メモリエラーへの耐性を有するオペレーティングシステム

井口 卓海[1]　山田 浩史[1]

概要：メモリにおける故障 (メモリエラー) はオペレーティングシステムをはじめとするシステム全体に致命的な影響を及ぼす．信頼性を高めるため，Ecc メモリを始めとする高信頼メモリと，Non-Ecc メモリと呼ばれる低信頼メモリを組み合わせるヘテロジニアスメモリシステムを構築することが可能である．しかし，現行の OS ではヘテロジニアスな構成に適応したデータの配置ができないことに加え，一部のメモリ上でメモリエラーが検知されると，その他のメモリの健全性を考慮せずに即座にシステム全体がクラッシュしてしまう．このため，ヘテロジニアスメモリシステムにおいて信頼性・可用性を向上させるためには，メモリエラーが発生しても即座にシステムをダウンさせず，動作を継続させるための仕組みが必要である．本研究では，メモリエラーを検知した際に破損箇所のデータを回復することでシステムを継続動作させる仕組みを提案する．システムの機能を管理するメモリオブジェクトと呼ばれるデータ構造のアドレスを保持しておき，メモリエラーが検知された場合にはそのリストから破損したオブジェクトを特定，リカバリ処理を施すことで回復を試みる．本研究では xv6 に対して実装を行い，フォールトインジェクションにより評価実験を実施した．実験では，メモリエラーが検知された状況を想定したクラッシュシナリオに基づきエラーを挿入し，リカバリを行った．実験の結果，提案手法によりメモリエラーが発生してもシステムがダウンすることなく継続動作することを確認した．

キーワード：オペレーティングシステム，ヘテロジニアスメモリシステム

1. はじめに

メモリにおける故障 (メモリエラー) が発生してしまうと，オペレーティングシステム (OS) をはじめとするシステム全体に致命的な影響を及ぼす．なぜなら，メモリにはアプリケーションデータのようなユーザが管理するデータだけではなく，ページテーブルなどのシステムの動作に関わる重要なデータ構造が配置されているためである．たとえば，UNIX 系 OS の一種である Linux においてメモリエラーが検知されると，kernel panic という安全策によってシステム全体が強制的に停止され，再起動せざるを得ない状況になり，データの損失や意図しない挙動といった悪影響を被る危険がある．メモリの中には，誤り訂正符号 (Error-Correcting Code:Ecc) [?] と呼ばれるメモリエラーの検知・訂正能力を持つ信頼性の高いメモリも存在するものの，その検出・訂正能力には方式ごとに限界があり，訂正可能なメモリエラーもソフト的なエラーに限られている．

また，異なる物性のメモリを組み合わせてメモリシステムを構築するヘテロジニアスメモリシステムが開発されている．このヘテロジニアスメモリシステムでは，電力供給が断たれてもデータを失わない不揮発性メモリ [] やメモリを 3 次元に積み重ねる 3D-stacked メモリ [] が登場している．これらのメモリでは，帯域幅やデータ I/O 速度などの性能，容量，電力消費量は一長一短の関係にあり，互いに組み合わせることで高性能・大容量 []，高性能・低消費電力量 [] を両立するシステムを構築できる．このように，信頼性の異なるメモリを組み合わせたヘテロジニアスメモリシステムを構築することが可能になっている．しかし，既存の OS ではメモリシステムが物性の均一なメモリから成るという前提のもとデータを配置するため，データの重要度に応じて信頼性の異なるメモリに適したデータの配置を行うことができず，ヘテロジニアスメモリシステムに対応できないという問題がある．また，高信頼メモリはメモリエラーの検出・訂正機能をモジュールにより付与するため，容量が DRAM のような Non-Ecc メモリに比べて少ない傾向にある．さらに，現行の OS ではメモリエラーが検出されると，その周囲のメモリの健全性に関わらず，前述の kernel panic をはじめとする安全策によってシステムダウンするように設計されており，メモリ全体における一部分のメモリエラーが致命的なクラッシュに直結してしまう．これらの点から，システム全体の信頼性・可用性を高めるには，データの重要度や破損時にシステムに与える影

