Vol.2009-ARC-183 No.37
Vol.2009-OS-111 No.37
2009/4/24

Cell/B.E. の SPE 向け軽量カーネルの設計と試作

太田　篤志 †　　並木　美太郎 ‡

Cell/B.E.のプログラミングにおいて，PLAYSTATION 3 上で動作する Linux には SPE を用いたプログラムを容易に開発することのできる環境が既に構築されているが，SPE のコンテキストスイッチのコストが高く, SPE の性能を引き出すことが難しくなる場合がある．本研究では論理的な SPE を提供する従来の仮想化手法ではなく，論理的なプロセスを提供する仮想化手法を提案し，SPE をより効率的に利用できるようにする．この手法を実現するために，本研究では SPE に対して軽量のカーネルを実装し，spufs および libspe を用いずに Cell/B.E.ハードウェアの直接操作による高速な動作を目標とする．試作した軽量カーネルを評価した結果, 空 SPE プロセスの実行速度は最大で MARS よりも 81 %, libspe2 よりも 334 %速いという結果が得られた．

Designing of the Lightweight Kernel for SPEs of Cell/B.E. and Development of Its Prototype

ATSUSHI OHTA †　and　MITARO NAMIKI ‡

This paper describes proposal of the virtualization method to use SPEs more effectively, designing of the lightweight kernel for SPEs to realize it, and development of its prototype. The programming environment, on which the programs using SPEs can be developed easily, in Linux on PLAYSTATION 3 is widely used. But it causes SPE context switches whose cost is so high that it can be hard to optimize SPEs. We propose the virtualization method which supplies logical processes instead of the existing which do logical SPEs. The lightweight kernel which loads and executes a process context is implemented on each SPE, and the Cell/B.E. hardware is directly operated without spufs and libspe to behave faster. The execution speed of empty processes on the prototype is up to 81% faster than on MARS, and up to 334% faster than on libspe2, as a result of the evaluation.

† 東京農工大学大学院工学府
Graduate School of Engineering, Tokyo University of Agriculture and Technology.
‡ 東京農工大学大学院共生科学技術研究院
Institute of Symbiotic Science and Technology, The Graduate School at Tokyo University of Agriculture and Technology.

1. はじめに

近年の CPU はプロセッサ単体の複雑化と高コスト化から性能向上が頭打ちになったことを受け，1 プロセッサに複数のコアを搭載する（ホモジニアス）マルチコア構成が主流となっている．しかしシングルコアのアーキテクチャをベースとしている以上，根本的に問題は解決していない．そこで注目を集めているのが，異なるアーキテクチャのコアを統合した「ヘテロジニアスマルチコア」と呼ばれるプロセッサ構成である．これは従来の単体で高速化された CPU コア 1 個～数個と，構造を単純化した多数の CPU コアを同居させることによって，シングルスレッド，マルチスレッド両方の性能を発揮しつつ，なおかつ同程度の性能のホモジニアスマルチコアと比べて発熱量や消費電力をも削減することができるというものである．本研究の対象となる「Cell Broadband Engine」（本論文では「Cell/B.E.」と表記する）も，汎用的なプロセッサコアである PPE（PowerPC Processor Element）と演算用プロセッサコアである複数の SPE（Synergistic Processor Element）により構成されたヘテロジニアスマルチコア CPU であり，SPE を用いた並列演算性能の高さが特徴である．2.4GHz 駆動の Cell/B.E. を用いたある計算実験では，SPE を 3 基用いた場合に Pentium4　3.6GHz クラス CPU の 2 倍のパフォーマンスを実現するのに対し，SPE を用いない場合は Pentium4 の 20%程度のパフォーマンスにまで落ち込んでしまう[1].

Cell/B.E.の持っている能力を最大限に引き出すための一番のポイントは“いかに SPE を効率的に利用するか”である．これは計算性能だけではなく，ヘテロジニアスマルチコアであるがゆえのプログラミングの難しさについても言えることである．SPE を仮想化したファイルシステムとそれを利用するためのライブラリにより，SPE を用いたプログラムを容易に作ることのできる環境は既に構築されているが，この環境では SPE のコンテキストスイッチが発生する可能性があり，しかもそれは Cell/B.E.の仕様上，高コストなものとなる．本研究では SPE を効率的に利用できるようにするために，従来の仮想化手法とは違い，SPE に実行させるプロセスを仮想化し，プロセスを順次 SPE に割り当てていくアプローチを取る．

2. 本研究の概要

2.1 既存のプログラミングモデル

Cell/B.E.のプラットフォームの一つである家庭用ゲーム機「PLAYSTATION 3」上で動作する Linux には，SPE を仮想化して管理するファイルシステム「spufs」[2]とそれを利用するためのライブラリ「libspe」[3]が用意されており，SPE を用いるアプリケーションを容易に開発

