Vol.2009-ARC-183 No.17
Vol.2009-OS-111 No.17
2009/4/23

ホモジニアスマルチコアCPUにおける異種OS間の連携機構の試作

磯部泰徳[†1]　佐藤未来子[†1]　並木美太郎[†1]

近年，マルチコアプロセッサは幅広く普及し，様々な機器に搭載されている．マルチコアプロセッサの性能を引き出すためには、専用 OS による適切なリソース管理やスケジューリング制御などを行い、コアを効率よく利用することが有効であるが，専用 OS 単体では、汎用 OS におけるファイルシステムや I/O などの機能は期待することが難しい。本研究では，汎用 OS と並列演算向け専用 OS を連携動作させる機構を試作した．これにより，専用 OS の制御や，専用 OS 用プログラムにおける I/O 関連の処理を汎用 OS 側で行うことが可能となり，専用 OS を肥大化・複雑化させずに，汎用 OS の機能を専用 OS 側から利用できる．評価では，OS 間通信のラウンドトリップ時間を計測し，90μsec という結果を得て，OS 間の連携での性能に問題のないことを確認した。

Prototyping of Multi Operating System Environment for Multi-core Processor

Hironori Isobe,[†1] Mikiko Sato[†1] and Mitaro Namiki[†1]

This paper described a prototyping of multi operating system environment for multi-core processor. The environment executes two different OSs (general-purpose OS and application oriented OS that is able to process multi-thread program) at the same time on multi-core processor. The environment enables general-purpose OS to collaborate with application oriented OS by inter-OS communication. As the result of evaluation, it turned out that inter-OS communication's round-trip time is 90 μsec, and efficient for application oriented computer system.

†1 東京農工大学
Tokyo University of Agriculture and Technology

1. はじめに

近年，マルチコアプロセッサはゲームや携帯電話を始め、様々な機器に搭載されている[1)]．マルチコアプロセッサの性能を引き出すためには，専用 OS による適切なリソース管理やスケジューリング制御などを行い，コアを効率よく利用することが有効である．しかし，そのような専用 OS だけでは従来の汎用システムのファイルシステムや API などのような利便性を望むことは難しい．

従来の汎用 OS の利便性を維持したまま，マルチコアプロセッサを効率よく利用できるシステムとするために，それぞれの処理内容に特化した複数のシステムソフトウェアを連携させる手法が有用である．本研究では，共有メモリ型ホモジニアスマルチコアプロセッサ（以下，マルチコアプロセッサ）を対象として，その性能を活かすためのプログラム実行基盤の実現を目指している．

マルチコアプロセッサ上で複数の OS を同時に動作させる機構について研究がおこなわれていた。[2)–6)] それらの研究では，ハードウェアなどの資源を各 OS に分割占有させて同時に動作させる方式を採用し，OS 間の通信はデータのやり取りのみであった．これらは 2 つの OS が 1 つのシステムとして連携して動作するものではなく，各 OS の環境を別々にしか利用できないという問題点があった．関連研究の詳細は第 8 節において述べる．

本研究では，汎用 OS と並列演算向け専用 OS を連携動作させる機構を試作し，特徴の異なる OS を 1 つのシステムとして利用可能とした．これにより，専用 OS の制御や，専用 OS 用プログラムにおける I/O 関連の処理を汎用 OS 側で行うことが可能となり，専用 OS を肥大化・複雑化させずに，汎用 OS の機能を専用 OS 側から利用できる．

2. 目　　標

本研究では，汎用 OS とマルチスレッドプログラムを軽量に管理・制御する並列演算向け専用 OS（以下 専用 OS）を連携させて動作させるヘテロジニアスハイブリッド OS 構成にすることによって，汎用 OS と専用 OS を適材適所に利用し，汎用 OS の利便性と専用 OS の演算性能を両立させることを目標とする．

汎用 OS では，既存のソフトウェア資源の利用環境や，ファイルシステム，I/O，デバイスの管理などに加え、専用 OS の管理制御の機能を提供する．専用 OS は，複数のプロセッサコアを管理し，効率よく利用することによって，汎用 OS に対して軽量・高速な並列演算実行環境を提供する．また，専用 OS から汎用 OS の I/O などの資源を利用することを可

