

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JECTION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

メモリエラーとは、ハードウェアの故障などによってメモリ上のビットが変化する現象であり、メモリエラーがオペレーティングシステムのメモリオブジェクトを破壊すると、コンピュータシステムの可用性に大きな影響を及ぼす。たとえば、ページテーブルがメモリエラーによって内容が変わってしまうと、正しくアドレス変換がなされず、該当プロセスが意図しないアドレスにアクセスしてしまう。Linux では、メモリエラーがハードウェアによって検知されると、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) [1] 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 1 ビットであり、複数ビットが故障すると訂正できない。実際のデータセンタでは毎年 32 % のサーバがメモリエラーを被っており、その内 14 % が訂正不可能であ

る [2]。

異なる物性のメモリを組み合わせる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登場しており、メモリエラーへの耐性向上が期待できる。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 [3] やメモリを 3 次元に積み重ねる 3D-stacked メモリ [4] 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 [5-7]、互いに組み合わせることで高性能・大容量 [3]、高性能・低消費電力量 [6, 8] 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るヘテロジニアスメモリシステムを構築し、重要なメモリオブジェクトを信頼性の高いメモリに配置することで、システムの可用性を高められる。しかしながら、既存の OS はメモリが DRAM のみで構成されていることを前提としており、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ない。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、DRAM のような Non-Ecc メモリに比べて容量が少ない傾向にあり [9]、ページテーブルやバッファキャッシュといったサイズの大きいメモリオブジェクトを搭載することが難しい。

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

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 [10] 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氣的

な要因もしくは宇宙線による外的要因[2]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[2]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[11]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[9]。

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、ソフトウェアに起因する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 [12]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ることで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を組み合わせる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 [13] やヘテロジニアスなリ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 [14] 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 [14, 15]。そのため、単に種類の異なる物理メモリを組み合わせるだけで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) [16]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 Vulnerability [9]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成本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 [17]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) [18]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 [14]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でオーバーヘッドとなっていたHot Pageの探索のコストを下げるが可能になる。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 [19]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

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ー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ジングの粒度向上によるエンドユーザのコスト削減を狙っており、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り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) 他のデータ構造から情報を抜き出して新しく割り当てたオブジェクトに適用する方法
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を取る方法である．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を取る方法である．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を取る手段である．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

図1に示すように，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．本章ではまず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．

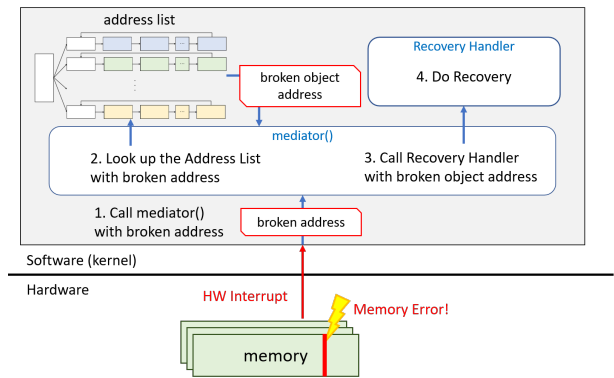


図1: 本手法の概観

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2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。

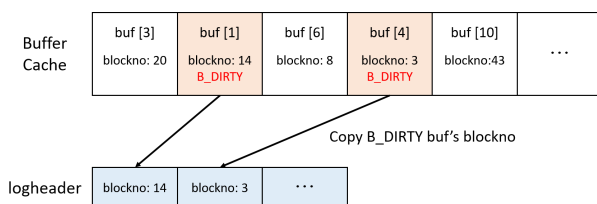


図2: log/logheader 構造体におけるリカバリの概観

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3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。

