Thin Hypervisor on AArch64

森　真誠

2021/12/1　BitVisor Summit 10

背景

OSのRoot権限を渡せるような共用HPC環境を提供できるようにするプロジェクトがある

- Thin Hypervisorを用いて実現したい
- スーパーコンピュータ富岳相当の環境での実現が目標

Arm版BitVisorが欲しい！！

共用 HPC における管理者権限の利用を許す計算資源提供
https://kaken.nii.ac.jp/ja/grant/KAKENHI-PROJECT-21K17727/

Armの概要

- ARM Ltdによって設計されライセンスされるアーキテクチャ
- RISC系のアーキテクチャ
- アーキテクチャはARM v1からARM v9まで
- ARM v7までは32bit動作モード(AArch 32)のみ、ARM v8-Aから
  64bit動作モード(AArch64)が存在
- AArch32での命令長は32bitまたは16bit(Thumb)、AArch64では32bitのみ
  - 某サイトには「オペランドがついて可変長」とあるが、そのような命令はないと思う

ここではAArch64のみを対象とする

現在のハイパーバイザーの進捗状況

細部の作り込みなどがまだまだだがFX700で

1. UEFIシェルからローダーが起動
2. 最低限の処理以外はパススルーだけのハイパーバイザ本体をロード
3. UEFIに処理を返却
4. UEFIシェルのbootコマンドでLinuxを通常通り起動
5. Hypervisor Callが実行可能

まで完成

2022/02/11 追記: ソースコードを公開しました RIKEN-RCCS / MilvusVisor

FX700での動作画面

1.
```
Shell> FS0:EFI\BOOT\HBOOTAA64.EFI
ConvertPages: failed to find range 140000000 - 14004FFFF
Hello,world!
Allocated 0x99F60FF000 ~ 0x99FE0FF000
Expected descriptor_size: 40, but returned descriptor_size: 48.
Memory Map
 Key: 0x4B3
 NumOfDescriptors: 83
```

2.
```
CurrentEL: 2
Switched TTBR0_EL2 from 0x883FD000 to 0x99FE0FE000
SegmentInfo { virtual_base_address: 548682072064, phys
Initial Page Table Level: 0, Initial Shift Bits: 39
0x7FC0000000: Level0's Table Index: 0x0
0x7FC0000000: Level1's Table Index: 0x1FF
Allocated: 0x99FE0E6000
0x7FC0000000: Level2's Table Index: 0x0
Allocated: 0x99FE0E5000
Mapped 0x1000 Bytes(1 Pages)
```

3,4.
```
Setup EL1
Hello,world! from EL1
CurrentEL: 1
Return to UEFI.
remove-symbol-file /h
Shell> boot
```

5.
```
Kernel 4.18.0-348.2.1.el8_5.aarch64 on an aarch64

     login: [   63.637513] hv_tools: loading out-
[   63.643820] hv_tools: module license 'MIT' tai
[   63.649569] Disabling lock debugging due to ke
[   63.655152] hv_tools: module verification fail
[   63.665866] Starting hv_tools
Synchronous Exception!!
ESR_EL2: 0x5A00FFFF
FAR_EL2: 0x0
HPFAR_EL2: 0x0
Hypervisor Call: 0xFFFF
Return to EL1.
```

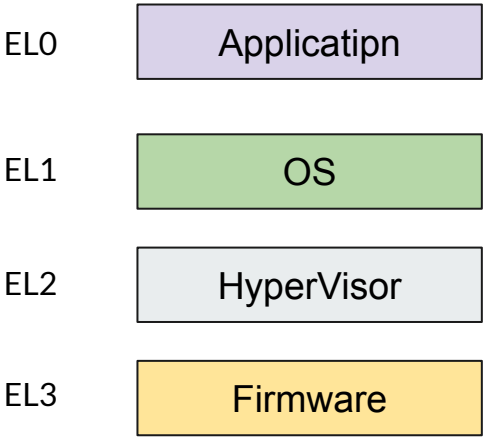
設計と方針について

(背景知識)AArch64の権限レベル

AArch64での権限レベルは権限が高い順に
EL3, EL2, EL1, EL0の4種類ある(x86_64と値が逆)