© 2009 Information Processing Society of Japan

Vol.2009-ARC-183 No.37
Vol.2009-OS-111 No.37
2009/4/24

することのできる環境が既に確立されている．本節では libspe の最新バージョンである「libspe2」を用いた時の簡単な流れを説明する．SPE を利用したいアプリケーションは，まず必要な数の SPE コンテキストを `spe_context_create` 関数により生成する．その後 `spe_image_open` 関数および `spe_program_load` 関数を用いて SPE 用のプログラムをコンテキストにロードし，`spe_context_run` 関数で SPE コンテキストを実行する．実行が完了した SPE コンテキストは `spe_context_destroy` 関数を用いて破棄される．libspe が生成する SPE コンテキストは論理的なものであり，実際に Cell/B.E.に搭載されている以上の数生成することができる．搭載数以上の SPE コンテキストが生成され，かつそれらが実行状態にあると，spufs は SPE のコンテキストスイッチを行う．Cell/B.E.は PPE と SPE でメモリ空間を共有しない分散メモリ型の構造を取っているため，このとき SPE 内の全てのデータをメインメモリに転送しなければならない．結果，コンテキストスイッチには時間がかかるため，可能な限り避けるべきであるとされている[4].

SPE を用いて高速な演算処理を実現するための研究は多岐に渡り行われているが，その多くは“SPE をフルに使って並列に処理することによりアプリケーションを高速化すること”だけに焦点が当てられており，そのようにして高速化されたアプリケーションを複数使いたい場合についての言及は少ないのが現状である．確かに単一のアプリケーションだけを考えれば，SPE の搭載数以上のコンテキストを生成しないように注意を払うこともできる．しかし実際には SPE を利用するアプリケーションが複数同時に実行されうることもあり，コンテキストスイッチが発生する可能性は依然として残る．SPE を利用するアプリケーションが複数実行されていたとしても，SPE のコンテキストスイッチはできる限り抑え，コンテキストスイッチに費やしていた時間を SPE の実行に充て，SPE を利用するアプリケーション全体の性能を向上することができるような手法が求められる．

2.2 本研究の方針

2.1 節で述べた内容を踏まえた上で，本研究では論理的な SPE を生成して実 SPE の実行時間を取り合うのではなく，図 1 のように，SPE に実行させたい処理を 1 ヶ所に集約して管理して，PPE 上で動作するアプリケーションに対して処理を登録するための窓口を用意し，SPE は実行待ちの処理を順次実行していくモデルを考える．PPE アプリケーションは SPE コンテキストを管理する必要がなくなり，SPE に実行させたい処理を集約・管理しているアプリケーションのみが SPE を利用する形となるため，搭載数以上のコンテキストが作られることはなくなる．よってコストの高い SPE のコンテキストスイッチを行わずに済み，結果として

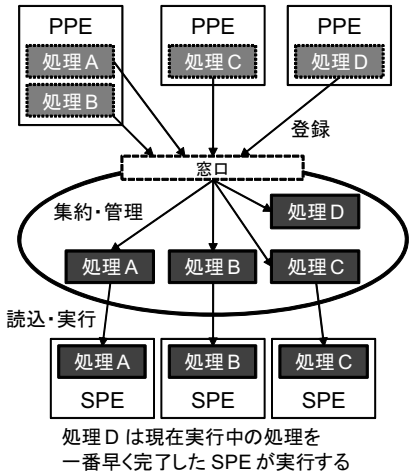

図 1　SPE に実行させたい処理を集約・管理するモデル

パフォーマンスが向上するのではないかと考えた．同時に PPE アプリケーションからは処理を登録した時点で SPE に割り当てられているように見えるため，spufs および libspe を用いたときと同じように仮想化が実現できるとも考えた．そのほか，本研究の方針を以下のように定めることとする．

- SPE に実行させたい処理「SPE プロセス」は SPE 用のプログラムを実行単位とする
  本節冒頭で述べた SPE に実行させたい処理を本論文では以降「SPE プロセス」と呼称する．2.1 節で述べたように PPE と SPE がメモリ空間を共有しないほか，SPE プロセスを登録した PPE アプリケーションとは独立したコンテキストを有しているため，「プロセス」という括りとした．また SPE プロセスは SPE 用のプログラム一つを実行する．
- SPE プロセスを登録，操作するためのインターフェース「SPE プロセスマネージャ」を用意する
  SPE プロセスを登録したり，あるいはプロセスの実行を指示したり実行状態や戻り値を取得したりといった操作を行うためのインターフェースとして「SPE プロセスマネージャ」を用意する．これは Linux のキャラクタデバイスとして実現し，`open`，`close`，`read`，`write` のようなシステムコールで容易にアクセスすることができるようにする．

