IC 2006

新しいマルチコアプロセッサ Cellで遊ぼう

Kinuko Yasuda
Fixstars Corp.

23 October, 2006

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

Outline

- **自己紹介**
- **Cellで遊びはじめるための情報と技術を紹介**
  - ✓ **Cellアーキテクチャの概要**
  - ✓ **Cell上のプログラミングの概要**
  - ✓ **Cell SDK 1.1のlibspeによるSPEの抽象化について**
  - ✓ **Cell LinuxのspufsによるSPEの抽象化について**
  - ✓ **SPEプログラムを扱う小テクニックの紹介**

Who We Are

株式会社フィックスターズと
Cellへの取り組みの紹介

23 October, 2006

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

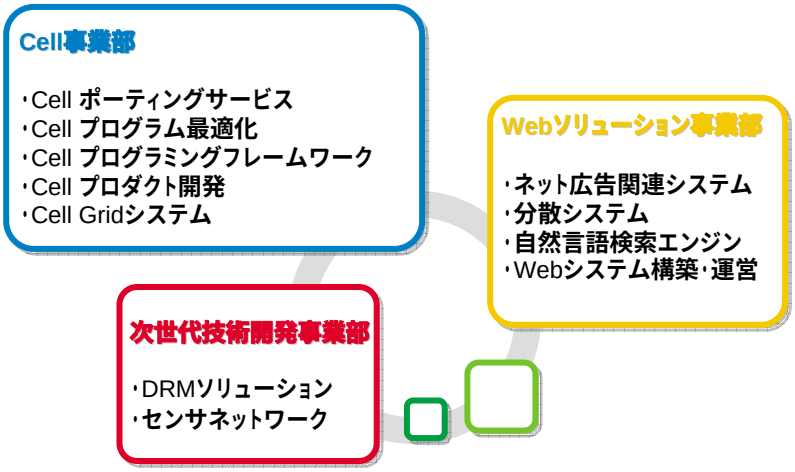
株式会社フィックスターズ

- **2002年設立、現在社員約40名**
- **2004年7月からCellへの取り組みをスタート**
  - ✓ 2004年から東芝様パートナーとしてCell事業を開始
  - ✓ 2006年からソニー様、SCE様とも取引開始

http://www.fixstars.com/

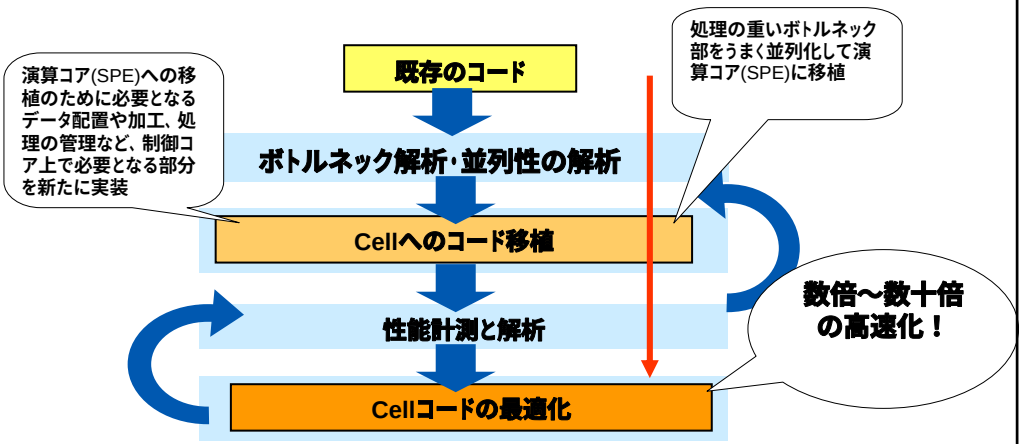
フィックスターズの事業内容

➔**Cell事業を中心に、最先端のソフトウェア技術をフルに使ったソリューションを提供しています**



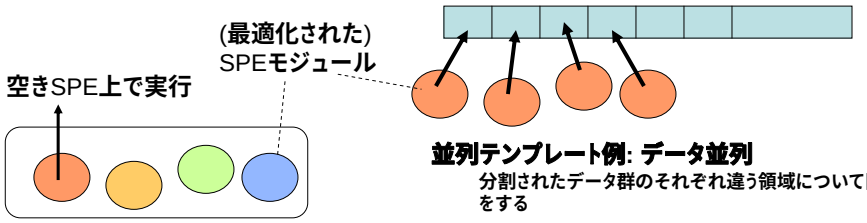
Cellへの取り組み(1) - Cellポーティングと最適化

➔**3DCGレンダリング、テンプレートマッチング、行列演算処理など、各種処理のCellプログラムの移植と最適化を請け負っています**



Cellへの取り組み(2) – IPAオープンソフト事業

- **Cell上のプログラム開発を支援するオープンソフトのフレームワークを開発中**
  - ✓ **コンピュータとしてCellの上で「遊んで」みたい人を支援するフレームワーク**
    - **誰かが書いたSPEモジュールを組み合わせてプログラミング**
    - **並列テンプレートによる並列化支援**
    - **「身近な」並列スーパーコンピュータ上のプログラミング環境を提供**



並列テンプレート例: プロセッサファーム
タスクキューから次々とSPEジョブを投入・実行

並列テンプレート例: データ並列
分割されたデータ群のそれぞれ違う領域について同じことをする

Cellへの取り組み(3) – その他

- **Cellプロダクト開発**
- **Cellグリッド実証実験**
- **Cellプログラミング普及活動**
  - ✓ **Cell開発者をバックアップするCell Wikiの運営**
    - **プログラミングチュートリアル**
    - **入門編・SIMD化編・ポーティング編**
  - ✓ **無料セミナー活動**
    - **Cellプログラミングと最適化**
  - ✓ **有料セミナー活動もやっています**
  - ✓ **書籍執筆 (予定)**

http://cell.fixstars.com/

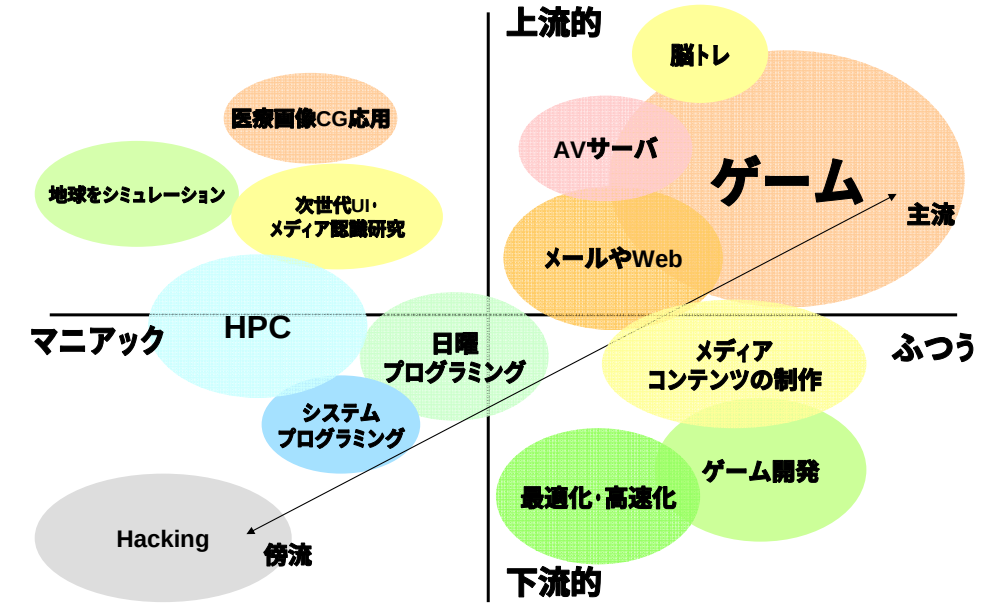
Introduction –
How We Can Play with the Cell

