AIPL による Xinu 風小型 OS の記述と実装

Actor-first 言語による教育用・研究用 OS カーネルの設計

Yasushi Kodama Repository Report

2026 年 5 月 4 日

概要

本稿は、Xinu のような小型 OS を、actor-first 記述言語 AIPL によって記述し、実装する研究構想を述べる。Xinu は教育用 OS として、プロセス、スケジューラ、メッセージ通信、デバイス、システムコールを小さなコードベースで提示する点に特徴がある。一方、AIPL は actor、message、future、select、capability を中核に持つ言語であり、OS 内部の構成要素を actor として記述するのに適している。

本稿では、Xinu 風 OS の構成要素を AIPL actor に対応づけ、プロセス管理、mailbox、スケジューリング、デバイスドライバ、システムコール、メモリ管理、割り込み処理をどのように記述するかを検討する。さらに、AIPL の型システムを用いてシステムコール引数、デバイス capability、プロセス間 message、kernel service の interface を検査する設計を提案する。提案の中心は、OS を monolithic な C 手続き群としてではなく、型付き actor 群による協調的な kernel として再構成する点にある。

目次

1　はじめに

小型 OS は、計算機システムの理解において重要な研究・教育対象である。商用 OS は巨大で複雑であり、初学者や研究者が OS の全体像を把握するには適さない場合が多い。Xinu はこの問題に対して、小さなコードベースでプロセス、スケジューラ、メッセージ、デバイス、システムコールを提示する教育用 OS として設計された。

しかし、古典的な小型 OS は C 言語による低水準実装を前提としている。C による実装はハードウェア制御には適しているが、OS 内部の component 間 interface、message の型、capability、並行性の構造を明示するには弱い。特に、プロセス、デバイス、通信、kernel service の境界は、関数呼び出しと共有データ構造の中に埋もれやすい。

AIPL は actor-first 言語である。actor は内部状態を閉じ込め、外部とは message によって通信する。この性質は、OS の構成要素を service として分離し、interface を型で管理する設計と相性が良い。本稿では、Xinu 風の小型 OS を AIPL で記述することで、教育的に見通しがよく、研究的には型付き OS service orchestration を実験できる環境を構築する構想を示す。

■本稿の貢献

- Xinu 風小型 OS の構成要素を AIPL actor に対応づける。
- AIPL による process、scheduler、mailbox、device、syscall の記述例を示す。
- OS kernel に型が必要な理由を、service interface と capability の観点から論じる。
- AIPL 実装 OS の実行アーキテクチャと段階的 bootstrapping 方法を提案する。
- 教育用 OS、研究用 OS、AIOS kernel への発展可能性を整理する。

2　Xinu 風小型 OS の要件

本稿でいう Xinu 風小型 OS は、Xinu の完全な再実装ではなく、Xinu が持つ教育的性質を継承した小型 OS を指す。すなわち、次の要件を満たす。

1. 小さい kernel core を持つ。
2. process または task の生成・終了・切替を持つ。
3. priority または round-robin scheduler を持つ。
4. process 間 message passing を持つ。
5. device を抽象化する。
6. syscall interface を持つ。
7. console、timer、memory、network など最小 service を持つ。

Xinu の重要な特徴は、OS の構成要素が比較的明快で、システム全体を読み切れることである。AIPL 版でもこの性質を維持する。したがって、最初の目標は高性能 OS ではなく、読める OS、変更できる OS、検査できる OS である。

3　AIPL による OS 記述の基本方針

AIPL では、OS 内部の service を actor として定義する。

$$Kernel = SchedulerActor + ProcessTableActor + DeviceActor^{*} + MemoryActor + IPCActor$$

各 actor は method interface を持ち、message により連携する。たとえば scheduler は次のような method を持つ。

$$\texttt{scheduler.ready} : Pid \to Unit$$

$$\texttt{scheduler.next} : Unit \to Pid$$

AIPL による OS 記述の基本方針は以下である。

- kernel service は actor として定義する。
- process は actor または actor によって管理される lightweight task とする。
- syscall は user actor から kernel actor への typed message とする。
- device driver は device actor として表現する。
- interrupt は event message として kernel actor に配送する。
- capability により、actor が呼び出せる kernel service を制限する。

