フルスクラッチで作る!
x86_64 自作 OS
パート 5

てっとりばやくマルチコア

大神祐真　著

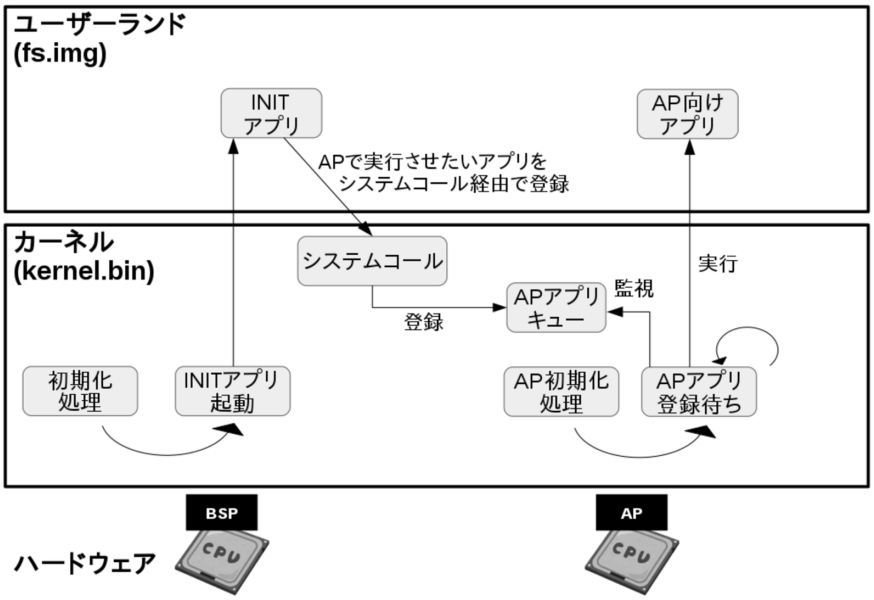
2019-08-12 版　　へにゃぺんて 発行

はじめに

本書をお手に取っていただきありがとうございます。

本書では、自作 OS によるマルチコア制御を、UEFI を活用することで、比較的手っ取り早く実現します。

本書でマルチコアの制御もできるようになったことで、自作 OS のアーキテクチャも図.1 のようになりました。



▲図 1　本書まででできあがる OS のアーキテクチャ図

本書の構成

本書は 2 年前に公開したブログ記事「UEFI ベアメタルプログラミング - マルチコアを制御する[*1]」の内容を元に、ブートローダーからカーネルへのジャンプ方法やカーネルへジャンプ後のマルチコア制御についても加筆してまとめ直したものです。

本書は以下の 2 章構成です。

- 第 1 章 ブートローダー編
  - ブートローダーを改造し、これまで使用していなかった残りのプロセッサもカーネルへジャンプさせます
- 第 2 章 カーネル・アプリ編
  - カーネルへジャンプしてきた後の、簡単な排他制御の方法や、外部アプリの実行のさせ方、各プロセッサへのタスク実行のシステムコール化を行います

なお、本書ではマルチコア時の割り込みについてはあまり立ち入りません[*2]。マルチコアとして新たに使用するプロセッサへ任意の処理を、ブートローダー/カーネル/アプリのレベルでそれぞれ行えるようになれば OK とします。

開発環境・動作確認環境

筆者が開発や動作確認を行っている環境を以下に記載します。

- PC: Lenovo ThinkPad E450
  - CPU: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz (x4 コア)
- OS: Debian GNU/Linux 8.11 (コードネーム: Jessie)
- コンパイラ: GCC 4.9.2
  - build-essential パッケージでインストールされるものです

言語としては、C 言語と一部アセンブラを使う程度なので、一般的な Linux 環境であれば問題ないと思います。

[*1] URL は本書末尾の参考情報を参照

[*2] 具体的には、システムコールを扱えるようにするために、ソフトウェア割り込みだけはマルチコア時にも使えるようにします。

本書の PDF 版/HTML 版やソースコードの公開場所について

これまでの当サークルの同人誌同様、本書も PDF 版/HTML 版は以下のページで無料で公開しています。

- `http://yuma.ohgami.jp`

サンプルコードは以下の GitHub リポジトリで公開しています。

- `https://github.com/cupnes/x86_64_jisaku_os_5_samples`

サンプルコードのリポジトリ内はディレクトリで分かれています。各ディレクトリが以降の各項で作るものです。各項の冒頭で「この項のサンプルディレクトリは XX です。」と紹介しますので適宜参照してみてください。

なお、ビルド方法は本書のパート 1 を参照してください。また、パート 2 でブートローダーにも手を加えています。実行の際はブートローダーも必要なので、そちらも併せてご参照ください。

付録サンプルコードについて

「A」から始まるディレクトリは参考用の付録サンプルコードです。今回は「A00」から「A02」の大きく 3 つの付録があります。

- A00_boot_base, A01_kern_app_base
  - 本書開始時点のブートローダー (A00_boot_base) とカーネル・アプリ (A01_kern_app_base) のソースコードを格納したディレクトリです
  - 本書開始時点からの各サンプルの差分確認等のために置いています
- A02_cover
  - 本書の最終時点に少し手を加えたマルチコアのデモプログラムです
  - 4 つのプロセッサで、それぞれが 1 ドットのランダムウォークの軌跡で画面の塗り潰しを行います
  - ちょっとスプラトゥーンっぽく、他陣に入るとスピードダウンします
  - apps ディレクトリ内の「08_dotoon」が起動時から動作している BSP 側のアプリで、「08_dotoon_cpu」が 2 つ目以降のプロセッサである AP 側のアプリです (BSP や AP について詳しくは次章以降で説明します)
  - 本書で紹介した所まででこんな風に遊べるよという一例として参考にしてみてください

- CPU が持つ乱数命令を使ったりしているので、その参考にもなるかと思います

目次

はじめに **2**
本書の構成 3
開発環境・動作確認環境 3
本書の PDF 版/HTML 版やソースコードの公開場所について 4

第 1 章 ブートローダーを変更し各 AP をカーネルへジャンプさせる **9**
1.1 マシンで有効なプロセッサ数を取得する 9
1.1.1 EFI_MP_SERVICES_PROTOCOL の準備 9
1.1.2 GetNumberOfProcessors() でプロセッサ数を取得する 11
1.1.3 動作確認 13
1.2 プロセッサ数をカーネルへ渡す 14
1.2.1 ブートローダー: カーネルへプロセッサ数を渡す 15
1.2.2 カーネル: ブートローダーから渡されたプロセッサ数を表示する . 16
1.2.3 動作確認 16
1.3 AP を動作させてみる 17
1.3.1 StartupAllAPs() について 17
1.3.2 動作確認 19
1.4 AP もカーネルへジャンプさせる 20
1.4.1 ap_main() でカーネルへジャンプするようにする 20
1.4.2 ap_main() の呼び出し側の変更点 22

第 2 章 カーネル側で AP を制御する **24**
2.1 ジャンプしてきた AP を認識する 24
2.1.1 Local APIC とは 24
2.1.2 Local APIC ID を取得してみる 25
2.1.3 AP の場合の条件分岐を追加する 26
2.1.4 動作確認 28

2.2 初期化前のコンソールへのアクセスを防ぐ 28
2.2.1 AP はコンソールの初期化を待つようにする 28
2.2.2 動作確認 29
2.3 spin lock を追加する 30
2.3.1 ロックとスピンロック 30
2.3.2 スピンロックを実装してみる 30
2.3.3 スピンロックを使ってみる 33
2.3.4 動作確認 34
2.4 AP に外部アプリを実行させる 34
2.4.1 AP の初期化処理 (ap_init()) を追加する 34
2.4.2 AP の外部アプリ実行 (ap_run()) を追加する 35
2.4.3 AP へ外部アプリを実行させる 37
2.4.4 動作確認 38
2.5 AP での外部アプリ実行をシステムコール化する 40
2.5.1 システムコールにエントリを追加する 40
2.5.2 システムコールを使ってみる 42
2.5.3 動作確認 44

おわりに **45**

参考情報 **46**
元ネタ 46
参考にさせてもらった情報 46
開発リポジトリ 46
本シリーズの過去の著作 47

第 1 章

ブートローダーを変更し各 AP をカーネルへジャンプさせる

マシンの電源を入れ、最初から動作している 1 つ目のプロセッサを「BSP(BootStrap Processor)」と呼びます。対して、電源を入れた直後は動作していない、2 つ目以降のプロセッサを「AP(Application Processor)」と呼びます。これまで、この自作 OS シリーズでは、BSP しか扱ってきませんでした。

この章では、ブートローダーの段階で UEFI 経由で AP をセットアップし、AP 側も BSP と同様にカーネルへジャンプさせる所までを順を追って紹介します。

1.1 マシンで有効なプロセッサ数を取得する

久々の UEFI です。この項では、UEFI の機能の呼び出し方の復習も兼ねて、UEFI 経由で使用可能プロセッサ数を取得してみます。

この項のサンプルディレクトリは「011_get_nproc」です。

1.1.1 EFI_MP_SERVICES_PROTOCOL の準備

UEFI が提供する色々な機能は「プロトコル」という単位でまとめられています。マルチコアの制御も同様で、「EFI_MP_SERVICES_PROTOCOL」にまとめられています。

