プロジェクト名：Capability-Based Microkernelによるセキュアなユーザーレベルメモリ管理システム
申請者名：伊組烈火

【提案プロジェクト詳細】

1. なにをつくるか

概要

Object-Capability ModelによるCapability-Based MicrokernelをベースにしたOSと，それを用いたユーザーランドメモリ管理システムを提案する.
この提案はセキュアかつ柔軟で安定したシステムの構築を可能とするものであり，ユーザーランドソフトウェアの互換性を保ちつつ，最適化されたメモリ管理システムを提供可能である.
これにより，現代のプログラム技術におけるコンテナやWasm, Hypervisorのような仮想化機構に対してより高レベルの最適化を提供することが可能になる.
本提案で開発するものは以下の3つである:

- A9N : Capability-Based Microkernel
- KOITO : A9Nをコアとするos
- ULMM Server : KOITO上で動作する，ユーザーレベルメモリ管理サーバー
    - SLAB Allocator
    - Buddy Memory Allocator

コンピューターの民主化によるIoTの拡大

近年，技術革新によるコンピュータの民主化は大きなムーブメントを引き起こしている.
ムーアの法則で予見されたように，CPUに含まれる面積当たりのトランジスタ数は指数関数的に増大した．半導体技術の進歩はよりコンパクトで低価格なコンピューターの発達を促し，21世紀の携帯端末は 従来持っていた電話機能のみならず， インターネットを介した広範なネットワークへの接続や高度なコンピューティング能力を獲得し，より賢いものへと進化している.
これによりIoTという概念が一般化し，生活と密接かつ重大な結合を果たしている現在，家の施錠システムまでもがネットワークに参加するようになり， 一つのモノに対して与えられるオペレーションが高度なものへと変貌を遂げている．そのニーズに応えるため，ありとあらゆるシステムにOSが搭載されるようになった．この現象は，コンピュータの民主化によるIoTの拡大と言えるだろう.

カーネルとOS

コンピュータの民主化が進む中で，カーネルとOSはますます重要な役割を果たすようになった.
カーネルは，システムとハードウェアを接続するための，プリミティブなAPIサブセットを提供するものである．このカーネルを基に，特定のユースケースを満たすためのソフトウェア群が一つのパッケージとなり，OSが構成される.
そのため，異なるOSでも同一のカーネルを使用しているケースが多数存在する．代表的なもので言えば，数多のLinuxディストリビューションやAndroidはLinuxカーネルが用いられている．また，Apple社のiOSやmacOSにはXNUカーネルが用いられている．カーネルが提供するAPIが共通であるため，各OSを構成するソフトウェアは高い互換性を持つ.

モノリシックなカーネル

従来のカーネルでは，一般的なシステム要求に対応するため，カーネル自体にその機能を含めてしまうというモノリシックな構造を持っている.
例えば，デバイスドライバやファイルシステム，ネットワークに接続するためのプロトコル・スタックなどはカーネル内部へ組み込まれる．このアプローチは高速な動作を可能とするが，コンポーネントの拡大に伴いカーネルが肥大化してしまうという問題を持っている．また，構成要素である各コンポーネントのコードを書き換えるたびにカーネルの再ビルドが必要となる．コードベースが十分に小さいうちは管理が可能であるが，開発の進行につれて保

守性と柔軟性を失ってしまい，誰もシステムの全容を把握できなくなる．この状態ではカーネルの汎用性が低下し，機能要件の小さな目的特化型のOSは，カーネルの提供する不要な機能に依存することとなる．これにより，同じカーネルを組み込みシステムとデスクトップシステムの両方で使用するのは困難になる．
また，カーネル内に存在するコンポーネントは当然ながらカーネル空間上で動作する．そのため，1つでも不安定なコンポーネントが存在すると，その影響が最重要であるはずのカーネル全体に波及してしまう．
簡単に言えば，カーネルの担う仕事があまりにも多すぎるのである．