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

響に応じて高信頼性メモリに置くデータを選別し，かつメモリエラーが発生しても即座にシステムを継続的に動作させるための仕組みが必要となる．

本研究では，高信頼性メモリ上で訂正できないメモリエラーが発生した場合でもシステム全体のダウンを回避して継続的な動作を可能にする手法を提案する．メモリエラーによってシステムがクラッシュしてしまう場合であっても，その周囲の大部分のメモリは健全であることに注目し，メモリエラーによって破損したデータをリカバリすることでシステム全体のクラッシュを回避する．Ecc メモリによって訂正不可能なメモリエラーが検出されると，メモリオブジェクトと呼ばれる，メモリ上にて役割ごとにデータを管理する構造体の粒度で破損箇所を特定，リカバリハンドラを呼び出すことで回復を実施する．リカバリハンドラでは，破損したメモリオブジェクトの中で無事なデータを退避させ，破損箇所に対応するデータを初期化した後，他データ構造との整合性を保持する．これにより，物理メモリ上でメモリエラーを抱えた状態でも kernel panic などを引き起こすことなくシステムを正常な状態で継続動作することを可能にする．

本論文における貢献を以下に示す．

- Ecc のような高信頼性メモリでも修正不能なメモリエラーに対して耐性を持つ OS を提案した．OS 自体にメモリエラーへの耐性を持たせることで，メモリと合わせてさらなる可用性の向上と，低信頼性のメモリを使用した場合においてもメモリエラーによる悪影響を軽減できると考えられる (3 章)．
- 提案手法を実現するため，OS のデータをメモリオブジェクトの単位でセマンティクス・使用される状況・データ構造間の関係性・回復可能性を明らかにし，設計を行った．メモリ側から故障が発覚したアドレスが渡されるという仮定の下，アドレスから故障したメモリオブジェクトを探索，合致したリカバリハンドラを呼び出すように実装した (4 章)．
- MIT の開発した小規模な教育用 OS である xv6 に対して実装を行った．想定したクラッシュシナリオ上において，システム全体がダウンすることなく継続して動作することを確認した (5，6 章)．

2. 背景

2.1 メモリエラー

メモリにおける故障 (メモリエラー) は，メモリ上のビットの反転や物理的破損を含む様々な原因によってメモリに保管されているデータが正しく読み取ることができない状態を指す．これらの故障はメモリ上で発生した時点で即座に発覚するわけではなく，メモリエラーが発生してからその箇所のデータにアクセスを試みた時点で初めて悪影響が発生する，もしくは Ecc メモリによって検出されることによって発覚する．

メモリエラーはその発生原因からソフトエラーとハードエラーの 2 種類に大別される．ソフトエラーは電磁気的な要因もしくは宇宙線による外的要因によるものを指し，1 から数ビットの規模でビット反転が生じる．ソフトエラーが発生した場合には，後述する Ecc 機能を有したメモリによって訂正，もしくは検出を行うことが可能な場合があるものの，メモリが持つ訂正・検出機能を上回る規模のエラーであったり，Ecc 機能を持たない Non-Ecc メモリが搭載されていた場合には，kernel panic などによるシステムダウンを引き起こす原因となる可能性がある．その一方でハードエラーは，メモリエラーのうち物的破損が原因であるエラーを指す．ハードエラーはメモリユニットそのものが物理的に破損しているため，データの読み書きそのものが不能に成る．このため，ハードエラーではソフトエラーと異なり，故障箇所において繰り返し永続的にエラーが発生し続け，かつメモリエラーの及ぶ範囲がソフトエラーに比べて広範囲に成る場合が多いという特徴がある．

様々な実環境下で生じるメモリエラーを観測した Schroeder らの調査によると，これらのメモリエラーの発生状況は以下のようにまとめられている．

- ソフトエラーよりもハードエラーの方がより多く発生する
- 商用のデータセンタでは，1 年に 8%以上のメモリに部分的な故障が発生する
- 1 年間で実験対象のマシンの約 3 分の 1 にメモリエラーが少なくとも 1 つ発生する
- 1 年間で訂正できないメモリエラーは全マシンのうち 1.3%に発生する

これらのメモリエラーのうち，ソフトエラーに対して訂正・検出を行うメモリ保護機能を有する信頼性の高いメモリも存在する．その訂正・検出方法にもいくつかの種類があり，その訂正・検出可能なメモリエラーの規模，メモリデバイスにより必要となる追加のメモリ容量は保護手法により異なる．

1 つ目の例として挙げる Mirroring [?] は，メモリ上のデータを複数保持しておくメモリ保護手法である．ミラーリングしている一方のデータに対してメモリエラーによるデータの損失が生じたとしても，もう一方の健全なデータから復元することが可能である．しかし，ひとつのデータを二重に保存することになるため，保護したいデータに対して追加で 125%分のメモリ容量が必要となる [?]．