4　全体アーキテクチャ

AIPL-Xinu の全体像を表 1 に示す。

表 1: AIPL-Xinu の kernel actor

Actor	役割
BootActor	初期化順序を制御し、kernel service を起動する
SchedulerActor	ready queue、priority、time slice を管理する
ProcessTableActor	pid、状態、context、mailbox、resource を管理する
IPCActor	send、receive、select、reply を管理する
MemoryActor	heap、page、allocation、ownership を管理する
TimerActor	tick、sleep、timeout、preemption event を生成する
ConsoleActor	console 入出力を提供する
DeviceManagerActor	device registration と driver lookup を管理する
NetworkActor	packet 入出力と protocol stack への入口を提供する
CapabilityActor	actor/process が持つ権限を検査する

この構成では、kernel は単一の巨大な制御構造ではなく、typed actor の集合である。ただし、すべてを分散 actor として実装すると overhead が大きい。したがって実装上は、同一 address space 内で lightweight actor runtime を使い、message delivery を最適化する。

5 プロセスモデル

AIPL-Xinu では、process を二通りに表現できる。

1. process 自体を actor とする方式。
2. process は kernel が管理する execution context とし、user program を actor-like task として扱う方式。

第一の方式は概念的に単純である。各 process は mailbox を持つ actor であり、message によって起床する。

```
class UserProcess {
  method main() {
    print("hello from process");
  }

  method signal(msg) {
    print("signal: " + msg);
  }
}
```

第二の方式は OS 実装に近い。ProcessTableActor が context を保持し、SchedulerActor が実行権を切り替える。AIPL 記述は control plane を表現し、低水準 context switch は C または assembly の small trusted runtime に委ねる。

教育用・研究用としては、最初に第一の actor process 方式で実装し、次に第二の native context 方式へ拡張するのが現実的である。

6 スケジューラ

Xinu 風 OS の中心は scheduler である。AIPL では scheduler を actor として記述する。

```
class Scheduler {
  var ready = array_empty();

  method ready(pid) {
    ready = array_push(ready, pid);
  }

  method next() {
    var pid = array_get(ready, 0);
    reply(pid);
  }
```

```
}
```

実際には priority、sleep queue、blocked state、time slice を扱う必要がある。

$$ProcState ::= Ready \mid Running \mid Blocked \mid Sleeping \mid Dead$$

SchedulerActor の interface は次のように設計できる。

$$\texttt{spawn} : Program * Priority \to Pid$$

$$\texttt{ready} : Pid \to Unit$$

$$\texttt{block} : Pid * Reason \to Unit$$

$$\texttt{sleep} : Pid * Time \to Unit$$

$$\texttt{next} : Unit \to Pid$$

AIPL の型により、`Pid` と単なる整数、`Time` と priority、blocking reason を区別できる。これは OS 内部の小さな取り違えを防ぐうえで重要である。

7 IPC と mailbox

Xinu は message passing を持つ。AIPL は actor 言語であるため、IPC は自然に message passing として表現できる。

$$\texttt{send} : Pid * Message \to Unit$$

$$\texttt{receive} : Unit \to Message$$

$$\texttt{select} : Pattern^* \to Message$$

AIPL の `select` は mailbox から pattern に合う message を受け取る構文であり、OS の IPC primitive として使える。

```
select {
  case timer_tick(t) -> {
    send scheduler.tick(t);
  }
  case io_ready(dev) -> {
    send device_manager.ready(dev);
  }
  timeout 10 -> {
    send scheduler.idle();
  }
}
```

IPC に型を導入すると、message payload の不一致を防げる。たとえば timer tick と device ready event は同じ整数を含むかもしれないが、意味は異なる。型はこの意味の違いを interface に反映する。

8 システムコール

AIPL-Xinu では syscall を user actor から kernel actor への typed message として扱う。

$$Syscall = UserActor \rightarrow KernelActor : Request$$

例として、console write は次である。

$$\texttt{console.write} : String \rightarrow Unit$$

process spawn は次である。

$$\texttt{kernel.spawn} : Program * Priority \rightarrow Pid$$

syscall を通常の関数呼び出しではなく message として扱う利点は、kernel boundary が明示されることである。さらに capability check を syscall の前に挿入できる。

```
class Kernel {
  method spawn(program, priority) {
    var ok = capability_check(sender, "Process.Spawn");
    if (ok) {
      var pid = process_create(program, priority);
      reply(pid);
    } else {
      reply("permission denied");
    }
  }
}
```

