単一アドレス空間オペレーティングシステム（SASOS）の包括的研究：アーキテクチャ、利点、欠点、および将来の展望

1. 序論

計算機科学の歴史において、オペレーティングシステム（OS）のメモリ管理アーキテクチャは、基盤となるハードウェアの制約と不可分に結びついて進化してきた。1960年代以降のマルチプログラミングの黎明期から現代に至るまで、UNIX、Linux、Windowsといった汎用オペレーティングシステムは、各プロセスに対して独立したプライベートな仮想アドレス空間を提供する「マルチアドレス空間」モデルを採用している。このモデルは、限られたアドレス空間（例えば32ビットアーキテクチャにおける4GB）を各プロセスに多重割り当てすることで、名前空間の枯渇を防ぐという歴史的必然性から生まれたものである[1]。プライベートアドレス空間はプロセス間のハードウェアレベルの隔離（アイソレーション）を容易にする一方で、仮想アドレスの解釈がプロセスごとに異なるという根本的な制約をもたらした[1]。
しかし、1990年代に入り、DEC Alpha、MIPS R4000、HP-PA RISCといった64ビットアーキテクチャのプロセッサが登場したことで、仮想メモリの設計前提は劇的な転換期を迎えた。40ビットの論理アドレス空間でさえ1テラバイトのメモリを管理可能であり、完全な64ビット空間であれば、毎秒100メガバイトの速度で消費し続けたとしても枯渇するまでに5,000年を要する計算となる[1]。仮想アドレスはもはや節約して多重利用すべき「希少資源」ではなくなったのである[1]。
この広大な名前空間を背景に考案された概念が、「単一アドレス空間オペレーティングシステム（Single Address Space Operating System: SASOS）」である。SASOSは、カーネルを含むすべてのプロセス、スレッド、およびデータ（ローカル、リモート、永続的、一時的を問わず）に対して、グローバルで共有される単一の仮想アドレス空間を提供するアーキテクチャである[3]。このモデルでは、異なるプロセスであっても、数値的に同一の仮想アドレスはシステム上の「全く同じ1バイトのデータ」を指し示す[2]。本報告書では、SASOSの基本アーキテクチャと多様な保護メカニズムを詳細に分析し、その決定的な利点と致命的な欠点、そして現代のクラウドネイティブ環境や最新のプロセッサ拡張アーキテクチャにおけるSASOSパラダイムの復活について深く考察する。

2. SASOSのコアアーキテクチャと基本原則

2.1 翻訳（Translation）と保護（Protection）の直交化

従来型のOSでは、メモリアドレスの「翻訳（仮想アドレスから物理アドレスへの変換）」と「保護（アクセス権の制御）」が密結合している。各プロセスは固有のページテーブルを持ち、そのテーブル内にエントリが存在しないアドレスにはアクセスできない。すなわち、アドレス空間の境界そのものが保護の境界（Address Space Isolation）として機能している[1]。

対照的に、SASOSでは翻訳と保護を完全に分離（直交化）する。システム全体のデータは単一のページテーブル（またはそれに類する広域データ構造）にマッピングされ、任意のアドレスの「名前（アドレス値）」を知ることはシステム内のあらゆるエンティティにとって可能である。しかし、名前を知っていることと、そのアドレスに「アクセス」できることは全く別の概念として扱われる[4]。SASOSにおいては、保護はページテーブルの境界ではなく、「保護ドメイン（Protection Domains）」や「ケーパビリティ（Capabilities）」と呼ばれる権限管理メカニズムによって提供される[1]。
この設計思想により、SASOSはフラットメモリモデル（初期のMac OSやMS-DOSのようにアドレス変換も保護も存在しないモデル）とは明確に一線を画し、高いセキュリティと隔離性を維持したまま全プロセスが同一のアドレス空間を共有することを可能にしている[1]。

2.2 単一レベルストア（Single-Level Store）と直交的永続性

広大な64ビットアドレス空間を利用する副次的な、しかし極めて重要なアーキテクチャ的特徴が「単一レベルストア（SLS: Single-Level Store）」の実現である。SASOSでは、データが存在する物理的な場所（メインメモリの揮発性DRAM上か、ハードディスクやSSDなどの不揮発性ストレージ上か）に関わらず、すべてのデータが仮想アドレス空間のどこかに一意にマッピングされる[5]。
これにより、プロセスは明示的なファイルシステムI/O（read()やwrite()などのシステムコール）を利用する必要がなくなる。二次ストレージは実質的に仮想メモリシステムのページング用バッキングストアとしてのみ機能し、ポインタを用いたメモリアクセスだけで永続データの読み書きが可能になる[8]。このような直交的永続性（Orthogonal Persistence）は、オブジェクトの生成時においてそのライフタイムを意識することなく、アドレスベースで一元的にデータを管理するパラダイムを提供する[9]。一度割り当てられたアドレスは、システムが再起動された後でも同一のデータを指し示し続けるため、永続的なデータ構造をそのままディスクに退避させることが可能である[7]。