Vol.2009-ARC-183 No.17
Vol.2009-OS-111 No.17
2009/4/23

表 1 Linux と Future の主な役割の分担
Table 1 Function of Linux and Future

Linux	Future/MULiTh
I/O の処理	スレッドプログラムの実行
Future の制御	複数の CPU コアの管理

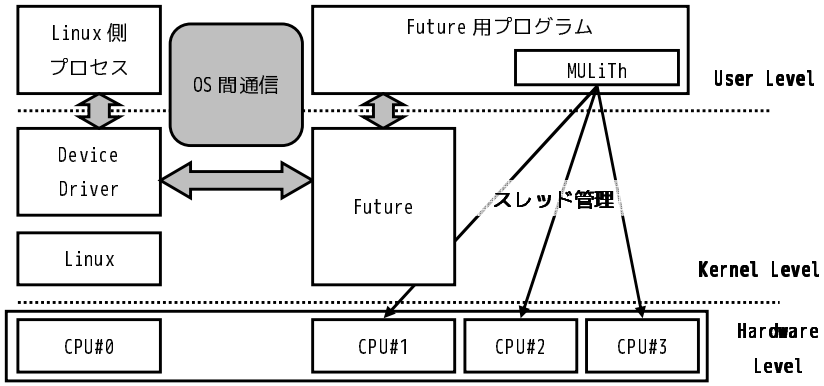

図 1 システムの概要
Fig. 1 The system structure

能にする機能を提供する．

3. システム構成

本システムでは，マルチコアプロセッサ上の異なるコア上で，汎用 OS と専用 OS を同時に動作させ互いに通信を行うことによって，一つのシステムとして動作する．汎用 OS である Linux がファイルシステムや I/O，ネットワーク，専用 OS の制御などの機能を提供し，専用 OS である Future[7] はマルチスレッドライブラリ MULiTh(Userlevel Thread Library for Multithreaded architecture) と連携して高速なマルチスレッドプログラムの実行環境を提供する．二つの OS は共有メモリを介した OS 間通信で互いに通信を行い連携する．これによって，専用 OS を肥大化・複雑化させることなく，汎用 OS 上で専用 OS 用プログラムを管理できるようにし，汎用 OS の機能を専用 OS 用プログラム中から利用することが可能となる．表 1 に各 OS の主な役割の分担を，図 1 にシステムの動作概要を示す．

3.1 Linux

Future のブートや Future 用プログラムのロードなどといった Future の制御を行う．また，Future 側で発生した I/O 処理も Linux が代行する．Future との通信はデバイスドライバを介して行う．

3.2 Future

MULiTh のスレッド制御をサポートするカーネルであり，OS 間連携機構を介して Linux に対しマルチスレッドプログラムの実行環境を提供する．Future においての「プロセス」とは，複数のコア，アドレス空間，I/O を割り当てた実行単位であり，Future はプロセスの管理や，プロセスへの実コアの割り当て，例外・割込み，OS 間通信の送受信等の特権処理を行う．

3.3 MULiTh

MULiTh は POSIX スレッドを管理するユーザーレベルのスレッドライブラリで，スレッドのコアへの割り当てや制御を行う．MULiTh が管理するスレッドは Future のプロセスに割り当てられた資源を共有する．スレッドは，同期やコア放棄等のタイミングで切り替わるがプリエンプションは行わない．

4. 設　　計

本節では，本システムの設計について述べる。

4.1 基 本 設 計

本システムでは，CPU とメモリは Linux と Future で分割して管理され，それ以外の周辺機器などは原則として Linux が管理する．基本的に Future は軽量で高性能なマルチスレッドプログラム実行環境を提供し，入出力デバイスの管理はすべて Linux が行う．そのため，Future が入出力処理などを行うには，Linux が代行する必要がある．

Linux がブート時の OS として最初に起動し，Future は Linux から起動される．Linux 側から Future 側に対しての通信や制御などは，全て専用のデバイスドライバを介して行われる．Future のプログラムの実行制御や，Future 上でのシステムコールの代行は Future プロセスに 1 対 1 で対応する Linux 側プロセスによって行われる．Linux 側プロセスと Future 側プロセスの対応関係を図 2 に示す．

4.2 CPU

Linux は単一のコアのみを利用し，残りの複数のコアは Future で管理するという構成をとる．そのため，Linux カーネルは 1 つ目の CPU のみを利用するだけでよく，2 つ目以降