マイクロカーネル

モノリシックなカーネルの持つ問題を解消するため，マイクロカーネルというアイデアが提案された．
カーネルは最小の特権操作のみをユーザーに提供し， モノリシックカーネルにおける多くのカーネル内コンポーネントをユーザーランドへ移動することで，文字通り”マイクロ”にするというものである． 多くのマイクロカーネルは割り込みやタイマー割り込み，IPCなどの切り離せない操作のみを提供する．
切り離されたコンポーネントはIPC機構によってクライアント-サーバーモデル[Tanenbaum, 2007]を実現し，協調動作を行う．
この設計は先述したモノリシックカーネルの問題点を解消しているが，完璧ではない．IPCはユーザーからカーネル， またその逆方向へのコンテキストスイッチを伴うため， 単なる関数呼び出しで済んでいたものに対してオーバーヘッドが課せられる．しかしながら，L4マイクロカーネルの作者であるJochen LiedtkeがIPCを注意深く設計することで高速なIPCを実現可能なことを示したことや[Liedtke, 1995]，技術革新によってコンピューターが高速化したことに伴い，現代においては十二分に実用可能なものとなった．そのため，ここでは問題としないことにする．

機構と方針の分離

マイクロカーネルが採用したアプローチは，機構と方針の分離[Levin et al, 1975]という設計原則に一般化できる．実装詳細である機構は，より抽象を表す方針を示してはならないという原則である．マイクロカーネルにおいては，機構がIPCや割り込みを操作するための仕組みであり，方針はユーザーであるクライアントの操作詳細を表す．

カーネルによるスケジューリング方針の決定

機構と方針の分離原則を適用すると，スケジューリングやメモリ管理システムの方針も分離できそうに思えるが，自体はそれほど単純ではない．
多くのマイクロカーネルにおいて， スケジューリングのための機構であるクロックドライバーはカーネル内部に存在する．何故なら，常にプリエンプティブマルチタスクを行うカーネルにとって，クロックサーバーからのIPCコストが無視できないものであるからだ．
通常のクライアント-サーバー間のIPCコストは全体から見たときにそれほど大きいものではない．しかし，クロックは100-500hzという高頻度で発生するため，処理の大半をIPCが占める可能性が大きいためである．
そのため，妥協案として実行可能コンテキストごとの優先度設定機能が提供されている．

カーネルによるメモリ管理方針の決定

では，メモリ管理システムではどうだろうか．
ユーザーランドでメモリ管理方針を決定可能なマイクロカーネルは存在するが， 多くの場合カーネル内に統合されている．
では，何故多くのケースにおいてメモリ管理方針が移動できないのだろうか．それは，セキュリティ上の問題に起因する．
ユーザーが直接ページテーブルを操作できる場合，自分の所有していない物理アドレスをマップすることや，他コンテキストが持つアドレス空間を破壊することが可能となり，コンテキスト間の分離が損なわれてしまう．
MINIXのように，VMサーバーのような限定されたプロセスのみが操作可能とすることも現実的ではない．このようなユーザーに対する無条件の信頼は， 悪意のあるサーバーや攻撃者に対する安全性が保証できないからである．このような設計は，与えたいアクセス権限以上の操作を可能としてしまうため，最小特権の原則[Saltzer, 1974]に反している．

Capability

スケジューリング方針はハードウェア制約によって決定されるため，変更は困難である．一方，メモリ管理方針は，最小特権の原則に基づいた適切なセキュリティ機構を導入することにより， ユーザーレベルで決定することが可

能になる.

Object-Capability Model

その機構の実現には, Object-Capability Modelを使用したCapability-Based Securityを導入する.
Object-Capability Modelとは, オブジェクトに対する操作権限を基にしたセキュリティモデルである.
ユーザーは特権的操作を行うために, 自分の持つCapabilityの指定子と, そのCapabilityに対する操作を指定する必要がある. ユーザーは自分の持たないCapabilityに対する操作手段は持ち得ず, 割り当てられた権限以上のことは行うことができない. これは最小特権の原則を満たす.

A9N Microkernel

A9N Microkernelは, これまで記した問題に対する解法を単一のマイクロカーネルとして統合したものであり, 私が以前から開発しているものである.
A9Nは, システム起動時にユーザーに許可するCapabilityを, UnixやLinuxのInitに該当する実行可能コンテキストAlphaへ渡す. このAlphaが与えられたCapabilityをどう使用するかは, A9NをコアとするOSの実装者に一任されている. ユーザー空間上で動作するAlphaこそがOSの性格を決定するものであり, カーネルよりも上位レイヤー部分の柔軟性を決定する.

シンプルなインターフェース

A9Nはすべての特権操作をIPCメカニズムによって行う. 具体的には, IPCシステムコールの第一引数としてCapabilityの識別子であるDescriptorを指定することによって行われる.
この機構により, A9Nのシステムコールはipc()とyield()の2つのみという, 究極的にシンプルなものとなる.