2.3 従来型OSとSASOSの比較分析

以下の表は、マルチアドレス空間を採用する従来型OSと、SASOSの基本パラダイムの違いを比較したものである。

アーキテクチャ要素	**従来型マルチアドレス空間OS (UNIX, Windows等)**	**単一アドレス空間OS (SASOS)**
仮想アドレスのスコープ	プロセスごとに独立・ローカル	システム全体で単一・グローバル
ポインタの有効性	コンテキスト依存（他プロセスでは無効）	コンテキスト非依存（全体で一意に解釈可能）
保護のメカニズム	アドレス空間の境界によるハード隔離	保護ドメイン、ケーパビリティ、言語レベルの型安全

		性
データ共有手法	共有メモリの明示的マッピング、メッセージの直列化	アドレス（ポインタ）の受け渡しのみで直接共有
ストレージ抽象化	ファイルシステムとメモリ空間が完全に分離	単一レベルストア（永続オブジェクトとメモリの統合）
IPC（プロセス間通信）	システムコール、データコピー、文脈切り替えの負荷大	ゼロコピーの関数呼び出し、ポインタの直接参照

3. アドレス保護メカニズムの多様な実装アプローチ

SASOSにおいて、単一空間内での不正アクセスを防ぐための保護メカニズムは、主にハードウェア支援アプローチ、ソフトウェア（言語）支援アプローチ、および暗号学的アプローチに大別される。各プロジェクトは、それぞれの目標に応じて異なる保護戦略を採用してきた。

3.1 ケーパビリティとパスワード保護（Mungiモデル）

オーストラリアのNICTAで開発された**Mungi**は、スパースな（空間に極めて疎に分布した）「パスワード・ケーパビリティ（Password Capabilities）」を用いたアクセス制御モデルを採用した[6]。ケーパビリティは、対象オブジェクトの64ビットベースアドレスと、所有者が生成した64ビットのランダムパスワードのペアで構成され、読み取り、書き込み、実行、削除などの権限を付与する[6]。

タスクが未参照のオブジェクトにアクセスしようとすると保護違反（ページフォルト）が発生し、カーネルはタスクの「アクティブ保護ドメイン（APD）」内に有効なケーパビリティが存在するかを自動的に検索する。合致するパスワードが見つかれば、以降のアクセスに対するハードウェアマッピングが確立される[6]。パスワードは巨大な乱数空間から選ばれるため、悪意のあるプロセスがブルートフォース攻撃や推測によって他者のパスワードを偽造することは統計的に不可能であるという性質に依存している[6]。この手法は、アプリケーション側にケーパビリティの明示的な管理を強要しない「非侵入的（Unintrusive）」なモデルを実現した[6]。

3.2 保護ドメインとアクセス制御リスト（Opalモデル）

ワシントン大学で開発された**Opal**プロジェクトでは、Mach 3.0マイクロカーネルをベースに、アクセス制御リスト（ACL）と権限ツリーを用いた「保護ドメイン」による制御が実装された[2]。Opalのスレッドはそれぞれ独自の保護ドメインに関連付けられており、このドメインが「どの仮想ページに対してどのようなアクセス権を持つか」を定義する[2]。

カーネルは、オブジェクト（セグメント）に対するアクセス権を管理し、プロセスは必要に応じてセグメントを自らの保護ドメインにアタッチ（接続）またはデタッチ（切断）すること

で、動的なアクセス権の変更を実現する[12]。さらに、クライアント・サーバーモデルにおいては、「ポータル（Portal）」と呼ばれる安全な入り口を通じて、クライアントスレッドがサーバーの保護ドメイン内で実行権限を得るRPC（リモートプロシージャコール）メカニズムが提供された[12]。