前置き - Cellをどう遊ぶか

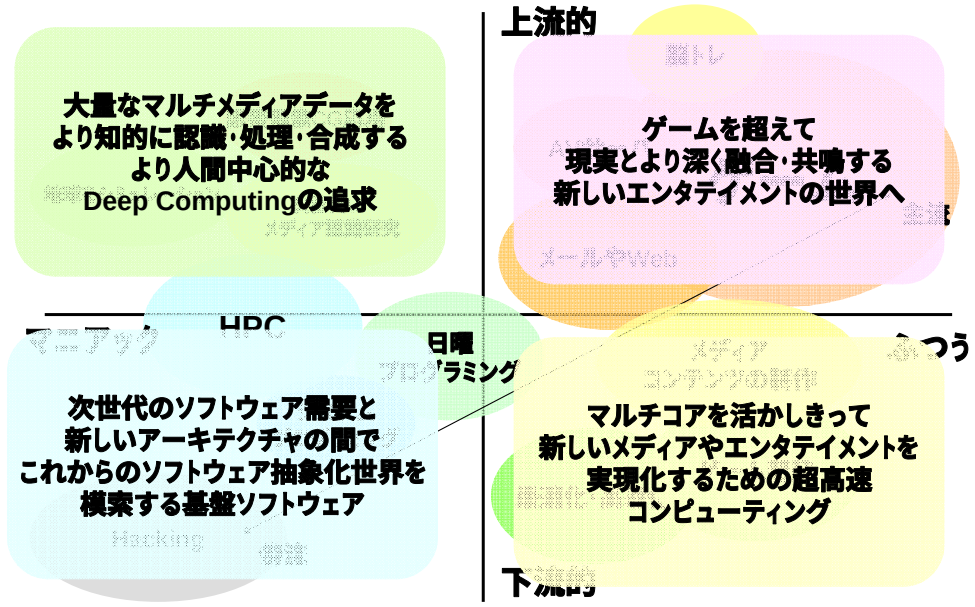
23 October, 2006

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

Cellの遊び方を想像する

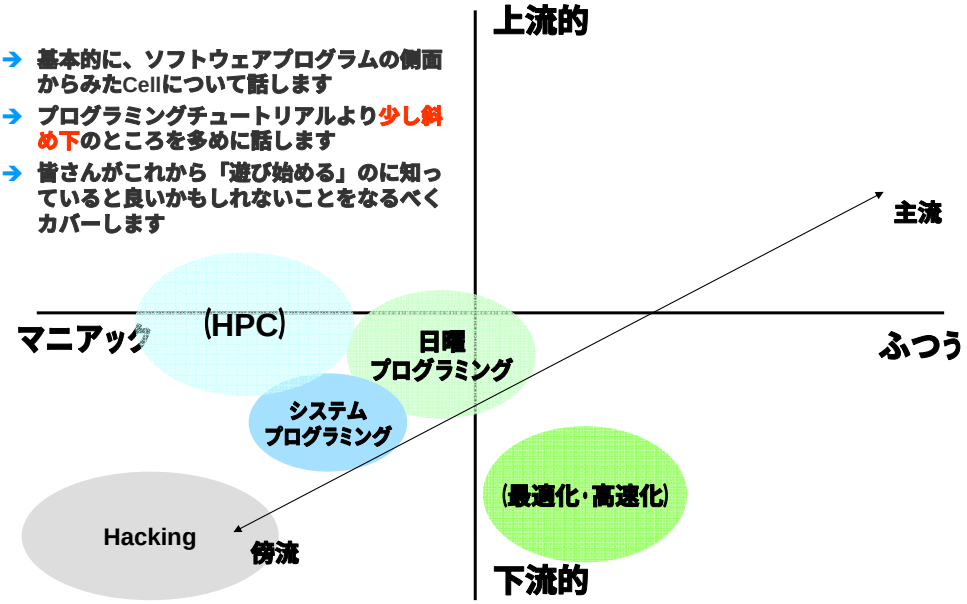


Cellの遊び方を想像(創造)する



この講演で導入する遊び方

- 基本的に、ソフトウェアプログラムの側面からみたCellについて話します
- プログラミングチュートリアルより少し斜め下のところを多めに話します
- 皆さんがこれから「遊び始める」のに知っていると良いかもしれないことをなるべくカバーします



なにはともあれ、情報を手に入れる

- **アーキテクチャドキュメント**
  - ✓ SCEI公式アーキテクチャドキュメント (日・英)
    - SCEIサイトから日・英両方のドキュメントを入手可能
    - http://cell.scei.co.jp/
- **情報公開サイト**
  - ✓ Cellユーザ情報公開サイト (日本語)
    - 2006年10月から東芝がcell user's groupサイトをオープン
    - http://www.cellusersgroup.com/
  - ✓ IBM Cell Resource Center (英語)
    - IBMのCell Linux, Cell SDK, Cell Bladeに関するサイト
    - http://www-128.ibm.com/developerworks/power/cell/
  - ✓ IBM Cellアーキテクチャ情報交換フォーラム (英語)
    - IBM開発者向けサイトのCell用フォーラム
    - http://www-128.ibm.com/developerworks/forums/dw_forum.jsp?forum=739&cat=46
  - ✓ Fixstars Cell Wiki (日本語)
    - プログラミングチュートリアルなど
    - http://cell.fixstars.com/

Cell実機の状況は？

- **Cellシミュレータ (2006年10月現在入手可能)**
  - ✓ IBMがIntel PCで動作するフルシステムシミュレータを公開
  - ✓ http://www.ibm.com/developerworks/power/cell/
- **東芝Cellリファレンスセット (2006年10月現在入手可能)**
- **IBM Cell Blade (2006年10月現在入手可能)**
  - ✓ 各社へお問い合わせください
- **Sony PLAYSTATION 3**
  - ✓ 2006年11月11日発売予定

今からCellで遊ぶなら…

- ✓ シミュレータであれば今すぐ遊べます
- ✓ あと少しなのでPLAYSTATION 3を待つ
- ✓ ちょっと奮発して東芝やIBMのCell実機を買う

Cell Linuxの状況は?