Vol.2009-ARC-183 No.17
Vol.2009-OS-111 No.17
2009/4/23

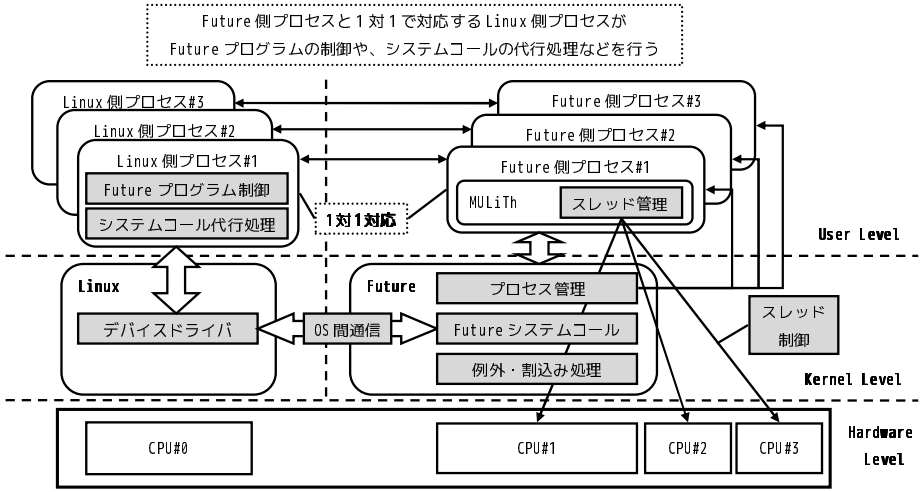

図 2 Linux 側プロセスと Future 側プロセスの対応
Fig. 2 Correspondence relation of Linux process between Future process

の CPU に関しては無視してよい。Linux から直接 Future の管理する CPU 資源を利用することはできない.

4.3 メ モ リ

メインメモリを Linux 用の領域と OS 間通信用の領域、Future 用の領域の 3 つに分割する．Linux カーネルは通常では存在するメモリの全領域を利用しようとするため，カーネルの mem オプションによって，Linux が利用するメモリを限定し，残りの部分を OS 間通信用と Future 用の領域として利用する．Future 用の領域は図 3 に示すように，mmap によって Linux 側の仮想アドレス空間にマップすることで，Linux 上から Future のメモリ領域をアクセスすることができる.

4.4 OS 間通信

Future へのプログラムの実行依頼や、Linux への I/O 処理の代行依頼などは OS 間通信という機構を用いて行われる。各種依頼は，OS 間通信メッセージ（以下，メッセージ）という単位でやり取りされる．メッセージは CPU 間割り込みによってそれぞれの OS に対して通知され，メモリ内の OS 間通信用の領域を用いて送受信される．OS 間通信の詳細は次節で述べる.

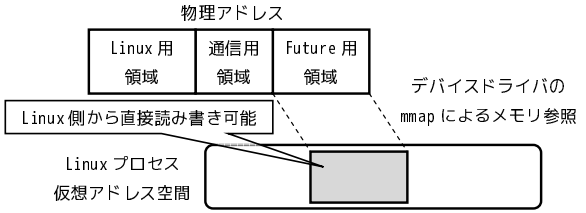

図 3 mmap による物理アドレスへのアクセス
Fig. 3 Access to physical address by mmap

5. OS 間通信

Future プログラムの起動依頼や，システムコールの代行処理は OS 間通信を利用して行われる．連携で用いる OS 間通信は CPU コア間の割込を用いて通知し，データの受け渡しは共有メモリ中に確保した通信用領域上の FIFO バッファを用いる．OS 間通信は，代行するシステムコールの種類などの情報をやり取りする.

代行依頼などは OS 間通信のメッセージとして依頼の種類や引数などの情報を渡す．実際のデータの入出力は，引数から得たアドレスから mmap による共有メモリよって行う．メモリを共有して読み書きするため，少ないオーバーヘッドで大量のデータのやり取りが可能となる.

5.1 OS 間通信メッセージ

メッセージは固定長のヘッダ部と可変長のデータ部から成り，図 5 に示すような構造になっている．以下にメッセージの各フィールドの内容を示す.

- message_type
  メッセージの種類を示す．これによって代行するシステムコールの種類を判別する.
- ltn
  Future プロセス上での論理スレッド番号を示す．これによって，どのスレッドによって発行された依頼メッセージか判別できる．また結果を返す際も，どのスレッドに対して結果を返すのかを判別するために用いる.
- data_size
  メッセージのヘッダの後に続く可変長の data_body 部分のサイズを示す.