© 2009 Information Processing Society of Japan

Vol.2009-ARC-183 No.37
Vol.2009-OS-111 No.37
2009/4/24

- SPE に軽量のカーネルを実装する
  SPE プロセスが登録された際に，プロセスの割り付けや初期化などによって PPE にかかる負荷を分散するため，SPE に軽量のカーネルを実装する．次に実行するべきプロセスの指示や SPE セグメント・ルックアサイド・バッファ（SLB）の設定など，PPE 側でしか行えない処理以外は軽量カーネルが担当する．つまり本研究では SPE プロセスマネージャ（を実現するデバイスドライバ）と SPE 向けの軽量カーネルを開発することになる．
- Cell/B.E.のハードウェアを直接操作する
  SPE プロセスマネージャと SPE 向けの軽量カーネルは spufs および libspe を用いた従来のプログラミングモデルの上で開発をすることも可能であるが，プロセスマネージャと軽量カーネルの二つによって SPE を仮想化することができるのであれば，spufs と libspe による論理的な SPE によって二重に仮想化されることになり無駄となる．そこで本研究では spufs および libspe を用いず，Cell/B.E.のハードウェアを直接操作することによってより高速に動作させることを目標にする．本研究の実行環境である PLAYSTATION 3 は専用のハイパーバイザによって資源が管理されており，Linux はその上層でゲスト OS として動作している．そのため“Cell/B.E.のハードウェアを直接操作する”ということは，ハイパーバイザコールを用いて Cell/B.E.の資源を制御することを意味している．

以上の方針により，本研究で開発する SPE プロセスマネージャおよび SPE 向け軽量カーネルの全体構成を図 2 に示す．この二つについて，詳細は次章以降で述べる．

3. SPE プロセスの管理

本研究で定義する SPE プロセスは，spufs および libspe を用いた際の SPE 用プログラムにそれ程大きな変更を加えることなく動作するものとしたい．これは spufs および libspe を用いた場合において，SPE 用プログラム自体には spufs や libspe に依存する部分がなく，本研究の手法にそのままプログラムを流用できる可能性が高いためである．

3.1 SPE プロセスの機能

SPE の持ち味は複数の SPE を用いた並列処理と SIMD 命令などを駆使して計算処理を高速に実行できるところにあり，PPE アプリケーションもその要求があるからこそ SPE を利用してくるはずである．またマイクロカーネルのように OS の機能がプロセスとして細かに分けられている場合に，その各々を SPE が実行するといったアイデアもある[5]．こういった状況

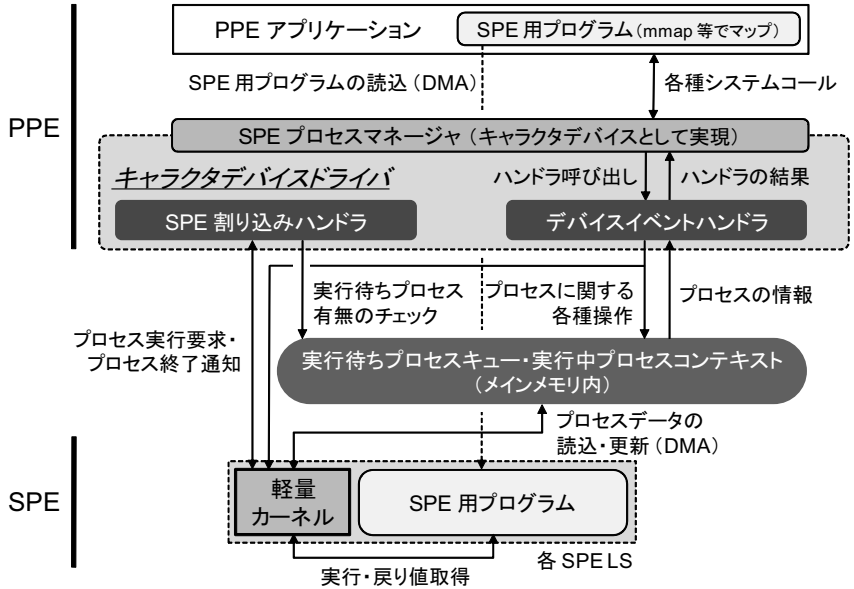

図 2　本研究の全体構成

下で SPE プロセスに必要とされる機能には以下が挙げられる．

- データのブロック転送
  計算処理においては，ある程度ブロック化されたデータを一度に読み込んで実行した方が効率がよく，また SPE の SIMD 命令の恩恵も受けやすい．SPE とメインメモリとの間のデータ転送手段である DMA は非同期命令のため，SPE の命令実行と並行して行うことができるが，この特徴を利用した SPE のプログラミングテクニックである「ダブルバッファリング」や「トリプルバッファリング」は，転送時間を隠蔽するために，ブロック化されたデータ転送が不可欠である．
- プロセス間通信
  SPE を利用する PPE アプリケーションと SPE プロセス，あるいは SPE プロセス間で通信を行えるようにし，プロセス間で連携を取ったり，情報を共有できるようにしたい．

