行し，複数ページにまたがるシーケンシャルな書き込みを繰り返すような場合があげられる．この例で従来手法を用いると物理メモリが離散してしまうが，代わりに提案方式を用いることで物理メモリの連続領域が保たれ， Hugepage による恩恵を受けることが可能となる．提案手法ではメモリの連続領域を保つ部分，保たない部分を選択することができる．よって上記の例のように書き込みが偏っていない部分には提案方式を用いないことで，内部フラグメンテーションの増加を抑えることが可能であると考えられる．

また，子プロセスが定期的に実行されるような場合にも有効であると考えられる．実際のアプリケーションでは，Redis のスナップショット作成が該当する．これについては次章のケーススタディにて説明をする．

5. ケーススタディ：Redis

本章では Redis についての基本的な情報を示してから提案手法の設計，実装について述べる．最後に提案手法の評価のための実験について述べる．

5.1 Redis について

Redis とは高速な key-value インメモリデータベースである．サーバはデータをメモリ内に配置することで高速なデータアクセスを可能にしており，クライアントからの要求に対して高速に応答を行う．また，Redis は基本的にシングルスレッドで動作するという特徴を持つが，スナップショットを作成する場合には fork() によってプロセスを複製し，親プロセスでのイベントループと子プロセスでのデータ永続化を並行して実行する．

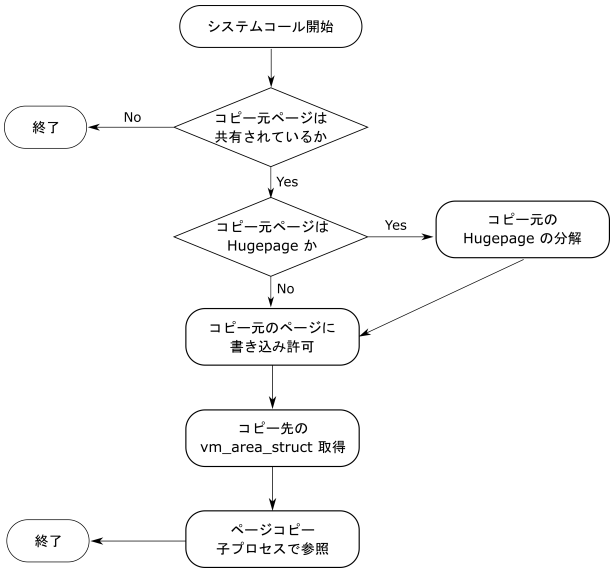

図 7: 提案手法に用いるシステムコールのフローチャート

5.2 スナップショット

本節では Redis におけるスナップショット作成 [30] の特性について述べる．Redis においてスナップショットを作成する場合， fork() によってプロセスを複製し子プロセスでデータの永続化を実行する．親プロセスでは fork() 実行前の処理に戻り，子プロセスと並行して処理が進められる．

そのため，スナップショット作成中に親プロセスにおいてデータの更新が発生すると CoW が発生する．もし CoW が Hugepage に対して発生した場合，親プロセスのメモリの連続領域が維持されずに分解される．その結果，子プロセス終了後にメモリの走査とコンパクションによる性能の劣化や再統合の遅延が発生するという問題がある [22]．スナップショットを定期的に取得する場合，そのたびにレイテンシが増加する可能性がある．

5.3 設計

5.3.1 メモリの連続領域の維持

4.2 節で述べたように，ページを共有している他のプロセスにコピーしたページを参照させるシステムをカーネルに作成し，自動的に提案手法によるページコピーを行う．このシステムについて説明するために，提案手法によるページコピーシステムの処理のフローチャートを図 7 で示す．

ページフォールト発生時，該当ページが複数プロセス間で共有されている場合にのみ提案した CoW を実行する．

ページフォールトは共有されたページへの書き込み以外にもスワップアウトされたページやマッピングされていないページへの書き込みなどによっても発生するが，それらの場合には CoW が発生しないためである．