プロトコルの実体は関数ポインタをメンバーに持つ構造体で、LocateProtocol() などの関数を使用すると構造体の先頭アドレスを取得できます。

それでは、まず、EFI_MP_SERVICES_PROTOCOL を、UEFI 関連の定義を記載している include/efi.h へ追加します (リスト 1.1)。

▼**リスト** 1.1　011_get_nproc/include/efi.h(**本書で使用する箇所のみ抜粋**)

```
/* ・・・ 省略 ・・・ */

struct EFI_DEVICE_PATH_UTILITIES_PROTOCOL {
        /* ・・・ 省略 ・・・ */
};

/* 追加（ここから） */
struct EFI_CPU_PHYSICAL_LOCATION {
        /* ・・・ 省略 ・・・ */
};

struct EFI_PROCESSOR_INFORMATION {
        /* ・・・ 省略 ・・・ */
};

struct EFI_MP_SERVICES_PROTOCOL {
        unsigned long long (*GetNumberOfProcessors)(
                struct EFI_MP_SERVICES_PROTOCOL *This,
                unsigned long long *NumberOfProcessors,
                unsigned long long *NumberOfEnabledProcessors);
        /* ・・・ 省略 ・・・ */
        unsigned long long (*StartupAllAPs)(
                struct EFI_MP_SERVICES_PROTOCOL *This,
                void (*Procedure)(void *ProcedureArgument),
                unsigned char SingleThread,
                void *WaitEvent,
                unsigned long long TimeoutInMicroSeconds,
                void *ProcedureArgument,
                unsigned long long **FailedCpuList);
        /* ・・・ 省略 ・・・ */
        unsigned long long (*WhoAmI)(
                struct EFI_MP_SERVICES_PROTOCOL *This,
                unsigned long long *ProcessorNumber);
};
/* 追加（ここまで） */

extern struct EFI_SYSTEM_TABLE *ST;
/* ・・・ 省略 ・・・ */
extern struct EFI_DEVICE_PATH_UTILITIES_PROTOCOL *DPUP;
extern struct EFI_MP_SERVICES_PROTOCOL *MSP;            /* 追加 */

/* ・・・ 省略 ・・・ */
```

EFI_MP_SERVICES_PROTOCOL と、そこから参照される新たな構造体も 2 つ追加しました。なお、構造体のメンバーは、本書で使用しないものは省略しています。構造体定義の全体像はサンプルディレクトリ内のファイルを参照してください。

EFI_MP_SERVICES_PROTOCOL 構造体定義に並ぶ各関数がマルチコア向けに UEFI が提供している関数です。プロセッサ数は「GetNumberOfProcessors」で取得します (使い方は後述)。

efi.h の最後に、EFI_MP_SERVICES_PROTOCOL 構造体のポインタ変数「MSP」を extern しています。ここへ、LocateProtocol 関数を使用して取得した EFI_MP_SERVICES_PROTOCOL 構造体の先頭アドレスを格納します。この行自

体は extern 宣言で、変数の実体は efi 周りの初期化処理を行う libuefi/efi.c で定義します。

それでは、libuefi/efi.c の変更点を示します (リスト 1.2)。

▼**リスト** 1.2　011_get_nproc/libuefi/efi.c

```
/* ・・・ 省略 ・・・ */

struct EFI_SYSTEM_TABLE *ST;
/* ・・・ 省略 ・・・ */
struct EFI_DEVICE_PATH_UTILITIES_PROTOCOL *DPUP;
struct EFI_MP_SERVICES_PROTOCOL *MSP;   /* 追加 */
struct EFI_GUID lip_guid = {0x5b1b31a1, 0x9562, 0x11d2,
/* ・・・ 省略 ・・・ */

void efi_init(struct EFI_SYSTEM_TABLE *SystemTable)
{
        /* ・・・ 省略 ・・・ */
        /* 追加（ここから） */
        struct EFI_GUID msp_guid = {0x3fdda605, 0xa76e, 0x4f46,
                                    {0xad, 0x29, 0x12, 0xf4,
                                     0x53, 0x1b, 0x3d, 0x08}};
        /* 追加（ここまで） */

        ST = SystemTable;
        ST->BootServices->SetWatchdogTimer(0, 0, 0, NULL);
        /* ・・・ 省略 ・・・ */
        ST->BootServices->LocateProtocol(&dpup_guid, NULL, (void **)&DPUP);
        ST->BootServices->LocateProtocol(&msp_guid, NULL, (void **)&MSP); //追加
}
/* ・・・ 省略 ・・・ */
```

EFI_MP_SERVICES_PROTOCOL の先頭アドレスを指すポインタ変数 MSP をグローバル変数として定義し、efi_init() の中で LocateProtocol() を使用して EFI_MP_SERVICES_PROTOCOL の先頭アドレスを取得しています。

これで、EFI_MP_SERVICES_PROTOCOL の機能をポインタ変数 MSP を通して呼び出せるようになりました。

1.1.2　GetNumberOfProcessors() でプロセッサ数を取得する

EFI_MP_SERVICES_PROTOCOL が持つ GetNumberOfProcessors() でマシンのプロセッサ数を取得できます。

GetNumberOfProcessors() のプロトタイプ宣言部分を再掲します (リスト 1.3)。

▼**リスト** 1.3　GetNumberOfProcessors()

```
unsigned long long (*GetNumberOfProcessors)(
        struct EFI_MP_SERVICES_PROTOCOL *This,
        unsigned long long *NumberOfProcessors,
        unsigned long long *NumberOfEnabledProcessors);
```

引数と戻り値は以下の通りです。

- 引数
  - This: EFI_MP_SERVICES_PROTOCOL の実体の先頭アドレス
  - NumberOfProcessors: マシンが持つプロセッサ数を格納する変数のポインタ
  - NumberOfEnabledProcessors: マシンが持つ有効状態のプロセッサ数を格納する変数へのポインタ
- 戻り値
  - 終了ステータス (0:成功、0 以外:警告あるいはエラーあり)

プロセッサの有効/無効は EFI_MP_SERVICES_PROTOCOL の EnableDisableAP() で変更できます。本書では特に使用しないため、詳しくは後述のコラム記載の仕様書を参照してみてください。なお、include/efi.h の EFI_MP_SERVICES_PROTOCOL の定義には、この関数の宣言も含めていますので、呼び出すことは可能です。

それではさっそく、この関数を使ってみます。poiboot.c へ処理を追加してみました (リスト 1.4)。

▼**リスト** 1.4　011_get_nproc/poiboot.c

```
/* ・・・ 省略 ・・・ */

void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
{
        /* ・・・ 省略 ・・・ */
        put_param(L"kernel_arg3", kernel_arg3);

        /* 追加（ここから） */
        /* プロセッサ数を表示してみる */
        unsigned long long nproc, nproc_en;
        status = MSP->GetNumberOfProcessors(MSP, &nproc, &nproc_en);
        assert(status, L"MSP->GetNumberOfProcessors");

        puts(L"nproc: ");
        puth(nproc, 1);
        puts(L"\r\n");
        puts(L"nproc_en: ");
        puth(nproc_en, 1);
        puts(L"\r\n");
        while (TRUE);
```

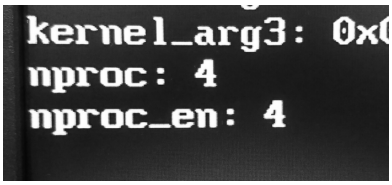
```
        /* 追加（ここまで）*/

        /* UEFI のブートローダー向け機能を終了させる */
        exit_boot_services(ImageHandle);

        /* ・・・ 省略 ・・・ */
```

GetNumberOfProcessors() を呼び出して、引数でポインタを渡している「nproc」と「nproc_en」へ、それぞれ「マシンが持つプロセッサ数」と「マシンが持つ有効なプロセッサ数」を格納させ、画面へ表示しているだけです。

なお、nproc と nproc_en を表示する際、16 進数でダンプする puth() へは、桁数を 1 で設定していますので、マシンのプロセッサ数が 16 以上の場合は[*1]良しなにこの桁数を増やしてください。

1.1.3 動作確認

ビルドし実行するとブートローダーのログの最後に nproc と nproc_en の値を表示して固まります (図 1.1)。



▲図 1.1　011_get_nproc の実行結果

筆者のマシンの場合、プロセッサの数 (NumberOfProcessors) は 4 で、全てが使用可能 (NumberOfEnabledProcessors も 4) でした[*2]。

[*1] 自作 OS でそんな環境を使えることはそうそう無いと思いますが。。

[*2] おそらく、一般的な PC においては、EFI_MP_SERVICES_PROTOCOL の関数で意図的に無効化しない限り、NumberOfProcessors と NumberOfEnabledProcessors は一致する (NumberOfProcessors の全てが NumberOfEnabledProcessors) かと思います。