3.3 ソフトウェアベースの分離と型安全性（Singularityモデル）

ハードウェア機能（MMUやページテーブル）に依存せず、プログラミング言語の型システムと静的検証によってSASOS空間内の保護を実現した画期的なアプローチが、Microsoft Researchによる**Singularity**プロジェクトである[13]。
Singularityは「ソフトウェア分離プロセス（Software-Isolated Processes: SIPs）」という概念を導入した。すべてのSIPは、単一の物理および仮想アドレス空間内で実行されるが、コンパイル時およびインストール時にC#などの安全な言語の型システム（MSIL: Microsoft Intermediate Language verifier）によってメモリ安全性が静的に証明される[13]。型システムによって「あるSIPから別のSIPのオブジェクト（メモリ領域）へのポインタの生成や参照変更が不可能である」ことが数学的に保証されるため、ハードウェアによる隔離を完全に省略できる[16]。
Singularityはこの閉じたオブジェクト空間を維持するため、プロセス間の通信を「契約ベースのチャネル（Contract-Based Channels）」を通じたメッセージパッシングのみに限定し、ポインタの共有を厳格に禁止している[16]。

4. 歴史的および代表的なSASOSプロジェクトの分析

SASOSの概念は理論上の産物に留まらず、数多くの研究プロトタイプや一部の商用システムとして結実してきた。以下に代表的なシステムを詳述する。

4.1 IBM i（旧 OS/400）

商用環境で最も成功を収め、現在も稼働し続けているSASOSの実装が、IBM System/38の系譜を継ぐIBM i（旧称: OS/400）である[4]。このシステムは「すべての要素はオブジェクトである」という強い設計思想に基づいており、単一レベルストア（SLS）と128ビットの太いポインタ（Thick Pointer）を採用している[8]。
IBM iの技術的根幹は、TIMI（Technology Independent Machine Interface）と呼ばれる抽象化された中間命令セットアーキテクチャにある[19]。アプリケーションは基盤ハードウェアではなくTIMIに対してコンパイルされ、OSが実行時にネイティブな機械語に翻訳する。この透過的なプロセスにより、1980年代の48ビットCISCプロセッサ向けに書かれたバイナリが、再コンパイルなしで現代の64ビットPowerPC RISCプロセッサ上で動作するという驚異的な後方互換性を実現している[8]。メモリ保護は、ハードウェアレベルでのタギング（タグ付きポインタ）によって強制され、ポインタ内のタグビットが正当でなければアクセスがハードウェアによって即座に拒否される仕組みとなっている[22]。

4.2 Mungi

分散環境の全ノードにまたがる巨大な単一アドレス空間の実現を目指したのが、オーストラリアの**Mungi**である[6]。L4マイクロカーネルアーキテクチャ上に構築されたMungiは、システムコールによる明示的なI/OやIPCを排除し、すべてを仮想メモリのマッピングとページフォルト

処理に委譲した「極めて純粋なSASOS」である[6]。
ネットワーク越しのノード間通信でさえ、リモートアドレス空間のページに対するフォルトと分散共有メモリ（DSM）のメカニズムによって解決される。Mungiは、プロセス生成や共有オブジェクト操作において、当時の商用UNIX（IRIXやLinux）を大幅に凌駕するパフォーマンスを記録した。具体的には、タスク生成速度やオブジェクト指向データベース（OO1）のベンチマークにおいて1桁以上の高速化を実証し、SASOSが専用ハードウェアなしでも効率的に動作することを証明した[5]。

4.3 Nemesis

ケンブリッジ大学などで1990年代後半に開発された**Nemesis**は、マルチメディア処理やQoS（Quality of Service）保証に特化したSASOSである[9]。当時のマイクロカーネル（Machなど）は、高帯域幅のオーディオやビデオストリーミング処理において、度重なるコンテキストスイッチとデータコピーによるペナルティが原因でQoSを保証できないという限界を抱えていた[26]。
Nemesisは単一アドレス空間を共有することで、広帯域データのコピーを完全に不要にし、コンテキストスイッチに伴うTLBのフラッシュペナルティを劇的に削減した。さらに、OSの機能をカーネル内部ではなく、共有ライブラリとしてユーザー空間に配置する「垂直統合型（Vertically-structured）」のアーキテクチャを採用した。これにより、カーネルは最低限の多重化と保護のみを担当し、ページングやスケジューリングといった機能の制御権をアプリケーション側に直接委ねる（ストレッチ抽象化）ことに成功した[26]。