- Cell SDK / Linux on Cell
  - ✓ IBM, BSC (Barcelona Supercomputing Center)がCell Linux (2.6.16ベース), コンパイルツール, デバッガ, ライブラリ類を公開
  - ✓ http://www.alphaworks.ibm.com/tech/cellsw
  - ✓ http://www.bsc.es/projects/deepcomputing/linuxoncell/
- Cell Linux
  - ✓ 本流ラインに最新Cellパッチがかなりマージされています
  - ✓ http://www.kernel.org/
  - ✓ http://patchwork.ozlabs.org/linuxppc/
- PS3 Linux
  - ✓ Terra SoftよりPS3用Linux “Yellow Dog Linux” 11月中旬発売予定
  - ✓ http://www.terrasoftsolutions.com/products/ydl/

今からCell Linuxで遊ぶなら…

- ✓ Cell SDK / Linux on Cell + IBMシミュレータなら今すぐ遊べます
- ✓ Cellリファレンスセットあるいは Cell Bladeを買えばLinux環境です
- ✓ PS3が出るのを待ってYellow Dog Linuxを買う

➔ 基本的にどのCellもLinuxコンピュータとして使えそう

この講演で仮定する環境

➔基本的に今すぐ一般入手可能な環境をベースにします

- ✓ Cell + Linux
- ✓ IBM, SCEI, BSCが開発・公開しているCell SDK 1.1環境
  - GNU toolchain 3.2 (gcc 4.0.2ベース)
  - Linux 2.6.16 + Cell Patches
  - SPE Management Library (libspe-1.1.0)
  - Full System Simulator (Cellマシンがない場合…)
- ✓ 環境の入手と構築方法
  - http://www.bsc.es/projects/deepcomputing/linuxoncell/
    - – ここの “Cell BE SDKs” ➞ ”SDK 1.1” から必要なrpmをダウンロードしてインストール
  - X86, x86-64, Powerpc64, IBM Cell blade serverであればそのまま使えます (x86, x86-64, ppc64ではsimulator使用が前提)

The Cell Architecture and Programming on the Cell

Cellのアーキテクチャと
Cell上のプログラムミングの概要

23 October, 2006

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

Cellアーキテクチャの概要

➔**9個のコアを持つ高性能マルチコアプロセッサ**

- ✓ **STI (Sony/SCEI, Toshiba, IBM) 共同開発**
- ✓ **Cellプロセッサ開発の背景**
  - ■ **家庭のエンタテイメントからグリッドコンピューティングまで、様々な演算要求に対応する新しいアーキテクチャ**
  - ■ **「スーパーコンピュータを1チップに」**

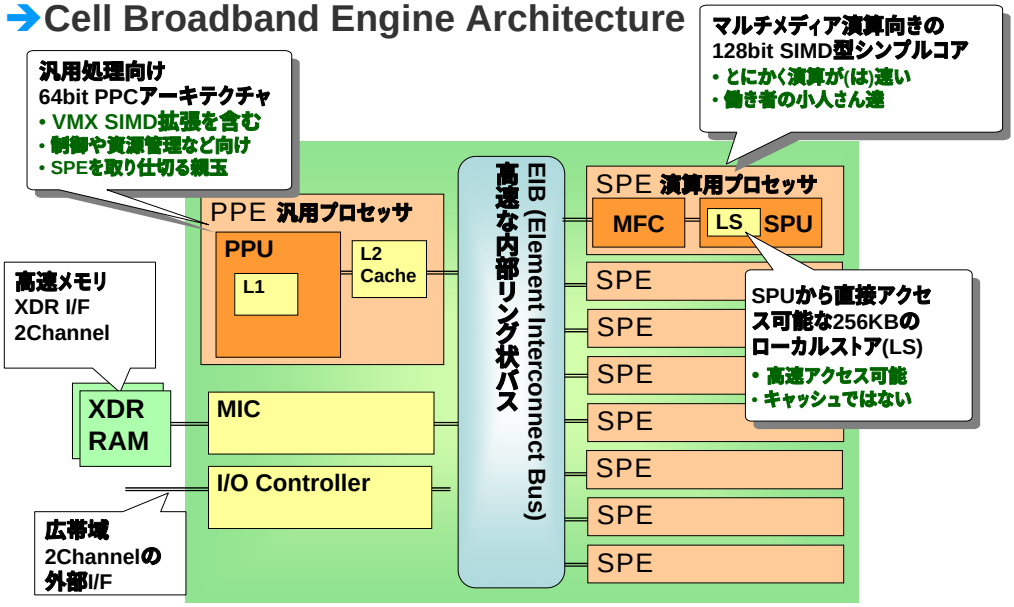
> **・最大浮動小数演算性能**
> **200GFlops以上 (単精度, 3.2GHz)**
> **・最大メモリ帯域幅 25.6GB/s (XDR RAM)**
> **・最大I/O帯域幅 76.8GB/s (FlexIO®)**

➔**2種類のコアを持つ非対称マルチコア**

- ✓ **低消費電力への要求とメディア演算処理への需要へのCellの解**
  - ■ **1個の制御用・汎用処理用コア…PPE (PowerPC Processor Element)**
  - ■ **複数のマルチメディア演算用コア…SPE (Synergistic Processor Element)**

Cellのハードウェア基本構成

Cell Broadband Engine Architecture



ソフトウェアから見たPPEの特徴 (Highlights)

PPE上のプログラミング

- ✓ 基本的に一般的なPowerPC上のプログラミングと同じ
- ✓ PPC, PPC64のrpmをそのままインストール・実行可能
  - インオーダー実行 (PPC 970などではアウトオブオーダー)
  - 2ウェイスレッディング (PPC 970では5段パイプライン)
  - VMX SIMD命令 (AltiVec) を標準搭載
- ✓ 演算処理は基本的にSPEに比べると苦手
  - SPEで実行するプログラム部分を増やせば増やすほど性能向上
  - SPE処理に対応する部分も高速化しないとボトルネックになる
    - SIMD化、2ウェイスレッディングの活用など
- ✓ プロセッサの性能特徴やハードウェア的な制限から、OSなどのシステムソフトウェア・制御ソフトウェアはPPE上で実行させるのが普通

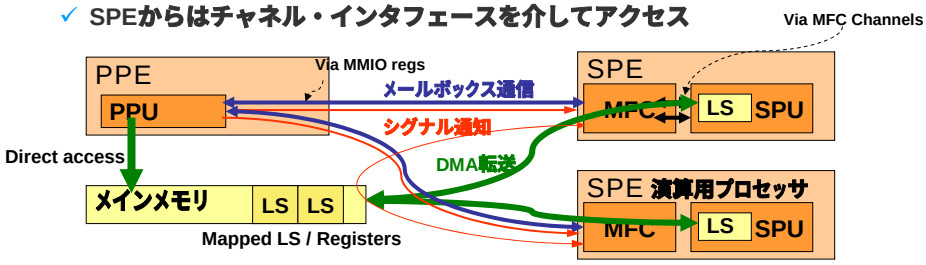
ソフトウェアから見たSPEの特徴 (Highlights)

- SPE上のプログラミング
  - ✓ ほぼフルC/C++プログラミングが可能
  - ✓ 128-bit SIMDプロセッサの特徴を活かすことで高速演算が可能
    - 潤沢な(128本の)128ビットレジスタ
    - SIMD命令による4並列同時演算処理
    - コンパイラによる自動最適化はまだまだ
  - ✓ “SPU Intrinsic” によりCの関数的にアセンブラ命令を記述可能
    - SIMD命令などもCの関数的にレジスタ名ではなく変数名を使って書ける
  - ✓ パイプラインを意識して最適化することで2命令同時発行可能
  - ✓ メインメモリへのアクセスはDMA転送が必要
    - DMA転送されるメモリ領域は128ビット境界などにアラインされている必要あり
  - ✓ システムプログラム的観点からは…
    - ユーザモードと特権モードの区別はない
    - ページテーブルなどはアクセスできない (PPE特権ソフトウェアのみアクセス可)
    - イベントを処理する割り込みハンドラ的なものを実装することは可能