Vol.2009-ARC-183 No.17
Vol.2009-OS-111 No.17
2009/4/23

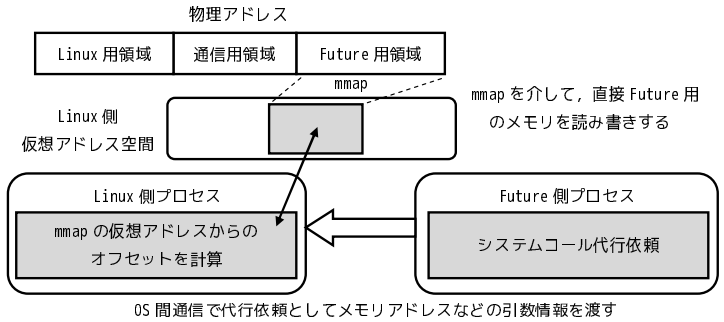

図 4 mmap による共有メモリアクセス
Fig. 4 Shared memory access by mmap

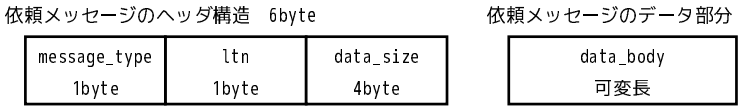

図 5 メッセージのデータ構造
Fig. 5 structure of message

- data_body
  代行するシステムコールの引数などの情報をシリアライズして送る．data_body のデータの内容は message_type ごとに判断される．

5.2 OS 間通信用バッファ

OS 間通信用のバッファは Linux 側プロセスと Future 側プロセスのペア一つに対して，全二重のバッファが一つ割り当てられる．また，Future の起動通知などの特殊な OS 間通信に用いる専用のバッファも存在する．各バッファは FIFO 方式のリングバッファで，システムコールの代行などのプロセスごとの通信は，各プロセスに割り当てられたバッファを利用する．OS 間通信のバッファを図 6 に示す．

5.3 OS 間の依頼処理

Future プロセスの起動依頼や，システムコールの代行処理依頼といった OS 間での依頼処理は，OS 間通信のメッセージを用いて行なう。図 7 に依頼処理時のシーケンスを示す．

Linux 側プロセスは，Future 側からシステムコールの代行依頼や，終了の通知があるまで

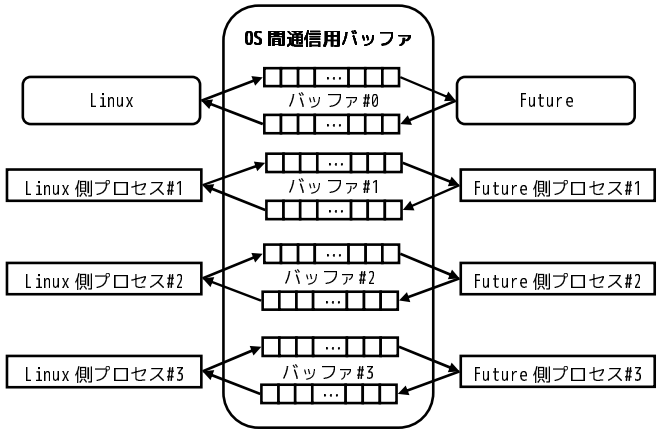

図 6 OS 間通信用バッファの構造
Fig. 6 structure of buffer for inter OS communication

は特に処理を行う必要はないため，Future からの通知を待ち，スリープすることで Linux 上で実行中の他のプロセスに対して CPU などのリソースを明け渡す．Future 側でシステムコール代行処理などの依頼が発生すると，まず OS 間通信用のバッファ領域にメッセージを書き出す．次に，依頼待ちでスリープしている Linux 側プロセスを CPU 間割込みを利用して，Linux 側のプロセスを起こす．依頼を発行したスレッドは結果が戻ってくるまでウェイト状態になる．

CPU 間割込みを受けた Linux は CPU 間割込みに対応した IRQ 番号に登録してある処理を実行し，スリープから目覚める．続いて OS 間通信用バッファからメッセージを読み取り，message_type ごとにそれぞれの処理を行う．システムコールの代行処理であった場合，Future 側に実行結果を返した上で，次の代行依頼が来るまで再びスリープする．終了通知であった場合は，Linux 側へ終了確認通知を Future へ送り，Linux 側プロセスは終了する．