© 2009 Information Processing Society of Japan

- 処理の移譲
  並列処理，計算処理を行うプログラム，マイクロカーネルの 1 機能の実行のどちらにおいても，別の処理の終了を待って同期を取る機会は多い．そこで終了待ちに時間がかかるような場合のために，SPE プロセスが処理を移譲できるようにして，移譲された際には“意図的に”コンテキストスイッチを発生させて他の実行待ち SPE プロセスの実行時間に充てる機能を提供した方がよい．

現時点での SPE 向け軽量カーネルは SPE プロセスの読み込みと実行のみを行う試作段階であり，SPE プロセスについて軽量カーネル側から提供している機能はない．しかし各 SPE に搭載されている MFC（Memory Flow Controller）が提供している機能である DMA 転送（3.2 節），メールボックス機構およびシグナル通知機構（3.3 節）と，SPU の命令の一つであるストップ・アンド・シグナル命令（3.4 節）を用いることで，以上に挙げた機能を実現することができる．ここではこの 3 つについて述べる．なお処理の移譲については 8 章にて今後の課題としている．

3.2 DMA 転送

SPE からメインメモリへのアクセスは直接には不可能であり，代わりに DMA を用いて SPE の作業領域である LS（Local Store）とのデータの読み書きを行う．DMA 転送は MFC が SPE SLB を参照し，プロセスごとに独立したアドレスである実効アドレスを一意のアドレスである仮想アドレスへ変換してその位置にあるデータへアクセスすることで実現される．つまり DMA 転送を行うためには，特権を持つソフトウェアによって SPE SLB を管理する必要がある．そこで本研究では SPE プロセスマネージャがその役割を担い，SPE プロセスを登録して実行する際に，あらかじめ DMA でアクセスされうるデータ領域などを SPE プロセスマネージャへアクセスヒントとして与えておく．このアクセスヒントにしたがい，SPE プロセスマネージャ側で SPE SLB に適切なエントリが設定され，SPE プロセスと登録元の PPE アプリケーションとの間でデータのやり取りが可能となる．アクセスヒントに関する詳細は 4.2 節で SPE プロセスマネージャを実現しているデバイスの `write` システムコールについて述べる際に併記する．

3.3 メールボックス機構およびシグナル通知機構

メールボックス機構とは PPE と SPE との間で通信を行うために MFC から提供されている機能である．PPE から SPE にデータを渡すための「インバウンド・メールボックス」と，SPE から PPE にデータを渡すための「アウトバウンド・メールボックス」があり，前者は 4 個 1 組，後者は 1 個 2 組の 32 ビットデータを保存する FIFO キュー構造となっている．またシグナル通知機構も通信を行うための機能であるが，これは制御メッセージやイベントをシグナル通知するために使われる．シグナル通知機構は 32 ビットのデータを保存するレジスタを 2 組有する．

SPE 向け軽量カーネルではメールボックス機構およびシグナル通知機構を一切使用していない．よって SPE プロセスにおいてこれらの機構を利用することは可能だが，現時点では SPE プロセスマネージャ側でメールボックス，シグナルを制御する仕組みをまだ実装していないため，PPE アプリケーション側で捕捉することはできない．

3.4 ストップ・アンド・シグナル（stop）命令

ストップ・アンド・シグナル（`stop`）命令は SPE の実行を停止し，PPE へ割り込みを通知するための命令である．現在は SPE プロセスの終了を SPE プロセスマネージャへ通知するために SPE 向け軽量カーネルが利用している．また spufs および libspe において，ストップ・アンド・シグナル命令は SPE から Linux のシステムコールを呼び出すために使われているが，現時点では同様の機能を SPE プロセスから利用することができない．今後対応していかなければならない課題である．

4. SPE プロセスマネージャ

2.2 節で述べたように，SPE プロセスマネージャは Linux のキャラクタデバイスとして実現されており，SPE プロセスの登録や操作がデバイスに対して発行するシステムコールと対応している．本章では PPE アプリケーションから利用する，SPE プロセスマネージャのシステムコールの仕様について述べる．