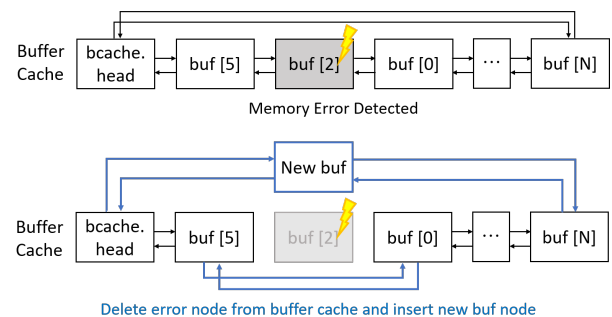


図3: buf 構造体におけるリカバリの概観

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4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。

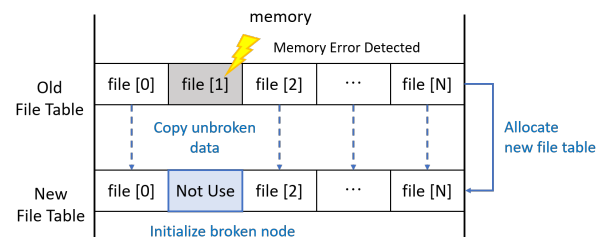


図4: file 構造体におけるリカバリの概観

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 5 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。

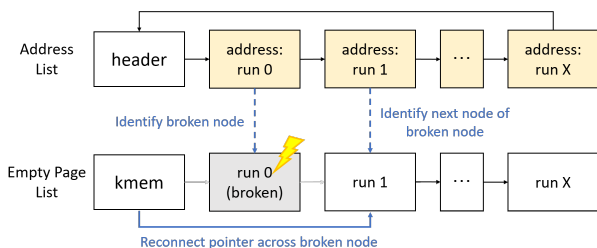


図 5: run 構造体におけるリカバリの概観

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 6 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。

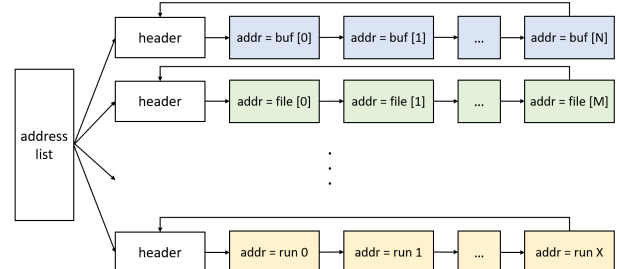


図 6: アドレスリスト全体の概観

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

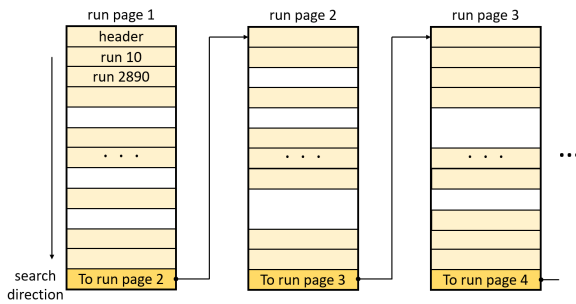


図 7: run 構造体のアドレスリストノードの格納方法

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 7 に示すように、1 ページをアドレス長 8byte 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に登録する。

アドレスリストからのメモリオブジェクトの削除もまた、登録時と同様にメモリオブジェクトが削除される関数内から登録抹消用の関数を呼び出し、削除されるメモリオブジェクトの先頭アドレスを渡すことで実施する。登録抹消関数内ではアドレスリスト内を走査し、引数と一致するアドレスを持つノードをリストから削除し、ノードに用いていた領域を解放することでアドレスリストからメモリオブジェクトを削除する。run 構造体においては、同様にリストを辿って一致したノードをリストから削除するものの、1 ページ内に複数のノードを格納しているため、ページそのものを解放することはしない。

run 構造体をアドレスリストに登録する際に通常のメモリ割り当て関数 `kalloc()` を用いると、解放しようとしているページに対して登録関数内で割り当てを行おうとしてしまう状況が発生することがあるため、その場合に対処する独自の割り当て関数 `my_kalloc()` を用意する。