PPEとSPE間のインタフェース

- MFCがSPUと外部とのインタフェースを提供
  - ✓ DMA転送
    - SPE LS (ローカルストレージ)とメインメモリ間の転送
    - メインメモリにLSをマップすることで、SPE間の転送も可能
  - ✓ メールボックス
    - PPE – SPE間で32bit整数をやり取りできるレジスタ
    - PPE → SPE (inbound) に4スロット
    - SPE → PPE (outbound) に1スロット + 割り込み用1スロット
  - ✓ シグナル通知
    - PPEからSPEへシグナル(32bit整数)を通知できる機構
    - レジスタをメインメモリにマップすることで、SPE - SPE間のシグナリングも可
- MFCへのアクセス
  - ✓ PPEからはMMIOレジスタを介してアクセス
  - ✓ SPEからはチャネル・インタフェースを介してアクセス

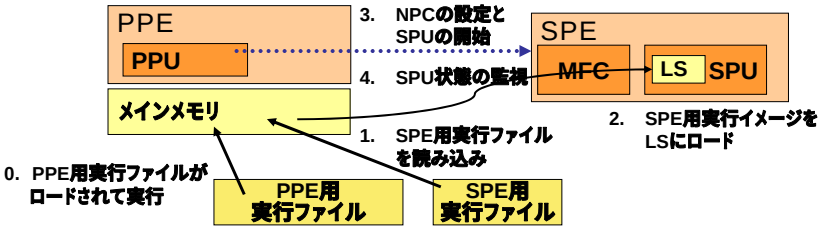
その他、PPEはメモリマップドのSPUレジスタを通じてSPUの実行制御が可能
- プログラムカウンタの設定
- SPUの開始・停止
- SPUの実行状態監視



PPEからSPEを使う

- 普通にしているとただのPowerPC
  - ✓ PowerPC用Linux, PowerPC用ユーザアプリケーション
  - ✓ OSはPPEで動作してプロセスもPPE上で起動

- PPEからSPEを使う超基本シーケンスは…
  1. SPU用にコンパイルされたSPEバイナリファイルを読み込む
  2. DMA転送やマップドエリアを介してSPEバイナリをLSにロード
  3. 引数と引数ローダをLSにロード (引数は後に汎用レジスタに置かれる)
  4. SPU NPC（Nextプログラムカウンタ）レジスタを設定
  5. SPU制御レジスタにSPU開始要求を書き込む
  6. SPU状態レジスタを見張って、停止状態になったら必要な処理をする



SPEプログラムを実行するPPEコードの骨格

- SPEコードを読み込んで実行するPPE側の擬似コード

```
main()
{
   open_spe_elf (&spe_elf, "SPE実行ファイル");
   load_elf_to_ls(spe_elf, &npc);
   start_spu(&spu, npc, 引数など);

    while (1) {
       if (stopped(spu, &stop_code)) {
          /* 停止タイプを見てイベントや
             システムコールなどの処理 */
          if (stop_code =
                stopped_by_stop)
             /* 実際に停止を示すなら終了 */
             break;
       }
       /* SPE実行中にPPEでやる処理など */
    }
    cleanup_spu(spu);
}
```

- SPEコードを読み込んで実行するだけのCell SDK libspe1.1 によるPPEコード
  - ✓ SPE上の実行体をスレッドとして抽象化

```
#include <libspe.h>

main()
{
   spe_program_handle_t *p;
   speid_t spe;
   int status;
   p = spe_open_image("SPE実行ファイル");
   spe = spe_create_thread(0, &p, 引数, NULL,
         -1, 0);

   /* SPE実行中にPPEでやる処理 */

   spe_wait(spe, &status, 0);
   spe_close_image(p);
}
```

SPE ELFファイルのオープンとmmap

SPU上での実行を開始 (スレッド的な抽象化)

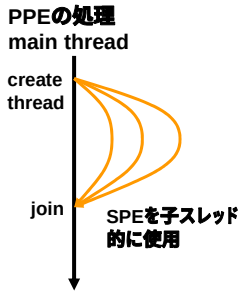
SPEスレッドの終了待ちと資源の解放

SPEプログラムは、最適化やDMA転送を考えなければ基本的なCプログラム (printf("hello, world"); など) を普通に書けば動きます

スレッド抽象化によるSPEプログラミング

➔スレッド抽象化によるSPEの利用

- ✓ プログラム全体の制御ロジックはPPE上で動作
- ✓ 処理が重いところをマルチコア並列化(SPE化)
  - ■ プロファイリングによりSPE化すべき部分を絞り込む
  - ■ データ並列、タスク並列、スレッドプールなどのマルチスレッドのプログラムモデルに似たモデルを適用可能
  - ■ 一般的なマルチスレッドモデルに対し、ソフトウェア的にはメモリ共有モデルではないので注意
- ✓ 並列化部分が終わったら処理をjoin



➔コードをSPE化するにあたってのコツ

- ■ SPEの処理に必要なデータはDMA転送しやすいようにアライン
- ■ さらにSIMDベクトル演算しやすいように128ビット単位でデータをパッキング
- ■ SPE処理中になるべくPPEも並列に処理をする
- ■ 互いに依存なく複数SPEに処理を分割できる場合は複数SPEで並列化
- ■ SPE中ではできる限りコードをSIMD化
- ■ ダブルバッファなどによってDMA転送とSPE処理もできるだけ並列化
- ■ 条件分岐外しやLoop Unrolling、Odd/Even命令並べ替えなどによりパイプラインが埋まるよう最適化

Cellにおけるプログラミングモデル

➔2種類のコアを活用するCellプログラミングとは

- ✓ 異なる特性を活かして機能分担する
  - ■ 制御的・管理的役割… PPE
  - ■ マルチメディア演算・浮動小数演算… SPE
- ✓ 複数のコアを活用して並列化する
  - ■ SIMD化、マルチバッファ化などの並列化も考慮

機能特性にこだわりすぎるとPPE部分がボトルネックになりやすいので注意
(PPEの制御がSPEに追いつかなくなりがち)

➔Cellにおける典型的プログラミングモデル

- ✓ PPE-Centric Model
  - ■ PPEプログラムを主体とする
  - ■ ボトルネック解析してSPEへ処理が重い部分をオフロード・最適化
- ✓ SPE-Centric Model
  - ■ SPEプログラムを主体とする
  - ■ 最初から並列分散プログラムとして設計
  - ■ 様々なモデルが考えられるが、下位フレームワークの支援が必須 (Ex. SPURS)

作成するプログラムの性質や要求にあわせて自由に開発方針や環境を選択できるのが望ましい
ハイブリッド的モデルもあり得る

Exploring the Libspe

Cell SDKのSPE管理ライブラリlibspe
について少し調べてみる

23 October, 2006

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

ユーザプログラムの下では何が起きているのか…

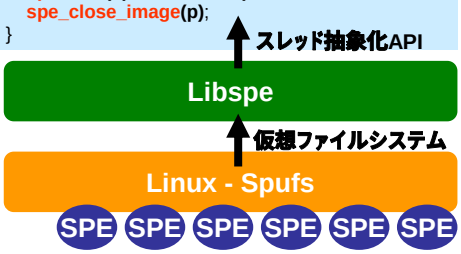
➔Cell SDK 1.1による最も単純なプログラム(再訪)

```
#include <libspe.h>

main()
{
   spe_program_handle_t *p;
   speid_t spe;
   int status;
   p = spe_open_image("SPE実行ファイル");
   spe = spe_create_thread(0, &p, 引数, NULL,
       -1, 0);

   /* SPE実行中にPPEでやる処理など */

   spe_wait(spe, &status, 0);
   spe_close_image(p);
}
```



