Vol.2009-ARC-183 No.15
Vol.2009-OS-111 No.15
2009/4/23

ヘテロジニアスマルチコアエミュレータにおけるコア間通信とアトミック命令の実装

青木　亮[†1]　追川修一[†1]

本研究では，ヘテロジニアスマルチコアにおけるボトルネックの発見や，新しい高速化手法の提案につなげることを目的に，QEMU をベースにした x86・ARM プロセッサによるヘテロジニアスマルチコアエミュレータ Gryphin を開発する．本論文では，異なる種類のプロセッサ間接続に専用の共有メモリを使用し，共有メモリ上でのリングバッファを用いた通信とプロセッサ間割り込み，およびホスト OS 経由によるアトミック命令の実現について述べる．

Implementation of the Inter-Core Communication and Atomic Instructions in a Heterogeneous Multi-Core Emulator

Ryo Aoki[†1] and Shuichi Oikawa[†1]

We are currently developing heterogeneous multi-core emulator, named Gryphin, that emulates the x86 and ARM processors based on QEMU. Gryphin aims at discovery of bottlenecks in heterogeneous multi-core and proposal of new speed-up techniques. This paper describes the implementation of inter-processor communication and atomic instructions on Gryphin. A ring buffer is used on dedicated shared memory for connection between emulated processors, and atomic instructions are correctly emulated by executing them on the host OS.

1. はじめに

近年の組み込み機器に要求される高機能性，多機能性により，機器に搭載される CPU も，x86 アーキテクチャなどの汎用 CPU 同様，マルチコア化という手段を取り始めた．汎用 CPU では，同種類のコアを複数搭載するホモジニアスマルチコアが一般的だが，比較的開発規模が小さい組み込み機器向け CPU では異なる種類のコアを複数搭載する，ヘテロジニアスマルチコア製品が発表されており，多種多様な要求への対応を試みている．また，最近の汎用 CPU にもマルチコアというアプローチから，IBM の Cell[1] や，Intel のメニーコアプロセッサ Larrabee のような新しいアーキテクチャを持つものも登場している．

今後も，汎用，組み込み CPU に関わらず，既存コアの組み合わせや，新しいアーキテクチャによる CPU が登場すると予測できる．しかしマルチコア化といっても，数十，数百ものコアの増加が予想され，多種多様なマルチコア構成が考えられる．また，実際のハードウェアでは，システムバスは非互換であり，接続することはできない．そのため，各構成ごとの開発やテストは，コストや時間の面から非効率的であり，マルチコアの研究を行う上で大きな問題となる．このような問題に対して，QEMU[2] のような PC エミュレーションという手段は，ソフトウェア上で様々な環境構築やテストを行えるため，非常に有効である．

そこで本研究では，ヘテロジニアスマルチコアに対応するシステムソフトウェアの研究や，従来のマルチコアにおけるボトルネックなどの発見，新しいマルチコア高速化手法の提案を目的に，QEMU をベースにした，ヘテロジニアスマルチコア対応エミュレータの開発を行う．ヘテロジニアスマルチコアを扱えるエミュレータを開発することで，既存のコアを組み合わせた CPU エミュレーションを可能にする．また，比較的容易に新しいコアの移植が可能な QEMU をベースにすることで，今後増加するであろう新しいアーキテクチャの CPU にも対応しやすい．将来的には様々なマルチコア構成を再現可能にし，ヘテロジニアスマルチコアのポテンシャル評価をも行える，より柔軟なフレームワークの提供を目標としている．

今回の開発では x86 プロセッサと ARM プロセッサを共有メモリにより接続する．共有メモリはホスト OS 上で mmap() システムコールを使用することで確保し，各ゲスト OS は，デバイスドライバ経由で共有メモリにアクセスすることで，相互通信を行うことができる．また，プロセッサ間の割り込みを実装し，相互通信時に割り込み処理を行うことができ，2 つの Gryphin プロセス間におけるアトミック性を保証したことで，共有メモリに対して安全な排他制御を行うことができる．

2. Gryphin アーキテクチャ

本研究では QEMU をベースにした，ヘテロジニアスマルチコア対応エミュレータを開

†1 筑波大学
University of Tsukuba

Vol.2009-ARC-183 No.15
Vol.2009-OS-111 No.15
2009/4/23

発する．このエミュレーションソフト名を *Gryphin* とし，以下同エミュレータを Gryphin と記載する．本章では，Gryphin のアーキテクチャについて述べる．