5.3 アドレスリスト走査・ハンドラ仲介関数

仲介関数 `meidator()` においてアドレスリストを走査する際には、メモリオブジェクトの開始アドレスと終端アドレスを用いて、メモリオブジェクトのデータの存在範囲で検索を行う。アドレスリストにおいて記録しているアドレスは各オブジェクトの先頭アドレスのみであるが、実際のメ

モリエラ発生時に物理メモリから渡されるアドレスは先頭アドレスとは限らない。このため、アドレスリストの探索時には、開始アドレスにメモリオブジェクトのサイズを足して終端アドレスを算出し、破損箇所のアドレスが登録しているメモリオブジェクトのデータ範囲内にあるかを確認する。

マルチプロセスの環境下では、リカバリハンドラが二重に呼ばれない工夫が必要となる。マルチコアプロセッサや CPU を複数搭載している構成などでは、複数のプロセスが同じコードを同時に実行する場合があるため、1 つのメモリエラが複数プロセスで同時に発覚する可能性がある。この場合、複数プロセスで同じリカバリハンドラが呼ばれ、リカバリ後のメモリオブジェクトが複数再構成されてしまうなど、データの不整合に繋がりがねない事態が生じる可能性がある。このため、図 8 のように、グローバル変数 `recovering` によるフラグを導入し、仲介関数内のプロセスの有無を管理することで、リカバリハンドラを呼び出すプロセスが 1 つになるように制御する。`recovering` が立っている状態で仲介関数に進入したプロセスは即座に `sleep` し、先に進入したプロセスによるリカバリが終了し、関数 `wakeup_others()` により起こされた後に `exit()` することでリカバリハンドラが複数呼ばれることに起因する不整合を防止する。

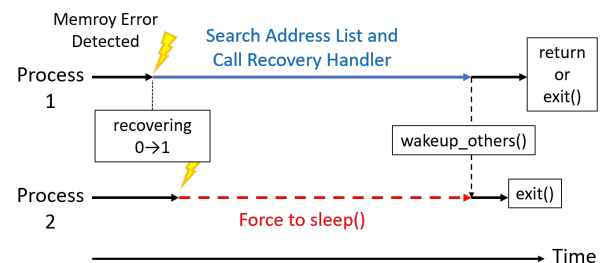


図 8: マルチプロセス下における処理の流れ

5.4 リカバリハンドラ

4.3 で例示した各オブジェクトに対応するリカバリハンドラについて、その実装方法を説明する。

5.4.1 ファイルシステム

`struct log/logheader` `log/logheader` 構造体のリカバリでは、書き込み予定のデータを格納している可能性のある `buf` 構造体をフラグをもとに判別し、`logheader` に反映させる。`logheader` 構造体には書き込む予定のデータブロック番号が記録されており、書き込むデータ本体はバッファキャッシュ内の `buf` 構造体にて管理されている。このデータに対して何らかの処理により変更が加わった際には、`buf` 構造体のメンバ `flags` に `B_DIRTY` というフラグが立ち、ディスクへの反映待ちの状態になる。このため、`logheader` のリカバリではバッファキャッシュ内で `B_DIRTY` が立っている

buf 構造体のブロック番号を再登録することで回復を行う。

log 構造体には主にログのメタデータを保持するため、リカバリではディスク上のスーパーブロックの情報を読み出す関数 `readsb()` によりメタデータを取得する。一方、メンバ `committing` はログがコミットの処理中か否かを示すフラグであるため、このフラグを立てた状態で初期化するとログへの反映が行われないことから、必ずフラグを立てない状態で初期化する。

`struct buf` buf 構造体は双方向リストで各ノードを接続することでバッファキャッシュを構成しているため、破損箇所のノードを削除するにはその前後のノードを特定する必要がある。そのため、図 9 に示すようにバッファキャッシュの先頭からメンバ `next` を辿ることで前ノードを、`prev` を辿ることで後ノードを探し当てる。前後のノードのメンバ `next`, `prev` が破損ノードを指している状態であるため、これを破損ノードをまたぐようにつなぎ替えることによって、リストから破損ノードを除外する。破損ノードの代わりとなるノードは、新しくメモリ領域を割り当てた後にバッファキャッシュの先頭に未使用状態として挿入する。

