Linux カーネルにおける Device Tree 情報のパース処理

坂本 貴史　Ubuntu Japanese Team

January 30, 2017

今日の内容

- Device Tree について、その概要の説明。
- Linux における Device Tree のアプリケーション、および Flattened Device Tree(FDT) の説明
- Linux の ARMv7-A/ARMv8-A アーキテクチャにおける、システムブート時の FDT の処理
- この資料は https://github.com/takaswie/presentations/ の 20161210 ブランチにあります

タイムテーブル

1. 導入
2. Device Tree の概要
3. Linux 4.9 のブート処理の要点
4. ブート時の Device Tree 関連処理

モチベーション

- ARM という名前で言及されるアプリケーションプロセッサはたくさんある
- そういったプロセッサを採用した製品がたくさん存在する
- それら製品向けに、Linux カーネルのポータビリティを維持する戦略に興味を持った

背景: ARM コアを集積したアプリケーションプロセッサの多様性

- ARMv7-A/ARMv8-A といったアーキテクチャを適用したプロセッサコアを集積したアプリケーションプロセッサがたくさんある。
- プロセッサコアの種類は同じだけれど、集積する機能を異にするバリアントが大量に存在する

背景: アプリケーションプロセッサの持つ機能の取捨選択

- 同一のアプリケーションプロセッサを採用した製品であっても、集積されたすべての機能は利用していない
- アプリケーションプロセッサの持つインターフェイスが「空いている」場合がある
- カーネルが初期化するべきサブシステムやドライバなどが、製品によって異なる
- 製品別にハードコードする方法も取れるが、コードの再利用性が損なわれるのに加え、コミュニティベースの開発にとってはデメリットとなる

背景: ハードウェア構成情報

- ハードウェアが持つ機能に関する情報
- PC/AT 仕様を適用した製品では、ACPI を採用している
  - システムから参照可能な特定のメモリ領域にハードウェア構成情報がある
- ARM という名前で言及されるアプリケーションプロセッサにおいては、ACPI の採用例は少なかった
  - ハードウェアに頼らずに、ハードウェア構成情報を表現する方法が必要

Linux カーネルと Device Tree

- Linux カーネルは、ハードウェア構成情報を表現するために Device Tree (DT) を採用している
- Device Tree のバイナリ表現として Flattened Device Tree (FDT) を規定している
- カーネルの初期化時に FDT をパースし、その内容に従い、初期化を行うコードを実行する

Device Tree

- Open Firmware 発祥
  - ファームウェア仕様の標準
  - PowerPC や Sparc が採用した
  - IEEE 1275:1994 で標準化された
- 2016 年に Device Tree の仕様をメンテナンスするためのコミュニティが発足
  - https://www.devicetree.org/
  - Linaro が中心メンバっぽい
- テキストによる表現のみを規定
  - ノード定義
  - プロパティ定義

Device Tree: ノード定義

ノード定義

```
[label:] node-name[@unit-address] {
    [properties definitions]
    [child nodes]
}
```

Device Tree: ノード名の表現

以下の 2 つは同じノードを表す

```
hoge/fuga/pero {
    ...
}

hoge {
    fuga {
        pero {
            ...
        }
    }
}
```

Device Tree: プロパティ定義

プロパティ定義

```
[label:] property-name = value;
```

Flattened Device Tree (FDT)

- Device Tree のバイナリ表現
- 多分 Linux 固有のバイナリフォーマット
- scripts/dtc/* にツールがある
  - ヘッダ
  - コンパイラ
  - デコンパイラ

Linux 4.9 のブート処理: 最後のほう

初期化処理を行ったコンテキストにおいて以下のように 2 つのカーネルスレッドを生成し、自身はスケジュール可能なタスクとなる。

```
(init/main.c)
->rest_init()
  ->kernel_thread(kernel_init)
  ->kernel_thread(kthreadd)
  ->schedule_preempt_disabled()
  (kernel/sched/idle.c)
  ->cpu_startup_entry()
    ->cpu_idle_loop()
```

もし自身がスケジュールされたら、それはその CPU 上に他のタスクがスケジュールされなかったことを意味する。これが CPU の idle 状態。

kthreadd()

2 つ目のカーネルスレッドのエントリー関数 kthreadd() はその名の通り、カーネルスレッドの生成要求を処理するサービスを実行する。

(kernel/kthread.c)

```
int kthreadd(void *unused)
{
    ...
    for (;;) {
        if (list_empty(&kthread_create_list))
            schedule()
        while (!list_empty(&kthread_create_list)) {
            ...
            create_kthread(create);
            ...
        }
    }
    ...
}
```

kernel_init()

1つ目のカーネルスレッドのエントリー関数である kernel_init() では、最終的にシステムの init プログラムを起動する。今時の Linux ディストリビューションだと、initramfs の'/init' プログラムが実行される。