2.1 Gryphin アーキテクチャ概要

2.1.1 ベースとなるエミュレータ

本研究では，ヘテロジニアスマルチコアをサポートするエミュレータを開発することが目的である．Gryphin では異なる種類のプロセッサを接続したいため，多くのプロセッサをエミュレートしなければならない．しかし，プロセッサ 1 つをエミュレートするには時間的，技術的な制約が厳しいため，できるだけ多くのプロセッサをサポートするエミュレータを土台にし，新たな機能としてヘテロジニアスマルチコア機能を追加するアプローチが望ましい．

今回は，ヘテロジニアスマルチコア対応エミュレータを開発するに当たって，多くのプロセッサアーキテクチャをサポートする QEMU を選択した．これは，Bochs[3] と比べても多くのプロセッサアーキテクチャをサポートしており，多種多様なコアの組み合わせを可能にすることや，QEMU には高い拡張性と移植性があり，使用実績も多いこと，ダイナミックトランスレーション機能により，プロセッサの追加実装がしやすいことから，研究の将来性向上が見込めるためである．

QEMU をベースとするため, 本研究ではホスト OS，およびゲスト OS 共に，親和性が高い Linux を用いることとした．

2.1.2 プロセッサ構成

QEMU は 最大 255 個の SMP（Symmetric Multi Processing） によるホモジニアスマルチコアをサポートしている．このホモジニアスマルチコア機能を変更し，SMP コアのひとつを x86 コアに，ひとつを ARM コアとすると変更が少なく，良い設計のように考えられる．図 1(a) は，QEMU の SMP コアの一方を単純に x86 から ARM に置き換えただけの構成である．しかし，QEMU はエミュレートするプロセッサごとにバイナリが作成されてしまうという問題がある．

QEMU はコンパイル時に，サポートするプロセッサリストから実際に使用するプロセッサを選択する必要がある．このとき選択したプロセッサごとに実行ファイルが作成される．つまり，QEMU は 1 つの実行ファイルでサポートする全てのプロセッサ環境をエミュレートするのではなく，各プロセッサ環境ごとにコンパイルされたプロセッサ専用の実行ファイルで環境をエミュレートする．そのため，x86 と ARM をサポートするように QEMU をコンパイルすると，x86 プロセッサをエミュレートする QEMU バイナリと ARM エミュレートする QEMU バイナリの 2 種類ができる．

図 1(a) のように 1 プログラムで複数のプロセッサをサポートするには QEMU の大幅な変更が必要である．また仮に，両コアを 1 つの QEMU プログラムで動作させたとしても，その上で OS を動作させるには，カーネルを変更する必要がある．さらにもし，各コアごとに異なる OS を動作させたとしても，メモリ全体をコア間で共有しているので，メモリを共有する部分でカーネルの変更を招く．

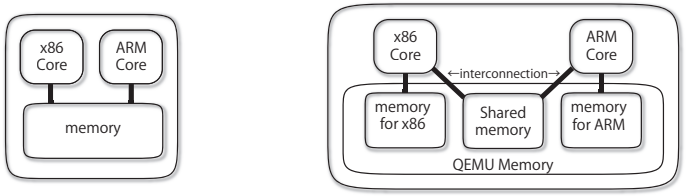

図 1 ヘテロジニアスマルチコアシミュレータ Gryphin の構成

そこで，今回は x86 用 QEMU と ARM 用 QEMU にそれぞれ手を加え，プロセス間で通信を行うことで，擬似的にひとつのプロセッサとして動作するように実装する．そのため，今回は図 1(b) に示すシステムを提案する．図 1(b) では，各プロセッサ専用のメモリと，その他に各コアから同様にアクセス可能な共有メモリを追加する．このように設計することで，各コア上で手を加えていない別の OS を動作させることが可能になる．

また本研究の目的上，アーキテクチャの異なるコアを多数接続したいが，研究は初期段階であるため，プロセッサ間の接続が容易になるように，x86 および ARM コアを 1 つずつ接続することとする．

2.1.3 プロセッサ（コア）間通信と OS 間通信の定義

ここまでの設計で，プロセッサは x86 と ARM コア各 1 つ，かつこれらを専用の共有メモリで接続することとした．これよりプロセッサ（コア）間の通信と OS 間の通信について述べるが，混乱を避けるため，本論文ではプロセッサ間通信，またはコア間通信といった場合，エミュレートされるハードウェア上での通信を意味するものとする．また OS 間通信といった場合は，ゲスト環境で動作しているソフトフェア間の通信を表している．ソフトウェアとは，カーネルやアプリケーションを指すものとする．

Vol.2009-ARC-183 No.15
Vol.2009-OS-111 No.15
2009/4/23

2.1.4 プロセッサ間接続と OS 間での通信方法

Gryphin と同じような構成を持つヘテロジニアスマルチコアプロセッサには Cell や，携帯向けの SH Mobile G3 などがあるが，Gryphin もこれらのプロセッサと同様に何らかの方法でコア間通信や OS 間通信を可能にする必要がある．