4.1 open システムコール

SPE プロセスを登録するためには，まずこのデバイスをオープンする．オープンするとファイルディスクリプタが返されるが，実行したい SPE プロセスとファイルディスクリプタは 1 対 1 で対応していなければならない．また一度プロセスが終了すると，このファイルディスクリプタを指定してプロセスを再度実行するといったことはできない．

4.2 write システムコール

オープンしたデバイスに対し，SPE プロセスの登録を行う．一般的に考えると，このシス

© 2009 Information Processing Society of Japan

Vol.2009-ARC-183 No.37
Vol.2009-OS-111 No.37
2009/4/24

表 1　spe_process_context_write_data_t 型

フィールド	型	説　　明
pid	int32_t	SPE プロセスを登録するアプリケーションのプロセス ID
program_start	uint64_t	SPE 用プログラムの先頭を指す実効アドレス
program_size	uint32_t	SPE 用プログラムのサイズ
arg	uint64_t	SPE 用プログラムに渡す引数
data_start	uint64_t	DMA 転送されうるデータの先頭を指す実効アドレス
data_size	uint32_t	DMA 転送されうるデータのサイズ

テムコールでは SPE 用のプログラムを書き込むのが妥当といえよう．しかし 3.2 節で述べたように，SPE プロセスの登録の際には DMA 転送のアクセスヒントも併せて登録する必要がある．よって本研究では表 1 に示す spe_process_context_write_data_t 型のデータを書き込むこととする．なお 5.1 節で改めて述べるが，ここで書き込んだデータは SPE プロセスコンテキストの一部となる．もし SPE 用のプログラムやデータがファイルになっている場合は，そのファイルシステムの mmap システムコールによってメモリ空間にマップし，そのアドレスを渡す．こうすることで，SPE 用のプログラムの情報とアクセスされるデータの情報を一続きに書き込むことができる．また SPE 用のプログラムはこの時点で，そのまま実行することのできる RAW バイナリ形式でなければならない．これは SPE 向け軽量カーネルがここで指定されたアドレスから DMA 転送を用いてプログラムを読み込むからである．プロセスを登録するアプリケーションのプロセス ID は，SPE プロセスマネージャが当該プロセスにおける実効アドレスから仮想アドレスを求めるために必要なデータである．

4.3　ioctl システムコール

登録を行った SPE プロセスの実行，中断，再開，強制終了といった操作は ioctl システムコールで行う．現在の試作段階では SPE プロセスの開始のみサポートしている．

4.4　read システムコール

SPE プロセス実行後は，read システムコールによって表 2 に示す spe_process_context_read_data_t 型のデータを取得することができる．4.2 節で述べた spe_process_context_write_data_t 型と同様に，ここで読み込むデータも SPE プロセスコンテキストの一部となる．このシステムコールはブロックしないようになっているため，SPE プロセスがまだ実行中の場合にはループ等で定期的に監視する必要がある．

表 2　spe_process_context_read_data_t 型

フィールド	型	説　　明
status	int32_t	SPE プロセス実行状況
ret	uint64_t	SPE プロセスの戻り値
spe_no	int32_t	プロセスの実行を担当した SPE

4.5　close システムコール

SPE プロセスの実行が完了しプロセスコンテキストが不要になったら，デバイスをクローズして SPE プロセスマネージャがプロセスを破棄できるようにする．

4.6　プログラム例

以上までに示した各システムコールを用いた簡単なプログラム例を図 3 に示す．この例では下の SPE 用プログラムの RAW バイナリが上の PPE アプリケーションの変数 spe_program に格納されている．また SPE プロセスで使用するデータは配列として用意されているが，外部のファイルを利用する場合には 4.2 節で述べたように mmap システムコールによってアプリケーションのプロセス空間にあらかじめマップしておく必要がある．

5. 実　　装

5.1　SPE プロセスコンテキスト

本研究における SPE プロセスコンテキストは，write システムコールで書き込む spe_process_context_write_data_t 型のデータおよび read システムコールで読み出す spe_process_context_read_data_t 型のデータの 2 つである．open システムコールが発行されると，SPE プロセスマネージャは SPE プロセスコンテキスト用の領域を確保する．この領域はその後 close システムコールが呼ばれた際に解放される[*1]．SPE プロセスコンテキストは ioctl イベント時に SPE の実行待ちプロセスキューに追加され，空いた SPE が実行を担当する際にキューから削除される．また各 SPE ごとに現在実行中のプロセスコンテキストを保持している．