(init/main.c)

```
static int __ref kernel_init(void *unused)
{
    ...
    if (ramdisk_execute_command) {
        ret = run_init_process(ramdisk_execute_command);
        if (!ret)
            return 0;
        pr_err("Failed to execute %s (error %d)\n",
               ramdisk_execute_command, ret);
    }
    ...
}
```

Linux 4.9 のブート処理: 中盤

先に述べた rest_init() は、start_kernel() の最後に呼ばれる。
start_kernel() は CPU やメモリなど、オペレーティングシステム成立に必須のシステム資源を初期化する。

```
(init/main.c)
->start_kernel()
    ...
    ->setup_arch()
    ...
    ->rest_init()
```

setup_arch()

- アーキテクチャ個別の初期化処理を行う。
- ARM の場合は以下のファイルに実装されている。
  - arch/arm/kernel/setup.c
  - arch/arm64/kernel/setup.c

Linux 4.9 のブート処理: 序盤

- ARM において、start_kernel() が呼ばれるまでは、アセンブラの世界が広がる
- 私はこの世界はそれほど詳しくない市井の普通のひとなので、この部分の説明はできません
- 解説には以下の知識がたぶん必要
  - アセンブリー言語 (GAS かあるいは NASM)
  - ARM アセンブラ
  - ARM コアの初期化処理
  - リンカースクリプトによる linux バイナリのレイアウト決定
  - 解凍直後の Linux カーネルイメージのメモリ上への配置状態

メモリへの FDT のロード

- ブートローダーの仕事
- Linux カーネルの圧縮イメージのロードと同時に行う

FDT の処理: ブート中盤

先述の setup_arch() の先頭部分で行う。FDT をパースして、その情報から CPU やメモリ、割り込みベクタなどの初期化を行う。

FDT の処理: ブート中盤: ARMv7-A の場合

Linux カーネルバイナリの.arch.info.init セクションにあるっぽい。

```
(arch/arm/kernel/setup.c)
->setup_arch()
  (arch/arm/kernel/devtree.c)
  ->setup_machine_fdt()
    ->メモリ上の FDT イメージの場所を、セクションを頼りに探す
  (drivers/of/fdt.c)
  ->unflatten_device_tree(of_root)
```

FDT の処理: ブート中盤: ARMv8-A の場合

ARMv7-A とは、メモリ上の情報へのアクセス方法が異なるっぽい。

```
(arch/arm64/kernel/setup.c)
->setup_arch()
  (ACPI サポートがない場合のみ)
  ->setup_machine_fdt()
    (arch/arm64/mm/mmu.c)
    ->fixmap_remap_fdt()
      ->メモリ上の FDT イメージを仮想アドレス空間にマップ
  (drivers/of/fdt.c)
  ->unflatten_device_tree()
```

unflatten_device_tree()

FDT のバイナリ表現を解釈し、カーネル内のメモリ構造へと展開する。ノードに対応するデータ構造は以下。

```
(include/linux/of.h)
struct device_node {
    ...
    struct property *properties;
    ...
    struct device_node *parent;
    struct device_node *child;
    struct device_node *sibling;
}
```

struct device_node の root ノード

struct device_node を利用したデータ構造は意外と単純。一個の root ノードからすべてをたどれる。

```
(drivers/of/base.c)
...
struct device_node *of_root;
EXPORT_SYMBOL(of_root);
...
```

device_node の利用: cpu 情報の場合

device_node データ構造のツリーを利用するには、of_xxx という一連のカーネル API を利用する。以下は “/cpus” の場合。

```
(arch/arm/kernel/setup.c)
->setup_arch()
  ->arm_dt_init_cpu_maps()
    ->of_find_node_by_path()
    ->for_each_child_of_node()
    ->of_node_cmp()
    ->of_get_property()
```

setup_arch() では他にもいろいろな箇所でこのデータを利用している。

FDT の処理: ブート後半

ペリフェラルバスやデバイスなどの初期化を行う。

```
(init/main.c)
->rest_init()
  ->kernel_thread(kernel_init)
->kernel_init()
  ->kernel_init_freeable()
    ->do_basic_setup()
      ->do_initcalls()
```

do_initcalls()

Linux カーネルが提供する様々なサービスの初期化を行う。

- ファイルシステム
- デバイスドライバ
- ネットワークプロトコルスタック
- 'Device Tree and Open Firmware support' スタック
  - of_platform_default_populate_init()

of_platform_default_populate_init()

ペリフェラルバス上のデバイスに対応するデータ構造をシステムに登録する。

```
(drivers/of/platform.c)
->of_platform_default_populate_init()
  ->of_platform_default_populate(of_default_bus_match_table
    ->of_platform_populate()
      ->of_platform_bus_create()
        ->of_platform_device_create_pdata()
          (drivers/of/device.c)
          ->of_device_add()
            ->device_add()
```

まとめ

Linux カーネルによる Device Tree 情報の利用について、概説しました。

参考資料

- 新装改訂版 Linux のブートプロセスをみる (白崎博生, 2014 年株式会社 KADOKAWA)
- リンカ・ローダ実践開発テクニック (坂井弘亮, 2010 年 CQ 出版社)