また，regular page 単位で CoW を実行するため，コピー元が Hugepage である場合は分解してから書き込み許可を与える．これによって，再び同じページに書き込みを行った場合にページフォールトが発生するのを防ぐ．fork() 実行直後は CoW を実行するためにすべてのページが書き込み不可になっている．ページを共有しているプロセスでは 5.4 節の処理により書き込み許可が与えられているが，このままではシステムコール呼び出し元プロセスの該当ページに書き込み許可が与えられていないままであるため，実際に書き込む場合に page fault が発生してしまう．それにより再びページコピーが行われるのを防ぐため，PTE の Write/Read ビットを立てて書き込み許可を与える．

さらに，後の処理に必要な vm_area_struct を取得してからページコピーを行う．提案手法ではページを共有している他のプロセスにコピーしたページを参照させる．これを実現するために，ページを共有するプロセスへ書き込みを行った場合のページフォールト時の処理を疑似的に再現する．ここでのページフォールト時の処理とは，Hugepage の分解や分解後のページのコピーである．fork() によって共有されたページに書き込むとページフォールトが発生し，ページフォールトハンドラによって Hugepage の分解，ページのコピー，書き込み許可等の付与が行われる．この処理を，システムコール呼び出し元で，ページを共有するプロセスに対して実行する．ここで必要になるのが vm_area_struct である．これによって，指定したアドレスが含まれる Hugepage を分解し，コピーした regular page を子プロセスのページテーブルに参照させる．また，対象のページが Hugepage ではない場合は分解処理を飛ばして同じ処理を行う．Redis においてこれらの処理を親プロセスで実行することによって，スナップショット作成中に書き込みが発生した場合でもメモリの連続領域が維持されると考えられる．

5.3.2 Hugepage 分解処理のトラップ

Hugepage への再統合を即座に行うために，物理的に連続かつ分解された Hugepage を追跡する．その領域は提案手法において Hugepage に対し CoW を行った場合に発生するが，Hugepage サイズの物理領域が連続しているためマイグレーションを行わずに Hugepage として再統合が可能である．図 6 の親プロセス PTE が示す領域である．ページの共有が終了した際に追跡したアドレスの各領域を Hugepage に戻すことで，低コストで Hugepage を再利用することが可能となる．

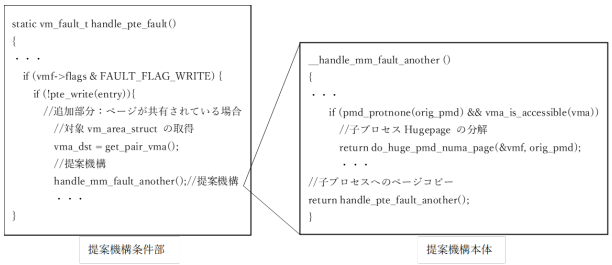

図 8: 連続領域維持機構実行までの流れ

5.4 実装

本章では提案手法の実装について述べる．実装は Linux 5.10.25，x86-64 アーキテクチャに行った．

5.4.1 ページフォールトによる CoW 先の変更

5.3.1 項で説明した，親プロセスの連続領域を維持する機構について説明する．2.4.2 項で述べたように，Linux 5.10.25 では Hugepage に対する CoW が発生した場合，それを分解した後に regular page 単位で CoW を実行する．提案手法では Hugepage が分解された直後に条件文を追加し，該当ページが複数プロセス間で共有されている場合は本機構を実行する．本機構が実行されるまでの流れを図 8 で示す．ページフォールトハンドラの handle_pte_fault() 関数において Hugepage が分解された後に本機構の関数 __handle_mm_fault_another() を呼ぶ．この関数では，従来の CoW 手法においてページフォールトが子プロセスで発生したのとほぼ同様の処理を実行する．異なるのは親プロセスのページにも書き込み許可を与えるという点である．

実行が終了すると，親プロセスでは連続領域が維持，子プロセスでは分離した状態となり，また両プロセスにおいて regular pages によるマッピングが行われている状態となる．その結果，親プロセスではページの共有が解除されるとマイグレーションを行わずに Hugepage として再利用可能となる．