破損箇所の回復を終えると、次にログを司る `log/logheader` 構造体、ディスク読み書きのキューである `idequeue` との整合性を確認する。

ログでは、`logheader` 構造体内にディスク書き込み予定のブロック番号が記録されており、その中に破損した buf 構造体に格納されていたブロック番号が存在しないか検査する。ログとバッファキャッシュ上で対応しないブロック番号が見つければ、破損したノードに格納されていたブロック番号と見なして削除する。これは、破損した書き込み予定のデータを他データ構造から復元することが困難であり、ログ書き込み時に対応するデータがないことによるクラッシュを防止する目的がある。

`idequeue` では、メモリエラーに起因する buf 構造体の破損によって生じたキューの切断箇所の有無を確認する。`idequeue` は buf 構造体のメンバ `qnext` を用いて、バッファキャッシュ内の buf 構造体を連結して入出力キューを構成しているため、`idequeue` を構成していた buf 構造体が破損すると、本来入出力予定であったデータを追跡できなくなる。これに対処するため、本来 `idequeue` の末尾ノードの `qnext` に NULL ポインタが代入されていた代わりに `idequeue` の先頭ノードを格納させるように変更し、NULL ポインタが格納されていた場合には必ず切断されていることを検知可能にする。リカバリ時には `idequeue` を走査し、切断箇所が見つければ、バッファキャッシュを走査して `idequeue` が NULL でないノードを再度繋げることで入出力キューの整合性を保つ。

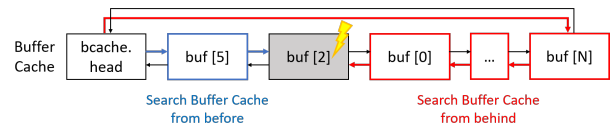


図 9: 破損要素の前後の buf ノードの特定

`struct file` file 構造体のリカバリでは、`ftable` そのものを再構成するために連続した 2 ページの物理メモリを確実に割り当てる必要がある。xv6 のメモリ割り当てを行う関数 `kalloc()` では未使用ページリスト上のページを単に割り当てるのみであるため、アドレスリストの登録時に用いる `my_kalloc()` を改良し、引数のフラグによって連続した 2 ページを割り当てるように変更している。

また、`ftable` の再構成を終えた後にパイプラインのコマンドを司る `pipe` 構造体と `inode` 構造体の 2 つのデータ構造との整合性を確認する。`pipe` 構造体は、パイプライン処理に必要なメタデータを保持するメモリオブジェクトである。`pipe` 構造体が存在する場合、file 構造体では `FD_PIPE` というフラグが立つノードが必ず 2 つ存在するため、`ftable` 構造体を走査して `FD_PIPE` が立つ file 構造体とともに存在しているかを確認する。パイプライン処理を行う file 構造体が片方しか見つからない場合には、破損した file 構造体がパイプライン処理の対象であったと判断し、パイプライン処理を中断する。

`inode` 構造体では、開かれているファイルに対応する `i` ノードのメタデータが保管されているため、`ftable` と `i` ノードキャッシュとを比較し、対応関係を調査する。対応していない `i` ノードが見つければ、破損した file 構造体のもものと判断し、キャッシュしている情報を削除する。

5.4.2 メモリアロケータ

`struct run` run 構造体のリカバリでは、アドレスリストを走査することで破損ノードの特定とリカバリに必要な情報を取得する。run 構造体が置かれている箇所に発生したメモリエラーが発覚するのは、メモリ割り当て関数 `kalloc()` においてリストの先頭の run 構造体にアクセスした場合である。このため、5.2 で述べた方法により、run 構造体のアドレスリストは実際の未使用ページリストの順序を反映した状態で適切に管理されていることから、アドレスリスト上で先頭ノードが破損したノードのアドレスを持ち、次ノードが空きページリストにおいて破損ノードの次ノードであることが分かる。したがって、破損したオブジェクトをアドレスリストと未使用ページリストから削除し、`kmem` 構造体に次ノードのアドレスを格納することで回復を行う。