カーネルオブジェクト作成時の問題

カーネルは当然ながらメモリを使用する. 例えば, ある実行可能コンテキストを作成するには, カーネル内にそれ専用のメタデータを作成する必要がある.
カーネルからメモリ管理方針をユーザーに移行した場合, 問題が発生する. ユーザー側で割り当てたメモリをカーネルへ渡した後, メタデータの情報が漏洩する可能性があるということだ.
例えば, ある物理アドレスをカーネルへマップし, その情報を渡す場合, 割り当て先のアドレスを作成したオブジェクトのメタデータを示すデータ構造へキャストすると, カーネル内で使用する情報がそのまま手に入ってしまう. これはユーザーによるシステムの改竄が容易にし, 信頼すべきセキュリティ機構を崩壊させてしまう.

メモリの抽象化

上記の問題を解決するためには, メモリ空間と操作権限をCapabilityとし, ユーザーが物理メモリアドレスを知ることが出来ないような設計にする必要がある.
そのために, 汎用メモリCapability : Genericと, Genericをカーネルオブジェクトへ変換する処理を導入する.
メモリを必要とするようなカーネルオブジェクトの作成と操作は, GenericをConvertして新たなCapabilityを作成し, そのCapabilityに対して操作を行うことで完了される.
物理メモリフレームもまたCapability : Frameであり, ユーザーのメモリ要求によってカーネルオブジェクトと同様に作成される.
アドレス空間へのマッピングはFrameのみが許可されているため, 所持していない物理メモリフレームは当然ながらマッピングすることはできない. また, カーネルオブジェクトはGenericからConvertすることでしか作成できないため, 情報漏洩を起こすようなこともない.
GenericをGenericへConvertすることにより, より小さなGenericを作成することも可能である.

システムのセキュアかつ柔軟な変更

メモリをCapabilityによって抽象化したことにより, 従来のマイクロカーネルでは困難であった安全かつ柔軟なメモリ管理方針の決定が可能となる. A9Nはカーネル内ヒープを持たず, すべてをユーザーレベルで管理するため, 特定のメモリ割り当てアルゴリズムに依存しない.
最初に作成したカーネルオブジェクトがそのままシステム終了時まで寿命を持ち, 途中で動的に作成されるようなことがない組み込みシステムの場合は, 一切の記録なしにGenericをただConvertするだけの単純な割り当てポリシーで良い.
カーネルオブジェクトの生成と破棄を繰り返すようなデスクトップシステムの場合は, サーバーとしてユーザーランドで動作するMemory Management Serverを作成し, メモリ割り当て要求はそのサーバーに対して行うことで

最適な動作を可能とする．また，新しいメモリ割り当てアルゴリズムを試したい場合も新たなサーバーとして作成するだけで完結し，処理に応じて動的にサーバーを切り替えることも容易である．
これはメモリ管理アルゴリズムに限らず， 従来モノリシックカーネルが担ってきた機能の殆どを切り替えることができる．

Capabilityのセキュアな**Addressing**

Capabilityの指定子であるDescriptorはメモリのアドレスを指し示すものではない． これらの指定はある種間接的なものとなっている．
ある実行可能コンテキストのグループがProcessであり， Processはメモリ空間とRoot Capability Nodeという Capabilityを格納するためのNodeを持っている．NodeもまたCapabilityであるため，ディレクトリのような構造を持つことができる．
入れ子になったCapabilityを指し示すための機構がDescriptorによる指定であり， ページテーブルと仮想アドレスの対応とよく似た構造を持つ．DescriptorはNodeごとにindexへ分解され，終端まで探索される．この方式による探索速度はO(log n)であり，非常に高速である．
Capabilityそのものを表すメタデータはカーネルの内部に隠蔽されているため，不正なアクセスは不可能である．
また，各Processの持つCapability Nodeの構造は異なるため，別プロセスにおいて有効なDescriptorのコピーは不可能である．

Capabilityに対する**CRUD**

Capabilityは，それを保持する親Nodeから

- remove
- revoke
- move
- copy

の操作が可能である．この機構により，カーネルが最初に起動するユーザーランドサーバーであるAlpha Serverは，子に自分の持つCapabilityを委譲し，それぞれが最小の特権を持つサーバーへ分散することが可能である．