*1 正確にはデバイスドライバの release イベントが発生したとき．dup システムコール等によりファイルディスクリプタが複製されている場合，一方を close しただけではデバイスデータ用の領域は解放されない．

© 2009 Information Processing Society of Japan

Vol.2009-ARC-183 No.37
Vol.2009-OS-111 No.37
2009/4/24

```
int main(int argc, char *argv[])
{
    // spe_program には下の SPE 用プログラムの raw バイナリが入る
    char spe_program[]  = { ... };
    int  data[128*1024] = { ... };
    spe_process_write_data_t write;
    spe_process_read_data_t  read;
    int fd, pid;

    fd  = open("/dev/...", O_RDWR);
    pid = getpid();

    write.pid = pid;
    write.program_start = (uint64_t)spe_program;
    write.program_size  = sizeof(spe_program);
    write.arg = 100;
    write.data_start = (uint64_t)data;
    write.data_size  = sizeof(data);

    write(fd, &write, sizeof(write));
    ioctl(fd, SPE_EXECUTER_IOCTL_START_PROCESS);
    do
    {
        read(fd, &read, sizeof(read));
    } while (read.status != SPE_PROCESS_STATUS_FINISHED);

    return 0;
}
```

```
uint64_t main(uint64_t arg)
{
    int data[128*1024] = { 0 };
    dma_get(data, arg, 128*1024);
    // Something...
    dma_put(data, arg, 128*1024);
    return 0;
}
```

図 3　プログラム例（上が PPE 用，下が SPE 用のプログラム）

5.2　割り込みハンドラと ioctl イベント

SPE 向け軽量カーネルが SPE プロセスの実行を完了すると，3.4 節で述べたように PPE に対して割り込みを送るようになっている（詳細については 5.3 節で述べる）．このとき SPE 実行待ちプロセスキューおよび実行中プロセスコンテキストを更新する必要が生じる．また SPE プロセスマネージャの `ioctl` イベントが発生した際にも，同様に実行待ちプロセスキューと実行中プロセスコンテキストを更新しなければならない．そこでスピンロックを用いてクリティカルセクション内で更新処理を行うようにしている．その後，割り込みハンドラの場合は実行待ちプロセスがあればそれを，`ioctl` イベントの場合は SPE プロセスを実行していない空きの SPE があれば当該プロセスを実行する処理 `spe_process_start` を行う．

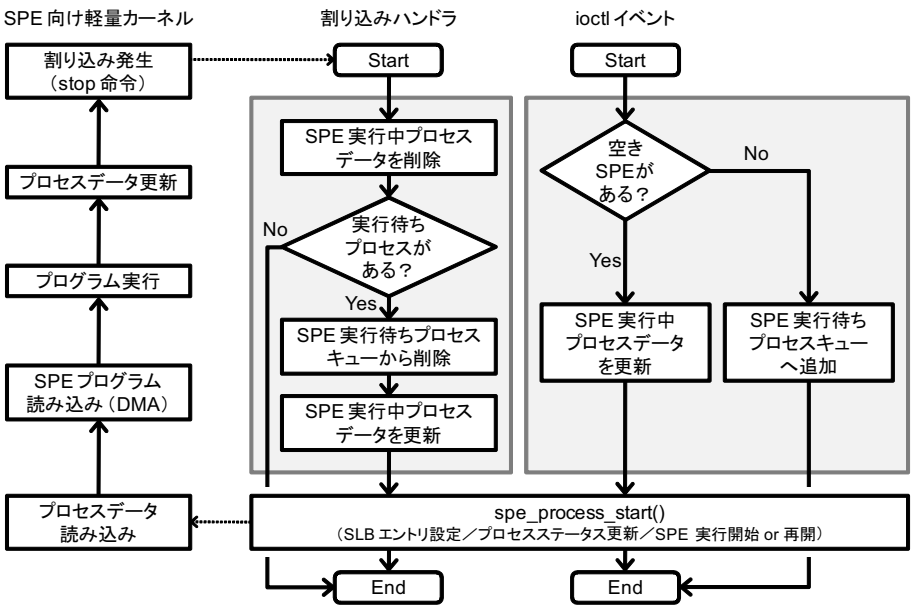

図 4　割り込みハンドラ，`ioctl` イベントおよび SPE 向け軽量カーネルの全体の処理の流れ
（網掛け部分はクリティカルセクションを示す）

5.3　SPE 向け軽量カーネル

SPE 向け軽量カーネルと，5.2 節で述べた割り込みハンドラおよび `ioctl` イベントの全体の処理の流れを図 4 に示す．SPE の実行が開始または再開されると，SPE 向け軽量カーネルは次に実行するべき SPE プロセスのプロセスデータを DMA により読み込む．ここで明らか