- ✓ **Cell SDK libspe1.1におけるSPEの抽象化は*SPE Thread***
- ✓ **その下のカーネルでは*Spufs*という仮想ファイルシステムとしてSPEを抽象化**
  - **この2つはどのようにレイヤを切ってどうインタラクションしているのか?**
  - **これとは違う抽象化のシステムを作る場合に、どういうことを考える必要がある?**

Libspe 1.1を追いかける

➔ソースコードの入手

- ✓ http://www.bsc.es/projects/deepcomputing/linuxoncell/
  - Cell BE Components → libSPE からソースパッケージをダウンロード
  - 主要部分は3,000行程度の小さなライブラリ
  - 参照コードとして大変手頃

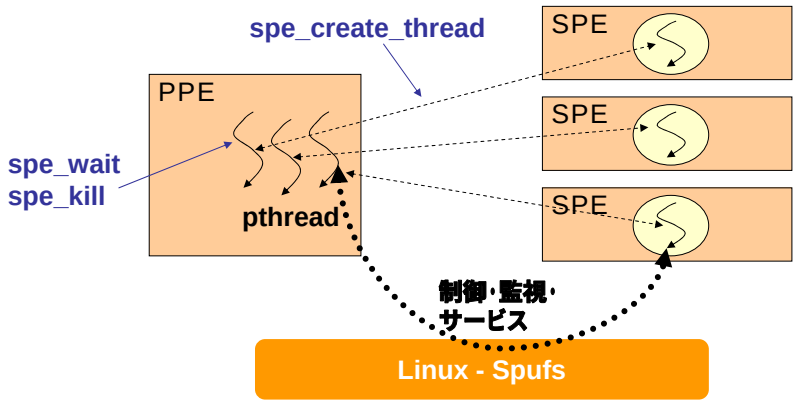
➔重要なライブラリコール

- ✓ spe_create_thread
  - ELF SPEイメージからSPEプログラムをSPEにロードして実行開始

他にもいろいろあるが、とりあえずspe_create_thread（とその周辺の関数）を見ればlibspe1.1の設計思想は大体わかる

Libspe 1.1におけるSPEスレッドの概要

- ➔ SPEスレッドを生成すると対になるpthreadが必ず作られ、そのpthreadがspufsを通じてSPE上の実行体の制御・監視・サービスを行う
- ➔ spe_waitやspe_killなどのAPIは実質そのpthreadに対するpthread_joinやpthread_cancelを呼び出す



Libspe 1.1によるSPEスレッド生成 (1)

➔spe_create_threadの擬似コード

```
speid_t
spe_create_thread(spe_program_handle_t *handle, void
*argp, void *envp)
{
    struct thread_store *thread;

    thread = calloc(1, sizeof(*thread));
    rc = spe_create ("/spu/spethread-スレッドID");
    thread->fd_run = rc;

    memfd = open("/spu/spethread-スレッドID/mem");
    void *spe_ls_mem = mmap(0, LS_SIZE, memfd);
    load_spe_elf(handle, spe_ls_mem);

    spe_open_files(thread,"/spu/spethread-スレッドID");

    pthread_create(&thread->spe_thread, NULL,
        spe_thread, thread);

    return thread;
}
```

Cell用システムコール spe_create を呼んでSPEスレッド用のspufsディレクトリを作成

spe_createは指定されたディレクトリ以下にSPEの各資源 (メモリ、レジスタなど) を仮想ファイルとして見せる

Spufsの見せる仮想ファイル"mem"をオープン (SPEのメモリ = LS) し、そこにELFイメージをELFプログラムヘッダに従ってロード (memcpy)

SPEスレッド制御のために、spufsの提供する主要なファイルをすべてオープンして内部構造体thread中のメンバ変数として保持

SPEスレッドのサービングスレッドとして動作するpthreadを1つ作って関数自体は終了

Libspe 1.1によるSPEスレッド生成 (2)

➔SPEスレッドと対になるpthread関数の擬似コード

```
Void *spe_thread(struct thread_store *thread)
{
   unsigned int npc, stop_code, status;

   do {
      int ret = spe_run (runfd, &npc, &status);
      stop_code = ret >> 16;

      if ((stop_code && 0xff00) == 0x2100) {
         int callnum = code & 0xff;
         handler = handlers[callnum];
         rc = handler(thread->mem_mmap, npc);
         npc += 4;
      } else if (code < 0x2000) {
         /* イベント待ち */
         pthread_cond_wait(&thread->event_deliver);
         thread->event_pending = 1;
      }
   } while (stop_code != spe_really_stopped);

   return stop_code;
}
```

Cell用システムコール spe_run を呼んでSPEスレッドのSPUを開始

spe_runはSPE側のプログラムが停止したら戻ってきて停止状態を示す"stop_code"を返す

Cell SDK的にstop_codeが0x21xxだったらPPU側のハンドラを呼び出すSPU-syscallとされているため、登録されているハンドラを呼び出してNPCを進める

Cell SDK1.1ではprintf, fopen, freadなどのC99/POSIX標準関数群をこれでサーブしている (SPE側libc内でstop命令を呼ぶ)

stop_codeが本当にSPUの停止や終了を示していそうだったらループを終了

spe_wait中のpthread_joinで回収

Libspe 1.1によるSPEスレッドのまとめ

➔特徴と長所

- ✓ シンプルな設計・馴染みやすいAPI
- ✓ SPE側のlibcと連携することで主要な標準関数を利用可能
- ✓ pthreadと一対一でSPEを管理する

補足：PS3発売と前後して、よりプリミティブなAPIを持つlibspe2が発表されました (2006/11/11)

libspe2?

➔短所

- ✓ SPEスレッド生成が比較的高コスト(3.2GHz環境で数msec)
  - ■ spe_create_thread中にシステムコールをやたらと呼ぶ
  - ■ 必ずpthreadを1つ新規に生成する
- ✓ API粒度がかなり荒い
  - ■ メモリへのロードとスレッド生成・開始が同じAPIなので、複数スレッドを低遅延でなるべく同時に開始させたい場合などには一工夫いる
  - ■ SPEスレッド待機と資源解放のタイミングが同じなので、一度停止するのを待って再び再開させるようなコードを書くには一工夫いる