Cell や SH Mobile G3 は各コア間との通信のためにコア同士を専用のバスで接続している．また，SH Mobile G3 のように OS が動作可能なプロセッサコアを複数搭載したチップは，各コア上で異なる OS を動作可能，また OS 間通信が可能であり，実際にそうした目的で利用が想定されている．本研究で開発する Gryphin も，プロセッサコアに OS が動作可能な 2 つのアーキテクチャを用いるため，同様に図 2 のような状況を想定した．

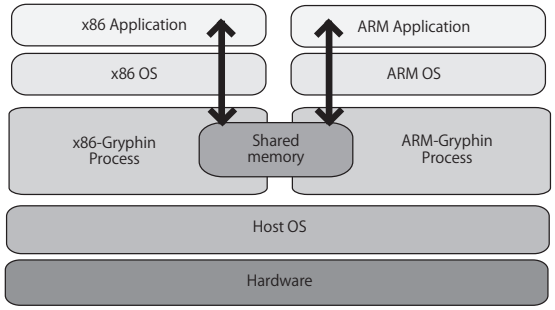

図 2 想定する動作環境

このような環境では，各プロセッサで異なったバイナリを処理し，各 OS では異なったアプリケーションサービスを提供することが考えられる．バイナリは異なるコアごとにしか処理できないが，データ自体はエンディアンの違いがあっても，プロセッサアーキテクチャに関わらず共有可能である．そのため，サービス間でデータを交換し，協調して動作することが想定される．

そこで，Gryphin では共有メモリを各コアごとに分配し，共有メモリ経由でコアを接続，共有メモリにリングバッファを実装することで，OS 間の通信を行うこととする．

2.2 共有メモリの設計

本節では，プロセッサ間を接続する共有メモリの設計について述べる．

2.2.1 ホスト OS 上での共有メモリの仮想化

Gryphin では，QEMU を改造し，共有メモリを追加する必要がある．つまり，ゲスト OS に対し，仮想化したメモリを提供する．メモリを仮想化するには，単にホスト OS 上に共有メモリに必要なサイズのメモリ領域を確保し，ゲスト OS からはその確保した領域を共有メモリとして使用してもらえばよい．ここで 1 つ問題となるのがホスト OS である Linux では記憶領域を確保する方法がいくつかあるということである．今回は x86-Gryphin と ARM-Gryphin の 2 プロセスを動作させ，プロセス間通信を行う必要がある．そのため，Linux 上で行えるプロセス間通信に限られる．

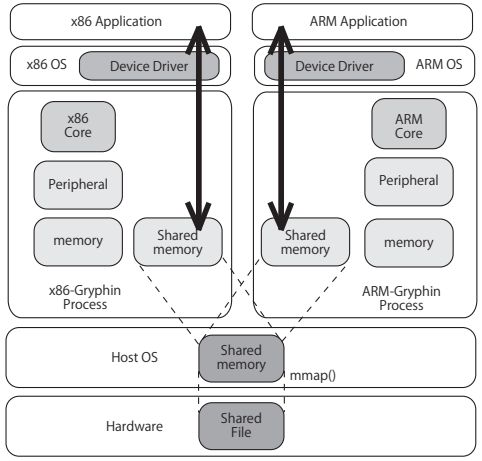

図 3 Gryphin アーキテクチャ

ここでは，図 3 のように，Linux が提供する mmap() システムコールを使用し，ホスト環境に用意した共有メモリ用のファイルを各 Gryphin プロセス空間上の共有メモリアドレスにマップする．Gryphin 上で動作する各 OS はこの共有メモリアドレスにアクセスすることで，OS 間の通信を行う．

2.3 デバイスドライバの設計

各ゲスト OS が動作するために必要なメモリと，データ伝送用の共有メモリを分けたことで，カーネルには変更を加えずに各コア上で，ゲスト OS を動作させることが可能になった．しかし，共有メモリにアクセスするには依然，カーネルを変更する必要がある．本研究では移植性を向上させるため，なるべくゲスト OS には変更を加えたくない．そのため，今回は共有メモリを新たなデバイスとみなし，共有メモリのデバイスドライバをカーネルモ

Vol.2009-ARC-183 No.15
Vol.2009-OS-111 No.15
2009/4/23

ジュールとして実装することで，カーネルへの変更を最小限にとどめる．

2.3.1 共有バッファの直接参照

デバッグや，アプリケーションによるデバイスの管理を実現するため，共有バッファを直接操作できる必要がある．デバイスドライバに mmap() を行える機能を追加することで，これを可能にする．この場合は，共有メモリの管理をアプリケーション側に任せることとなる．そして共有メモリは，複数のアプリケーションから同時に使用されることが考えられるため，同じ領域にアクセスする場合には，何らかの排他制御が必要になる．排他制御には何らかのアトミック命令が必要になるが，これは 2.5 節で述べる．