6. 連 携 機 構

Linux と Future は互いに役割を分担し，連携することによって一つのシステムとして動

Vol.2009-ARC-183 No.17
Vol.2009-OS-111 No.17
2009/4/23

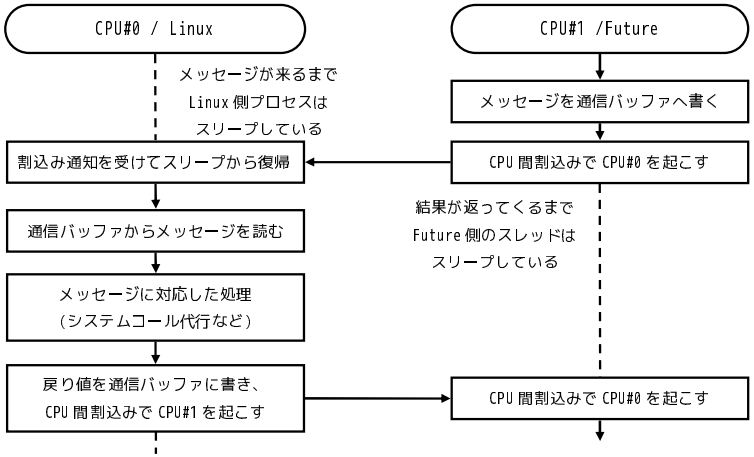

図 7 OS 間の依頼処理
Fig. 7 processing sequence of request

作する．OS 間連携機構では，次の機能を提供する．

Linux からの Future の起動

あらかじめ起動した Linux から Future のカーネルをロードし，ブートする．ロード及びブートは Linux のユーザープロセスからデバイスドライバを経由して行う．

Future 用プログラムのロード及び実行

Linux のファイルシステム上で管理している Future 用プログラムを Linux からロードし，Future 上で実行させる．また，実行時にプログラムで利用する CPU コアの数を引数として指定できる．

Future 用プログラムのシステムコール代行処理

Future 用プログラム中で発行されたシステムコールを Linux に代行させることによって，Linux 側で管理している I/O などの資源を利用可能にする．

6.1 Future の起動処理

Linux から Future を起動させるシーケンスを図 8 に示す．

(1) リセット解除によって CPU#0 起動し，Linux のカーネルが起動する．この時点では，CPU#1 は停止状態のままである．

(2) 起動した Linux から Future 用のメモリ領域にカーネルプログラムをロードする．カーネルは mmap を介して Future 領域へロードする．

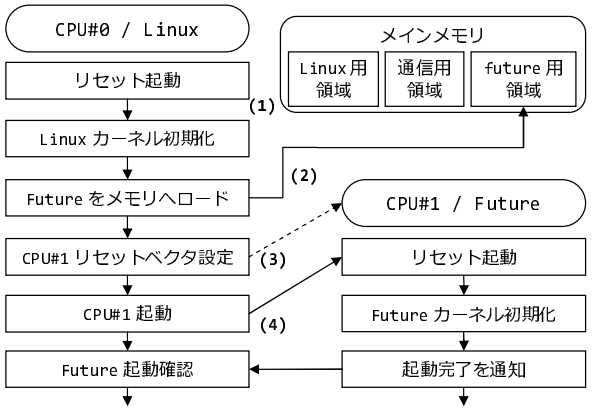

図 8 Future の起動処理
Fig. 8 Future OS boot process

(3) Linux から CPU#1 のリセットベクタアドレスをロードしたカーネルのエントリポイントに設定する．これによって，CPU#1 は起動時に指定したアドレスから実行を開始する。

(4) CPU#1 を起動させる．CPU#1 は Future カーネルのエントリポイントから実行を開始し，Future カーネルが立ち上がる．起動した Future カーネルは起動後に Linux に対して起動完了通知を行う．Linux 側は起動完了通知が一定時間内に受け取れなかった場合に，異常処理を行う．

6.2 Future 用プログラムのロード及び制御

Future プログラムのプログラム制御ルーチンはプログラムのロード・実行依頼後は，終了までスリープして待機する．システムコールの代行が必要な場合は，このプログラム制御ルーチンが依頼を受け取りシステムコールを代行する．Linux から Future 用プログラムを起動させるシーケンスを図 9 に示す．

(1) 起動した Linux から Future 用のメモリ領域にプログラムをロードする．

(2) OS 間通信で Future に対して実行依頼を行う．この際にロード先のアドレスや利用するコア数などを通知する．依頼を受けた Future は通知されたロード先アドレスから実行を開始する．