実体を持たない**Capability**

Node内のCapability SlotにCapabilityを表す情報が格納される． Capabilityの実体を指す個別のポインタと，Slot固有のSlot Dataである．
Slot Dataの存在により，GenericやFrameなどの個別にインスタンスを持たせたくないCapabilityに対する処理を単一のManager Objectへ委譲するだけで完結できる．

HAL

カーネルはハードウェアとの接続にHALを使用する．ハードウェア依存部分はHALに集約されているため，異なるアーキテクチャやボードに移植することが容易となる．デバイスドライバーはDMAを行うためこの限りではないが， ユーザーから見えるカーネルのAPIは等しいため，殆どのサーバーはアーキテクチャを変更し再ビルドするだけで動作する．
これにより， デスクトップ環境向けやインターネット・サーバー環境向けのメモリ管理システムを組み込み向け環境へ移植することや，その逆を変更なしに動作させることが可能となり，各システム間の統合を可能とする．
本提案ではカーネルのターゲットアーキテクチャをx86_64に絞るが， 基本的にサーバーはアーキテクチャ非依存である．

KOITO

KOITOはA9Nマイクロカーネルをコアとするシステムである．
実装するユーザーレベルメモリ管理システムはKOITO上に起動するサーバーによって動作する．また，システムのテストや各種アプリケーションの実装もKOITO上で行う．
本提案に対するKOITOの比重はそこまで大きくなく， あくまでもカーネルと各サーバーを接続するために実装される．

ライブラリ

liba9n

liba9nは，Cによって実装する，A9Nカーネルを呼び出すための最もプリミティブなAPIライブラリである．IPCや，それをラップする各Capabilityへの操作はこのライブラリを用いて行う.
Capabilityによるカーネルコールと，　それに必要な共通の定数やアーキテクチャ非依存のデータ構造を提供する. liba9nにおける実際のIPC呼び出し部分は，アセンブリで記述されるためハードウェアに依存する.

koito-libc

koit-libcは，Cによって実装する，KOITO上で動作させるソフトウェアが個別に使用するlibc互換ライブラリであり，これによるUnixソフトウェアの移植を目指す.
libcには標準ユーティリティライブラリとしての側面と，システムコールラッパーとしての側面があるが，koito-libc内のシステムコールに該当する関数呼び出しは内部で機能を持つサーバーにメッセージを送信し，　処理を委譲することによって行う.

ULMM Server

ULMM　Serverは，　本プロジェクトのメインであるCapabilityを用いたユーザーレベルメモリ管理を実装するサーバーである. Capabilityを管理・分割し，要求されたメモリ関連呼び出しに対して適切に割り当てを行う.
このサーバーで実装するのは，

- カーネルオブジェクトをAllocateするためのSLAB Allocator
- ページフレーム単位でユーザーにメモリを割り当てるためのBuddy Allocator

の2つである.

SLAB Allocator

ULMM ServerのSLABはカーネルオブジェクトを効率的に割り当てるためのAllocatorである.
Capabilityのタイプと個数をリクエストすることによってキャッシュが作成され，Generic内の領域を予約する.
各Capabilityのサイズはカーネルにより2の累乗バイトであると規定されているため，　Genericに対するConvertの比較的薄いラッパーとなる.

Buddy Allocator

ULMM ServerのBuddy Allocatorは，Buddy Systemを用いてGenericを管理し，要求されたページフレームを割り当てるAllocatorである.
Genericは派生したオブジェクトをすべて破棄しなければ再利用ができない．また，Genericのサイズを後から知ることもできないため，　GenericからカーネルオブジェクトをConvertするタイミングで管理情報をユーザーレベルで保持しなければならない.
そのため，SLABとは異なり，このサーバーは管理情報分のFrameを自分自身に割り当て，そこに管理情報を書き込み，Allocation時のヒントとしなければならない. そのため，SLABよりも実装コストは高くなる.

2. どんな出し方を考えているか

提案の実装であるA9N Microkernel + OS + Userlevel Memory Allocatorは，MITライセンスのもとGitHub上に公開するものとする. また，それらを適切に使用するためのマニュアルも公開する.
マニュアル作成用の組版エンジンとしてTypstを用い，Typstコードも公開する.
本システムのユーザー拡大のためにはそれらは必須事項である. セキュアを謳うクローズドなシステムには透明性が欠けていることや，オープンなシステムであっても使用方法が理解できなければ意味が無いためである.