2 つ目の例はパリティ付き DRAM である．これは，保護したいデータに対して余分なビットにパリティ符号を付与しておき，メモリ上のデータとパリティビットの間でパリティチェックを実施することで，ソフトエラーに起因する誤りを検知することが可能なメモリである．メモリの誤り検出能力は 1 ビットの誤りであり，メモリ単体での訂正

はできない．

次に紹介するのは，Error-Checking Code(Ecc) メモリである．これは Ecc 機能を持つモジュールを付加したメモリであり，余分なビットに誤り訂正符号と呼ばれる検証用のデータを保持しておき，保護対象のデータとすり合わせることで誤りの検出を行う．誤り訂正符号としてよく用いられるのはハミング符号であり，ソフトエラーに起因する 1 ビットの誤り訂正・2 ビットの誤り検出能力を有する．

また，IBM が開発した，Ecc よりも更に信頼性を高めた誤り訂正・検出機能である Chipkill [?] を搭載したより信頼性の高い Chipkill メモリも存在する．これは，メモリチップ 1 つ分のデータ破損及び単一のメモリチップ内の複数ビットの誤りを訂正する能力を持つ．　最後に紹介するのは Non-Ecc メモリである．Ecc メモリを代表とする誤り訂正・検出機能を持つメモリに対して，一般的な DRAM などを含むメモリ保護機能を持たないメモリ全般を指す．Ecc メモリであれば訂正可能なビット反転であっても，そのまま直接メモリエラーとしてシステム全体のクラッシュを引き起こしてしまうことから，誤り訂正・検出機能を持つメモリよりも信頼性が低い傾向にある．その一方で，メモリエラー訂正・検出のためのメモリデバイスにメモリ容量を割く必要がないため，高信頼性のメモリよりも容量が多い傾向にある．

2.2 ヘテロジニアスメモリシステム

CPU や GPU，メモリなどのリソースの中で，特性の異なるものを組み合わせてシステムを構築する技術であるヘテロジニアスコンピューティングに関する研究が盛んに行われている．たとえば，異なる性能・消費電力の CPU を組み合わせたアーキテクチャを構築することにより，システムの消費電力低下を実現した例 [?] やヘテロジニアスなりソースを認識するスケジューラによって，データセンタにおけるアプリケーションの性能向上と Quality of Service(QoS) の維持を実現した例 [?] が存在する．

特にメモリにおいて，帯域幅やレイテンシのような性能面，メモリエラーへの耐性のような信頼性の面，消費電力量などの物性の異なるメモリを組み合わせることで構築したメモリシステムをヘテロジニアスメモリシステムと呼ぶ．しかし，既存の OS や Virtual Machine Monitor(VMM) では，単一の物理メモリから構成される従来のホモジニアスメモリシステムが前提となっており，メモリに対して適切なデータを配置する制御機能が欠如している [?, ?]．そのため，単に種類の異なる物理メモリを組み合わせてシステムを構築するだけでは，OS 側においてその構成を認識するできないために，構築時の狙いに沿ったデータの配置を行うことができない．それゆえ，現行の OS を用いてヘテロジニアスメモリシステムを利用するためには，サーバアーキテクチャの 1 つである Non-Uniform Memory Access(NUMA) [?] のような，既存の物理リソース管理手法を改良することによる対策，もしくは VMM においてヘテロジニアスな構成を認識し，VM 上で稼働する OS から見てヘテロジニアスな構成が透過的に扱われる方法などによる対策を打つ必要がある．

2.3 関連研究

ヘテロジニアスメモリシステムに関する研究は，システム全体の信頼性を高める目的とシステムの性能を高める目的の 2 つに大別される．

Characterizing Application Memory Error Vulnerablity [?] では，大規模なデータセンタにおける物理メモリの金銭的コストと高い信頼性の維持を両立するヘテロジニアスメモリシステムの構築と目的としてメモリエラー耐性に関する safe ratio と recoverability という新しい指標を提案し，それをもとに実際にヘテロジニアスメモリシステムを構築・分析を実施している．この評価指標を用いてデータのメモリエラーへの耐性を評価することにより，耐性の有無に応じて配置するメモリの信頼性を変えることで，ヘテロジニアスメモリシステムを活用するシステムを実現している．