4.4 Sombrero

Sombreroは、クラスタ環境全体をNUMA（Non-Uniform Memory Access）マルチプロセッサのように見立てることを目指した分散SASOSである[28]。Sombreroの特徴は、プロセッサ内に保護ハードウェアエミュレーション（RPLB: Range Protection Lookaside Buffer）を導入した点にある[28]。これにより、スレッド単位での保護ドメインと、通常の関数呼び出し命令による暗黙のドメイン横断（コンテキストスイッチ）をサポートした。開発者は、リモートノードのAPI呼び出しを、ローカルのサブルーチン呼び出しと全く同じコストと直感で記述することが可能となり、分散システムにおけるソフトウェア開発の複雑さを大幅に低減した[28]。

5. SASOSの決定的な利点とパフォーマンス特性

SASOSアーキテクチャの導入は、システムパフォーマンス、データ共有メカニズム、そしてソフトウェア開発のパラダイムにおいて、従来のマルチアドレス空間OSでは到達不可能なレベルの利点をもたらす。

5.1 データ共有の極限的な簡素化とゼロコピー

従来型のマルチアドレス空間では、プロセス間で複雑なデータ構造（ポインタで相互参照されたグラフ、ツリー、データベースインデックスなど）を共有する場合、そのままポインタ領域を渡すことはできない。仮想アドレスの解釈がプロセスごとに異なるためである[1]。結果として、送信側でデータを直列化（シリアライズ）し、パイプやソケットを通じて送信し、受信側で逆直列化するか、あるいはポインタの書き換え（ポインタスウィズリング）を行う必要があ

り、著しいパフォーマンスの低下を招いていた[2]。
SASOSでは、すべてのアドレスがグローバルに一意（Context-independent）であるため、ポインタの意味はシステム内のどこでも共通である[2]。これにより、巨大で複雑なデータ構造をコピーや直列化なしでそのまま他プロセスと共有することが可能になる。これは、CADツール、大規模な統合開発環境、分散オブジェクト指向データベースといった、データ共有が頻発する高負荷なアプリケーションにおいて決定的なパフォーマンス優位性をもたらす[2]。

5.2 コンテキストスイッチとIPC（プロセス間通信）の劇的な高速化

一般的なOSでのプロセス切り替えには、ページテーブルディレクトリの物理的な切り替えや、CPUのTLB（Translation Lookaside Buffer）のフラッシュが伴う[5]。TLBがフラッシュされると、その後のメモリアクセスでページテーブルウォークが頻発し、システム全体のパフォーマンスが著しく低下する。
SASOSでは、全スレッドが同一のページテーブル（仮想アドレス空間）を共有しているため、スレッド間の切り替え時にページテーブルの切り替えやTLBのフラッシュを行う必要が一切ない[31]。保護ドメインの権限のみを変更するだけで済むため、切り替えコストは極めて小さくなる[33]。さらに、共有メモリを介したRPC（リモートプロシージャコール）も、パラメータをレジスタで渡してドメインを切り替えるだけの単純な関数呼び出しとして実装できるため、従来のメッセージパッシングに比べてレイテンシが大幅に削減される[6]。また、すべてのデータが常に同じ仮想アドレスでアクセスされるため、マルチアドレス空間OSで問題となる「仮想キャッシュのエイリアシング（同一データが異なる仮想アドレスでキャッシュされ不整合を起こす問題）」も構造的に回避される[6]。
以下の表は、ソフトウェアベースのSASOSであるSingularityと、従来のハードウェア保護を用いるOSとの間で、基本的なプロセス操作のCPUサイクル数を比較したものである。ハードウェア境界の切り替えを排除したSASOSの圧倒的な効率性が示されている。

OS名 (アーキテクチャ)	API呼び出し (サイクル)	スレッド切り替え (サイクル)	IPC Ping/Pong (サイクル)	プロセス生成 (サイクル)
Singularity (SASOS/SIP)	80	365	1,040	388,000
FreeBSD	878	911	13,300	1,030,000
Linux	437	906	5,800	719,000
Windows	627	753	6,340	5,380,000

出典: Singularity: Rethinking the Software Stack (AMD Athlon 64 3000+ 1.8 GHzでの測定)[18]。

5.3 ソフトウェアアーキテクチャの簡素化

単一レベルストアにより、開発者は「揮発性メモリ」と「不揮発性ストレージ」の違いを意識して明示的なファイル入出力をコーディングする負担から解放される[9]。変数が割り当てられたアドレスを永続オブジェクトとして指定するだけで、背後の仮想メモリマネージャが透過的にページングを行うため、アプリケーションのコードベースは劇的に削減され、バグの発生源となる複雑な状態管理が排除される[7]。