5.4.2 Hugepage 分解処理のトラップ

5.4.1 で述べたように Hugepage と同サイズの物理領域が連続していて regular pages でマッピングされている領域は低コストで Hugepage として利用できる．しかし，既存手法では提案手法のような CoW を行わないためそれらの領域を追跡する機構が存在しない．提案手法ではその追跡のために Hugepgae を分解する処理をトラップし，該当する仮想アドレスを構造体のリストで管理する機構を作成した．構造体をコード 1 に示す．

Listing 1: struct splitted thp

```
1 struct splitted_thp{
2         unsigned long haddr;
3         struct list_head list;
4 };
```

構造体リストの先頭は mm_struct が保持する．

5.4.3 ページ共有解除に伴う Hugepage への再統合

コード 1 の構造体で追跡される領域はプロセス間の共有が終了した場合即座に Hugepage へ統合することが可能だが，既存の Linux にはそのような領域であってもマイグレーションを行い新たな Hugepage 領域を確保してしまう．そこで，提案手法ではマイグレーションを行わずにそのままの領域を Hugepage として再利用する処理を行う．ページの共有が終了した場合，コード 1 の構造体リストを参照し，該当する領域を Hugepage に統合する．

5.5 実験

5.5.1 実験環境

実験に使用するマシンの構成を表 1 に示す．OS カーネルには提案手法が実装された Linux 5.10.25， Linux ディストーションには Ubuntu 20.04 LTS，x86-64 アーキテクチャを使用した．

表 1: 実験マシン構成

構成	内容
CPU	Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz
メモリ	256GB

5.5.2 目的

提案手法により CoW 発生時にメモリの連続領域が維持されること，ページの共有終了時に Hugepag への再統合が行われることを確認する．Redis においてスナップショット作成時に書き込みが発生した状況を再現するために，子プロセスを生成し，親プロセスにおいて書き込みを行うことで CoW を発生させ，ページの共有終了後に Hugepage の再統合を行う．

5.5.3 実験内容

提案手法を用いて，fork() によって共有された Hugepage に対し親プロセスから書き込みを行い，連続領域の維持と Hugepage への再統合が実行されるか確認する．図 9 で示すように書き込みは regular page 単位で 4 ページごとに行う．その後，各プロセスで 512 の PTE 内容を表示し，親プロセスで連続領域が保持されていることを確認する．さらに子プロセス終了直後， pmd エントリの内容を表示し Hugepage への統合が成功しているか確認する．pte または pmd エントリの内容は下位 12 ビットがフラグ，それよりも上位のビットが物理ページフレームを表すため，後者が連続していれば連続領域が保持されていることが確認できる．16 進数では 下位 3 桁がフラグを示す．

5.5.4 実験結果

実験結果を図 10 で示す．親プロセスでは共有 Hugepage への書き込み後も連続領域が維持され，子プロセスでは親プロセスで書き込まれたページが別の物理ページを参照していることが分かる．従って，提案機構により CoW が書き込みを行った親プロセスではなく子プロセスで発生したことが確認できる．