EFI_MP_SERVICES_PROTOCOL のドキュメントは？

EFI_MP_SERVICES_PROTOCOL については、「フルスクラッチで作る!UEFI ベアメタルプログラミング」で紹介した UEFI の仕様書には書かれていません。マルチコア等、プラットフォームに依存する UEFI の仕様は仕様書が分かれていて、通常の仕様書と同じく UEFI 公式の仕様書のページ[a]にある「UEFI Platform Initialization Specification」に書かれています。2019 年 6 月現在はバージョン 1.7 が公開されていて、EFI_MP_SERVICES_PROTOCOL の仕様は「13 DXE Boot Services Protocol(P.462-)」に書かれています。

[a] `https://uefi.org/specifications`

GetNumberOfProcessors() が返すプロセッサ数は論理プロセッサ数

GetNumberOfProcessors() が返すプロセッサ数は、論理プロセッサ (実行ユニット) の数です。そのため、例えばハイパースレッディングにより、各物理プロセッサに 2 つの論理プロセッサが動作する場合、物理プロセッサ数の 2 倍の値が返ります。

1.2 プロセッサ数をカーネルへ渡す

今後、カーネル側で「任意のタスクを任意のプロセッサで実行させる」という事を行うにあたり、プロセッサ数の情報はカーネル側でも必要です。

そこで、前項で取得したプロセッサ数をカーネルへ渡すようにします。

この項では、カーネル側もセットで変更します。サンプルディレクトリは以下の通りです。

- ブートローダー: 012_1_add_nproc_kern_param
- カーネル: 012_2_dump_nproc_at_kern

1.2.1 ブートローダー: カーネルへプロセッサ数を渡す

カーネルへのプラットフォーム情報の受け渡しには platform_info という構造体を用意していました。そのため、プロセッサ数はこの構造体へメンバーとして追加することにします。

前項の poiboot.c をリスト 1.5 のように変更します。

▼**リスト** 1.5　012_1_add_nproc_kern_param/poiboot.c

```
/* ・・・ 省略 ・・・ */

struct __attribute__((packed)) platform_info {
        struct fb fb;
        void *rsdp;
        unsigned int nproc;     /* 追加 */
} pi;

/* ・・・ 省略 ・・・ */

void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
{
        /* ・・・ 省略 ・・・ */

        /* カーネルへ引数として渡す内容を変数に準備する */
        unsigned long long kernel_arg1 = (unsigned long long)ST;
        put_param(L"kernel_arg1", kernel_arg1);
        init_fb();
        pi.fb.base = fb.base;
        pi.fb.size = fb.size;
        pi.fb.hr = fb.hr;
        pi.fb.vr = fb.vr;
        pi.rsdp = find_efi_acpi_table();
        /* 追加（ここから）*/
        unsigned long long nproc, nproc_en;
        status = MSP->GetNumberOfProcessors(MSP, &nproc, &nproc_en);
        assert(status, L"MSP->GetNumberOfProcessors");
        pi.nproc = nproc_en;
        /* 追加（ここまで）*/
        unsigned long long kernel_arg2 = (unsigned long long)&pi;
        put_param(L"kernel_arg2", kernel_arg2);
        unsigned long long kernel_arg3;
        if (has_fs == TRUE)
                kernel_arg3 = fs_start;
        else
                kernel_arg3 = 0;
        put_param(L"kernel_arg3", kernel_arg3);

        /* 「プロセッサ数を表示してみる」の処理は削除 */

        /* UEFI のブートローダー向け機能を終了させる */
        exit_boot_services(ImageHandle);

        /* ・・・ 省略 ・・・ */
```

platform_info 構造体にプロセッサ数 (nproc) のメンバーを増やし、nproc_en(有効なプロセッサ数) を設定しています。

1.2.2 カーネル: ブートローダーから渡されたプロセッサ数を表示する

platform_info 構造体に nproc メンバーを増やしたので、カーネル側も併せて変更し、ブートローダーから受け取った nproc を表示してみます。

main.c をリスト 1.6 のように変更します。

▼**リスト** 1.6　012_2_dump_nproc_at_kern/main.c

```
/* ・・・ 省略 ・・・ */

struct __attribute__((packed)) platform_info {
        struct framebuffer fb;
        void *rsdp;
        unsigned int nproc;     /* 追加 */
};

#define INIT_APP       "test"

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* CPU 周りの初期化 */
        gdt_init();
        intr_init();

        /* 追加（ここから） */
        /* ブートローダーから渡されたプロセッサ数を表示 */
        puts("NPROC ");
        puth(pi->nproc, 1);

        /* halt して待つ */
        while (1)
                cpu_halt();
        /* 追加（ここまで） */

        /* ・・・ 省略 ・・・ */
```

特に解説することはありません。ブートローダーから受け取った nproc を画面へ表示し、それ以降の処理へ進まないよう while() で止めているだけです。

1.2.3 動作確認

実行すると、図 1.2 のように画面にプロセッサ数が表示されることを確認できます。

▲図 1.2　012 のブートローダーとカーネルの実行結果

現状の自作カーネルは、起動の始めに画面を青で塗りつぶします。モノクロ印刷なので分かりにくいですが、背景が青になっているので、カーネルのレベルでプロセッサ数を表示できていることが分かります。

1.3　AP を動作させてみる

前項までで EFI_MP_SERVICES_PROTOCOL の使い方とカーネルへのパラメータの受け渡しを紹介しました。AP を動作させてみる事も、プロセッサ数の確認と同様に、EFI_MP_SERVICES_PROTOCOL の関数を呼び出すことで行えます。

この項では、またブートローダーに少し手を加え、EFI_MP_SERVICES_PROTOCOL の StartupAllAPs() を使用して全ての AP を動作させてみます。

この項のサンプルディレクトリは「013_start_ap」です。

1.3.1　StartupAllAPs() について

StartupAllAPs() は、その名の通り、全ての AP の動作を開始させる関数[*3]です。AP に実行させたい関数を引数に渡すことで、全ての AP に指定した関数を実行させることができます。

- 引数
  - This: EFI_MP_SERVICES_PROTOCOL の実体の先頭アドレス
  - Procedure: AP に実行させる関数
  - SingleThread: 各 AP を直列に実行する (=1) か、並列に実行する (=0) か
  - WaitEvent: 各 AP を BSP に対して非同期に実行する際、AP での関数の実

[*3] 指定した特定の AP のみ動作を開始させる関数 (StartupThisAP()) もあります。本書では使用しないのですが、include/efi.h の EFI_MP_SERVICES_PROTOCOL の定義には含めていますので呼び出すことは可能です。詳しくは仕様書を見てみてください。

行完了の通知に使用するイベントオブジェクト
    * NULL を指定した場合、AP での関数実行は BSP に対して同期型で実行される (AP での関数実行が終わるまで BSP へ処理が戻ってこない)
  - TimeoutInMicroSeconds: AP での関数実行を同期型で実施する際のタイムアウト時間 (単位:マイクロ秒)
    * 0 を指定した場合、無限に待ち続ける
  - ProcedureArgument: AP で実行する関数へ渡す引数
  - FailedCpuList: NULL 以外のアドレスを指定しておくと、エラー終了した AP があったとき、そのような AP のリストを指定されたアドレスに格納してくれる
- 戻り値
  - 終了ステータス (0:成功、0 以外:警告あるいはエラーあり)

StartupAllAPs() をリスト 1.7 のように使用します。

▼**リスト** 1.7　013_start_ap/poiboot.c

```
/* ・・・ 省略 ・・・ */
void ap_main(void *_st);        /* 追加 */

void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
{
        /* ・・・省略  ・・・ */
        put_param(L"kernel_arg3", kernel_arg3);

        /* 追加（ここから） */
        /* BSP 自身のプロセッサ番号を表示 */
        unsigned long long pnum;
        status = MSP->WhoAmI(MSP, &pnum);
        assert(status, L"MSP->WhoAmI");
        puth(pnum, 1);

        /* 全ての AP をスタートする */
        status = MSP->StartupAllAPs(MSP, ap_main, 0, NULL, 0, ST, NULL);
        assert(status, L"MSP->StartupAllAPs");

        /* BSP/AP のプロセッサ番号を表示できたらそのまま止める */
        while (TRUE);
        /* 追加（ここまで） */

        /* UEFI のブートローダー向け機能を終了させる */
        exit_boot_services(ImageHandle);

        /* ・・・ 省略 ・・・ */
}

/* ・・・ 省略 ・・・ */

/* 追加（ここから） */
```

```
void ap_main(void *_st)
{
        efi_init(_st);

        /* 自身のプロセッサ番号を表示 */
        unsigned long long pnum;
        unsigned long long status = MSP->WhoAmI(MSP, &pnum);
        assert(status, L"MSP->WhoAmI");
        puth(pnum, 1);
}
/* 追加（ここまで） */
```

まず「BSP 自身のプロセッサ番号を表示」で、そのコードを実行しているプロセッサのプロセッサ番号を取得するために「WhoAmI()」を使用しています。この関数は、第 2 引数で与えるポインタの指す先へ WhoAmI() を呼び出したプロセッサ番号を格納してくれます。そのため、このコードブロックでは、BSP のプロセッサ番号 (=0) を取得し puth() で画面表示しているだけです。