6. SASOSの欠点と普及を阻んだ技術的課題

これほどまでに理論的優位性を持つ強力なアーキテクチャであるにも関わらず、IBM iを除いてSASOSがメインストリームの汎用デスクトップ・サーバーOSとして普及しなかった背景には、いくつかの致命的な理由と技術的課題が存在する。

6.1 POSIX互換性の欠如とfork()プリミティブの矛盾

UNIXおよびLinuxソフトウェアエコシステムの根幹を成すPOSIX規格において、新しいプロセスを生成する標準的なシステムコールは fork() である。fork() の本来のセマンティクスは、「親プロセスのメモリ空間の完全なコピー（同じアドレス配置）を持つ、新しい独立した仮想アドレス空間を子プロセスとして作成する」ことである[35]。
SASOSは定義上「システム内に単一のアドレス空間しか存在しない」ため、fork() を文字通り実装することは根本的に不可能である[35]。例えば、親プロセスがアドレス 0x1000 に重要なデータを持っていた場合、子プロセスも自身のメモリとして 0x1000 を参照しようとする。しかし、SASOSにおいて 0x1000 はグローバルに1つのデータしか指し示せないため、親と子のデータが衝突してしまう。これを回避するために子プロセスのデータをアドレス空間内の全く別の場所（例えば 0x8000）にコピーすると、今度はデータ内に含まれる絶対ポインタ（0x1000 への参照）が古い領域を指したままとなり、プログラムが確実にクラッシュする[35]。この根本的な不整合が、レガシーソフトウェアのSASOSへの移植を絶望的に困難なものにしていた。

6.2 プライベートな静的データと動的リンクの競合

単一アドレス空間では、コードの共有（共有ライブラリの展開）が極めて容易である。全プロセスが同一アドレスにロードされたライブラリ関数を呼び出せばよいためである[7]。しかし一方で、「共有したくないデータ」の隔離には複雑な機構が必要となる[7]。
標準Cライブラリを複数のプログラムが使用する場合、ライブラリ内部に存在する「静的変数（.bss や .data セグメント、グローバル変数など）」は、プログラムのインスタンス（プロセス）ごとに別々の独立した値を持つ必要がある（Privatization）[7]。SASOSではすべてのアドレスがグローバルであるため、単純に共有ライブラリを実行すると、全プロセスが同じ静的変数のアドレスを読み書きしてしまい、システム全体の状態が破壊される。これを回避するために、OSとコンパイラはグローバルオフセットテーブル（GOT）やベースレジスタを用いた間接参照を利用し、スレッドコンテキストごとに静的データのポインタを切り替えるという追加の実行時オーバーヘッドを払う必要がある[7]。これにより、SASOSの最大の利点であった「アドレスの直接的かつ一意な解釈」という原則が損なわれる結果となった。

6.3 アドレス空間の枯渇と分散ガベージコレクション

64ビットアドレス空間は事実上無限に見えるが、オブジェクトの動的割り当てと解放を長期間繰り返すうちに、広域的なメモリ断片化（フラグメンテーション）が発生する可能性がある[5]。また、一度削除されたオブジェクトの仮想アドレスを再利用する場合、他のプロセスがその古いアドレスを保持したまま（ダングリングポインタ）新しいオブジェクトに不正アクセスする重大なセキュリティリスクが生じる[6]。
Mungi等のシステムでは、再利用されるアドレスには常に新しいランダムパスワードを割り当てることで統計的にセキュリティを確保しているが、根本的な解決策としては、システム全体で不要なオブジェクトを特定・回収するガベージコレクション（GC）が必要となる[6]。Angelのような分散SASOSにおいて、ネットワーク上の全ノードにまたがるポインタの参照グラフをたどり、孤立した永続オブジェクトを特定する分散GCを構築・維持することは、理論的にも実装上も極めて複雑であり、スケーラビリティの足かせとなった[10]。

6.4 ハードウェア・ソフトウェア基盤への過度な依存と移行コスト

多くのSASOSモデルは、プロセッサによる特殊なハードウェア保護機能（RPLBやタグ付きポインタなど）を前提としているか、Singularityのようにシステム全体を特定の型安全言語（C#やRustなど）で書き直すことを要求する[4]。現代のITインフラは、CやC++で書かれた膨大な「メモリ安全でない」レガシーコードの蓄積の上で稼働している。これらをすべてSASOSのパラダイム（安全な言語への移行、ポインタ演算の厳格な制限、独自のIPCモデル）に適応させるためのサンクコストは天文学的であり、実用化への大きな障壁となった[15]。