FIXSTARS

Exploring the Cell Linux

Cell LinuxのSpufsにも詳しくなってみる

23 October, 2006

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

Cell Linuxを追いかける

- ➔Spufs抽象化に関する情報ポインタ
  - ✓ http://www-128.ibm.com/developerworks/power/library/pa-cell/
- ➔Cell Linuxのソースコードの入手
  - ✓ http://www.bsc.es/projects/deepcomputing/linuxoncell/
    - Cell BE Components → Linux Kernel からパッチをダウンロードできる
    - Linux 2.6.16 に対するおよそ97個のパッチ
    - Kernel.orgからダウンロードできる Linux-2.6.16に対し、 BSCサイトからダウンロードできるapply_patches.shかquiltパッチツールで全パッチを適用
      - Linux 2.6.16 ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.16.tar.bz2
  - ✓ 現在ではLinux-2.6.18, Linux-2.6.19-rc2などにかなりのパッチが取り込まれている
- ➔Cell特有コードを追いかけるときの主要ポイント
  - ✓ arch/powerpc/platforms/cell 以下
  - ✓ include/asm-powerpc/spu*.h

Spufsの概観とLibspe 1.1のSPEスレッド再考

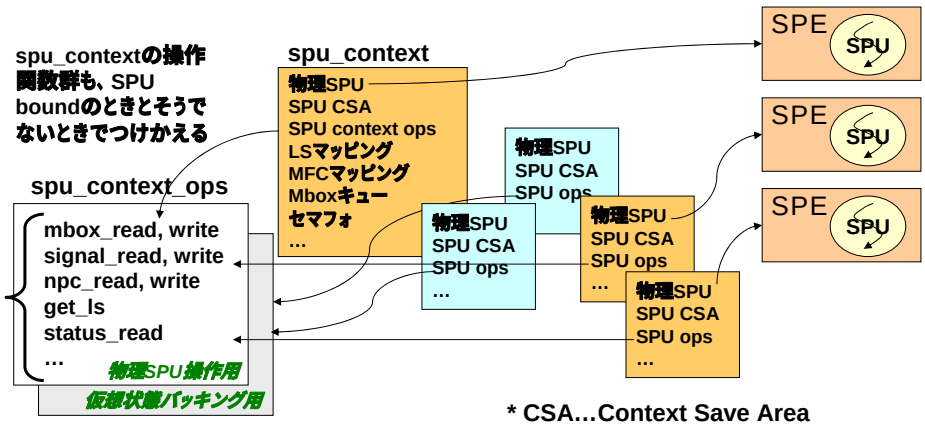
- ➔Libspeのコードからspufsの大体のusageは分かる
  - ✓ Spu 1つを1ディレクトリとして抽象化
  - ✓ Spu に関連する各資源や制御を仮想ファイルとして抽象化
  - ✓ Spu利用を開始するための新システムコールとして spe_create
  - ✓ Spuの実行を開始するための新システムコールとしてspe_run
  - … なんとなく満足？
- ➔「SPU1つ」？
  - ✓ 古来より「スレッド」とか「プロセス」と言えば「プロセッサの仮想化」
  - ✓ 仮想化の1つのポイントは、「物理プロセッサ数の隠蔽」
  - ✓ SPEスレッドも物理SPU数を超えてSPEスレッドを作れるの？

→ 作れます。Spufsが仮想化しています

SpufsによるSPEの仮想化の概要

➔物理SPUを仮想化してユーザレベルプログラムに提供

- ✓ 仮想化されたSPUはspu_context構造体として管理
- ✓ 物理SPUにバインドされている状態とされていない状態がある



Spufs用新システムコール: spe_create

➔ カーネル内のspe_createのエントリ: sys_spu_create

- ✓ spe_create_threadの頭の方で呼ばれてSPEスレッドのためのSPU contextを初期化する

sys_spu_createの擬似コード

```
asmlinkage long sys_spu_create(const char *pathname,
unsigned int flags, mode_t mode)
{
   structu nameidata nd;
   path_lookup(pathname, &nd);
   struct dentry *dentry = lookup_create(nd, 1);

   struct inode *inode = spufs_new_inode(dentry);

   struct spu_context *ctx = alloc_spu_context();
   inode->i_ctx = ctx;

   while ((name = spufs_dir_contents[i++]) != NULL) {
      spufs_new_file(dentry, name, ops, mode);
   }

   spufs_context_open(dget(dentry));
}
```

spufsの1spuをあらわすディレクトリ用inodeの確保と初期化

新しいspu_contextの生成と初期化
まだ物理SPUとはバインドしない

spufsが提供する仮想ファイル群の初期化
(“mem”, “regs”, “mbox”, “mbox_stat”, “signal”, “mfc”, etc…)

Spufs用新システムコール: spe_run

➔カーネル内のspe_createのエントリ: sys_spu_run

- ✓ SPEファイルをLSに読み込んだ後、実際にSPUの実行を開始
- ✓ 処理の実体は arch/powerpc/platforms/cell/spufs/run.c内のspufs_run_spu

➔spufs_run_spuの擬似コード

```
long spufs_run_spu(struct spu_context *ctx, u32 *npc)
{
   down_write(&ctx->state_sema);
   spu_activate(ctx, 0);
   ctx->state = SPU_STATE_RUNNABLE;
   downgrade_write(&ctx->state_sema);

   ctx->ops->npc_write(ctx, *npc);
   ctx->ops->npc_writecntl(ctx, SPU_RUNNABLE);

   do {
      spufs_wait(ctx->stop_wq, spu_stopped(ctx, &code));
      if (code indicates syscall) {
        spu_process_callback(ctx);
      }
   } while (!(code indicates SPU is really stopped));

   spu_yield(ctx);
}
```

ここではじめて物理SPUとバインドする

NPCを書き込んでRUNNABLEに設定 (ここで、context操作関数群opsはハードウェア操作に直結)

SPUがstopped状態になるまでwait (他のカーネルタスクをschedule)

SPUが(PPE側のユーザレベルライブラリコール要求や本当の停止のために)停止したとわかるまでwaitを繰り返す. ブロッキングなため、libspeのようにpthreadなどから呼び出す方が普通

他に走行可能なSPUが見つかったらyield

SPEのコンテキストスイッチ (1)

➔コンテキストスイッチを起こす関数：spu_yield()

- ✓ arch/powerpc/platforms/cell/spufs/sched.c中で定義
- ✓ スケジューリング的にはほとんど何も特別なことはしていない

```
void spu_yield(struct spu_context *ctx)
{
   struct spu *spu;
   int need_yield = 0;
   if (down_write_trylock(&ctx->state_sema)) {
      if ((spu = ctx->spu) != NULL) {
         int best = sched_find_first_bit(spu_prio->bitmap);
         if (best < MAX_PRIO) {
            spu_deactivate(ctx);
            ctx->state = SPU_STATE_SAVED;
            need_yield = 1;
         } else {
           spu->prio = MAX_PRIO;
         }
      }
      up_write(&ctx->state_sema);
   }
   if (unlikely(need_yield))  yield();
}
```

より優先度が高い他に走れる人がいたらyieldする

ここで物理SPUからアンバインド spu_activateと対になる

SPEのコンテキストスイッチ (2)

➔対になる関数spu_activate()とspu_deactivate()によってSPU contextと物理SPUをバインド/アンバインド

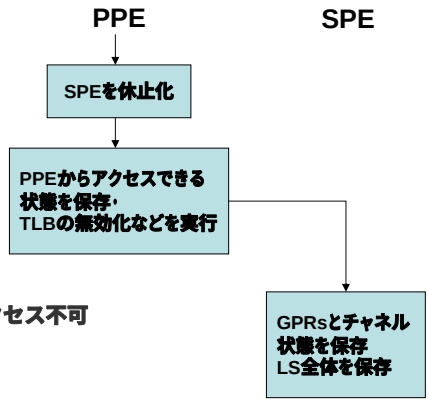
➔spu_activate (spufs/sched.c)
- ✓ spu_get() でSPUを取得
- ✓ bind_context()でSPUとコンテキストをバインド
  - ■ ctx->ops を backing_ops から hw_ops に切り替える
  - ■ spu_restore() でSPU状態とLSの中身をcontext中のCSAからリストア