現在の AIPL 実装では `sender` や戻り値型に動的境界が残るが、研究目標としては `Result Pid Error` のような型付き syscall result を導入する。

9 デバイスドライバ

小型 OS では device abstraction が重要である。AIPL-Xinu では device driver を actor として記述する。

$$DeviceActor = open/read/write/close/ioctl$$

```
class ConsoleDriver {
  method write(s) {
    lowlevel_console_write(s);
    reply("ok");
  }
```

```
  method read() {
    var c = lowlevel_console_read();
    reply(c);
  }
}
```

DeviceManagerActor は device 名から driver actor を引く。

$$\texttt{device.open} : DeviceName \to DeviceHandle$$

$$\texttt{device.write} : DeviceHandle * Bytes \to Result$$

device capability は重要である。すべての process が network や raw disk にアクセスできてはならない。AIPL の capability layer により、device access を actor interface として制限する。

10 メモリ管理

AIPL-Xinu の初期段階では、memory management を単純な heap allocator actor として実装できる。

$$\texttt{memory.alloc} : Size \to Ptr$$

$$\texttt{memory.free} : Ptr \to Unit$$

より進んだ段階では、ownership、region、page、capability を型に反映する。

$$Ptr\langle Region, Permission\rangle$$

ただし、AIPL 自体を安全な高水準 kernel 記述言語として使う場合、raw pointer を直接 user actor に渡さない方がよい。MemoryActor が buffer handle を管理し、process は handle を通して read/write を要求する。

$$BufferHandle \neq RawPointer$$

この設計により、小型 OS でありながら memory safety の研究対象になる。

11 割り込みと event

割り込みは低水準 hardware event である。AIPL で直接 interrupt handler を全て書くのは現実的でないため、最小 trusted runtime が interrupt を受け、AIPL runtime に event message として渡す。

$$Interrupt \to EventMessage$$

たとえば timer interrupt は TimerActor への `tick` message になる。

$$\texttt{timer.tick} : Time \to Unit$$

device interrupt は DeviceManagerActor または各 DeviceActor へ配送される。

$$\texttt{device.interrupt} : DeviceId * InterruptInfo \to Unit$$

この設計では、割り込み処理の低水準部分を小さく保ち、それ以外の制御ロジックを AIPL の型付き actor として記述できる。

12　型システムの役割

OS を AIPL で書く最大の利点は、kernel internal interface に型を与えられる点である。

12.1　Pid と整数の区別

C で書かれた小型 OS では、pid、priority、device id、error code が同じ整数型で表されがちである。AIPL では、これらを異なる抽象型として扱う。

$$Pid \neq Priority \neq DeviceId \neq ErrorCode$$

これにより、scheduler に device id を渡すような誤りを静的に防げる。

12.2　syscall interface の検査

syscall は kernel の公開 interface である。型付き syscall により、user process が不正な引数で kernel service を呼ぶことを防げる。

$$\texttt{spawn} : Program * Priority \to Result\ Pid$$

12.3　capability の明示

OS では、権限の管理が不可欠である。AIPL の capability 型は、actor が呼び出せる kernel service を制限する。

$$Capability ::= ConsoleWrite \mid SpawnProcess \mid OpenDevice \mid NetworkSend$$

$$\Gamma \vdash actor : CapabilitySet$$

syscall typing は capability を前提にできる。

$$\frac{ConsoleWrite \in Cap(actor) \quad \Gamma \vdash s : String}{\Gamma \vdash actor \to console.write(s) : Unit}$$

12.4 message protocol の検査

OS 内部には protocol がある。たとえば device read は、request、block、interrupt、wakeup、reply という順序を持つ。この順序を session type として記述できる。

$$read.request \rightarrow process.block \rightarrow device.interrupt \rightarrow process.ready \rightarrow read.reply$$

通常の型だけでは、この順序は保証できない。AIPL-Xinu では session protocol を runtime checker または静的 session type として導入することで、kernel protocol の破綻を検出する。