`struct kmem` kmem 構造体のリカバリでは、run 構造体のアドレスリスト及びコールスタックの情報をもとにオブジェクトデータの再現を行う。kmem 構造体のメンバには空きページリストの先頭、排他制御用のロック、ロックの

有効/無効を管理するフラグがある．実際の空きページリストの先頭は `kmem` 構造体が破損したことで追跡できないため，代わりにアドレスリストの情報をを用いることで復元する．ロックの有効/無効を管理するフラグは，ブート時にロックが使えない状況に対応する目的で設置されていることから，関数 `getcallerpcs()` を用いてコールスタックを取得し，呼び出し元にブートで用いられる関数の有無を確認することでフラグの値を判断する．

5.4.3 コンソール

`struct cons` `cons` 構造体のリカバリでは，`spinlock` 構造体のリカバリハンドラとコールスタックの情報を利用する．`cons` 構造体のメンバには，コンソールの排他制御を担うロックとロックの有効/無効を管理するフラグが存在する．フラグは `struct kmem` と同様に，`getcallerpcs()` を呼び出してコールスタックの情報を取得し，フラグが無効化される `panic()` 内でメモリエラーが発覚しているかを確認することで復元する値を決定する．

`struct devsw` `devsw` 構造体のリカバリでは，関数 `console_read()/console_write()` の関数ポインタを使用し，`log` 構造体との整合性を保持する．`devsw` 構造体はコンソールへの入出力を司る 2 つの関数ポインタを保持しているため，その復元には同じ値を代入することで容易に復元可能である．一方，`devsw` 構造体を再構成することで，メモリエラーによるオブジェクトの破損が発覚した関数内で使用されていた元々の `devsw` 構造体を指すポインタが使用できなくなるため，システムコールを中断せざるを得ない．それゆえ，`log` 構造体においてシステムコールの発行状況を示すフラグ `outstanding` が立っているとその後のシステムコールによる変更がディスクに反映されなくなることから，このフラグをリセットすることでログ書き込みを有効にする．

6. 実験

6.1 目的

本実験の目的は，実装したアドレスリスト，仲介関数 `mediator()` と各メモリオブジェクトのリカバリハンドラによる回復により，メモリオブジェクトが破損した状態におけるシステム全体の継続動作に対する有効性を検証することである．メモリエラーが実際に発生した場合に検出されると考えられる箇所において，メモリオブジェクトにフォールトインジェクションを行い，仲介関数 `mediator()` を呼び出してリカバリ処理を行わせ，システムの継続処理が可能かを確認する．

6.2 実験環境

実験を実施する物理マシンと QEMU [20] によりエミュレートするマシン構成を表 1, 2 に示す．基本的にシングルコアの構成において実験を行うが，6.4 においてのみマルチコアの構成を用いて実験を実施する．また，物理マシ

ンの OS には Ubuntu18.04 を用い，エミュレータ上で実験を行う．xv6 では 5 において述べた実装を適用し，クラッシュシナリオにおいて対象となるインジェクション部分のみが作動するようにインジェクションコードを追加し，フラグ `broken_flag` をシステムコールで立てることによって発火する．