7. 現代コンピューティングにおけるSASOSのルネサンス（再評価）

過去の学術的な実験に留まるかに見えたSASOSの概念は、現在、クラウドネイティブ環境、エッジコンピューティング、および最新のプロセッサ拡張アーキテクチャの文脈において、形を変えて静かなルネサンス（復活）を迎えている。

7.1 WebAssemblyとUnikernelの融合による軽量隔離

FaaS（Function as a Service）やサーバーレスアーキテクチャでは、何千もの細粒度な関数をミリ秒以下のコールドスタート時間で立ち上げ、サーバー上に高密度に配置する必要がある。しかし、従来のコンテナや仮想マシンは、基盤となるOSカーネルの肥大化したオーバーヘッドと、マルチアドレス空間のコンテキストスイッチに伴うコストを免れず、応答時間の限界に直面している[39]。
この課題に対するブレイクスルーとして注目されているのが、**WebAssembly (Wasm)** と **Unikernel** の組み合わせである[39]。Wasmランタイム（Wasmtimeなど）は、サンドボックス化された環境で多数のWasmモジュールを「単一のホストプロセス（単一アドレス空間）」内で実行する。Wasmのメモリはモジュールごとに「線形メモリ」として独立して割り当てられ、静的検証によってメモリ外へのアクセスが厳格に防止される[42]。これは、Singularityが「SIPs」と型安全性を用いてハードウェアMMUの隔離をソフトウェアで代替した手法の、現代的かつ言

語非依存な再構築であると見ることができる[14]。Unikernel技術と組み合わせた「単一アドレス空間内での複数コンパートメント隔離」は、OSのアタックサーフェスを数千のシステムコールから数十に抑え込みつつ、コンテナの数十倍の起動速度とニアネイティブな実行効率を両立させる次世代インフラとなりつつある[39]。

7.2 CHERIアーキテクチャによるハードウェア強制保護

SASOSの「すべてのデータがフラットに番地付けされている」という特性は、セキュリティにおいて「もろ刃の剣」であった。C言語のようなメモリ安全でない言語でポインタの演算ミスが発生した場合、マルチアドレス空間であれば自プロセスがクラッシュするだけで済むが、SASOSにおいては理論上、システムカーネルや他のアプリケーションのデータを破壊するリスクを孕んでいた[31]。
この根本的な脆弱性をハードウェアレベルで完全に塞ぐ技術が、ケンブリッジ大学とSRI Internationalが開発した **CHERI（Capability Hardware Enhanced RISC Instructions）** アーキテクチャである[4]。CHERIは、メモリアクセスの際に通常の整数ポインタではなく、ハードウェアレベルで圧縮された「制限境界付きポインタ（Tagged Pointers）」を使用する機能を提供する[35]。このポインタにはアクセス可能な上限・下限のアドレスとパーミッションが埋め込まれており、CPU自体がポインタの偽造や範囲外アクセスを物理的にブロックする。CHERIを用いることで、単一のアドレス空間内であっても、関数やモジュール単位での極めて細粒度なメモリ保護（Compartmentalization）がハードウェアの速度で実現可能となった[44]。

7.3 μFork：SASOSとPOSIXの架け橋

前述した fork() の根本的な不整合問題に対処するため、近年 **μFork** と呼ばれる先進的な研究が提案された[35]。μForkは、CHERIのハードウェアメモリタギングを利用して、メモリ空間内に絶対ポインタが存在するかどうかを確実かつ高速に識別する。
fork() が呼び出されると、システムは親のメモリ領域を単一アドレス空間内の別の領域にコピーしつつ、CHERIタグを頼りにポインタのアドレス値を新しい領域のオフセットに合わせて動的に再配置（Relocation）する[35]。さらに、PIC（位置独立コード）を活用して再配置が必要な絶対ポインタの数自体を最小限に抑えることで、従来のFreeBSDの fork よりも3.7倍高速（わずか54μs）という驚異的な軽量性を達成した[35]。これにより、SASOSが長年抱えていた「POSIXレガシーソフトウェアとの互換性」という最大の障壁を打破する道筋が示されたのである。