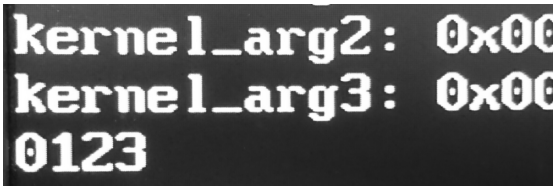
次の「全ての AP をスタートする」のコードブロックで StartupAllAPs() を使用して全ての AP の実行を開始させています。各 AP に実行させる関数は「ap_main()」です (ここより下の方で定義を追加しています)。見てもらえば分かる通りですが、処理としては WhoAmI() を使用して自身のプロセッサ番号を取得して画面表示しているだけです。AP 側でも UEFI の機能を呼び出すため、ap_main() の引数 (StartupAllAPs() の第 6 引数) では、EFI_SYSTEM_TABLE のポインタを渡しています。

StartupAllAPs() に関しては他に、第 3 引数 (SingleThread) に 0(各 AP は非同期) を、第 4 引数 (WaitEvent) で NULL(BSP に対し AP は同期型) を指定しています。そのため、全ての AP で並列に ap_main() が実行された後、StartupAllAPs() から戻ってくる挙動になります。

表示したプロセッサ番号を確認したいので、StartupAllAPs() から戻ってきた後は、無限ループで止めています。

1.3.2 動作確認

実行すると図 1.3 のように搭載しているプロセッサの数だけ番号が表示されます。



▲図 1.3　013_start_ap の実行結果

筆者の PC は論理プロセッサ 4 つなので、0 から 3 までの番号が表示されています。

1.4　AP もカーネルへジャンプさせる

AP へ任意の関数を実行させることができました。それでは、この章の最後に、BSP 同様に AP もカーネルへジャンプさせてみます。

この項のサンプルディレクトリは「014_jump_to_kern」です。これが本書で作成するブートローダーの最終版になります。

なお、AP がカーネルへジャンプしてくる際は、カーネル側もそれに応じて変更する必要があるのですが、新しい内容も出てくるので次章でカーネル側の変更を説明します。

1.4.1　ap_main() でカーネルへジャンプするようにする

前項で、各 AP に ap_main() を実行させるようにしました。基本的には、この ap_main() でカーネルへジャンプするようにするだけです。

早速ですが、変更後の ap_main() はリスト 1.8 の通りです。

▼リスト 1.8　014_jump_to_kern/poiboot.c

```
/* ・・・ 省略 ・・・ */

struct __attribute__((packed)) platform_info {
        struct fb fb;
        void *rsdp;
        unsigned int nproc;
} pi;

/* 追加（ここから） */
struct ap_info {
        unsigned long long kernel_start;
        unsigned long long stack_space_start;
        struct EFI_SYSTEM_TABLE *system_table;
} ai;
/* 追加（ここまで） */
```

```
/* ・・・ 省略 ・・・ */
void ap_main(void *_ai);        /* 変更 */

void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
{
        /* ・・・ 省略 ・・・ */
}

/* 変更（ここから） */
void ap_main(void *_ai)
{
        struct ap_info *ai = _ai;
        efi_init(ai->system_table);

        unsigned long long pnum;
        unsigned long long status = MSP->WhoAmI(MSP, &pnum);
        assert(status, L"MSP->WhoAmI");

        unsigned long long stack_base = ai->stack_space_start + (pnum * MB);
        unsigned long long kernel_arg1 = 0;
        unsigned long long kernel_arg2 = 0;
        unsigned long long kernel_arg3 = 0;

        /* カーネルへ渡す引数設定（引数に使うレジスタへセットする） */
        unsigned long long _sb = stack_base, _ks = ai->kernel_start;
        __asm__ ("      mov     %0, %%rdx\n"
                 "      mov     %1, %%rsi\n"
                 "      mov     %2, %%rdi\n"
                 "      mov     %3, %%rsp\n"
                 "      jmp     *%4\n"
                 ::"m"(kernel_arg3), "m"(kernel_arg2), "m"(kernel_arg1),
                  "m"(_sb), "m"(_ks));

        while (TRUE);
}
/* 変更（ここまで） */
```

引数として「ap_info」という構造体を渡すようにしました。StartupAllAPs() で渡すことのできる引数は 1 つなので、AP 側へ渡すべき情報はこの構造体にまとめます。そして、platform_info と同様にグローバル変数を定義しています。

AP がカーネルへジャンプするにあたって必要な情報は「カーネルの領域の先頭アドレス」と「AP 毎のスタックポインタ」です。「カーネルの領域の先頭アドレス」は ap_info の kernel_start で渡します。「AP 毎のスタックポインタ」は、AP 毎にアドレスを変える必要があります。ここでは各 AP のスタックサイズを 1MB として、「AP 用スタック領域の先頭アドレス + (プロセッサ番号 * 1MB)」を AP 毎のスタックポインタとしています。ap_info の stack_space_start が「AP 毎のスタック領域の先頭アドレス」です。「プロセッサ番号」は、ap_main() 内で WhoAmI() を実行して取得しています。

なお、カーネルへ渡す「kernel_arg1」から「kernel_arg3」の引数は、(少なくとも当面は、)AP 側で使用しないため、全て 0 にしています。kernel_arg1 は

EFI_SYSTEM_TABLE へのポインタで、今の所、BSP 側でカーネルが動作する際も使用していません。kernel_arg2 は platform_info へのポインタで、kernel_arg3 はファイルシステムの先頭アドレスです。これらに関しては、各種デバイスの初期化処理などは基本的には BSP 側で一度実施しておけば十分です。一部、AP 側でも初期化が必要なものもありますが、そのために必要な情報はカーネルへジャンプした後、自ら取得可能です (少なくとも本書で扱う範囲では)。

ap_main() の最後で行っているカーネルへのジャンプのインラインアセンブラ自体は、BSP 側で実行しているものと同じです。

1.4.2 ap_main() の呼び出し側の変更点

ap_main() の変更に合わせて、poiboot.c のリスト 1.9 の変更を行えば、この項の変更は完了です。

▼**リスト** 1.9　014_jump_to_kern/poiboot.c

```
/* ・・・ 省略 ・・・ */

#define MB             1048576 /* 1024 * 1024 */

/* 追加（ここから） */
/* AP 側の UEFI 処理完了までの BSP の待ち時間（単位: マイクロ秒） */
#define WAIT_FOR_AP_USECS      100000 /* 100ms */
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */

void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
{
        /* ・・・ 省略 ・・・ */
                kernel_arg3 = 0;
        put_param(L"kernel_arg3", kernel_arg3);

        /* 変更（ここから） */
        /* 全ての AP をスタートする */
        ai.kernel_start = kernel_start;
        ai.stack_space_start = stack_base;
        ai.system_table = ST;
        status = MSP->StartupAllAPs(
                MSP, ap_main, 0, NULL, WAIT_FOR_AP_USECS, &ai, NULL);
        /* 変更（ここから） */

        /* UEFI のブートローダー向け機能を終了させる */
        exit_boot_services(ImageHandle);

        /* ・・・ 省略 ・・・ */
```

グローバル変数として定義していた ap_info 型の変数 ai を、ap_main() の引数として渡すようにしました。

また、StartupAllAPs() の第 5 引数のタイムアウト時間を 100ms に設定しています。今回の ap_main() は、カーネルへジャンプした後、戻って来ることはありません。そのため、タイムアウト時間が 0 のままだと BSP 側での StartupAllAPs() の呼び出しが、AP 側で ap_main() から返ってくるまで無限に待ち続けてしまいます。そのため、タイムアウトを設定しています。なお、BSP は StartupAllAPs() から戻ってくると、UEFI の機能を終了させる exit_boot_services() を実行します。ap_main() で UEFI の機能として WhoAmI() を実行するため、AP 側でこれを終えるまでは BSP 側で UEFI 終了の処理に進んでしまうことが無いようにタイムアウト時間を設定したいです。ここではざっくりと 100ms くらいでタイムアウトするようにしてみました。

AP がカーネルへジャンプするためのブートローダー側の変更は以上です。カーネル側の変更点は次章で説明します。

第 2 章

カーネル側で AP を制御する

前章では、UEFI の機能を使用して AP を制御する方法を紹介しました。この章では、UEFI の世界を抜けてカーネルへジャンプしてきた後で AP を制御する方法を紹介します。

2.1 ジャンプしてきた AP を認識する

前章の最後で、AP をカーネルへジャンプさせました。これで、BSP 同様に、カーネルへ来ることはできているのですが、カーネル側のエントリー関数である現状の start_kernel() には、「BSP であるか AP であるか」の条件分岐が無いです。そのため、BSP/AP に関わらずカーネルの初期化処理が実施されます。デバドラの初期化など、カーネルの初期化処理はたいてい、BSP 側で一度実施しておけば良いので、start_kernel() 冒頭に BSP/AP の条件分岐を入れます。