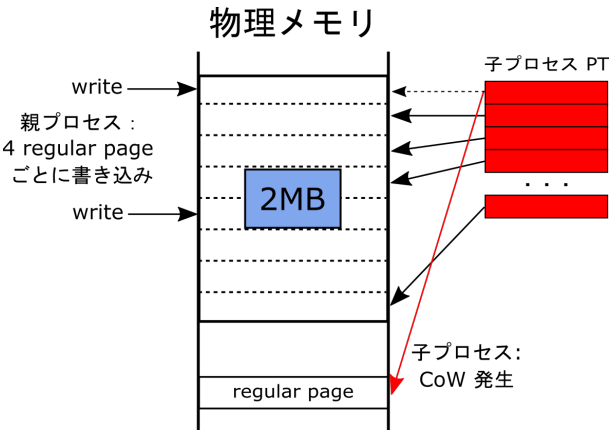

図 9: 実験における書き込み方法

```
parent
huge[4096*  0] *pte = 0x8000000136a00 867
huge[4096*  1] *pte = 0x8000000136a01 825
huge[4096*  2] *pte = 0x8000000136a02 825
huge[4096*  3] *pte = 0x8000000136a03 825
 . . .
huge[4096*508] *pte = 0x8000000136bfc 867
huge[4096*509] *pte = 0x8000000136bfd 825
huge[4096*510] *pte = 0x8000000136bfe 825
huge[4096*511] *pte = 0x8000000136bff 825

child
huge[4096*  0] *pte = 0x8000000144893 827
huge[4096*  1] *pte = 0x8000000136a01 805
huge[4096*  2] *pte = 0x8000000136a02 805
huge[4096*  3] *pte = 0x8000000136a03 805
 . . .
huge[4096*508] *pte = 0x80000001384b4 827
huge[4096*509] *pte = 0x8000000136bfd 805
huge[4096*510] *pte = 0x8000000136bfe 805
huge[4096*511] *pte = 0x8000000136bff 805

parent(after child process exits)
huge[4096*  0] *pmd = 0x8000000136a00 8e7
 . . .
huge[4096*511] *pmd = 0x8000000136a00 8e7
```

図 10: 各プロセスのページテーブルエントリ内容

また，子プロセス終了後には親プロセスで pmd エントリの内容が表示され，Hugepage フレームを指し示した．これらの結果より，提案機構による連続領域の維持と Hugepage への再統合が即座に行われたことが確認された．

6. おわりに

本研究では，CoW 発生時にメモリの連続領域を維持するプロセスを選択可能にし，ページ共有終了後即座に

Hugepage に統合する機構を提案した．CoW によりコピーしたページを子または親プロセスに参照させる方式を実装することで提案手法を実現した．従来手法では， fork() により共有された Hugepage へ書き込みを行うとその領域が物理的に連続ではなくなるが，提案方式を用いるとメモリの連続領域を保ったまま書き込みを行うことができる．これにより，マイグレーションを必要としない Hugepage への再統合やメモリコンパクション時間の短縮が可能になると考えられる．

今後の課題としては，実際の Redis におけるベンチマークによる評価，三つ以上のプロセスでページが共有されている場合への対応，Hugepage 割り当ての最適化が挙げられる．Redis ベンチマークによる評価は， Hugepage による性能向上が得られるものに対してスナップショットを作成しながら実行し， THP と提案手法間で性能や Hugepgae 割り当て数を比較する方法が考えられる．

参考文献

[1] Panwar, A., Bansal, S. and Gopinath, K.: HawkEye: Efficient Fine-grained OS Support for Huge Pages, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, New York, NY, USA, pp. 347–360 (online), DOI: 10.1145/3297858.3304064 (2019).

[2] Basu, A., Gandhi, J., Chang, J. et al.: Efficient virtual memory for big memory servers, *Proceedings of the 40th Annual International Symposium on Computer Architecture*, New York, NY, USA, pp. 237–248 (2013).

[3] Panwar, A., Prasad, A. and Gopinath, K.: Making Huge Pages Actually Useful, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, New York, NY, USA, pp. 679–692 (online), DOI: 10.1145/3296957.3173203 (2018).

[4] Ryoo, J. H., Gulur, N., Song, S. et al.: Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB, *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, New York, NY, USA, pp. 469–480 (online), DOI: 10.1145/3079856.3080210 (2017).

[5] Pham, B., Vaidyanathan, V., Jaleel, A. and Bhattacharjee, A.: CoLT: Coalesced Large-Reach TLBs, *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 258–269 (online), DOI: 10.1109/MICRO.2012.32 (2012).

[6] Intel: Intel 64 and IA-32 architectures software developer's manual volume 3A: System programming guide, part 1 (2020).

[7] Pham, B., Veselý, J., Loh, G. H. and Bhattacharjee, A.: Large pages and lightweight memory management in virtualized environments: can you have it both ways?, *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, New York, NY, USA, pp. 1–12 (online), DOI: 10.1145/2830772.2830773 (2015).