表 1: 物理マシンの構成

構成	内容
CPU	Intel(R) Core(TM) i5-4210M CPU @ 2.60GHz
コア数	2 コア 4 スレッド
メモリ	32GB

表 2: 仮想マシンの構成

構成	シングルコア	マルチコア
コア数	1 コア 1 スレッド	2 コア 2 スレッド
メモリ	4GB	4GB

6.3 クラッシュシナリオ

想定したクラッシュシナリオに基づいて，シナリオの内容とインジェクション対象となる関数，リカバリを行わない場合に観測された挙動とリカバリを行った場合に見られた挙動について述べる．シングルコアにおけるクラッシュシナリオは，全メモリオブジェクトに対して 1 つ以上の状況を想定し，計 18 個について実験を行った．いずれのシナリオにおいてもリカバリを行わない場合にはクラッシュしてしまうシステムが，実装したリカバリハンドラによる回復を行うことによって継続的に動作することが確認できた．以下では 18 個のシナリオのうち，`log` 構造体，`kmem` 構造体，`buf` 構造体に関する 3 つのシナリオについて述べる．

6.3.1 クラッシュシナリオ 1

1 つ目のシナリオでは，ブート時に `log` 構造体においてメモリ破損が発覚し，ログからコミットされたログが復旧できない場合を想定する．コード 1 は，`log.c` 内の `install_trans()` でのインジェクション部分を示している．`install_trans()` はコミットされたログをディスクに書き込む関数であり，毎回の起動時にも実行され，復旧する必要のあるログが存在するか，存在するならば復旧を実施する．この場所でメモリエラーが発覚したと仮定し，`log` 構造体のポインタに 0 を代入してインジェクションを行い，その状態から `mediator()` を呼ぶことでリカバリを開始する．

リカバリを行わない場合では xv6 自体の起動が終了せず，画面上では反応が無くなった．GDB による内部の挙動の観察を試みるも，ソースコードの表示やスタックの内容も判読可能なものがない状態であった．一方で，リカバリを実施すると起動に成功してシェルに入力することが可能になり，`mkdir` や `rm` のようなログ書き込みが生じるコマ

ンドの処理も受け付けることが確認された。

コード 1: クラッシュシナリオ 1

```

1 static void
2 install_trans(void)
3 {
4     int tail;
5
6     // Case 1: When this xv6 boots, the memobj
7     // log was broken.
8     if(broken_flag == 1){
9         struct log *alt = log;
10        log = 0; // break log.
11        broken_flag = 0;
12        mediator((void*)alt);
13    }
14
15    for (tail = 0; tail < log->lh.n; tail++) {
16        . . .

```

6.3.2 クラッシュシナリオ 2

2 目目のシナリオでは、新しいメモリページを割り当てる際に kmem 構造体が破損していた場合を想定する。コード 2 は、kalloc.c 内の関数 kalloc() におけるインジェクション部分を示している。シナリオ 1 と同様に、kmem 構造体のポインタを 0 で初期化することでフォールトインジェクションを実施し、mediator() を呼び出すことでリカバリを開始する。

リカバリを実施しない場合、インジェクション直後にコンソールからの入力を受け付けなくなった。GDB により内部動作を調査すると、kmem 構造体の排他制御を担うメンバ lock が破損していることによりロックをかけられないため、デッドロックのような状態で延々とロックの取得を試みていたことが原因であった。一方でリカバリを実施すると、コンソールがコマンド入力を受け付ける状態になり、mkdir や rm のようなログ書き込みやページの割り当て・解放が生じるであろうコマンドの処理も受け付けることが確認された。

コード 2: クラッシュシナリオ 2

```

1 char*
2 kalloc(void)
3 {
4     struct run *r;
5
6     // Case 2: When try to allocate a page, it
7     // turns out kmem is broken.
8     if(broken_flag == 1){
9         broken_flag = 0;
10        struct kmem *alt = kmem;
11        kmem = (struct kmem*)0;
12        mediator((void*)alt);
13    }

```

```

14 r = kmem->freelist;
15 . . .