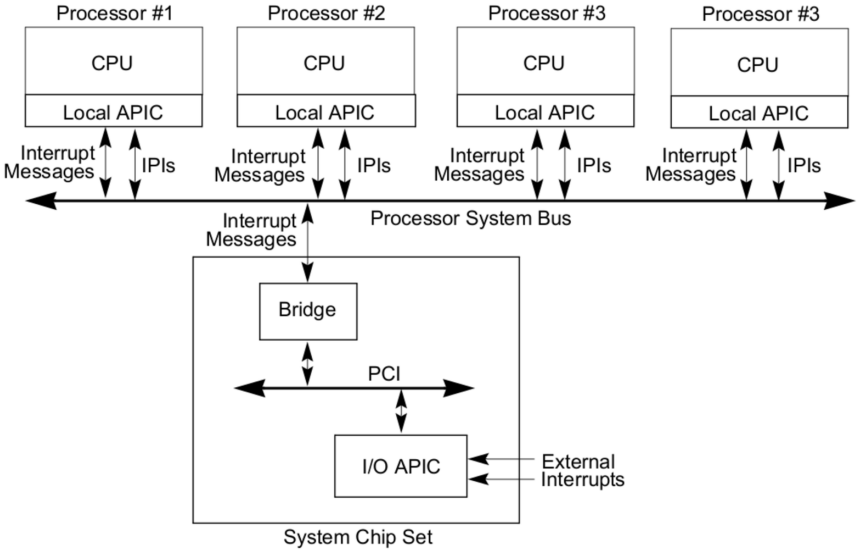
BSP か AP かの判別にプロセッサ番号を見たいところですが、UEFI の世界は既に抜けているので、WhoAmI() は使えません。実は、プロセッサ番号は「Local APIC」と呼ばれる IC が持っています。ここでは、Local APIC の情報から AP 自身のプロセッサ番号を確認してみます。

この項のサンプルディレクトリは「021_lapic_id」です。

2.1.1 Local APIC とは

「APIC」は「Advanced Programmable Interrupt Controller」の略で、旧来の割り込みコントローラ (PIC) の進化版です。各プロセッサに一つずつ付いているため「Local APIC」と呼ばれています。(Local ではない APIC が居たりする訳ではないです。)APIC

周りの構成例を Intel SDM[*1]から引用します (図 2.1)。



▲図 2.1　APIC 周りの構成例

前章で WhoAmI() で取得したプロセッサ番号の実体は「Local APIC ID」と呼ばれるもので、プロセッサ毎の Local APIC のレジスタに格納されています。

2.1.2　Local APIC ID を取得してみる

それでは、Local APIC のレジスタにアクセスし、Local APIC ID を取得してみます。

まず、Local APIC のレジスタへはメモリへマップされたアドレスでアクセスします。

Local APIC の領域の先頭アドレスは 0xfee00000 です。これは物理アドレスなのですが、このシリーズの自作 OS はページテーブルを、UEFI が設定しておいてくれるまま、物理アドレス=仮想アドレスで使用しているので、そのまま 0xfee00000 の領域にアクセスすれば良いです。

そして、Local APIC ID が格納されているレジスタは 0xfee00020 の「Local APIC ID Register」です。4 バイトのレジスタで、上位 8 ビットに実際の Local APIC ID が格納されています (その他は予約ビットです)。

[*1] Intel が提供している開発者向けのマニュアル (Intel Software Developer's Manual) です。配布場所などは本書末尾の「参考情報」を参照してください。

これを踏まえて、プロセッサ番号 (=Local APIC ID) を取得する関数「get_pnum」を「apic.c」というファイルを作って追加してみます (リスト 2.1)。

▼**リスト** 2.1　021_lapic_id/apic.c

```
/* 追加（ここから）*/
#define LAPIC_REG_BASE 0xfee00000
#define LAPIC_ID_REG   (*(volatile unsigned int *)(LAPIC_REG_BASE + 0x20))

unsigned char get_pnum(void)
{
        return LAPIC_ID_REG >> 24;
}
/* 追加（ここまで）*/
```

0xfee00020 から unsigned int(4 バイト) でレジスタ値を取得し、24 ビット右シフトすることで上位 8 バイトを得ています。

また、併せて include ディレクトリに「apic.h」を追加し、get_pnum() のプロトタイプ宣言を追加しておきます (コードは割愛)。

加えて、"acpi.o"を Makefile の"OBJS"へ追加しておいてください (こちらもコードは割愛)。

2.1.3　AP の場合の条件分岐を追加する

get_pnum() を実装したことで、カーネル側でも BSP/AP 自身がプロセッサ番号を取得できるようになりました。

これを利用して、start_kernel() に AP の場合の条件分岐を追加します。BSP/AP それぞれで、自身のプロセッサ番号を表示させるようにしてみました (リスト 2.2)。

▼**リスト** 2.2　021_lapic_id/main.c

```
/* ・・・ 省略 ・・・ */
#include <pic.h>
#include <apic.h>       /* 追加 */
/* ・・・ 省略 ・・・ */

#define INIT_APP       "test"

/* 追加（ここから）*/
/* コンソールへアクセス可能な CPU を管理（同時アクセスを簡易的に防ぐ）*/
unsigned char con_access_perm = 0;
/* 追加（ここまで）*/

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
```

```
        /* 追加（ここから） */
        unsigned char pnum = get_pnum();

        /* AP の場合、初期化処理をスキップ */
        if (pnum) {
                /* 自分の番まで待つ */
                while (con_access_perm != pnum);

                /* AP 自身のプロセッサ番号を表示 */
                puth(pnum, 1);

                /* 次の番へ回す */
                con_access_perm++;

                /* halt して待つ */
                while (1)
                        cpu_halt();
        }
        /* 追加（ここまで） */

        /* フレームバッファ周りの初期化 */
        fb_init(&pi->fb);
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        clear_screen();

        /* 追加（ここから） */
        /* BSP 自身のプロセッサ番号を表示 */
        puth(pnum, 1);

        /* 次の番へ回す */
        con_access_perm++;

        /* halt して待つ */
        while (1)
                cpu_halt();
        /* 追加（ここまで） */

        /* ACPI の初期化 */
        acpi_init(pi->rsdp);

        /* ・・・ 省略 ・・・ */
```

start_kernel() 冒頭に、プロセッサ番号を確認して 0(BSP) でなければ AP 用の処理を実施する処理を追加しました。BSP/AP 共に、実験として、自身のプロセッサ番号を表示した後、無限ループで止めています。BSP の場合のみ、コンソールの初期化をしてからプロセッサ番号表示を行います。

なお、自作カーネルにはまだロック機構が無いため、ここでは、BSP と各 AP が文字表示を順番に行うように「con_access_perm」変数で、「今コンソールへアクセスして良いプロセッサ番号」を管理しています。プロセッサ番号は、BSP が 0、AP は 1 以降となりますので、con_access_perm を 0 から始め、各プロセッサで文字表示し次第、con_access_perm をインクリメントすることで、各プロセッサが順番にコンソールへ出

力するようにしています。なお、これはあくまでも仮の実装です。ロックについては次項以降で改めて説明、実装します。

2.1.4 動作確認

実行すると、プロセッサの数だけ 0 から順に番号が表示されます (図 2.2)。

▲図 2.2　021_lapic_id の実行結果

筆者の環境の場合、プロセッサは 4 つなので「0123」と表示されています。

2.2 初期化前のコンソールへのアクセスを防ぐ

前項では、BSP/AP でのコンソールへの同時アクセスを防ぐため、順番にアクセスする仕組みを入れていました。

汎用的なロックを実装する前に、そもそも AP 側で何らかのリソースを使うとき、自身で初期化しないなら誰か (主に BSP) が初期化するのを待つ必要があります。この項では、AP がコンソール初期化を待つ処理を追加してみます。

この項のサンプルディレクトリは「022_wait_con_init」です。

2.2.1 AP はコンソールの初期化を待つようにする

コンソールの初期化が完了したことを示すフラグ変数「is_con_inited」を追加し、AP はこのフラグがセットされるのを待つようにします (リスト 2.3)。

▼リスト 2.3　022_wait_con_init/main.c

```
/* ・・・ 省略 ・・・ */
#define INIT_APP       "test"

/* 変更（ここから） */
/* コンソールの初期化が完了したか否か */
unsigned char is_con_inited = 0;
```

```
/* 変更（ここまで） */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        unsigned char pnum = get_pnum();

        /* AP の場合、初期化処理をスキップ */
        if (pnum) {
                /* 変更（ここから） */
                /* コンソールの初期化が完了するまで待つ */
                while (!is_con_inited);

                /* AP 自身のプロセッサ番号を表示 */
                puth(pnum, 1);
                /* 変更（ここまで） */

                /* halt して待つ */
                while (1)
                        cpu_halt();
        }

        /* フレームバッファ周りの初期化 */
        fb_init(&pi->fb);
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        clear_screen();
        is_con_inited = 1;      /* 追加 */

        /* BSP 自身のプロセッサ番号を表示 */
        puth(pnum, 1);

        /* halt して待つ */
        while (1)
                cpu_halt();

        /* ・・・ 省略 ・・・ */
```