8. 結論

単一アドレス空間オペレーティングシステム（SASOS）は、論理アドレスと物理アドレスの境界を取り払い、システム全体のメモリとストレージをフラットな単一の名前空間として統合する極めて野心的なアーキテクチャである。
本研究の分析が示す通り、このモデルはアドレス変換と保護を直交化することにより、コンテキストスイッチのオーバーヘッドを劇的に削減し、ゼロコピーでの複雑なデータ共有を実現する。Mungi、Opal、Nemesis、IBM i といった先行研究や商用実装は、タスク生成、IPC、マルチメディアストリーミング、直交的永続性の各分野において、マルチアドレス空間を採用する従来型OSを圧倒的に凌駕するポテンシャルを示してきた。さらに、Singularityは、その隔離メカニズムをソフトウェアの型安全性で実現することで、ハードウェアMMUの制約すらも超越可

能であることを実証した。
しかしながら、その広範な普及を阻んだ最大の壁は、POSIX互換性（特にアドレス空間の複製を前提とする fork() プリミティブ）の欠如や、動的リンク時のプライベート静的データの管理といった、既存のソフトウェアパラダイムとの決定的な不整合であった。これらの課題により、長らくSASOSは特定用途の組み込みシステムや研究プラットフォームに留まることを余儀なくされてきた。
今日、コンピューティングの潮流がサーバーレス、エッジコンピューティング、関数ベースのマイクロサービスへと急速に移行する中、ミリ秒単位のコールドスタートと超高密度なプロセス実行基盤が切実に求められている。このような状況下において、仮想メモリの重厚な切り替えコストを排除したSASOSの基本設計は、WebAssemblyや軽量コンテキスト（lwCs）として形を変え、再び表舞台に立ちつつある。加えて、CHERIに代表される新しいハードウェアタギング技術や μFork といった革新的な互換性レイヤーの登場により、レガシーソフトウェアと次世代アーキテクチャを隔てていた厚い壁は崩れ去ろうとしている。
SASOSは決して「過去の失敗した概念」ではなく、ハードウェアの進化とクラウドコンピューティング環境の成熟を待っていた「早すぎた正解」であったと結論づけられる。計算資源の極限的な実行効率と、ハードウェアレベルの厳格なセキュリティの両立が求められるこれからの時代において、SASOSの基本原則は、未来のシステムソフトウェア設計において不可欠な強固な基礎となるだろう。

引用文献

1. Single Address Space Operating System - C2 Wiki, https://wiki.c2.com/?SingleAddressSpaceOperatingSystem
2. Sharing and Protection in a Single Address Space Operating System - University of Washington, https://homes.cs.washington.edu/~levy/opal/opal-tocs.pdf
3. https://en.wikipedia.org/wiki/Single_address_space_operating_system#:~:text=In%20computer%20science%2C%20a%20single,address%20space%20for%20all%20processes.
4. Single address space operating system - Wikipedia, https://en.wikipedia.org/wiki/Single_address_space_operating_system
5. Single address space operating system - Grokipedia, https://grokipedia.com/page/Single_address_space_operating_system
6. Implementation and Performance of the Mungi Single-Address-Space Operating System - cgi .cse. unsw. edu.a u, https://cgi.cse.unsw.edu.au/~reports/papers/9704.pdf
7. Linking Programs in a Single Address Space - USENIX, https://www.usenix.org/events/usenix99/full_papers/deller/deller.pdf
8. Everything is memory? : r/osdev - Reddit, https://www.reddit.com/r/osdev/comments/1u4ync5/everything_is_memory/
9. The Mungi Single-Address-Space Operating System, https://trustworthy.systems/publications/papers/Heiser_EVRL_98.pdf
10. (PDF) Single Address Space Operating Systems - ResearchGate, https://www.researchgate.net/publication/2570875_Single_Address_Space_Operating_Systems