```

6.3.3 クラッシュシナリオ 3

3 目目のシナリオでは、ディスクへの入出力を行うタイミングで、データを格納している buf 構造体の破損が発覚した状況を想定する。コード 3 は、ide.c 内の関数 idestart() におけるインジェクション部分である。idestart() はディスクへの読み書きを行う際に呼び出され、buf 構造体の内容からセクタ番号などを特定する。このため、buf が破損しているとディスクへの読み書きに支障が出てしまうことが予想される。

リカバリを実施しない場合では、インジェクションコードに続く 12 行目の if 文に引っかかり、関数 panic() を呼ぶことで kernel panic を引き起こしてしまい、システムを再起動しなければならない状態に陥る。一方でリカバリを実施すると、リカバリハンドラによる処理の終了後強制的にシェルに戻り、その後コマンドを受け付ける状態になっていることが確認された。

コード 3: シナリオ 3: idestart() における buf へのインジェクション

```

1 static void
2 idestart(struct buf *b)
3 {
4     // Case 3: when xv6 tries to disk I/O, it
5     // turns out that the struct buf which
6     // the b pointing is broken.
7     if(broken_flag == 1){
8         struct buf* alt = b;
9         b = 0x0;
10        broken_flag = 0;
11        mediator((void*)alt);
12    }
13
14    if(b == 0)
15        panic("idestart");
16    if(b->blockno >= FSSIZE)
17        panic("incorrect blockno");

```

6.4 クラッシュシナリオ (マルチプロセス)

エミュレートするマシンの構成を変更し、5.3 で述べた仲介関数とリカバリハンドラを二重に呼び出してしまう状況への対応を検証する。マルチプロセスを想定したクラッシュシナリオは log 構造体が破損したケースと buf 構造体が破損したケースの計 2 個を想定した。以下では log 構造体が破損したシナリオについて述べる。

6.4.1 クラッシュシナリオ 1

マルチプロセスにおけるクラッシュシナリオ 1 では、log 構造体のリカバリ中に別プロセスが log のロックを取得しようとしたことで二重にリカバリハンドラが呼ばれてしま

うケースを想定する．システムコールを発行する際にログの処理を開始するための関数 `begin_op()` におけるインジェクションにより，`mediator()` 及びリカバリハンドラが呼ばれている最中に，別プロセスが `log` 構造体にロックをかけようとすると，`spinlock.c` 内のロック取得関数 `acquire()` から二重に `mediator()` が呼ばれる状況が起こる．この状況を再現するため，コード 4，5 に示すインジェクション部分を追加している．

インジェクションを実施したところ，マルチプロセスへの対処を講じていない場合には新しい `log` 構造体が二重に生成されていた．一方でマルチプロセス時の対処を施していた場合には，最初のプロセスのみがリカバリハンドラの処理を行い，後続のプロセスではハンドラの処理が終わるまでスリープした後強制終了した．このため，生成される新しい `log` 構造体は 1 つのみに限定することができ，システムも継続的に動作していることが確認できた．

コード 4: `log/logheader` へのインジェクション

```

1 void
2 begin_op(void)
3 {
4     // Case d-1: when xv6 calls begin_op() to
5     // use cprintf(), it turns out that log is
6     // broken, and later access again in
7     // acquire().
8     if(broken_flag == 1){
9         broken_flag = 2;
10        struct log *alt = log;
11        log = 0;
12        mediator((void*)alt);
13    }
14    acquire(&log->lock);
15    while(1){
16        if(log->committing){
17            . . .

```

コード 5: `spinlock` のインジェクション

```

1 void
2 acquire(struct spinlock *lk)
3 {
4     pushcli(); // disable interrupts to avoid
5     // deadlock.
6     // Case d-1: when xv6 calls begin_op() to
7     // use cprintf(), it turns out that log is
8     // broken, and later access again in
9     // acquire().
10    if(broken_flag >= 1){
11        idelock = (struct spinlock*)0;
12        broken_flag = 0;
13        mediator((void*)&idelock);
14    }
15    if(holding(lk))

```

```

13 panic("acquire");
14 . . .

```

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ぐず，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

- [1] Chen, C. L. and Hsiao, M. Y.: Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review, No. 2, pp. 124-134 (online), DOI: <https://doi.org/10.1147/rd.282.0124> (1984).