2.2.2 動作確認

実行すると図 2.3 のように表示されました。

▲図 2.3　022_wait_con_init の実行結果

AP がコンソールの初期化を待つようにはなったのですが、put 系の関数を同時に呼び出すため、文字と文字の出力が被って表示が壊れてしまっています。

2.3 spin lock を追加する

それでは、コンソールへのアクセスを保護するロック機構を実装します。

この項のサンプルディレクトリは「023_spin_lock」です。

なお、ここで保護する対象は、「putc() による 1 文字の出力」とします。それにより、前項で確認した「文字出力同士がかぶり、文字が壊れる事態」を防ぎます。文字を出力する順序に対しては特になにもしないので、同時に putc() が呼ばれたときの文字の表示順は任意となります。

2.3.1 ロックとスピンロック

ソフトウェアの処理において、「ここの処理は、別のスレッド等から同時実行されると困る」場合があります。今回の putc() が正にそれで、putc() である 1 文字を描画中に putc() が呼ばれると、同じ位置に文字を書こうとして文字の表示が壊れます。そのような場合に特定の処理の実行を順番待ちさせる仕組みが「ロック」です。

ロックにはいくつか種類があるのですが、ここでは比較的シンプルで Intel SDM にも実装例がある「スピンロック (Spin Lock)」を実装してみます。スピンロックは、ある区間を誰かがロックを取って実行中の時、その区間を実行しようとした 2 つ目以降のプロセッサをロックが解放されるまでビジーループで待たせる方式です。シンプルなものなので詳しくは実装で説明します。

2.3.2 スピンロックを実装してみる

さっそくコードを紹介します (リスト 2.4)。

▼**リスト** 2.4　023_spin_lock/x86.c

```
/* ・・・ 省略 ・・・ */
/* 追加（ここから）*/
/*
Spin_Lock:
        CMP lockvar, 0          ;Check if lock is free
        JE Get_Lock
        PAUSE                   ;Short delay
        JMP Spin_Lock
Get_Lock:
        MOV EAX, 1
```

```
        XCHG EAX, lockvar       ;Try to get lock
        CMP EAX, 0              ;Test if successful
        JNE Spin_Lock
Critical_Section:
        <critical section code>
        MOV lockvar, 0
        ...
Continue:

# ref:
# 8.10.6.1 Use the PAUSE Instruction in Spin-Wait Loops
# - Intel(R) 64 and IA-32 Architectures Software Developer's Manual
#   Volume 3 System Programming Guide
*/

void spin_lock(unsigned int *lockvar)
{
        unsigned char got_lock = 0;
        do {
                while (*lockvar)
                        CPU_PAUSE();

                unsigned int lock = 1;
                asm volatile ("xchg %[lock], %[lockvar]"
                              : [lock]"+r"(lock), [lockvar]"+m"(*lockvar)
                              :: "memory", "cc");

                if (!lock)
                        got_lock = 1;
        } while (!got_lock);
}

void spin_unlock(volatile unsigned int *lockvar __attribute__((unused)))
{
        *lockvar = 0;
}
/* 追加（ここまで）*/
```

spin_lock() がロックを取得する関数で、spin_unlock() で解放します。ロック状態は変数で管理し、これらの関数へポインタで渡します (lockvar 変数)。

冒頭のコメントが、Intel SDM 記載のサンプルで、spin_lock() と spin_unlock() は、C 言語で書き下したものです。

spin_lock()/spin_unlock() 共にロック状態を保持する変数のポインタを渡すようになっています。

spin_lock() で行っていることは以下の 3 つです。

1. ロック状態 (lockvar ポインタの指す先) が解放済み (0) になるのを待つ。
2. atomic 命令でロック状態を変更 (ロックを取得 (1)) する。
3. ロックの取得に失敗した場合、再度 1. から実施する。成功した場合は関数を return する。

1. について、「CPU_PAUSE()」は今回追加したマクロで、x86.h へリスト 2.5 の様にマクロ定義を追加しました。

▼**リスト** 2.5　023_spin_lock/include/x86.h

```
/* ・・・ 省略 ・・・ */

#define MAX_INTR_NO    256

#define CPU_PAUSE()    asm volatile ("pause")  /* 追加 */

struct __attribute__((packed)) interrupt_descriptor {
        /* ・・・ 省略 ・・・ */
```

「pause」という命令を呼び出すだけのマクロです。pause 命令は、CPU へビジーループがあることを伝えるヒントとして機能する命令です。コレを入れておくと CPU がよしなにウェイト時間を設けて、省電力化へ寄与します。これまで知らなかったので使っていませんでしたが、ビジーループで待機させている箇所には pause 命令を入れておくと良さそうです。

そして、2. について、「atomic 命令」は、他の一般的な命令では複数の命令になってしまうものを 1 命令で行ってくれるものです。1 命令 (不可分、atomic) なのでその間に割り込まれる事が無いと保証できます。

今回使用している atomic 命令は「xchg(Exchange Register/Memory with Register)」です。これはレジスタあるいはメモリの内容を別のレジスタと入れ換える命令です。インラインアセンブラの書き方についてここでは説明しませんが、この様に書くことで lockvar 変数と lock 変数の内容の入れ替えを 1 命令で行うことができます[*2]。lock 変数には予め、ロックを示す値 (1) を入れているので、入れ替えによって lockvar へ 1 がセットされます。

なぜこのような方法で lockvar の書き換えを行っているのかというと、これによってロックを取得できたのか否かを確実に知る事ができるからです。確実にロックを取得する上で問題となるのが 1. でロックが解放済み (0) であることを確認した後 xchg 命令を実行するまでの間で、他のプロセッサ等が先に xchg 命令を実行してロックを取ってしまうことです。そんな場合でも lockvar 自体は 1 になるので、ロックが取れたと思って目的の処理を実施すると先にロックを取ったプロセッサ側と処理が競合します。

ロックが取れなかった時は取れなかったと知るために lock 変数側を使います。別のプ

[*2] xchg 命令は「メモリ領域とレジスタ」あるいは「レジスタとレジスタ」の入れ替えしかできません。ここでは「lockvar(メモリ領域) と lock の内容を格納したレジスタ」の入れ替えとなるようにインラインアセンブラを書いています。asm volatile () のコロンで区切られた 2 つ目に、「lock」に対しては「+r(レジスタ)」を、「lockvar」に対しては「+m(メモリ)」を指定しているのがそうです。

ロセッサ等によりロックが取られてしまっている時は、xchg 命令実行時、既に lockvar は 1 になっています。そのため xchg の実行後、lock の内容は 1 になります。他方、ロックを正常に取得できた場合、xchg 命令実行時、lockvar は 0 なので、lock の内容は 0 になります。

そのため、最後に 3. では lock の内容を確認し、その内容が 1(ロックを取得できなかった) ならば、もう一度 1. から繰り返すようにしています。そうではなく、lock が 0 ならばロックを取得できたとして got_lock に 1 を代入して while を抜け、関数からも return します。

spin_unlock() はもっと単純で、単に lockvar の指す先に 0 を代入するだけです。spin_unlock() を実行する時、既にロックは取っているので、lockvar を書き換える何かが競合してくることはありません。心置きなくロックを解放するだけです。

あとは、x86.c へ追加した spin_lock()/spin_unlock() のプロトタイプ宣言を x86.h へ追加しておきます (コードを引用しての紹介は割愛)。

2.3.3 スピンロックを使ってみる

それでは、spin_lock()/spin_unlock() を使ってみます。

文字出力系の関数で最終的に 1 文字を描画する putc() にロック処理を追加します (リスト 2.6)。

▼**リスト** 2.6　023_spin_lock/fbcon.c

```
#include <x86.h>        /* 追加 */
#include <fbcon.h>
#include <fb.h>
#include <font.h>

/* 64bit unsigned の最大値 0xffffffffffffffff は
 * 10 進で 18446744073709551615(20 桁) なので'\0' 含め 21 文字分のバッファで足りる */
#define MAX_STR_BUF    21

unsigned int cursor_x = 0, cursor_y = 0;
unsigned int putc_lock = 0;     /* 追加 */

void putc(char c)
{
        spin_lock(&putc_lock); /* 追加 */

        unsigned int x, y;

        /* ・・・ 省略 ・・・ */

        }

        spin_unlock(&putc_lock);       /* 追加 */
```

```
}
/* ・・・ 省略 ・・・ */
```

putc() のロック状態を管理する変数として putc_lock をグローバル変数として用意し、putc() の冒頭に spin_lock() を、末尾に spin_unlock() を追加しました。これで、putc() を複数のプロセッサから同時に呼び出された場合も、先に spin_lock() を取得した側でのみ putc() の中身が実行され、取得できなかった方は spin_lock() 内のビジーループで待ち続ける様になります。

2.3.4 動作確認

実行すると、今度は文字が壊れること無く BSP と各 AP のプロセッサ番号が表示されるようになりました (図 2.4)。

▲図 2.4　023_spin_lock の実行結果

2.4 AP に外部アプリを実行させる

ここまでで、AP 側をカーネルへ遷移させ、カーネル内のコードを実行させることができました。

次に、AP へ外部アプリを実行させてみます。

この項のサンプルディレクトリは「024_exec」です。

2.4.1 AP の初期化処理 (ap_init()) を追加する

外部アプリは、何らかの処理を行うためにシステムコールを発行します。システムコールはソフトウェア割り込みなので、外部アプリを実行するには AP 側でも割り込みの設定を行う必要があります。