13 実装計画

AIPL-Xinu の実装は段階的に行う。

表 2: 実装段階

段階	内容
Stage 0	既存 OS 上で AIPL runtime として kernel actor 群を動かす
Stage 1	Scheduler、ProcessTable、IPC、Console を AIPL で実装する
Stage 2	Timer event、sleep、timeout、preemption simulation を追加する
Stage 3	DeviceActor と DeviceManagerActor を追加する
Stage 4	native context switch を trusted runtime に追加する
Stage 5	bare-metal または emulator 上で boot する最小 kernel へ移行する
Stage 6	capability type と session protocol checker を kernel 全体へ適用する

最初から bare-metal を目指すと、hardware initialization と toolchain の複雑さに研究の焦点が奪われる。したがって、最初は hosted kernel、すなわち既存 OS 上で動く AIPL kernel simulator として実装する。その後、trusted runtime を薄くして bare-metal へ近づける。

14 教育的意義

AIPL-Xinu は教育用 OS として有用である。理由は三つある。

1. OS component が actor として分離され、役割が見えやすい。
2. message と syscall の型が明示され、interface を理解しやすい。
3. scheduler、IPC、device、memory を段階的に差し替えられる。

従来の C ベース小型 OS では、低水準 pointer 操作や header 依存に学習者の注意が向かいやすい。AIPL-Xinu では、まず OS の概念構造を actor と message として学び、必要に応じて低水準実装へ降りることができる。

15 研究的意義

AIPL-Xinu は、単なる教育用 OS ではなく、型付き OS service orchestration の研究基盤になる。

- kernel service の actor 化
- syscall の型検査
- capability 型
- session type による kernel protocol 検査
- actor runtime と OS scheduler の対応
- AIOS kernel への拡張

特に AIOS との接続が重要である。将来の OS では、AI agent、model adapter、memory service、tool service が OS service として常駐する可能性がある。AIPL-Xinu は、それらを kernel actor または user actor として安全に接続する実験環境になる。

16 課題

AIPL-Xinu には多くの課題がある。

16.1 性能

actor message passing は関数呼び出しより overhead が大きい。kernel 内部の hot path、特に scheduler、interrupt、memory allocation では最適化が必要である。解決策として、同一 actor group 内 message の inline 化、静的 dispatch、zero-copy buffer handle が考えられる。

16.2 信頼基盤

すべてを AIPL で書くことは現実的でない。boot、interrupt entry、context switch、MMU setup などは trusted runtime として C/assembly で書く必要がある。研究上は、trusted computing base を最小化し、AIPL で書ける部分を最大化することが目標になる。

16.3 型と低水準資源

raw pointer、DMA buffer、device register、interrupt mask のような低水準資源を高水準型でどう表現するかは難しい。capability と linear type、region type の導入が必要になる可能性がある。

16.4 deadlock

kernel actor 同士が同期 `now` を循環的に呼ぶと deadlock が起こり得る。OS kernel では deadlock は致命的である。session type、effect type、lock ordering の型検査が必要になる。

17 評価方法

AIPL-Xinu の評価は、性能だけでなく、教育性と検査可能性を含めるべきである。

表 3: 評価項目

評価項目	内容
コード規模	kernel actor ごとの行数、trusted runtime の行数
理解容易性	学習者が scheduler、IPC、device を追跡できるか
型エラー検出	syscall 引数、pid/device id 取り違え、message shape mismatch を検出できるか
性能	context switch、message delivery、syscall latency
拡張性	新しい device actor や syscall を追加しやすいか
安全性	capability violation、protocol violation、deadlock risk を検出できるか

Xinu 風 OS としては、単純な shell、process spawn、message passing、timer sleep、console IO が動作することを最初の milestone とする。

18 結論

本稿では、Xinu のような小型 OS を AIPL で記述し実装する研究構想を提案した。AIPL-Xinu は、OS を型付き actor 群として構成する。Scheduler、ProcessTable、IPC、Memory、Device、Timer、Capability は actor として分離され、syscall は typed message として表現される。

この設計により、教育用 OS としては構成要素の役割が明確になり、研究用 OS としては型付き syscall、capability、session protocol、actor-based kernel orchestration を実験できる。Xinu が小型 OS の理解を容易にしたように、AIPL-Xinu は actor-first な型付き OS 設計の理解と実験を容易にすることを目指す。

今後の作業は、hosted AIPL kernel simulator の実装、scheduler と IPC の実測評価、capability 型の導入、session type による kernel protocol 検査、そして trusted runtime を薄くした bare-metal 版への移行である。