Vol.2009-ARC-183 No.17
Vol.2009-OS-111 No.17
2009/4/23

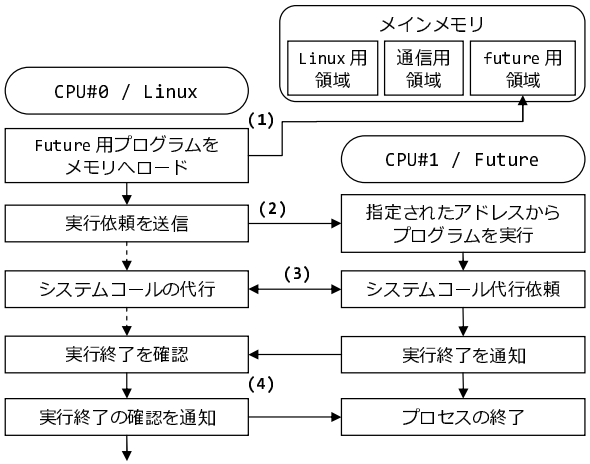

図 9 Future 用プログラムのロード及び実行
Fig. 9 Load and execute of program for Future

(3) システムコールが発行された場合は，Linux に対して代行依頼を行う．Linux は代行した結果を Future へ返す．

(4) Future のプログラムが終了したら Linux へ通知を行う．Linux は通知を受けると Future へプロセスの終了依頼を出す．

6.3 システムコールの代行処理

システムコール代行処理では Future プログラム内のファイル入出力に関するシステムコール (open,close,read,write など) とネットワークに関するシステムコール (socket,connect など) を Linux 側で代行する．現段階では，ファイルシステムに関するシステムコールの一部のみを実装している．Future プログラム中のシステムコールの Linux での代行処理のシーケンスを図 10 に示す．

(1) 標準ライブラリ中のシステムコール呼び出しをフックし，Future のシステムコールを呼び出す．

(2) Future のシステムコールは OS 間通信を利用して，Linux 側のプログラムに対してシステムコール番号，引数を渡す．read/write を例に挙げると，引数として，ファイルディスクリプタ，読み書きするメモリのアドレス，読み書きするバイト数を Linux 側に渡す．

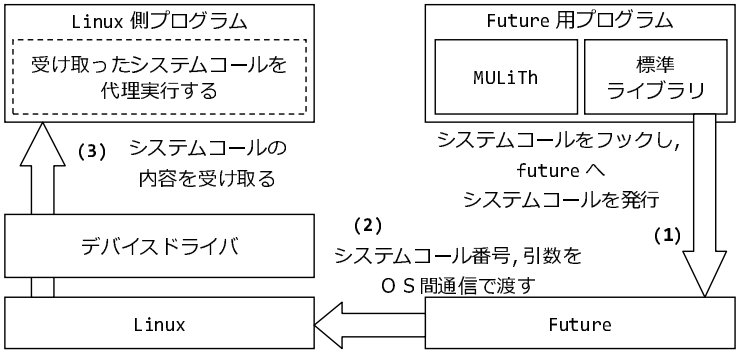

図 10 システムコールの代行処理
Fig. 10 Systemcall emulation

(3) Linux 側のプログラムは OS 間通信を通して受け取った情報からシステムコールの種類を判別し，代わりにシステムコールを Linux へ発行する．read/write 等のメモリの読み書きを伴うシステムコールの場合は，受け取った引数の情報より，読み書きするメモリアドレスの mmap してある Future 領域からのオフセットを求め，直接データの読み書きを行う．メモリを共有しているので，オーバーヘッドが少なくデータの読み書きが可能である．また，代行したシステムコールの結果は，OS 間通信を用いて Linux から Future へ通知される．

6.4 OS 間通信のインターフェース

Linux 側はデバイスドライバを介して Future との連携を行う．通信時に利用するバッファの場所はドライバ側で決定されるので，送受信時にその都度指定する必要はない．表 2 に Linux 側のデバイスドライバのインターフェースの一覧を示す．

- read
  OS 間通信用バッファにデータを読み込む．メッセージ到着の CPU 間割込みを受け取るまで，スリープする．
- write
  OS 間通信用バッファにデータを書き込む．指定したデータをそのまま書き込むので，予めメッセージ用のデータを作っておく必要がある．