そろそろ AP に関する処理は別ファイルへ切り出そうと思います。ap.c を作成しそこに初期化処理 (ap_init()) を実装します (リスト 2.7)。

▼**リスト** 2.7　024_exec/ap.c

```
/* 追加（ここから） */
#include <x86.h>
#include <intr.h>
#include <syscall.h>

void ap_init(void)
{
        /* CPU 周りの初期化 */
        gdt_init();
        intr_init();

        /* システムコールの初期化 */
        syscall_init();
}
/* 追加（ここまで） */
```

BSP 同様に GDT/IDT の初期化と、システムコールの初期化を行いました。syscall_init() の中でソフトウェア割り込みの割り込みハンドラの設定を行っています。

2.4.2　AP の外部アプリ実行 (ap_run()) を追加する

次に AP が外部アプリを実行する処理を「ap_run」という関数で追加します (リスト 2.8)。

▼**リスト** 2.8　024_exec/ap.c

```
#include <x86.h>
#include <intr.h>
#include <proc.h>       /* 追加 */
#include <syscall.h>
#include <fs.h> /* 追加 */
#include <common.h>     /* 追加 */

#define MAX_APS                 16      /* 追加 */

struct file *ap_task[MAX_APS] = { NULL };     /* 追加 */

void ap_init(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void ap_run(unsigned char pnum)
{
```

```
        while (1) {
                /* 自分用のタスクが登録されるのを待つ */
                while (!ap_task[pnum - 1])
                        CPU_PAUSE();

                /* 実行 */
                exec(ap_task[pnum - 1]);

                /* 空に戻す */
                ap_task[pnum - 1] = NULL;
        }
}
/* 追加（ここまで）*/
```

AP に実行させる外部アプリ (タスク) を登録するための変数「ap_task」を用意し、ap_run() では、自分用のタスクが登録され次第、exec() で実行します。exec() は同期型で外部アプリを実行するので、外部アプリの実行が終わり、戻ってきたら ap_task を空 (NULL) に戻します。

また、タスク登録用の関数「ap_enq_task」も追加しておきます (リスト 2.9)。

▼**リスト** 2.9　024_exec/ap.c

```
/* ・・・ 省略 ・・・ */

struct file *ap_task[MAX_APS] = { NULL };
unsigned int ap_task_lock[MAX_APS] = { 0 };   /* 追加 */

/* ・・・ 省略 ・・・ */

void ap_run(unsigned char pnum)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから）*/
int ap_enq_task(struct file *f, unsigned char pnum)
{
        int result = -1;

        spin_lock(&ap_task_lock[pnum - 1]);

        /* 空いていればタスクを登録 */
        if (!ap_task[pnum - 1]) {
                ap_task[pnum - 1] = f;
                result = 0;
        }

        spin_unlock(&ap_task_lock[pnum - 1]);

        return result;
}
/* 追加（ここまで）*/
```

空きがあれば登録し、無ければ何もせずエラーを返すだけです。

この関数も複数のコンテキストから呼ばれる可能性があるので、スピンロックでロックをとるようにしています。

以上で ap.c への関数追加は完了です。併せて include/ap.h を作成してこれらの関数のプロトタイプ宣言と、Makefile の"OBJS="への"ap.o"の追加を行っておいてください (共にコードは割愛)。

2.4.3 AP へ外部アプリを実行させる

作成した関数を使用して AP へ外部アプリを実行させてみます (リスト 2.10)。

▼**リスト** 2.10　024_exec/main.c

```
#include <x86.h>
#include <ap.h> /* 追加 */
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        unsigned char pnum = get_pnum();

        /* 変更（ここから） */
        /* 専用の初期化処理を行い、実行を開始する */
        if (pnum) {
                ap_init();
                ap_run(pnum);
        }
        /* 変更（ここまで） */

        /* フレームバッファ周りの初期化 */
        fb_init(&pi->fb);
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        clear_screen();
        is_con_inited = 1;

        /* プロセッサ番号表示処理は削除 */

        /* ACPI の初期化 */
        acpi_init(pi->rsdp);

        /* ・・・ 省略 ・・・ */

        /* ファイルシステムの初期化 */
        fs_init(_fs_start);

        /* 追加（ここから） */
        /* AP1 で外部アプリ実行 */
        struct file *app = open("puta");
```

```
        ap_enq_task(app, 1);    /* 1 回目 */
        while (ap_enq_task(app, 1) != 0) /* 2 回目を試みる */
                CPU_PAUSE();

        /* halt して待つ */
        while (1)
                cpu_halt();
        /* 追加（ここまで） */

        /* スケジューラの初期化 */
        sched_init();

        /* ・・・ 省略 ・・・ */
```

start_kernel() 冒頭の AP 用の条件分岐で、AP の初期化 (ap_init()) と実行 (ap_run()) を行っています。

その後、AP は ap_run() の中でタスク登録待ちになりますので、BSP 側はファイルシステム初期化 (fs_init()) 後、プロセッサ番号 1 の AP に外部アプリを実行させます。繰り返し実行できることの確認で 2 度実行しています。

最後に Makefile の OBJS へ ap.o を追加すれば完了です (コードの紹介は割愛)。

2.4.4 動作確認

動作確認用の外部アプリには、本シリーズのパート 3「システムコールの薄い本」で作成した「02_putc」というアプリを使用します。putc システムコールの実験アプリで、「A」という 1 文字を画面表示するだけのアプリです。このアプリのソースコードも「024_exec」内に置いています。

- 024_exec/apps/02_putc/

このディレクトリへ移動した後、「make」コマンドでアプリケーションバイナリができあがります。

```
$ make
gcc -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common -Iinclude -fPIE \
-c -o app.o app.c
gcc -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common -Iinclude -fPIE \
-c -o lib.o lib.c
ld -Map app.map -s -x -T ../app.ld -pie -o test app.o lib.o
```

「test」というバイナリ名で生成されますので、「puta」へリネームしてください。

```
$ ls test
test
$ mv test puta
$ ls puta
puta
```

そして、ファイルシステムイメージを生成するスクリプトも「024_exec」に含めています。以下のようにファイルシステムイメージ「fs.img」を生成します。

```
$ ../../tools/create_fs.sh puta
$ ls fs.img
fs.img
```

この fs.img をカーネル (kernel.bin) と同じ階層に配置してください。ファイルシステムイメージについて詳しくは、当シリーズのパート 1 である「フルスクラッチで作る!x86_64 自作 OS」をご覧ください。

実行すると、図 2.5 の様に「A」が 2 回表示される事が確認できました。puta を 2 回実行できているので良さそうです。

▲図 2.5　024_exec の実行結果

カーネルとアプリバイナリのグローバル変数に注意

今回まででカーネルと外部アプリを AP へ実行させることができるようになりました。
AP 側でカーネルやアプリを実行する際も、BSP 側と同様に、それらが配置されているアドレスへジャンプする方針で進めてきました。ただ、ここまで作ってきた自作 OS は、実行コンテキスト毎にメモリ空間を分けていたりもしないので[a]、グローバル変数やスタティック変数等のように、実行バイナリ内にその変数の実体がある変数は注意が必要です。BSP/AP 間で共に同じアドレスへアクセスすることになるので変数の内容がプロセッサ間で共有されます。スタックに実体がある関数内のローカル変数等では、当然、そのようなことは起きません。
本書では、その事をわかった上で、BSP/AP 間で同時に実行されるカーネルやアプリでは、プロセッサ間で共有してほしくない変数はスタックから確保されるように実装しています[b]。

[a] URFI が用意してくれた「物理アドレス=仮想アドレス」のページテーブルをそのまま使っているので、プロセス毎に別々のアドレス空間を用意していたりはしません。
[b] カーネルは、世の中のものも、BSP と AP でアドレス空間を分けていたりはしない (同じアドレスで動作しカーネルバイナリ内に BSP/AP の分岐がある) のかと思いますが、アプリは流石に BSP/AP を意識していちいち実装するのは面倒です。なので、やはりプロセス毎のメモリ空間の分離は実現しておいた方が便利です。が、今回に関して言えば、複数のプロセッサで同時に実行する際は、そのアプリをファイルごとコピーして物理的に別々のアドレスにする、という方法でも実現可能です。