[8] RedHat: 5.2. Huge Pages and Transparent Huge Pages Red Hat Enterprise Linux 6, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-transhuge (2020).

[9] kernel development community, T.: HugeTLBpage on ARM64 — The Linux Kernel documentation, https://www.kernel.org/doc/html/latest/arm64/hugetlbpage.html (2021).

[10] Documents, L. K.: Transparent Hugepage Suppor, https://www.kernel.org/doc/Documentation/vm/transhuge.txt. (accessed Janualy, 2022).

[11] Redis: Redis, https://redis.io/. (accessed Janualy, 2022).

[12] Corbet, J.: Memory compaction, https://lwn.net/Articles/368869/. (accessed Janualy, 2022).

[13] Zhu, W., Cox, A. L. and Rixner, S.: A Comprehensive Analysis of Superpage Management Mechanisms and Policies, *Proceedings of the 2020 USENIX Annual Technical Conference*, USENIX ATC 20, pp. 829–842 (online), available from ⟨https://www.usenix.org/conference/atc20/presentation/zhu-weixi⟩ (2020).

[14] cloudera: Cloudera recommends turning off memory compaction due to high CPU utilization, https://docs.cloudera.com/documentation/enterprise/latest/topics/cdh_admin_performance.html. (accessed Janualy, 2022).

[15] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y. and Brooks, D.: Surpassing the TLB Performance of Superpages with Less Operating System Support, *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, p. 158–169 (online), DOI: 10.1145/2749469.2750392 (2015).

[16] Hunter, A., Kennelly, C., Turner, P., Gove, D., Moseley, T. and Ranganathan, P.: Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator, *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 21, pp. 257–273 (online), available from ⟨https://www.usenix.org/conference/osdi21/presentation/hunter⟩ (2021).

[17] Intel: 5-Level Paging and 5-Level EPT (2017).

[18] : Intel Skylake, https://www.7-cpu.com/cpu/Skylake.html.

[19] Zhao, K., Gong, S. and Fonseca, P.: On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications, *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, New York, NY, USA, pp. 540–555 (2021).

[20] Li, X., Liu, L., Yang, S. et al.: Thinking about A New Mechanism for Huge Page Management, *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, New York, NY, USA, pp. 40–46 (online), DOI: 10.1145/3343737.3343745 (2019).

[21] Kwon, Y., Yu, H., Peter, S. et al.: Coordinated and Efficient Huge Page Management with Ingens, *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI 16, pp. 705–721 (online), DOI: 10.5555/3026877.3026931 (2016).

[22] Redis: Redis latency problems troubleshooting, http://redis.io/topics/latency. (accessed Janualy, 2022).

[23] MongoDB: Disable Transparent Huge Pages (THP) - MongoDB Manual, https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/ (2020).

[24] VoltDB: Recommendation to disable huge pages for VoltDB, https://docs.voltdb.com/AdminGuide/adminmemmgt.php (2019).

[25] Couchbase: Disabling Transparent Huge Pages (THP), https://docs.couchbase.com/server/current/install/thp-disable.html (2020).

[26] Navarro, J., Iyer, S., Druschel, P. and Cox, A.: Practical, Transparent Operating System Support for Superpages, *SIGOPS Oper. Syst. Rev.*, Vol. 36, No. S (online), DOI: 10.1145/844128.844138 (2003).
[27] Talluri, M. and Hill, M. D.: Surpassing the TLB Performance of Superpages with Less Operating System Support, *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, p. 171–182 (online), DOI: 10.1145/195473.195531 (1994).
[28] Zhu, W.: Exploring superpage promotion policies for efficient address translation, Master ' s thesis, Rice University, 6100 Main St, Houston, TX 77005 (2019).
[29] Navarro, J., Iyer, S. and Cox, A.: Practical, Transparent Operating System Support for Superpages, *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI 02, Boston, MA, (online), available from ⟨https://www.usenix.org/conference/osdi-02/practical-transparent-operating-system-support-superpages⟩ (2002).
[30] Redis: Redis persistence, `https://redis.io/docs/manual/persistence/`. (accessed April, 2022).