3. 斬新さの主張、期待される効果など

先行研究

Capability-Based　Microkernelであり，　カーネル内ヒープを持たない先行事例としてseL4[seL4]が挙げられる.

seL4もまたlibseL4[libsel4](libsel4allocman, libsel4vka)によるユーザーレベルメモリ割り当てシステムを持っているが, 本提案のようなAllocatorよりも低レベルを示す,生のCapability操作に近いものである.
また, seL4のAPIは本質的にハードウェアに依存しており, アーキテクチャ固有のPage Tableを自らマップしなければならないなど, APIレベルでの互換性が考慮されていない. このことからseL4はA9Nよりも柔軟度が低いといえる.
本提案はメモリ管理機構そのものの柔軟性を謳っているため, 新規性が認められると考える.
また, 前年度未踏ITプロジェクトであるCaprese[Caprese]は, Capabilityを主軸とするマイクロカーネルの開発がメインであり, 一見領域が同一な提案に思えるが, 本提案におけるカーネル部分はほぼ完成していて, メインとなる部分はあくまでもユーザーレベルメモリ管理システムであるため, 全く別の提案であるといえる.

仮想化システムにおける2レベルのセキュリティと最適化

近年のセキュアな仮想化システムにはContainer, WASM, Hypervisorなどが挙げられるが, 本提案はそのいずれとも親和性が高い.
カーネルレベルのセキュリティとそれらの持つセキュリティは, 冗長なようでそうではない. 各システムが存在するレイヤーごとにセキュリティ機構が必要であり, むしろ必要なものである.
また, システムごとに最適なメモリ割り当て方針を決定できることにより, 従来のものと比較して高速なものを提供することが可能となる. これにより, 本提案のユーザーはよりセキュアで高速な実行環境を手に入れることができる.

4. 具体的な進め方と予算

開発を行う場所

自宅

使用するハードウェア

メイン開発用計算機 : Apple MacBook Pro 2023 (AArch64 : Apple M2 Pro, 16GB RAM, 512GB SSD)
サブ開発用計算機 : 自作デスクトップPC (x86_64 : AMD Ryzen7 2700, 32GB RAM, 512GB SSD)

使用するソフトウェア/ツール

OS

基本的にDocker Container上で作業を行う.
Host : macOS Ventura 13.5
Host (Sub) : Windows 11 23H2
Guest : Debian 11 Bullseye on Docker : Apple Siliconの場合, Rosetta2による仮想環境を使用する

Language

C++20 : メインで使用
C11 : ブートローダー, ASMとのグルーコード, liba9n, koito-libcの標準実装に使用
ASM (x86_64) : A9N Microkernel, liba9nのハードウェア依存部分に使用

Build

Clang
Clang++
lld
NASM
EDK2 : ブートローダーのビルドに使用する

Test

Google Test : テスト駆動開発を行う

Debug
LLVM Binutils
QEMU

Manual
Typst

開発線表

	6月	7月	8月	9月	10月	11月	12月	1月	2月
A9N	Capabilityの実装			レガシーコードの置き換え	すり合わせ	ドキュメント整備			発表準備
KOITO			koito-libcの実装		Unixソフトウェアの移植				
ULMM	SLABの実装		Buddy Systemの実装			リファクタリング			

開発に関わる時間帯と時間数

現在学生ではなく, また就職もしていないため, フルタイムで一日当たり8時間, 週平均48時間程度作業を行う予定である.

予算内訳

活動時間 (h)	予算 (円)
1440	2,880,000

5. 提案者の腕前を証明できるもの

本提案で使用するCapability-Based Microkernelであ A9Nは, 一般社団法人未踏による未踏ジュニアに採択され作成したものである[A9N]. このカーネルは当初, HALを備えたマイクロカーネルであったが, 採択期間の終了後からCapability-Basedなものへと書き換えを行っている. このA9Nが高い評価を得たため, スーパークリエータの称号を獲得した[スーパークリエータ]. また, 未踏会議2024への展示を行った.
このカーネルを作成するにあたり, オペレーティングシステムの理論, C, C++, ASM(x86_64)やDockerによるエコシステムの深い理解を得た.
得られた知見はScrapbox[Scrapbox]へ出力し, x86_64アーキテクチャやseL4に関する記事を作成した, また, 採択期間で得られたカーネルを書くための手法をZenn[カーネルことはじめ]の記事として作成し, Ideas Trendの2位となった.
Github[Github]ではPython + DDDによるRestfulなOAuth2 ROPC Serverの実装や, Object-Orientedなアプローチを用いたNESエミュレータの実装を行っている. A9Nは, Capability Systemが完成次第Publicなものとなる.