2.5 AP での外部アプリ実行をシステムコール化する

前項で追加した AP での外部アプリ実行の仕組みをシステムコール化します。
この項のサンプルディレクトリは「025_syscall」です。

2.5.1 システムコールにエントリを追加する

「SYSCALL_EXEC_AP」というシステムコールのエントリを syscall.c に追加します (リスト 2.11)。

▼**リスト** 2.11 025_syscall/syscall.c

```
#include <ap.h> /* 追加 */
#include <intr.h>
/* ・・・ 省略 ・・・ */

enum SYSCCALL_NO {
        SYSCALL_PUTC,
        SYSCALL_OPEN,
        SYSCALL_EXEC,
        SYSCALL_ENQ_TASK,
        SYSCALL_RCV_FRAME,
        SYSCALL_SND_FRAME,
        SYSCALL_EXEC_AP,        /* 追加 */
        MAX_SYSCALL_NUM
};

unsigned long long do_syscall_interrupt(
        unsigned long long syscall_id, unsigned long long arg1,
        unsigned long long arg2 __attribute__((unused)),
        unsigned long long arg3 __attribute__((unused)))
{
        unsigned long long ret_val = 0;

        switch (syscall_id) {
        /* ・・・ 省略 ・・・ */

        /* 追加（ここから）*/
        case SYSCALL_EXEC_AP:
                ret_val = ap_enq_task((struct file *)arg1, arg2);
                break;
        /* 追加（ここまで）*/
        }

        /* ・・・ 省略 ・・・ */
```

前項で追加した ap_enq_task() をシステムコール経由で呼び出せるようにしました。システムコールの第 1・第 2 引数を ap_enq_task() の第 1・第 2 引数に渡し、ap_enq_task() の戻り値をシステムコールの戻り値として返すようにしています。

併せて、main.c の start_kernel() でプロセッサ番号 1 の AP に外部アプリを実行させて止めていた処理を削除し、通常通りスケジューラまで実行されるようにします (リスト 2.15)。

▼**リスト** 2.15　025_syscall/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* ファイルシステムの初期化 */
```

```
        fs_init(_fs_start);

        /* AP1 への外部アプリ実行と、
         * その後の処理を止めていた while() を削除 */

        /* スケジューラの初期化 */
        sched_init();

        /* ・・・ 省略 ・・・ */
```

2.5.2 システムコールを使ってみる

それでは、追加した SYSCALL_EXEC_AP システムコールを使ってみます。

システムコールを呼び出すアプリケーションはサンプルディレクトリ内の以下の場所に作成することにします。

- 025_syscall/apps/07_exec_ap/

ここでは、前著「ぼくらのイーサネットフレーム」で最後に作成した「06_beef_server」からの差分のみ紹介します。(06_beef_server も apps ディレクトリに入っています。)

まず、アプリ側のライブラリ関数を定義している lib.c へ、SYSCALL_EXEC_AP を関数化した「exec_ap()」を追加します (リスト 2.13)。

▼**リスト** 2.13　025_syscall/apps/07_exec_ap/lib.c

```
/* ・・・ 省略 ・・・ */

enum SYSCCALL_NO {
        SYSCALL_PUTC,
        SYSCALL_OPEN,
        SYSCALL_EXEC,
        SYSCALL_ENQ_TASK,
        SYSCALL_RCV_FRAME,
        SYSCALL_SND_FRAME,
        SYSCALL_EXEC_AP,        /* 追加 */
        MAX_SYSCALL_NUM
};

/* ・・・ 省略 ・・・ */

void exec(struct file *file)
{
        syscall(SYSCALL_EXEC, (unsigned long long)file, 0, 0);
}

/* 追加（ここから） */
int exec_ap(struct file *file, unsigned char pnum)
{
```

```
        return (int)syscall(SYSCALL_EXEC_AP, (unsigned long long)file, pnum, 0);
}
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */
```

定数 SYSCALL_EXEC_AP と、exec_ap() を追加し、exec_ap() では syscall() を使ってシステムコールを呼び出しています[*3]。

また、この関数追加に併せて、include/lib.h へ exec_ap() のプロトタイプ宣言を追加しておいてください (コードは割愛)。

そして、アプリ本体である app.c はリスト 2.14 のように作成します。

▼**リスト** 2.14　025_syscall/apps/07_exec_app/app.c

```
#include <lib.h>

#define MAX_EXEC_COUNT 2

int main(void)
{
        unsigned char i;
        for (i = 0; i < MAX_EXEC_COUNT;) {
                if (exec_ap(open("puta"), 1) == 0)
                        i++;
        }

        return 0;
}
```

exec_ap() が 2 回成功するまで繰り返しています。実行する外部アプリは前項と同じく puta です。

このアプリは BSP 側に実行させます。このアプリから return してくると、halt して待機し続けます (リスト 2.15)。

▼**リスト** 2.15　025_syscall/main.c

```
/* ・・・ 省略 ・・・ */

#define INIT_APP       "test"

/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
```

[*3] システムコールの仕組みについて詳しくは、当シリーズのパート 3 である「システムコールの薄い本」をご覧下さい。

```
                    void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* init アプリ起動 */
        exec(open(INIT_APP));

        /* halt して待つ */
        while (1)
                cpu_halt();
}
```

07_exec_app のビルドも 02_putc と同様です。BSP で実行する init アプリ名 (INIT_APP 定数) は「test」なので、生成される test バイナリをリネームする必要はありません。

ただし、fs.img には test と puta の両方を含めておく必要があるので、test と puta を同じディレクトリに配置し、以下のように create_fs.sh を実行します。(create_fs.sh への相対パスは適宜変更してください。)

```
$ ls test puta
test puta
$ ../../tools/create_fs.sh test puta
$ ls fs.img
fs.img
```

2.5.3 動作確認

実行すると、前項と同様に A が 2 回表示されます。(前項と同じなので実行結果の画像は割愛)

おわりに

本書をお読みいただきありがとうございました！

本書では、マルチコアを手っ取り早く制御する方法として UEFI の機能を利用して実現する方法を紹介しました。16 ビットのリアルモードから 32 ビットのプロテクトモード、そしてそこから 64 ビットのロングモードへのモード移行に関わる処理を書かなくて済むのは大分楽ですね。

カーネルへジャンプした後は、特定のプロセッサへ任意のアプリを実行させるところまでを、手っ取り早い方法で紹介しました。「同一実行ファイルを複数のプロセッサで実行するとグローバル変数が共有される」など、まだまだ作り込みが雑なところはありますが、マルチコアを試す第一歩には十分かと思います。

ともかくこれで、マルチコアを動かすことができました！ この後は、「マルチコア時のスケジューリングはどうするか」とか、「アプリケーションでマルチコアを扱うときのフレームワーク」とか、あるいは「マルチコアを使った面白いアプリを作ってみる」などなど、独自な自作 OS へ発展していくと面白いんじゃないかと思います。

参考情報

元ネタ

- 「UEFI ベアメタルプログラミング - マルチコアを制御する」
  - https://yohgami.hateblo.jp/entry/20170503/1493787477

参考にさせてもらった情報

- UEFI Platform Initialization Specification
  - https://uefi.org/specifications
  - UEFI の EFI_MP_SERVICES_PROTOCOL の仕様が書かれています
    - バージョン 1.7 現在、13 章の「DXE Boot Services Protocol」に書かれています
- Intel 64 and IA-32 Architectures Software Developer Manuals
  - https://software.intel.com/en-us/articles/intel-sdm
  - Intel 公式の x86 CPU のソフトウェア開発マニュアルです
  - 2019 年 5 月更新の最新版時点で、マルチコアについては Volume 3 の「CHAPTER 8 MULTIPLE-PROCESSOR MANAGEMENT」に、APIC についても同じく Volume 3 の「CHAPTER 10 ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER(APIC)」に記載されています。

開発リポジトリ

本にする前のネタ作り等、本書で扱った OS の開発は以下のリポジトリで行っていますので、興味があれば見てみてください。

- ビルドスクリプト
  - https://github.com/cupnes/yuaos

- ブートローダー
  - https://github.com/cupnes/poiboot
- カーネル
  - https://github.com/cupnes/yuakernel
- アプリ
  - https://github.com/cupnes/yuaapps

本シリーズの過去の著作

サークル「へにゃぺんて」の同人誌は、「フルスクラッチで作る!」シリーズ含め、すべて以下の URL から PDF 版/HTML 版は無料でダウンロード/閲覧できます

- http://yuma.ohgami.jp

「フルスクラッチで作る!」シリーズの著作としては以下があります。

- フルスクラッチで作る!UEFI ベアメタルプログラミング! パート 1、パート 2
  - ブートローダーを作る上で必要な UEFI を直接叩く方法を紹介
- フルスクラッチで作る!x86_64 自作 OS(パート 1)
  - ハードウェアを抽象化しアプリへ提供するカーネルの基本的な枠組みを作り上げる
  - カーネルの機能を利用するアプリとして「画像ビューア」を作る
- フルスクラッチで作る!x86_64 自作 OS パート 2
  - 副題: ACPI で HPET 取得してスケジューラを作る本
- フルスクラッチで作る!x86_64 自作 OS パート 3
  - 副題: システムコールの薄い本
  - システムコールを実装しカーネルとアプリを分離する
- フルスクラッチで作る!x86_64 自作 OS パート 4
  - 副題: ぼくらのイーサネットフレーム
  - NIC ドライバを自作し、受信したイーサネットフレームを表示してみたり、独自のバイナリ列を送信してみたりする本

フルスクラッチで作る!x86__64 自作 OS パート 5 てっとりばやくマルチコア

2019 年 8 月 12 日　コミックマーケット 96 v1.0
著　者　大神祐真
発行所　へにゃぺんて
連絡先　yuma@ohgami.jp
印刷所　日光企画

(C) 2019 へにゃぺんて