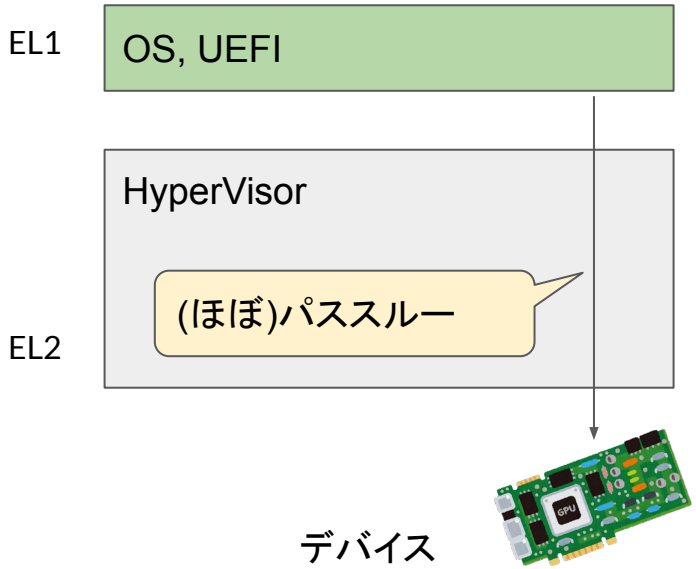
EL3: セキュアモニタ(ファームウェアなど)
EL2: ハイパーバイザー
EL1: スーパーバイザー(OS)
EL0: アプリケーション

(Secure-WorldやTrusted-OSなどあるがここでは割愛)



ハイパーバイザーの設計

- EL2に存在し、トラップ時のみ動作
- EL0&1からのハードウェアクセスはほぼパススルー
- デバイスのアクセスを一部トラップ（予定）
    - メモリ管理機構を用いて MMIOを使ってトラップ

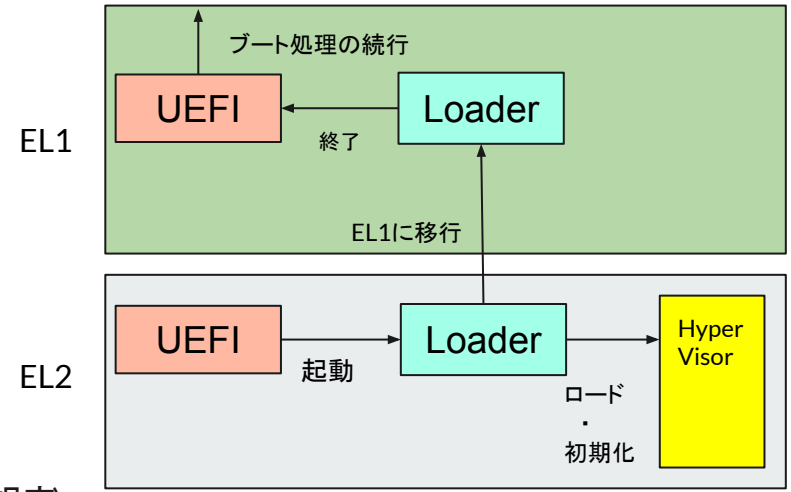


起動方針

ローダーがUEFIアプリケーションとして起動し
ハイパーバイザー本体をロードした後、EL1に移行して
UEFIに制御を返す

- OSを無改変で使用可能
- ゲストOSのブートローダを作らなくていい

(UEFI SpecificationではUEFIはEL1またはEL2で動作すると規定)



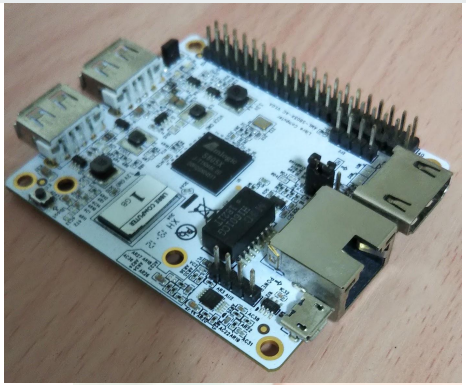
実装項目

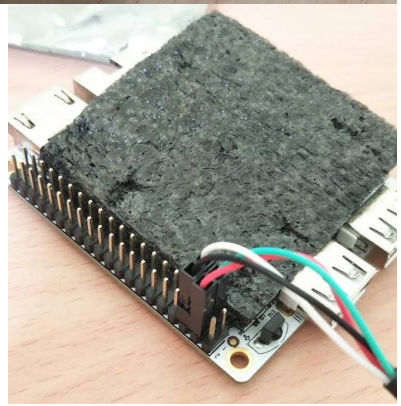
- メモリ管理
  - ハイパーバイザー本体用のページングテーブルの実装
  - 2段階ページングのセットアップ
- EL2からEL1への切り替え処理
  - 割り込みハンドラのセットアップ
  - UEFI環境のEL2からEL1へのシステムレジスタのコピー
- マルチコア対応