6. プロジェクト遂行にあたっての特記事項

大学入学にあたる年齢だが, 受験においては春AOを考えているため現在はどこにも所属していない.

7. IT以外の勉強、特技、生活、趣味など

IT以外の趣味は多岐に渡る. というのも, 面白そうなことはまずやってみるというスタンスで生きてきた結果, 消えることなく上に積み上がっていったからである.
音楽に関しては, 制作面でいうと電子音楽の作曲[SoundCloud]や和声法, 対位法の学習を行っている. また, 演奏面に関してはピアノ, ギター, ベース, シンセサイザーなどを楽しんでいる.
グラフィックスに関しては, 美術解剖学を学んだり, パースを学んだり, デザインを行っている. 3DCGでいうとモデリングやVFXを行っており, 実写合成[実写合成動画]が評価されCPUメーカーのAMDからノベルティ贈呈を得たことがある.
考古学に関しては, 主に古代エジプトを嗜んでいる. ヒエログリフの解読が多少できる.

8. 将来のITについて思うこと・期すること

レガシーなコードは将来のユーザーを苦しめる. 特に, カーネルやOSのような形を変えにくいものに関しては, 永久に更新できない負の遺産が何重にも積み重なることになる.
変えにくいからこそ, 慎重に設計しなければならないし, レガシーな部分を置換できるような設計でなければならないのに, 殆どのシステム・ソフトウェアについてその観点が欠けているように思える.
現代, それらのシステムは肥大し, 誰も全容を把握できなくなった. モノリスな構造はAPIの肥大を招き, もはや置き換えることは不可能になり, Attack Surfaceの拡大によるセキュリティ上のリスクはそれに比例して大きくなっていった.
今こそマイクロカーネルが復興のときである. 第1世代マイクロカーネルのMachはその遅さから普及しなかった. 普及のために脱マイクロカーネルへ舵を切り, BSDのコードと統合され, XNUへと名前を変えた.
第2世代マイクロカーネルはIPCの高速化によって実用に近いものとなり, seL4の前身であるL4-embeddedは世界中のiOS端末に搭載されるようになった.
そして第3世代マイクロカーネルが誕生した. seL4やFiasco.OC, Novaの台頭により, セキュアかつ高速なマイクロカーネルが完全に実用的になったのである. コンピューターアーキテクチャとソフトウェアアーキテクチャの進化が融合し, 仮想化技術に迎合する新たなシステムの時代が幕を開けたのである.
これらは長い研究と苦労の産物である. 先人たちがそうであったように, 第4世代のマイクロカーネルに向けた歩みを止めてはならないと強く感じる. そして, それを実現し得る若い世代に期待したい.

9. 参考文献

[Tanenbaum, 2007] Tanenbaum, A S.: “モダン オペレーティング システム 原著第二版”, (水野忠則・太田剛・最所 圭三・福田晃・吉沢康文 訳), ピアソン・エデュケーション・ジャパン, 2007.
[Liedtke, 1995] Liedtke, J.: “Improving IPC by Kernel Design” Proc. 14th Symp. on Operating Systems Principles, ACM, pp. 237-150, 1995.
[Levin et al, 1975] Levin, R., Cochen, E.S., Corwin, W.M., Pollack, F.J., and Wulf., W.A.: “Policy/Mechanism Separation in Hydra”, Proc. 5th Symp, on Operating Systems Principles, ACM, pp. 132-140, 1975.
[Saltzer, 1974] Saltzer, J.H., and Schroeder, M.D.: “Protection and Control of Information Sharing in MULTICS”, Commun. of the ACM, vol.17, pp.388-402, 1974
[seL4] https://sel4.systems/
[libsel4] https://github.com/seL4/seL4_libs
[Caprese] https://www.ipa.go.jp/jinzai/mitou/it/2023/gaiyou_tn-4.html
[A9N] https://jr.mitou.org/projects/2023/a9n
[スーパークリエータ] https://jr.mitou.org/projects/
[Scrapbox] https://scrapbox.io/horizon2038/
[カーネルことはじめ] https://zenn.dev/horizon2k38/articles/6e0ea0570c98fb
[Github] https://github.com/horizon2038
[SoundCloud] https://soundcloud.com/user-571923716
[実写合成] https://twitter.com/horizon2k38/status/1270246466217336832