HeteroVisor [?] では，ヘテロジニアスな CPU とメモリの構成を用いることで，クラウドコンピューティングサービスにおけるゲスト Virtual Machine(VM) への割り当てリソースを調整する粒度を高めることを目的としたハイパーバイザを提案している．Amazon Elastic Computing Cloud(AmazonEC2) [?] をはじめとする現在のクラウドコンピューティングでは，ゲストに割り当てるリソース量の調整は VM インスタンスという粗い単位で行われていた．本手法では E-state と呼ばれる新しい概念を導入し，ゲスト VM からの CPU・メモリリソースの要求度合いを HeteroVisor 側に伝達し，それをもとにシステム全体の性能を左右するデータである Hot Page を高速/低速メモリ間でマイグレーションすることでリソース割り当ての調整を行う．これにより，ヘテロジニアスなリソースを段階的に割り当てることでヘテロジニアスプラットフォームを活用し，高い性能と QoS の維持の両立を実現している．

HeteroOS [?] は，HeteroVisor と同様にヘテロジニアスなプラットフォームを用いてゲスト VM へのリソーススケーリングの粒度を高めることを目的とし，ハイパーバイザだけではなくゲスト VM 上で稼働する OS(以下ゲスト OS と呼ぶ) にもヘテロジニアスなメモリ構成を認識させることで性能向上を実現する手法である．これにより，Hot Page にまつわる情報がゲスト OS 側から共有されるため，HeteroVisor でオーバヘッドとなっていた Hot Page の探索のコストを下げることが可能になる．加えて，本手法ではヘテロジニアスメモリシステムをゲスト OS 側に認識させるため，サーバアーキテクチャの 1 つである NUMA を

改良し，高速/低速なメモリを NUMA ノードとして扱うことにより実現している．

KLOCs [?] は，システム全体の性能を向上させるために，I/O-intensive，memory-intensive なアプリケーションにおいて kernel オブジェクトのメモリ占有率・アクセス率が高いことに注目し，HeteroVisor や HeteroOS のようにアプリケーションデータだけではなく，kernel の機能を司るデータ構造である kernel オブジェクトに対しても Hot Page の検査と高速/低速なメモリ間のマイグレーションの対象としている．knode と呼ばれるデータ構造により，関連するオブジェクトをまとめて Hot/Cold の検査とマイグレーションの単位とすることで，Hot なオブジェクトの走査にかかるオーバヘッドを削減している．

2.4 問題点

前述のように，これらの先行研究ではデータセンタにおけるサーバマシンを対象とし，主に VMM に手を加えており，OS 単体を対象としたシステムの信頼性・可用性の向上を目指しているものではない．たとえば，HeteroVisor と HeteroOS では，ヘテロジニアスなプラットフォームの導入によって，ゲスト VM の性能向上やリソーススケーリングの粒度向上によるエンドユーザのコスト削減を狙いとしており，信頼性に目を向けた研究ではない．Characterizing Application Memory Error Vulnerability は，データそのもののメモリエラーへの耐性に着目して信頼性に目を向けた研究ではあるものの，実施されたエラー耐性の分析はユーザデータにとどまっており，kernel のようなシステム内部に関わる箇所は対象外とされている．また，これらの研究では対象としているデータの粒度がページ単位であるものが多く，メモリオブジェクトの粒度で言及がなされている研究は少ない．KLOCs は kernel のオブジェクトに着目した研究ではあるものの，マイグレーションで扱う単位としては knode によりまとめられたオブジェクトグループであり，また性能向上を目的としたものである．

PC ではメモリエラーの発生がシステムのクラッシュに直結してしまう．VM 上で稼働する OS とは異なり，OS 側が直接 CPU・メモリなどの物理的リソースを管理するため，VMM でヘテロジニアスな構成への対応とメモリエラーへの対処を代わりに実施することができない環境である．このため，ヘテロジニアスメモリシステムを導入することでシステムの信頼性を高めるためには，OS 自体にヘテロジニアスな構成を認識し，メモリエラーへの対処を行う手法が必要となる．

3. 提案

本研究では，メモリ側で訂正不可能なメモリエラーが発生した際に，破損箇所のオブジェクトに対してリカバリを実施することでシステム全体のダウンを回避し，継続動作を可能にする OS を提案する．提案手法では，破損した箇所のメモリオブジェクトにおける正常なデータを移動，オブジェクトを再構成することで，可能な限りメモリエラーに起因する kernel panic やクラッシュにつなげることなくシステムを継続動作させることで，システム全体のメモリエラーへの耐性及び可用性を向上させることを目的とする．

本研究において想定するメモリエラー及びメモリの条件は以下の通りである．

- 物理メモリとして Ecc メモリを使用する．
- 修正できないメモリエラーを検知した箇所のアドレスは，破損箇所にアクセスした時点で割り込みによって OS 側に通知される．