開発環境

- 開発言語としてRustを採用
  - 普段Rustを使用しているため
  - 所有権などで不正なメモリアクセスを防ぐことが可能
  - ベアメタル環境でもIteratorなどのモダンな構文を使用可能
- ブート環境はUEFI
- デバッグ環境
  - QEMU
  - AML-S805X-AC
  - FX700

(余談) AML-S805X-AC





$10でクラウドファンディングしていた

U-Bootでの起動に失敗するとUEFIが立ち上がってくる謎ボード

(UEFI搭載のArmデバイスが少ない中での貴重なデバイス)

半年遅れて到着しLinuxがうまく起動させられず役目を失い放置されていた

⇨ UART完備でUEFIアプリケーション開発では最適なので使用

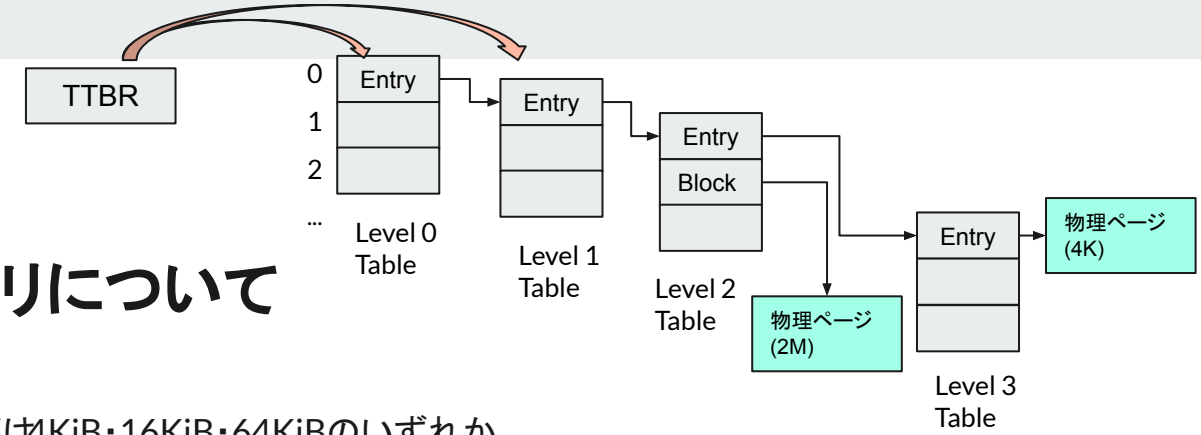
初期のデバッグではとても役立った

実装について

メモリ管理について

- メモリの確保
  - UEFIから128MB固定長で確保
  - 確保エリアをEfiUnusableAreaとして設定
- ハイパーバイザーを読み込むためのEL2用のページテーブルの用意
- EL0&1から見える物理メモリの制御のための2段階目のページテーブルの用意

AArch64の仮想メモリについて



- ページングの1ページのサイズは4KiB・16KiB・64KiBのいずれか
- ページングの段数は1~5段階まで変動
  - 最大仮想アドレスと1ページ単位のサイズに応じて自動で決定
  - 一番下の階層がLevel 3で上に行くほどLevel 2, 1, 0, -1とレベルが減少
- ページテーブルのサイズは4KiB(1エントリ64Bitより512エントリ)
- x86_64のHugeページのようなブロックエントリというエントリが存在
  - 広範囲を1エントリで一気にマップ可能でテーブル数を削減可能
  - ただしブロックエントリが使用可能な階層には制限あり

AArch64の仮想メモリについて

- EL1ではTTBR0_EL1というページテーブル(ユーザ用、ローメモリ)と TBBR1_EL1(カーネル用、ハイメモリ)という2種類のページテーブルが存在
  - EL2ではArm v8.1まではTTBR0_EL2のみ
- ページング関係の設定はTCR_EL2レジスタで設定
- RAMはメモリ空間の0からマップされているとは限らない
  - ROMやデバイスなどと共有しているため
  - 搭載しているメモリのサイズ分だけページテーブルを用意すればよいわけではない
  - 場合によってはメモリアドレス 1TiB付近にマップされていることあり