➔spu_deactivate (spufs/sched.c)
- ✓ unbind_context()でSPUとコンテキストをアンバインド
  - ■ spu_save() でSPU状態とLSの中身をcontext中のCSAにセーブ
  - ■ ctx->ops を hw_ops から backing_ops に切り替える

➔SPUのコンテキストのセーブとリストアはspu_save(), spu_restore()によって行われる

SPEコンテキストのセーブとリストア

➔ SPUのコンテキストとは…
- ✓ 256KBのLS
- ✓ 128本の128-bit汎用レジスタ (GPRs)
- ✓ Special Purpose Registers (SPRs)
- ✓ インタラプトマスク
- ✓ MFCコマンドキュー
- ✓ SPEチャネル状態とデータ
- ✓ MFC Synergisticメモリ管理(SMM)状態
- ✓ その他の特権状態レジスタ



➔ このうち、
- ✓ GPRs, SPRs, チャネル状態などはPPEからはアクセス不可
- ✓ 特権状態レジスタはSPEからはアクセス不可

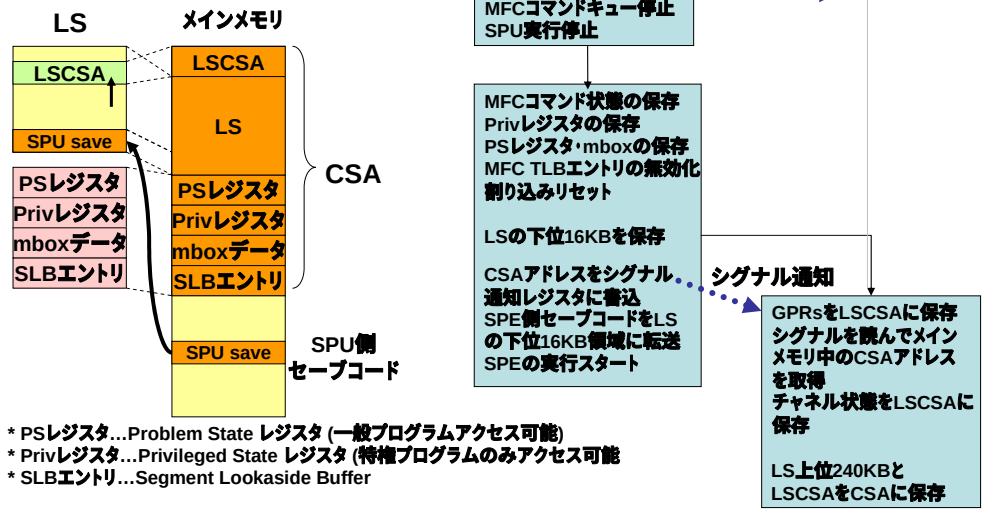
→ PPEとSPE両側でセーブ/リストア処理が必要

➔ コンテキストセーブとリストアに使用する構造体
- ✓ CSA: Context Save Area
  PPE側がメインメモリ上に保持するコンテキスト情報
- ✓ LSCSA: Local Store Context Save Area
  SPU側でしかセーブできない情報をLS上に保持するための構造体
  後にメインメモリに転送されてCSA内にコピーされる

SpufsにおけるSPEコンテキストのセーブ

➔spu_saveを追いかける

- ✓ spu_restoreは基本的にこの逆



* PSレジスタ…Problem State レジスタ (一般プログラムアクセス可能)
* Privレジスタ…Privileged State レジスタ (特権プログラムのみアクセス可能
* SLBエントリ…Segment Lookaside Buffer

SPEのコンテキストスイッチ：まとめ

➔PPE側とSPE側両方の協力が必要

- ✓ SPE側で実行してほしいセーブ・リストアコードをPPEからDMA転送して実行させる
- ✓ コード的にはややトリッキー

➔LSのすべての状態を保存するのはかなりのオーバヘッド

- ✓ PPEの特権ソフトウェアによるPreemptive Context Switchは最も効率の悪いコンテキストスイッチのシーケンスとなる
- ✓ コンテキストスイッチはなるべく起こさないほうが良い
  - ■ 現在のspufsではstopコード発行時と終了時のみ
- ✓ アプリケーションによる自発的なコンテキストスイッチが望ましい
  - ■ Mboxチャネル状態やデータ、割り込み状態など、状態が well known であれば保存しなくて良い可能性の高いデータはたくさんある

Libspeとspufsのまとめと性能上のTips

- Libspe 1.1
  - ✓ シンプルなSPEスレッド抽象化を提供するライブラリ
  - ✓ spufsを利用する参照実装の1つとも言える
  - ✓ SPEからのC99/POSIX関数のサポートなど、使い勝手はかなり良い
  - ✓ とりあえずCellプログラムを実装するならはじめるのには最適
- Spufs
  - ✓ 仮想ファイルシステムとしてSPUを抽象化
  - ✓ 物理SPU数を隠蔽して仮想化
- 性能上のTips
  - ✓ SPEスレッドは決して軽くない機構として使う
  - ✓ デバッグ以外の目的でSPEからのPPEライブラリコール呼出しを多用しない
    - PPE側で処理中、SPEはブロックしているだけ
    - PPE側のpthreadのスケジューリング状況によってブロック時間が左右される
  - ✓ spufsの仮想化には大きく頼らない
    - SPEスレッド数を無闇に増やさない

FIXSTARS

Some More Techniques to Tame Your SPE Programs

SPEプログラムを扱うための
いくつかのTIpsと技術

23 October, 2006

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

ここで話す小トピックス

➔**SPEプログラムを扱う上で少し役に立つ(かもしれない)トピックをいくつか紹介します**

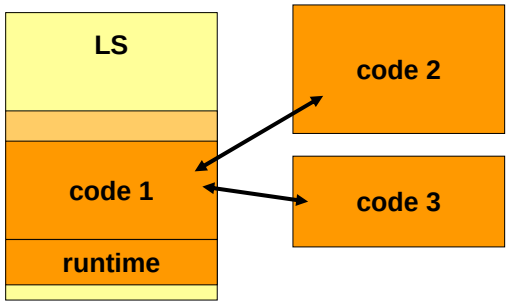
- ✓ **トピック1: オーバレイによるアプリケーションタスクの実現**
- ✓ **トピック2: CESOFによるSPEバイナリの埋め込み**
- ✓ **トピック3: SPEプログラムをいきなり実行**

トピック1:オーバレイ

➔**オーバレイ技術**

- ✓ **プログラムをあらかじめ幾つかに分けておいてメモリ上に必要な部分だけを読み込みなおす**
- ✓ **ページングによる仮想メモリが利用可能ではなく、かつメモリ制限がある場合に大きなプログラムを実行させることができる**