本研究では，Non-Ecc メモリではなく，誤り検知機能を持つ Ecc メモリを物理メモリとして使用する状況を想定する．これは，リカバリを実施する際にメモリエラーが発生した箇所のメモリに格納されていたオブジェクトを特定するには，破損箇所のアドレスが必要となるためである．また，メモリエラーのうち Ecc メモリで訂正できないエラーは，write 命令によりエラー箇所が上書きされることによってエラーとして認識されない (マスクされる) 状況を想定しない．Ecc メモリで訂正不可能なエラーには規模の大きいソフトエラーもしくはハードエラーが考えられる．前者の場合にはメモリエラーのマスクが起きる可能性があるため，メモリエラー箇所のデータを正しいデータで置き換えるという対策を講じることが可能である．しかし，本研究ではそのマスクが起きる状況を考慮せず，一律に全メモリエラーをハードエラーとして，よりシステムに悪影響を与えるものとしてみなす．

以上の仮定のもと，本手法では以下のアプローチにより設計を行う．

- Ecc メモリから受け渡されたアドレスからメモリエラーによって破損した箇所に置かれていたメモリオブジェクトを特定する．
- メモリオブジェクトのセマンティクス・使用状況・他のメモリオブジェクトやデータ構造との関連性に応じたリカバリハンドラにより回復を試みる．

以上のアプローチに基づいた設計を行う際に要求されるデザインチャレンジを以下に示す．

- **破損が検知された箇所のアドレスから対応するメモリオブジェクトを特定する方法**
  Ecc メモリから OS 側に破損箇所のアドレスが渡されるものの，それ単体で破損したメモリオブジェクトを同定するのは困難であり，破損箇所のアドレスから対応するメモリオブジェクトを探し当てるための機構が必要になる．このため，事前に各メモリオブジェクトの開始アドレスをまとめて管理するリスト (以下アドレスリストと呼ぶ) を事前に用意し，破損の発覚時に

はそのリストを引くことで当該オブジェクトを特定する．

- **多様なメモリオブジェクトに対するリカバリの実施方法**
  システムの諸機能を司るメモリオブジェクトは，その構成，セマンティクス，他データ構造との関連性といった要素が多岐にわたり，1 つの汎用的なリカバリハンドラによって対応することが困難である．このため，メモリオブジェクトごとに異なるリカバリハンドラを用意し，破損したオブジェクトによって呼び出すリカバリハンドラを変えることで対処する．

また，メモリオブジェクトごとにリカバリハンドラが存在するものの，その回復方法は，大きく以下の 3 つのアプローチに分かれる．

(1) **新しく用意したオブジェクトに他のデータ構造から情報を抜き出してオブジェクトの回復に用いる方法**
  他データ構造内に破損したオブジェクトにまつわる情報が残っており，それを抜き出すことで破損前の状態のメモリオブジェクトを再現できる場合に取る方法である．この方法では，場合によってはリカバリ処理が終わった段階でもとの処理に復帰することも可能である．

(2) **破損したオブジェクトに代わるオブジェクトを割り当てて初期化する方法**
  他データ構造から破損前の状態を復元することが難しく，かつ破損したメモリオブジェクトの代替となるものを用意しなければならない場合に取る方法である．この方法では，それまでメモリオブジェクト内に格納されていたデータを復元できないため，ユーザ側で同じ処理の再実行が必要になる場合がある．

(3) **破損したオブジェクトそのものを破棄してしまう方法**
  破損したオブジェクトそのものを破棄し，代替オブジェクトを構築しなくてもシステムの継続動作に問題ない場合に取る手段である．この方法では，他の 2 つの方法のように初期化やデータの再現を行う必要がない場合に実施し，再度メモリエラーが検出された場所へのアクセスを防止するのみでリカバリとする．

4. 設計

前章で述べたように，メモリエラーが検出された場合，事前に用意していたアドレスリストを引くことでメモリエラーによって破損したオブジェクトを特定し，オブジェクトに対応した適したリカバリハンドラを呼び出すことで回復を試みる．本章ではますアドレスリストの設計について述べた後，メモリからメモリエラーが発生した際にアドレスを受け取り，かつ適切なリカバリハンドラを呼び出すためにアドレスリストとリカバリハンドラを仲介するための機構 (以下仲介関数と呼ぶ) について説明する．最後に今回保護対象としたファイルシステム・メモリアロケータ・コンソールのメモリオブジェクトを挙げ，その中から代表的な 4 つに関してその役割，破損時に予想される影響を紹介し，ハンドラの設計に関して説明する．