ハイパーバイザーでの実装

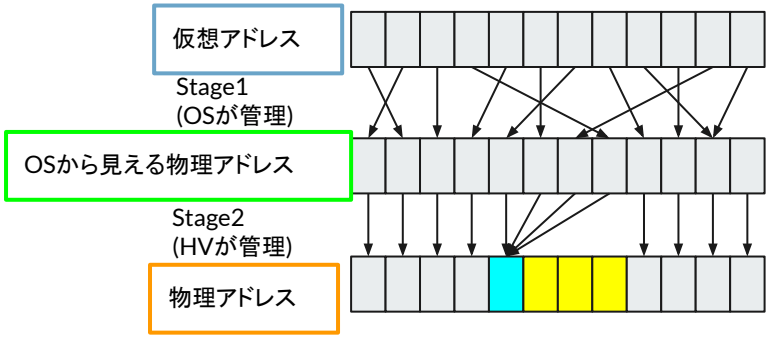
- UEFIの用意したページテーブルをコピーして使用
- 1ページあたり4KiB
- 適宜ブロックディスクリプタを設定し必要に応じて細分化する処理を搭載
- 3段以上のページテーブル(一番上のテーブルがLevel 1以上)に動的に対応
  - デバイスによって段数が違う
- ハイパーバイザー本体は0x7FC0000000に配置
  - ELFのリロケーション処理を避けるため固定アドレス
  - Level 1の511番目のエントリを使用するため

AArch64の2段階目ページング

- 2段目のページテーブルVTTBR_EL2と設定用レジスタVTCR_EL2が存在
- 同じくページテーブルサイズは4/16/64KiB、最大5段階のページテーブル
  - ページテーブルの段数は VTCR_EL2に設定
  - テーブル段数の限界はそれぞれのハードウェアに存在していて、それに応じて設定
- 最大中間物理アドレス・ページテーブルのサイズ・段数によっては自動的に最初のページテーブルの
  ページのサイズが8KiBや16KiBになることあり
  - Concatenated PageTable(気づくのに時間がかかった)
  - ページテーブルの段数を増やせば回避できるが、ハードウェアの制限によりできないことも

ハイパーバイザーでの 2段ページング

- 基本的にはストレートマッピング(IPA = PA)
  - 一部のデバイスへのアクセスをトラップするためマップしない箇所を作成（未実装）
- ハイパーバイザー本体へのアクセス・書き込みを防ぐため該当領域はOS側からアクセス不可に
  - 4KiB領域を一つ用意してハイパーバイザーの存在する物理メモリ領域へのアクセスを全てその1ページのアドレスへ変換
- ブロックエントリを利用してページテーブルを削減



EL1への切り替え処理

一部のレジスタを除いて、EL1/EL2/(EL3)には同じ名前のシステムレジスタが存在

環境をコピーするため、EL2のシステムレジスタの値をEL1レジスタにコピー
(ただしビットフィールドが一致していない場合も)

その後、ハイパーバイザー用の設定に切り替え

- 一部を除いてEL0&1での動作をトラップしないように設定
- 割り込みベクタなどもここでセット

切り替え時にコピーするレジスター覧

以下のシステムレジスタの値をEL2からEL1にコピー

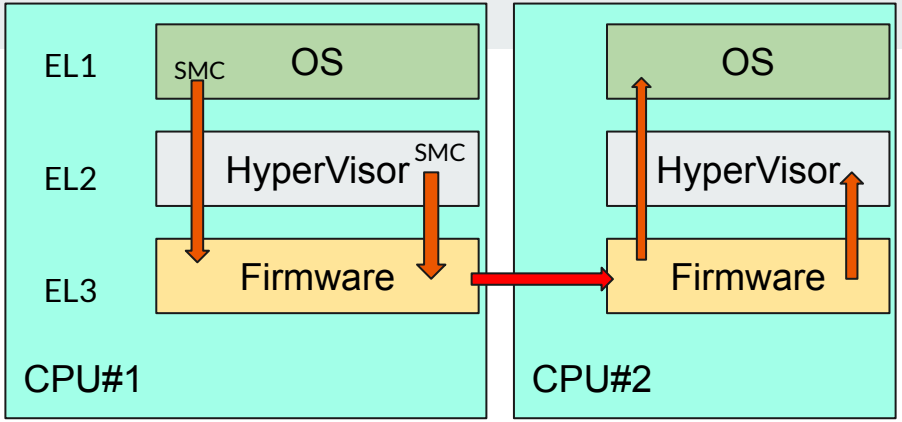
- CPTR_EL2 / CPACR_EL1: 各機能(SIMDやSVE)のアクセス制御に関する設定
- TTBR0_EL2 / TTBR0_EL1 : ページテーブル
- TCR_EL2 / TCR_EL1: MMUに関する設定
- VBAR_EL2 / VBAR_EL1: 割り込みベクタ
- SCTLR_EL2 / SCTLR_EL1: システム制御に関する設定