© 2009 Information Processing Society of Japan

Vol.2009-ARC-183 No.37
Vol.2009-OS-111 No.37
2009/4/24

表 3　開発・実行環境

プラットフォーム	PLAYSTATION 3（Cell/B.E. 3.2GHz, 7SPEs）
OS	Yellow Dog Linux 6.1 (20090201)
Linux カーネルバージョン	2.6.28
gcc/spu-gccバージョン	4.1.2 (gcc) / 4.1.1 (spu-gcc)

となる SPE 用プログラムが配置されている実効アドレスとサイズをもとに，同じく DMA によってプログラムを読み込み，プログラムに処理を渡す．プログラムが完了すると再び軽量カーネル側に処理が返るので，ここで SPE プロセスの実行状態を“完了状態”にし，SPE 用プログラムの戻り値とともにプロセスデータを DMA で書き出して更新し，SPE プロセスマネージャに SPE プロセスの終了を伝えるため `stop` 命令による割り込みを発生させる．この時点で SPE は動作を停止し[6]，割り込みハンドラが開始される．以降の動作は 5.2 節で述べた通りである．

6. 評 価 ・ 考 察

6.1　開発・実行環境

SPE プロセスマネージャおよび SPE 軽量カーネルは，表 3 に示す環境で開発および実行を行った．なお Yellow Dog Linuxのカーネルイメージには標準で spufs が含まれているため，カーネルの再コンパイルを行って spufs の含まれていないカスタムカーネルイメージを作成し，これを用いている．

6.2　空タスクの実行時間の計測と考察

ここでは libspe2 を用いた従来の環境，7.1 節で後述する MARS[7]を用いた環境，および本研究の環境それぞれにおいて，SPE に何もしない空のタスク（本研究においては空の SPE プロセス）を実行させたときの実行時間を比較する．PLAYSTATION 3 で使用できる 6 個の SPE 全てを使い，空のタスクを 10，50，100，500，1,000 個を作り，実行開始から全てのタスクの実行が終了するまでの時間を PPE アプリケーション側で計測する．計測の結果を表 4 および図 5 に示す．libspe2 については 1,000 プロセス実行時にはセグメンテーション違反が発生し，計測を行うことはできなかった．

空タスクの実行時間を比較すると，本研究での実行時間は MARS の環境下で実行するよりも最大で 81 %速く，libspe2 に至っては最大で 334 %速いという結果を得ることができた．

表 4　空タスクの実行時間（単位：ミリ秒）

タスク数	libspe2	MARS	本研究
10	15.492	5.031	4.240
50	66.914	26.783	17.547
100	141.764	59.183	32.599
500	692.639	249.081	160.219
1,000	(SEGV)	557.675	333.434

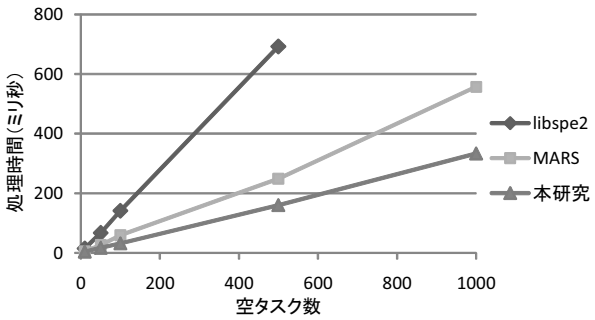

図 5　空タスクの実行時間

libspe2，および libspe2 を使用している MARS と比較してこれだけ速い結果となったのは，Cell/B.E.のハードウェアを直接制御したことによる成果とみてよいであろう．本研究の SPE 向け軽量カーネルはまだ試作の段階にあり，機能拡張をしていく中で今後実行時間は延びることが予想される．機能の拡張だけでなく，現行機能のチューニングにも今後目を光らせていかなければならない．

7. 関 連 研 究

本章では SPE に実行させたい処理（本章でのみ便宜上「タスク」と表記する）を 1 箇所で管理する機構を有する既存の研究について述べる．

7.1　MARS（Multicore Application Runtime System）

MARS は 1 個以上の MPU と，それを管理，制御するホストプロセッサで構成されるマル

© 2009 Information Processing Society of Japan

チコアアーキテクチャ上で動作するプログラムの開発を容易にするためのライブラリであるが，その特徴から明らかなように Cell/B.E.を強く意識したものとなっている．MARS は MPU に軽量のカーネルをロードし，実行待ちタスクのチェックやその読み込みと実行，さらにタスクが同期をとる際の他のタスクとのコンテキストスイッチ処理などをこのカーネルが担当している．MPU を Cell/B.E.の SPE と置き換えれば，本研究でも SPE 向け軽量カーネルを実装している点では非常に類似したものといえる．MARS は MPU を仮想化するというよりはマルチコアプログラミングを簡単にすることに焦点が当てられており，また MARS を用いている複数のアプリケーション間でタスクのキューは共有しない．本研究では単一の実行待ちプロセスキューを有し，空いた SPE が実行待ちプロセスを順次実行することにより SPE のパフォーマンスの改善を目指している．