4.1 アドレスリスト

アドレスリストは回復の対象となる全メモリオブジェクトの開始アドレスを記録しておき，メモリエラーの検出時にそれを引くことで破損したオブジェクトを特定するために用いられる．このアドレスリストが OS 内に存在するメモリオブジェクトの状態を正常に示すためには，保護対象のメモリオブジェクトの生成・削除に合わせてアドレスリストへ登録・削除する必要がある．アドレスリストを実現するため，専用の構造体を用意して，システムのブートやメモリオブジェクトの初期化を行うタイミングでオブジェクトの先頭アドレスを登録する．なお，オブジェクトによってはアドレスリストを準備する前から存在するものもあるため，その場合にはアドレスリストが構築された段階でリストへ追加する．同様に，オブジェクトが削除・解放されるタイミングでアドレスリストから記録していたアドレスを削除することにより，存在しないオブジェクトに対する誤ったリカバリを防止する．

4.2 アドレスリスト探索・ハンドラ仲介関数

破損箇所に対応した適切なリカバリハンドラを呼び出すために，物理メモリから受け取ったメモリエラーの発生箇所のアドレスでアドレスリストを引き，メモリオブジェクトを特定する機能が必要である．というのも，アドレスリストが全メモリオブジェクトを網羅しているのに対して，リカバリハンドラは個々のオブジェクトにつき 1 つ存在するためである．したがって，破損が発覚した時点で直接リカバリハンドラを呼ぶのではなく，アドレスリストを探索して特定されたメモリオブジェクトをもとにリカバリハンドラを呼ぶ仲介関数 mediator() を用意する．もしリカバリハンドラに対応したメモリオブジェクト以外のデータにおいてメモリエラーによる破損があった場合には，本手法による回復が不可能であるため kernel panic によってシステムを停止させる．

4.3 リカバリハンドラ

前章において述べたように，リカバリハンドラはメモリオブジェクトごとに存在するが，回復の方法論は関連するデータ構造からの情報が抜き出せるかどうか，そして破損したメモリオブジェクトの回復必要性によって 3 つに分けられる．以下では本研究において対象としたファイルシステム・メモリアロケータ・コンソールの 3 つのモジュールに関して，モジュール内のメモリオブジェクトが各方法論

のいずれによって回復を行うのかについて説明する．

4.3.1 ファイルシステム

ファイルシステムに関するメモリオブジェクトには，ファイル・i ノードのメタデータは勿論のこと，ファイルのデータをバッファリングしておくためのバッファ，ディスクに変更を書き込むためのログ，そしてこれらのデータ構造における排他制御を担うロック構造が含まれる．本研究において保護の対象としたのは，struct buf，file，inode，log/logheader，spinlock，sleeplock の計 6 つのオブジェクトである．

他データ構造からの情報をもとにメモリオブジェクトのデータの復元を図る方法を採用したのは，struct log/logheader のみであり，残りの 5 つのオブジェクトに関しては 2 つ目の方法である初期化を採用した．このうち struct log/logheader 及び struct buf，struct file に関してその役割とリカバリの方針について述べる．

struct log/logheader　struct log と struct logheader はともにファイルシステムにおけるログを担うメモリオブジェクトである．struct log がログのメタデータを保持し，ログ本体の情報は struct logheader 内に記録され，書き込み予定のデータそのものは後述のバッファキャッシュ内に保管されている．この log/logheader が破損した場合には，新しいデータ領域を割り当てて再構築した後に，ディスクのデータとバッファキャッシュから内容を再現する．log 構造体のメンバの多くはディスク上のスーパーブロックに保管されているため，ディスクから superblock を読み出して新しく用意した log 構造体に代入することで回復可能である．logheader 構造体のリカバリでは，バッファキャッシュ内にて書き込み予定のデータの eviction を防ぐために特定のフラグが立っていることから，図**??**に示すようにバッファキャッシュ内でフラグの立っているデータから書き込み予定のデータを特定し，ログに再登録することで復元することが可能である．

struct buf　buf 構造体はファイル，ログの実データを保持しておくためのバッファキャッシュを構成するメモリオブジェクトである．xv6 のファイルシステムでは，図**??**に示すように，この buf 構造体の配列を組むことによってバッファキャッシュ bcache 構造体として構築し，配列の各ノードに対して双方向リストの形式でアクセスしている．このため，bcache 構造体内部の buf 構造体が破損した場合には，bcache 構造体全体を再構築するのではなく，破損した buf 構造体の代替ノードを割り当て，リストに追加することでリカバリを実施する．こうすることにより，メモリエラーが発生したメモリ領域へのアクセスを防止するとともに，欠損した分のバッファキャッシュを補うことが可能になる．