切り替え時に設定するレジスター覧

以下のレジスタの値を設定

- CNTHCTL_EL2: タイマー割り込みに関する設定
- HCR_EL2: ハイパーバイザー全体の設定
- TTBR0_EL2とVBAR_EL2を設定

Armでのマルチコア



ArmにはPower State Coordination Interface(PSCI)
という規格が存在

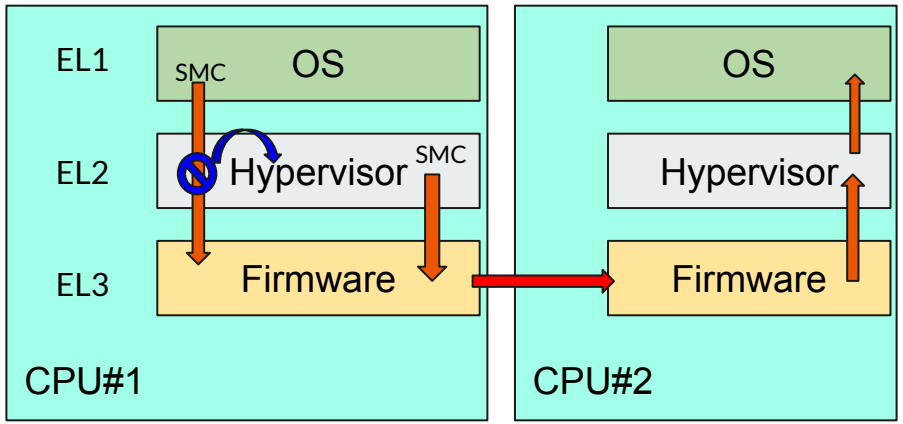
- PSCIは電源やCPUの制御を行う
- PSCIは主にEL3に存在するファームウェアを呼び出しで実現SMC)

各コアのONにはMPIDRレジスタの値が必要

MPIDRの値のリストはACPIのMADTか、DeviceTreeのCPUノードのregに存在

ハイパーバイザーでのマルチコア制御

1. SMCをトラップ(HCR_EL2.TSC == 1)をしてPSCIの各関数呼び出しをトラップ
2. CPU_ONの場合、OSのエントリポイントをハイパーバイザーのエントリポイントに書き換えてSMCを発行、それ以外はそのままSMC発行
3. 起動したコアのシステムレジスタの値を設定
4. EL2からEL1に移行するERETの戻り先をOSのエントリポイントに設定
5. OSが処理を続行



開発で苦労したところ

開発で苦労したところ

ページングの実装

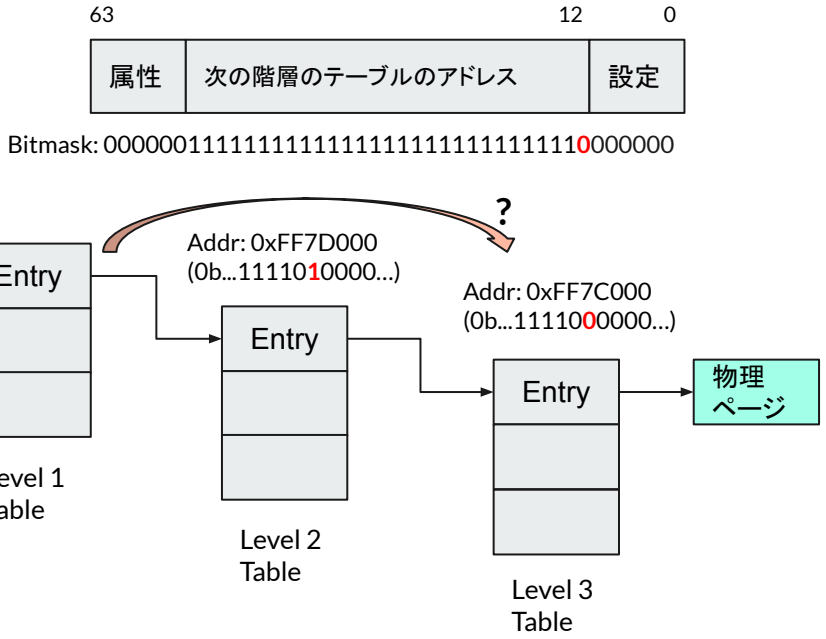
デバイスごとにRAM位置が異なる・ページテーブルの段数が違うなどで挙動が異なるため、コケる場所がQEMUや実機それぞれで全て異なったため、開発・デバッグに苦戦