- [2] Bianca Schroeder, E. P. and Weber, W.-D.: DRAM Errors in the Wild: A Large-Scale Field Study, *Proceedings of the eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, ACM, pp. 193–204 (online), DOI: <https://doi.org/10.1145/1555349.1555372> (2009).
- [3] Moinuddin K. Qureshi, V. S. and Rivers, J. A.: Scalable high performance main memory system using phase-change memory technology, *Proceedings of the 36th annual international symposium on Computer architecture*, ACM, pp. 24–33 (online), DOI: <https://doi.org/10.1145/1555754.1555760> (2009).
- [4] Black, B. et al.: Die Stacking (3D) Microarchitecture, *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, (online), DOI: <https://doi.org/10.1109/MICRO.2006.18> (2006).
- [5] Dulloor, S. R. et al.: Data tiering in heterogeneous memory systems, *Proceedings of the Eleventh European Conference on Computer Systems*, ACM, pp. 1–16 (online), DOI: <https://doi.org/10.1145/2901318.2901344> (2016).
- [6] Phadke, S. and Nararanasamy, S.: MLP aware heterogeneous memory system, *Proceedings of the 2011 Design, Automation & Test in Europe*, IEEE, (online), DOI: <https://doi.org/10.1109/DATE.2011.5763155> (2011).
- [7] Luiz E. Ramos, E. G. and Bianchini, R.: Page placement in hybrid memory systems, *Proceedings of the international conference on Supercomputing*, ACM, pp. 85–95 (online), DOI: <https://doi.org/10.1145/1995896.1995911> (2011).
- [8] Chatterjee, N. et al.: Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access, *In Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, pp. 13–24 (online), DOI: <https://doi.org/10.1109/MICRO.2012.11> (2012).
- [9] Luo, Y. et al.: Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory, *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE, pp. 1–12 (online), DOI: <https://doi.org/10.1109/DSN.2014.50> (2014).
- [10] of Technology, M. I.: xv6-public, <https://github.com/mit-pdos/xv6-public>. (accessed 2021-01-08).
- [11] Daniel Henderson, J. M. and Ahrens, G.: POWER7 System RAS, [urlhttp://www.s7.com/Hardware/IBM-Power-Systems/PDFs/POWER7-System-RAS.PDF](http://www.s7.com/Hardware/IBM-Power-Systems/PDFs/POWER7-System-RAS.PDF) (2010).
- [12] IBM: IBM Chipkill Memory, <https://www.ibm.com/support/pages/ibm-chipkill-memory-netfinity-7000-m10>. (accessed 2021-01-08).
- [13] Kumar, R. et al.: Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction, *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, (online), DOI: <https://doi.org/10.1109/MICRO.2003.1253185> (2003).
- [14] Kannan, S. et al.: HeteroOS – OS Design for Heterogeneous Memory Management in Datacenter, *Proceedings of the 44th Annual Symposium on Computer Architecture*, ACM, pp. 521–534 (online), DOI: <https://doi.org/10.1145/3079856.3080245> (2017).
- [15] Narayan, A. et al.: MOCA: Memory Object Classification and Allocation in Heterogeneous Memory Systems, *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, pp. 326–335 (online), DOI: <https://doi.org/10.1109/IPDPS.2018.00042> (2008).
- [16] : VMWare ESX Server 2 NUMA Support, https://www.vmware.com/pdf/esx2_NUMA.pdf.
- [17] Vishal Gupta, M. L. and Schwan, K.: HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms, *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ACM, pp. 79–92 (online), DOI: <https://doi.org/10.1145/2731186.2731191> (2015).
- [18] : Amazon EC2, <https://aws.amazon.com/ec2>. (accessed 2021-01-09).
- [19] Sudarsun Kannan, Y. R. and Bhattacharjee, A.: KLOCs: kernel-level object contexts for heterogeneous memory systems, *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 65–78 (online), DOI: <https://doi.org/10.1145/3445814.3446745> (2015).
- [20] : QEMU, <https://www.qemu.org/>. (accessed 2021-01-15).