struct file　file 構造体は，ファイルシステム内で使用しているファイルのメタデータを管理するメモリオブジェクトであり，buf 構造体と同様に file 構造体の配列を組むことで使用ファイルのテーブル ftable を構成している．しかし，bcache とは異なりアクセスする際にリストではなく配列としてアクセスするため，buf 構造体のようにリストから削除することで破損領域を排除することができない．このため，図**??**に示すように，ftable 内の file 構造体が破損した場合には，ftable 自体を新しく再構成し，その中で破損したノードに対応する箇所を初期化することで回復を行う．

4.3.2 メモリアロケータ

メモリアロケータに関するメモリオブジェクトは，物理メモリの割り当てを司る構造体は勿論，kernel のマッピングを管理するメモリオブジェクトを含めており，本モジュールでは struct kmap，kmem，run，pde_t *kpgdir の計 4 つをリカバリの対象とした．このうち，struct kmap，kmem，kpgdir は他データ構造からの情報を持ち込むことにより回復を行う一方，struct run では，後述の理由からメモリオブジェクトの破棄が適切な手段であるため，再構成・初期化を行わずにリカバリを施す．以下では，struct kmem，run を具体例にリカバリの方針を説明する．

struct run　run 構造体は，システムにおいて未使用の物理ページの先頭に格納され，メンバとして次の run 構造体のポインタを持つことで未使用ページリストを構成するメモリオブジェクトである．新しく物理メモリを割り当てる際には，そのリストから 1 つ run 構造体を取り出してそのアドレスを渡すことでメモリを 1 ページ分割り当てる．この run 構造体のリカバリでは，図**??**アドレスリストによって破損したノードの次ノードを把握し，破損したノード本体は破棄することで回復を図る．アドレスリストには全 run 構造体の先頭アドレスが記載されているため，破損した前後のノードが特定できることから，破損したノードの前後をつなぎ合わせることで，破損ノード以降のリストに正常にアクセスすることが可能になる．また，破損した run 構造体が管理していたページにはメモリエラーの発生箇所が含まれているため，そのまま破棄することでエラー箇所へのアクセスを防止することができる．

struct kmem　kmem 構造体は，前述の run 構造体からなる未使用ページリストの先頭アドレスを管理しており，run 構造体にアクセスする際にはこの kmem 構造体を通すことで，メンバのロックによって排他制御を行うことができる．この kmem 構造体のリカバリ時には，run 構造体のリストを復元することが必要不可欠であるため，アドレスリストに記載されている run 構造体のリストの先頭を代入することで容易に復元が可能である．また，排他制御を担うロックは spinlock のリカバリハンドラを呼び出すことで回復を行う．

4.3.3 コンソール

コンソールに関わるメモリオブジェクトは，コンソール

のロックを管理する struct cons，特殊デバイスへの I/O を行う関数ポインタを管理する struct devsw の 2 つをリカバリの対象とした．いずれのメモリオブジェクトについても，他データ構造及び前述のリカバリハンドラの併用することで回復が可能である．

5. 実装

本研究では，実装対象に xv6 を使用し，アドレスリスト・仲介関数 meidator()・リカバリハンドラ及びそれらに必要な諸機能の追加を行った．

5.1 アドレスリストの作成

アドレスリストは，メモリオブジェクトの開始アドレスを保持するノードを単方向循環リスト構造により接続したリストを 4.3 で挙げたメモリオブジェクトの全てにおいて作成し，それらのヘッダノードを 1 つの構造体でまとめて管理することで実現する．各ノードではアドレスリストのノード専用の構造体 alist_node を用意し，オブジェクトの先頭アドレスを void 型ポインタの形で保存する．各メモリオブジェクトのアドレスリストは，この alist_node 構造体を図**??**のように単方向リストの先頭・末尾を繋げた循環リスト構造の形式をとる．この全循環リストへのポインタを持つ addr_list_header 構造体を窓口とすることで，各アドレスリストを 1 つの入り口から管理する．このように実装している理由は，アドレスリストを走査する際の終了条件として，ヘッダから出発してヘッダに戻ってきた場合を設定でき，誤って NULL ポインタを参照してしまう事態を防ぐ目的がある．また，配列としてではなくリスト構造として実装することで，メモリオブジェクトの頻繁な増減に対応することが可能である．