Vol.2009-ARC-183 No.17
Vol.2009-OS-111 No.17
2009/4/23

表 2 Linux 側のデバイスドライバのインターフェース
Table 2 interface of Linux side device driver

ハンドラ名	機能
read	Future からメッセージを受信（メッセージが来るまでスリープ）
write	Future へメッセージを送信
mmap	Future 用領域を Linux プロセスのアドレス空間にマップ
ioctl(BOOT_FUTURE)	CPU#1 のリセットベクタを指定し，CPU#1 を起動させる
ioctl(CREATE_PROC)	Future のプロセス情報の生成（pid やロード先アドレスなど）
ioctl(RUN_PROC)	Future プロセスの実行依頼を送信する

表 3 Future 側の OS 間通信用関数
Table 3 interface of Linux side device driver

ハンドラ名	機能
l2f_read	Linux からメッセージを受信
f2l_write	Linux へメッセージを送信

- mmap
  Future 用領域を Linux プロセスのアドレス空間にマップする．Future カーネルのロード，Future 用プログラムのロード，システムコール代行処理時のデータのやり取りなどで利用される．
- ioctl(BOOT_FUTURE)
  CPU#1 のリセットベクタを指定し，CPU#1 を起動させる．予め，mmap を利用して Future カーネルを特定のアドレスにロードしておく必要がある．
- ioctl(CREATE_PROC)
  Future 用プログラムのロード先アドレスや，Future 側のプロセス ID などを設定する．Future 用プログラムのロード先は，Linux 側で管理し，このインターフェースを用いて設定される．利用する通信バッファの場所もここで決定される．
- ioctl(RUN_PROC)
  Future プロセスの実行依頼を送信する ioctl(CREATE_PROC) で指定されたロード先に mmap を利用してプログラムをロードしておく必要がある．

Future 側はカーネル内の関数を介して OS 間通信を行う．表 3 に Future 側の OS 間通信用関数の一覧を示す．

- l2f_read
  通信用バッファからメッセージを読み込む．どの通信用バッファから読み込むか指定する必要がある．
- f2l_write
  通信用バッファからメッセージへ書き出す．どの通信用バッファへ書き込むか指定する必要がある．

現段階の実装状況では，連携機構によって，Future の起動，Future 用プログラムのロード及び実行をすることが可能となっている．また，システムコールの代行処理についても，open，write を実装し，Future 用プログラム中の出力を Linux 側へ書き出すことが可能である．

```
int main(void){
 return;
}
```

図 11 計測用プログラム
Fig. 11 program for evaluation

表 4 情報家電用マルチコアプロセッサ仕様
Table 4 Spec of multicore processor for consumer electronics

項目	仕様
CPU コア	SH-4A(600MHz) × 4 core
システムバス周波数	300MHz

7. 評　　価

本システムを情報家電用マルチコアプロセッサ[8](表 4) へ試作，実装した．実装は，Linux からの Future の起動，Future 用プログラムの制御，システムコール代行処理の一部を実装した．今回の評価では，即時終了する Future プログラム（図 11）を実行して OS 間通信の基本的なラウンドトリップ時間を計測した。計測は，実行依頼の送信から終了通知の受信までの時間を計測した．計測には gettimeofday 関数を利用し，10 回計測を行った平均値を結果として示した．

また、Linux から Future 用プログラムのロード，Future プロセスの生成を含めた，終了までの所要時間を計測した．こちらは，Future 用プログラムを Future 用領域へロードする部分から終了確認通知を Future に送信するまでの時間を計測した．こちらも計測には gettimeofday 関数を利用し，10 回計測を行った平均値を結果として示した．

これらの結果を 5 に示す．また，参考データとして，SH-Linux(kernel-2.6.19) の fork によるプロセス生成の所要時間を併せて示す．

まず，OS 間通信ラウンドトリップ時間について考察する．このオーバーヘッドが発生するのは Future 用プログラムの実行とシステムコール代行依頼を発行するときだけである．

Vol.2009-ARC-183 No.17
Vol.2009-OS-111 No.17
2009/4/23

表 5 計測結果
Table 5 measuring result

項目	所要時間
OS 間通信ラウンドトリップ時間	90[μsec]
Future : ロード〜プロセス生成〜終了	11.53[msec]
（参考）SH-Linux : fork〜execv〜wait	6.37[msec]

システムコール代行に伴う OS 間のデータのやり取りは，mmap による共有メモリアクセスによって行われるため，オーバーヘッドはほとんど発生しないと考えられる．よって，プログラム全体の実行時間から見れば 1 回あたりの時間は 90[μsec] と極めて短い時間であるため，OS 連携機構を実現するのに十分な性能があると判断できる．