11. The OPAL Operating System Project - University of Washington, https://homes.cs.washington.edu/~levy/opal/
12. CS 736 Reviews - Spring 2015: Sharing and Protection in a Single Address Space Operating System., https://pages.cs.wisc.edu/~swift/classes/cs736-sp15/blog/2015/02/sharing_and_protection_in_a_si.html
13. Singularity (operating system) - Wikipedia, https://en.wikipedia.org/wiki/Singularity_(operating_system)
14. Singularity - Microsoft Research, https://www.microsoft.com/en-us/research/project/singularity/
15. Deconstructing Process Isolation, https://cs.uwaterloo.ca/~brecht/courses/702/Possible-Readings/oses/singularity-deconstructing-process-isolation-mem-system-perf-2006.pdf
16. An Overview of the Singularity Project1 - Microsoft, https://www.microsoft.com/en-us/research/wp-content/uploads/2005/10/tr-2005-135.pdf
17. Singularity vs. the Hard Way Part 1 | Semantic Scholar, https://pdfs.semanticscholar.org/d8eb/38c1e39eedf645cbd9e99f555836188556db.pdf
18. Singularity: Rethinking the Software Stack | Microsoft Research, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/osr2007_rethinkingsoftwarestack.pdf
19. IBM AS/400 - Wikipedia, https://en.wikipedia.org/wiki/IBM_AS/400
20. Below MI - IBM i for Hackers - GitHub Pages, https://silentsignal.github.io/BelowMI/
21. Features that separate the AS/400 from common computer systems, https://try-as400.pocnet.net/wiki/Features_that_separate_the_AS/400_from_common_computer_systems
22. The Linux Kernel on iSeries - LWN.net, https://lwn.net/2001/features/OLS/pdf/pdf/iseries.pdf
23. Capability Addressing - Mark Funk's Blog, https://mrfunk.info/?page_id=5
24. [PDF] Single Address Space Operating Systems | Semantic Scholar, https://www.semanticscholar.org/paper/Single-Address-Space-Operating-Systems-Wilkinson-Murray/8c65379022a3d45b6a07771618919f318195b742
25. Mungi — Single-Address-Space OS - Trustworthy Systems, https://trustworthy.systems/projects/OLD/mungi/
26. Nemesis (operating system) - Grokipedia, https://grokipedia.com/page/nemesis_operating_system
27. Nemesis, https://www.cl.cam.ac.uk/research/srg/netos/projects/archive/nemesis/
28. The Sombrero Single Address Space Operating System Prototype A Testbed for Evaluating Distributed Persistent System Concepts and Implementation., http://sombrero.rightnet.com/pdpta2000.pdf
29. Sombrero Project, http://sombrero.rightnet.com/
30. The Sombrero Distributed Single Address Space Operating System Project - USENIX,

https://www.usenix.org/publications/library/proceedings/usenix-nt98/full_papers/poster_skousen/skousen.pdf
31. Address-Space Multiplexing revisited on AMD64/x86_64 - ITEC-OS Start, https://os.itec.kit.edu/2769.php
32. Predictable Virtualization on Memory Protection Unit-based Microcontrollers - School of Engineering & Applied Science - The George Washington University, https://www2.seas.gwu.edu/~gparmer/publications/rtas18mpu.pdf
33. Super Fast Single Address Space Operating System : r/osdev - Reddit, https://www.reddit.com/r/osdev/comments/1sr9224/super_fast_single_address_space_operating_system/
34. Light-Weight Contexts: An OS Abstraction for Safety and Performance - USENIX, https://www.usenix.org/system/files/conference/osdi16/osdi16-litton.pdf
35. µFork: Supporting POSIX fork Within a Single-Address-Space OS - arXiv, https://arxiv.org/html/2509.09439v2
36. µFork: Supporting POSIX fork Within a Single-Address-Space OS - arXiv, https://arxiv.org/pdf/2509.09439
37. Single address space or private address spaces? - National Open Access Monitor, Ireland, https://oamonitor.ireland.openaire.eu/rpo/rcsi/search/publication?pid=10.1145%2F504409.504410
38. (PDF) An Overview of the Singularity Project - ResearchGate, https://www.researchgate.net/publication/236160050_An_Overview_of_the_Singularity_Project
39. WebAssembly and Unikernels: A Comparative Study for Serverless at the Edge - arXiv, https://arxiv.org/html/2509.09400v1
40. HybridServe: Adaptive WebAssembly-Container Runtime Selection for Edge Serverless Computing - Kyungyong Lee, https://leeky.me/publications/hybridserve.pdf
41. Nanvix: A Multikernel OS Design for High-Density Serverless Deployments - arXiv, https://arxiv.org/pdf/2604.11669
42. WASM Meets Unikernels: Next-Gen Cloud Native Deployments - sanj.dev, https://sanj.dev/post/wasm-meets-unikernels-secure-cloud-deployments/
43. Dynamic linking in WebAssembly: Architecture and Performance Evaluation - Helda - University of Helsinki, https://helda.helsinki.fi/server/api/core/bitstreams/f5038e2b-43ef-4a83-ada3-5cb6da0c9e9b/content
44. CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor - Department of Computer Science and Technology | - University of Cambridge, https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-961.pdf
45. MMU-less Systems - CHERI Alliance, https://cheri-alliance.org/who-we-are/working-groups/mmu-less-systems/