7.2 Cell 用の Fine-grain Task Manager [8]

この研究ではメニーコアアーキテクチャ向けの Fine-grain タスク OS を設計し，それをゲームプログラム用 OS「Cerium」へ実装している．Cerium のタスクである SceneGraph，PolygonPack，SpanPack の 3 つに対して並列実行を行い，アムダールの法則を考慮した細粒度タスクを管理し，またタスク同士の依存関係を柔軟に指定でき，パイプライン化して実行するなどの工夫が凝らされている．本研究では Cerium が扱うフレームバッファ上での描画処理のようにデータ並列性の高いタスクだけでなく，計算処理も含め，タスクの特性を選ばない汎用的な構成となっている．

8. おわりに

本研究では論理的な SPE を提供する従来の仮想化手法とは異なる新しい仮想化手法を提案し，これを実現するための SPE 向け軽量カーネルの設計および試作を行った．SPE プロセスマネージャと SPE プロセスを実行するための軽量カーネルを連携させ，SPE の搭載数以上にプロセスを登録できるようにすることで仮想化を実現しただけでなく，Cell/B.E.のハードウェアを直接操作するようにしたことで，空プロセスの実行速度は MARS に比べて 81 %，libspe に比べて 334 %向上した．

今後の課題は，まず SPE 向け軽量カーネルの機能拡張がある．今回は試作ということでプログラムを読み込んで実行するだけの簡素なものであったが，yield によって明示的な制御譲渡を行いノンプリエンプティブ SPE マルチタスクを実現したり，DMA 転送の際に空いたタグ番号を自動的に割り振るような機能を SPE プロセスに対して提供できないか思案している段階にある．また SPE プロセスマネージャ側の機能拡張も必要であり，SPE プロセスに対する操作はプロセスの実行を指示するのみであったが，中断，再開，強制終了といった操作は後々必要となるであろう．中断をサポートするのであれば，他の実行待ちプロセスと入れ替えるコンテキストスイッチがなければ空いた SPE を生かすことができない．また SPE から Linux のシステムコールを発行するような場合にも対応できるようにしたい．SPE プロセスマネージャ，SPE 向け軽量カーネル以外の部分では，SPE プロセス全体の実行状況を可視化できるようにしたいと考えている．例えば procfs ファイルシステムを介して実行中プロセス，実行待ちプロセスの状況を逐一確認できるようにすることなどである．

参考文献

[1] 西川 善司．西川善司の *3D* ゲームファンのための *PS3* パフォーマンス考察講座～ *PS3* は *SPE* で加速する！ *SPE* がなければ性能は *Pentium 4* の *1/5* 以下？. GAME Watch, 2006.

[2] Arnd Bergmann. *Spufs: The Cell Synergistic Processing Unit as a virtual file system*. IBM DeveloperWorks, 2005.

[3] International Business Machines Corporation. *SPE Runtime Management Library Version 2.2*. Cell Broadband Engine Architecture Joint Software Reference Environment Series, 2007.

[4] International Business Machines Corporation, Sony Computer Entertainment Incorporated and Toshiba Corporation. *Cell Broadband Engine Architecture Version 1.01*. 2006.

[5] 野尻 祐亮，並木 美太郎．マイクロカーネル方式による *Cell/B.E.*向け *OS* 構成法の提案と *MINIX 3* による実現. 情報処理学会研究報告 2009-OS-110, Vol.2009, No.6, pp.91-98, 2009.

[6] International Business Machines Corporation, Sony Computer Entertainment Incorporated and Toshiba Corporation. *Synergistic Processor Unit Instruction Set Architecture Version 1.2*. 2007.

[7] Sony Corporation of America. *MARS - Multicore Application Runtime System*. http://ftp.uk.linux.org/pub/linux/Sony-PS3/mars/latest/mars-docs-1.1.1/html/, 2008.

[8] 宮國 渡, 河野 真治, 神里 晃, 杉山 千秋. *Cell* 用の *Fine-grain Task Manager* の実装. 情報処理学会研究報告 2008-OS-108, Vol.2008, No.35, pp.91-98, 2008.

© 2009 Information Processing Society of Japan