5.2 アドレスリストの登録・削除

アドレスリストへのメモリオブジェクトの登録は，オブジェクトを生成する関数内から登録用関数を呼び出すことで，オブジェクト生成のタイミングに合わせて行われる．各オブジェクトの生成が行われる関数は限定されているため，その各関数内で登録用関数に対し，引数として生成したオブジェクトの先頭アドレスを渡すことで，アドレスリストへの登録を行う．登録関数内では，アドレスリストノード用に新たにメモリを動的に割り当て，前述の alist_node 構造体に引数として渡されたアドレスと次ノードへのポインタを設定して該当するリストに追加する．アドレスリストへ追加する際には常にヘッダノードの次ノードに差し込むことでリストへの追加による無駄なリストの走査時間を削減している．

この方法は，処理が簡潔に記述できる一方で run 構造体のようなオブジェクト数が非常に多いオブジェクトでは無駄にメモリ容量を割いてしまうという問題がある．加えて，xv6 のメモリ割り当て方式では，割り当てるデータ構造の大きさに関わらず 4KB のページをそのまま渡してしまうため，1 ページ全体のうち一部しか使わないこの方式を数の多いメモリオブジェクトにも採用してしまうと内部断片化が深刻になる．これに対処するため，run 構造体では図**??**に示すように，1 ページをアドレス長 4byte を 1 エントリとする配列に見立て，ページ内にアドレスリストのノードを複数設置し，実際の空きページのリストとアドレスリストを同順で登録する．

- 空きページを管理するオブジェクトのような，数が非常に多いオブジェクトは 1 ページ内にノードを複数埋める
- メモリオブジェクトの登録だけでなく，削除・再登録に対応

5.3 アドレスリスト走査・ハンドラ仲介関数

- メモリオブジェクトの種類によっては，他のリカバリハンドラが動かせない状況になるため，リストを探索する順序を調整
- マルチプロセスを使用している際等，二重にリカバリハンドラを呼び出す場合に対処

5.4 リカバリハンドラ

- 各メモリオブジェクトのリカバリハンドラについて記述 or 工夫を凝らしたものやわかりやすいものについてのみ記述

6. 実験

6.1 目的
6.2 実験環境
6.3 クラッシュシナリオ
6.3.1 クラッシュシナリオ 1
6.4 クラッシュシナリオ (マルチプロセス)
6.4.1 クラッシュシナリオ 1

7. おわりに

本研究では，Ecc メモリにおいても訂正不可能なメモリエラーを検知した場合にでも，リカバリを実施することでシステム全体の継続動作を可能にする手法を提案した．既存研究におけるヘテロジニアスメモリシステムを用いた手法では，主に性能向上に重きを置いてシステム全体の性能に関わる Hot Page を検査・マイグレーションする方式を取っており，OS 内部のメモリオブジェクトという細かい粒度で，かつメモリエラーに注目して信頼性を高めることは考慮されてこなかった．本手法ではメモリエラーが発生した場合に備えて事前に各メモリオブジェクトの先頭アドレスを保持するアドレスリストを構築しておき，メモリエラーが実際に検知された際にはそのリストを引くことで適

切なリカバリハンドラを適用する．リカバリハンドラでは，破損したメモリオブジェクトの再構築及び関連するデータ構造との整合性の保持を行うことで，継続してシステムが動作できる状態にすることを可能にした．

提案手法を xv6 に対して実装し，メモリエラーが起きた状況を想定したクラッシュシナリオに基づいてフォールトインジェンクションによる実験を実施したところ，想定したシナリオのもとではシステムが形像動作することを確認した．

今後として，本研究では基本的に 1 コア 1 スレッドのマシンにおけるクラッシュシナリオを想定したため，よりコア数，スレッド数が増加した場合におけるリカバリハンドラや仲介関数の排他制御，リカバリ対象のデータ構造と関連するデータ構造間での整合性の保ち方を考える必要がある．また，今回メモリエラーは 1 つのオブジェクトに対してのみ発生する，つまり複数オブジェクトを跨がず，複数同時に発生しないものとして実装・実験を実施した．このため，より信頼性を高めるために，同時多発的かつオブジェクトを跨ぐメモリエラーにも対応可能な手法を考える必要がある．

加えて，今回メモリ内でより大きなウェイトを占めるページテーブルは対象外としたが，実際メモリ内を多く専有するページテーブルはメモリエラー発生時に破損する確率が高いことが考えられる．それゆえ，ページテーブルに対してもリカバリを行うことができる機構の開発が求められる．

参考文献