更にプログラミングミスや仕様の理解不足で不思議なバグが発生

- 幻の(?)二段ページング
- コンパイルするディレクトリによって動いたり動かなかったりするブートローダ

幻の二段ページング

TCR_EL2の値と仕様を見る限りでは
ページテーブルが3段階層のはずなのに
どう見ても2段階層になっている



原因: テーブルのエントリから次階層の
テーブルアドレスを取り出すビットマスクが
1bit間違っていていて1個先のテーブルを参照していたため

(原因を突き止めた時絶叫し家族に怒られた)

(反省: ビットマスクを直打ちしない)

コンパイルするディレクトリによってコケるバグ

作業ディレクトリでコンパイルすると実行時に意図しないページフォルト割り込みが発生し
停止するが、~/Downloads/でコンパイルするとページフォルト割り込みが発生せず正しく動作する不具合に遭遇

- コンパイルするマシンによっても発生したりしなかったり.
- QEMUと実機(AML-S805X-AC)でも挙動が違う

コードのバグなどを調査するもエラー番号が微妙に変わるだけ

ディレクトリの違いによるバイナリと動作の変化

```
---- panic ----
Line 138 in src/main.rs
Message: Test
```

Rustはパニックのコードの位置情報をバイナリに埋め込んでいるが、

その際にフルパスで保存(/home/hoge-user/hypervisor/src/main.rs)

コンパイルするディレクトリによってバイナリファイルサイズが変化し、

ハイパーバイザー本体をRAM上に展開する際に必要なページ数が変化

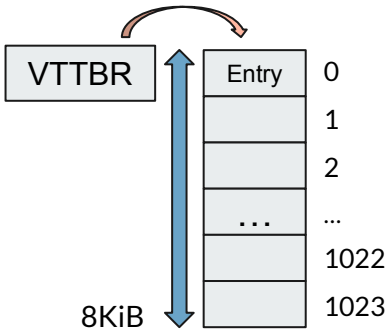
その後のメモリアロケーションで返されるメモリアドレスがコンパイルディレクトリによって変化

コンパイルするディレクトリによってコケるバグの正体

Concatenated PageTable

- 2段階目の最初のページテーブルにのみ適用
- 最大物理アドレスやページテーブルの段階に応じて**自動で**適用
- 1個のページテーブルのサイズが8KiB～64KiBまで変化
- テーブルの先頭アドレスはテーブルサイズで**要アライメント**

この仕様を理解しておらず4KiBのテーブルを作成していたため、
テーブルのアドレスが変化することで、アライメントが変化し
アライメント違反になったりならなかったりした



その他開発で苦労したところ

シリアルポートの実装

- x86の様にI/O Portの0x3F8とほぼ決まっているわけではない
- UEFIは教えてくれない

ACPIのSPCRテーブルかDeviceTreeからシリアルポートのアドレスを取得

(QEMUとFX700はSPCRテーブル、AML-S805X-ACはDeviceTreeを解析)

コントローラも複数存在し、送受信の制御方法もそれぞれ

まとめ

- 理化学研究所でAArch64向けのThin Hypervisorを開発
- 現状としてはUEFI用のローダーと何もしないHypervisorを実装
  - UEFIのページテーブルをコピーしたものを Hypervisor用のページテーブルとして使用
  - ハイパーバイザーへのアクセスを防ぐだけの IPA→PAのページテーブルを作成
  - システムレジスタのコピーや設定を行い、EL2からEL1へ移行
  - CPU_ONのSMCをトラップしオンデマンドのマルチコア初期化実装
- FX700のマルチコア環境上でHypervisorの起動後、Linuxが起動し動作

今後の方針

- メモリ管理の整備・充実
  - 物理メモリの開放は不可能
  - TLB最適化のために2段ページを最適なブロックサイズでメモリマップできるように
- Hypervisorの有無による性能への影響の調査
- デバイスアクセスのトラップの実装