➔**メモリ制限の厳しいLS上では比較的有効なテクニック**



オーバレイによるアプリケーションレベルタスク

➔動作概要

- ✓ SPEに常駐するランタイムとなる実行ファイルをまずロードして実行
- ✓ ランタイムはアプリケーションの指示に従って次に実行する「タスク」のテキストとデータをLS上に読み込む
- ✓ タスクの実行が終わったら、それまでタスクをロードしていたLS領域にまた次に実行する「タスク」を読み込みなおす

➔性能的利点

- ✓ 高価なSPEスレッドは一度だけ生成すれば良い
- ✓ タスクスイッチはPPEに処理を戻さなくてもSPEで自律的に実行可能
- ✓ 別のタスクを読み込み直すコストは本質的にはDMA転送のみ
  - ■ スレッド生成の1000分の1程度のコストですむ
- ✓ コードの入れ替えはアプリケーションレベルで行うため、タスク終了毎に必要最低限な部分だけセーブ・リストアすれば良い

SPE上のオーバレイの実現

➔SPE上で実行させるプログラムはある程度小さなタスクとして分割して実装

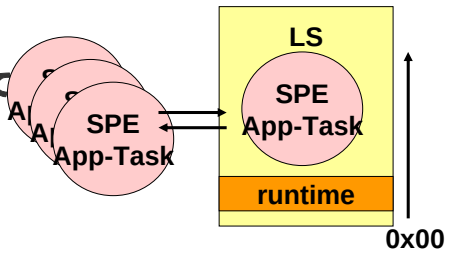
- ✓ これがアプリケーションレベルのタスクとなる
- ✓ PIC (位置独立コード)としてコンパイル

➔オーバレイの親となるruntimeを作る

- ✓ ここが“タスク”を入れ替えてDMA転送する部分となる

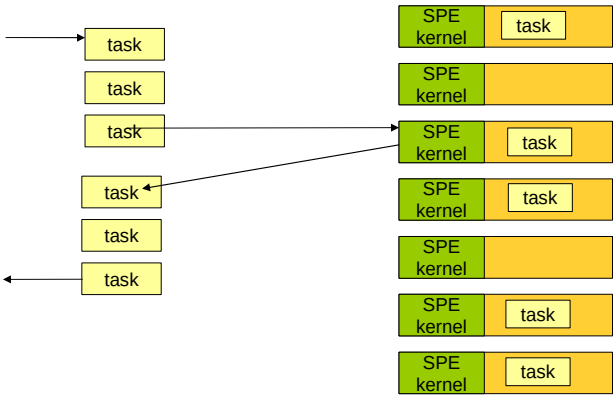
➔オーバレイされるタスクを再配置するコードを書く

- ✓ 簡単にやるならリンカスクリプトとreadelfなどの結果を組み合わせて静的に解決してしまう
- ✓ そうではない場合、メインメモリにロード時に適当に解決してタスクイメージを作っておく



アプリケーションレベルタスクキューの実装例

- SPE上で動作させたい処理を「タスク」として抽象化
  - ✓ タスクは入力データと出力データによって規定されるプログラム片
  - ✓ 入力データと処理コードとともにエンキューされ、SPE上のランタイムカーネルによって空きSPE上で次々と実行される
  - ✓ タスクの入れ替えはオーバレイで行う



トピック２:CESOF

- PPEとSPEが別実行ファイルのために良く起こる事態
  - ✓ PPEアプリケーション実行時にSPE実行ファイルが見つからなくて失敗する
  - ✓ エラー処理をちゃんとやっていなくてSegmentation Fault
  - ✓ プログラムファイルの検索やロードに時間がかかる
- 1つの実行ファイルにできないの？→できます。
  - ✓ CESOF (Cell Embedded SPU Object Format)
  - ✓ SPE実行ファイルを作った後、Cell SDKについてくるembedspuスクリプトを使うことでPPEリンク可能なオブジェクトを生成できます
- embedspuとlibspeで可能になること
  - ✓ SPEオブジェクトをPPE実行ファイルに埋め込み可能
  - ✓ PPE側のグローバルシンボルのアドレスを “_EAR_” というprefixを使うことで参照可能 (EAR: Effective Address Reference)
    - Ex. PPE側プログラム: void *foo;
    - Ex SPE側プログラムからの参照: extern unsigned long long _EAR_foo;

Embedspuのやっていること

➔ **embedspuの処理の中身**

1. SPEバイナリイメージへのポインタを指定されたシンボル名で参照可できるように宣言する .s ファイルを生成
   - 後で定義するセクション“.rodata.speelf”と“.data.spetoe”をさすポインタを持つspe_program_handle_t構造体の変数を定義 → spe_open_imageが不要に
2. SPUバイナリイメージ自体をインクルードする .s ファイルを生成
   - “.rodata.speelf”というセクション名で.incbin を使ってバイナリファイルを直接インクルードする.sファイルを生成
3. SPUバイナリをreadelfにかけて“_EAR_”を持つシンボル名を抜き出し、“.data.spetoe”というセクション中にそのPPEシンボルのアドレスを格納した領域を宣言する .s を生成
   - ➔ SPE中のEAR変数の値は、libspeによってロードされるときにここの領域からコピーされる→SPEからPPEシンボルアドレスを参照
4. 1,2,3をリンクしたオブジェクトを生成

トピック3: SPE実行ファイルを直接実行

➔SPEプログラムを書くときに、いちいちそれをロードするPPEとセットで書かなければならないのが面倒

➔いきなりコマンドラインから実行できない？→ できます。

- ✓ Cell SDKに含まれるelfspeコマンドとelfspe-registerスクリプトによって speバイナリを直接コマンドラインから実行できます
- ✓ elfspe はspe実行ファイルを実行するだけのPPEアプリケーション
- ✓ elfspe-register は以下のようなスクリプトで、SPE ELF形式実行ファイルが実行されたら elfspe を呼び出すフックを登録

```
#!/bin/sh echo
':spe:M::¥x7fELF¥x01¥x02¥x01¥x00¥x00¥x00¥x00¥x00¥x00¥x00¥x00¥x00¥x
00¥x02¥x00¥x17::/usr/bin/elfspe:' ¥ >/proc/sys/fs/binfmt_misc/register
```

- ✓ これをルートになってあらかじめ登録しておけばOK

FIXSTARS

Implementing and Optimizing Cell Programs

いよいよ、ちょっぱやの
Cellプログラムを書いてみる

23 October, 2006

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002

…と思いましたが、今回はパス

➔**たまにセミナーをやっているのでご参加ください**

まとめ

➔**Cellは**

- ✓ **ヘテロジニアス・マルチコアで**
- ✓ **スパコンなみのパワーを持ち**
- ✓ **大規模に各家庭に普及されることが予想され**
- ✓ **オープンなアーキテクチャと開発環境を持つ**

興味深いコンピュータです。

➔**これまで確立されてきた**

アプリケーション、プログラミング方法、実行モデル、バイナリファイルの概念、スケジューリング、OS、VM、リンカやローダの、コンパイラなどなど…

などの考え方を超えて、新しい遊び方を考えてみませんか？

FIXSTARS

Thank you for listening.

あなたのCellの遊び方は見つかりそうでしょうか？
ご清聴ありがとうございました。

23 October, 2006

Fixstars Corporation
Nisshin Building 3F, 1-8-27, kounan, Minato-ku, Tokyo 108-0075, Japan
Tel:+81-3-5781-5001 Fax:+81-3-5781-5002