次に，プログラムのロードなどを含めた計測結果であるが，Linux でのプロセス生成及び実行のに比べて 5.16[msec] の時間が掛かっているが，Future 上での実行を想定しているプログラム本体の実行時間に対しては十分に短い時間であり，このオーバーヘッドによる遅延よりも Future 用プログラムの管理を Linux 上で行えることの利便性の方が高いと考えられる．

8. 関 連 研 究

本節では，既存のハイブリッド OS 構成の研究についてまとめる．

マルチコア SH における複数カーネルの実行機構の設計と実装 [2)]

この研究では，CPU やメモリなどの資源を分割し，同時に動作する複数のカーネルに占有させる仕組みを提案，実装している．資源を分割占有させることでオーバーヘッドなく複数の OS を同時に実行することを可能にしている．この研究では，OS 間の通信は仮想ネットワークによるデータのやり取りのみであり，複数の OS はそれぞれ別々に利用するしかない．一方，本研究では，OS 間で連携することで，汎用 OS の I/O 資源を専用 OS で利用することを可能とし，１つのシステムとして利用可能としている．

シングルチップマルチプロセッサ上のハイブリッド OS 環境の実現 [3)–4)]

マルチコア上の異種 OS 間通信機能の設計と評価 [5)]

これらの研究では，μITRON や T-Kernel といったリアルタイム OS と Linux をマルチプロセッサ上で独立に動作させ，汎用性とリアルタイム性を両立する仕組みを提案，実装している．また，この研究も，OS 同士が連携して動作する機構はない．本研究では，並列演算向けの専用 OS と連携させることで，汎用性と演算性能の両立した１つのシステムとして利用可能である．

9. お わ り に

本稿では，汎用ホモジニアスマルチコアにおける異種 OS の連携機構について述べた．連携機構の一部を試作，実装し，評価の結果、OS 間通信の性能は問題がないことを確認した．今後の課題としては，現段階では，open，write のシステムコールの代行処理のみの実装となっているので，read や socket，connect などの代行処理の追加実装をし，Future のマルチプロセス化への対応，GUI の代行などの機能強化を図る予定である．また，OS 連携機構を利用した実践的なプログラムによる性能の検証を行い，評価したい．

謝辞 本論文おけるマルチコアチップ及び実行環境は NEDO リアルタイム情報家電用マルチコアプロジェクトにて早稲田大学，(株）ルネサステクノロジ，(株）日立製作所により提供された．関係者各位に感謝する．

参 考 文 献

1) S. Torii et al., "A 600MIPS 120 mW 70 μA Leakage Triple-CPU Mobile Application Processor Chip," Proc. 2005 IEEE Int'l Solid-State Circuits Conf., Digest of Technical Papers (ISSCC 05), IEEE Press, 2005, pp. 136-137.
2) 下澤拓，藤田肇，石川裕．マルチコア SH における複数カーネルの実行機構の設計と実装．情報処理学会研究報告,2008-OS-109,pp.25-32(2008)
3) 遠藤幸典，菅井尚人，山口義一，近藤弘郁．シングルチップマルチプロセッサ上のハイブリッド OS 環境の実現 - システムアーキテクチャ -．情報処理学会第 66 回全国大会 2D-5(2004)
4) 菅井尚人，遠藤幸典，山口義一，近藤弘郁．シングルチップマルチプロセッサ上のハイブリッド OS 環境の実現 - OS 間インターフェースの実装 -．情報処理学会第 66 回全国大会 2D-6(2004)
5) 遠藤幸典，菅井尚人，山口義一，近藤弘郁．マルチコア上の異種 OS 間通信機能の設計と評価．情報処理学会第 68 回全国大会 5A-4(2006)
6) 日本エンベデッド・リナックス・コンソーシアム (Emblix)．Linux における RTOS とのハイブリッド構成に関する仕様 (第 1 版)．(2002)
7) 佐藤未来子，磯部泰徳，十山圭介，野尻徹，入江直彦，内山邦夫，並木美太郎．汎用ホモジニアスマルチコアプロセッサにおける OS とスレッドライブラリ．情報処理学会第 20 回コンピュータシステムシンポジウム, ポスターセッション,(2008)
8) 早瀬清ほか，独立に周波数制御可能な 4320MIPS、SMP/AMP 対応 4 プロセッサ LSI の開発，情報処理学会研究報告,2007-ARC-55,pp.31-35(2007)