2.3.2 リングバッファ

ゲスト OS 間での通信の際，データを確実に受け渡すためのバッファリングが必要となる．今回は共有メモリを各プロセッサごとに分割し，リングバッファとして実装することとした．リングバッファとは，論理的に環状構造となるデータ構造のことで，通信を行う際は，よく用いられる手法である．ここでは，送信用と受信用の二つのリングバッファを用いて，相手と通信を行うこととする．こうすることで，送信データと，受信データを区別することなく，スムーズにデータをやり取りすることが可能である．今回は 1 対 1 の通信だけを考慮し，共有メモリを 2 つに分割し，1 つを送信用，もう 1 つを受信用とした．このように実装することで，排他制御を行うことなく通信ができる．また，2 者間でバッファの位置を相互に知る必要があるため，このバッファの位置をオフセットとして保存するためのレジスタを共有メモリの最初のページフレームに割り当てる．

2.4 プロセッサ間割り込み

Gryphin による OS 間通信は，リングバッファによって行うが，相手先の負荷が上がり，データを処理しきれなくなることでバッファがいっぱいになると，通常，相手の準備が整うまで自分のプロセスをブロックしなければならない．ブロックを解除するためには，カーネルの割り込み機能使用するが，Gryphin では，プロセッサごとに異なる OS が動作しており，この方法は使用できない．また，単に専用の割り込みハードウェアを作り，そのハードウェアを利用して実装しようとすると，カーネルに大きな変更が必要になってしまう．そこで Gryphin では，エミュレートされている既存の割り込みコントローラを使い，プロセッサ間割り込みを実現する．

既存の割り込みコントローラを使うことで，他のデバイスと同様に割り込みコントローラに共有メモリ用の割り込み線を接続し，Linux に備わっている既存の割り込みシステムを使うことで，変更を最小限にとどめることができる．

実際にプロセッサ間割り込みを行うには，割り込みコントローラと接続された共有メモリ用コントローラを作成し，共有メモリ用コントローラに割り込み要求がきたときに，相手側の割り込みコントローラにその割り込みを伝えなければならない．共有メモリ用コントローラはプロセッサ間の割り込み要求を受信するインターフェイスとその割り込みを相手に送信する役割をもつ仮想的なコントローラである．

通常なら，単に相手側の割り込みコントローラに割り込み要求を送ればよいが，Gryphin では各プロセッサは異なるプロセスで動作しているため，共有メモリ用コントローラ間で何らかのプロセス間通信を行う必要がある．また，割り込みの非同期性を保つため，非同期なプロセス間通信を使うことが望ましい．

そこで，今回はソフトウェア割り込みであるシグナルを使用することでプロセス間の割り込みを実現する．

2.5 共有メモリ上のアトミック命令

共有メモリを直接操作している時の共有メモリや共有メモリ用コントローラのレジスタは，各プロセッサからは mmap() によりマップされた共有メモリとして扱われるため，各プロセッサから同時に書き込まれる恐れがある．こういった場合，共有メモリをロックして安全にデータを書き込みたい．資源の排他利用（ロック）には様々な方法があるが，どの方法においても，アトミックな命令が必要である．

QEMU はプロセス単体で動作することが想定されているが，Gryphin の各プロセッサは，ホスト OS 上では別プロセスで動作しており，エミュレータ自体がホスト OS によってコンテキストスイッチされてしまうため，そのままではアトミック性が保証できない．

そこで，Gryphin では，共有メモリをロックする際のアトミック命令を実装する．ソフトウェアのみでアトミック命令をエミュレートしようとするとオーバーヘッドが大きくなってしまうため，Gryphin ではホスト環境のロック機構を用いて，アトミック命令のエミュレーション機能を実装する．つまり，ホスト環境で共有メモリをロックし，アトミック命令をエミュレーションする．このように実装することで，エミュレートしている環境に左右されることなくアトミック性を保証できる．

3. 実　　装

本章では，2 章で述べた設計内容に基づき実装した Gryphin の実装内容について述べる．

3.1 全 体 構 成

Gryphin は QEMU をベースとしているため，QEMU の基本的な制約を継承している．

Vol.2009-ARC-183 No.15
Vol.2009-OS-111 No.15
2009/4/23

それは，各プロセッサアーキテクチャごとにバイナリがコンパイルされてしまうことである．今回は x86 アーキテクチャと ARM アーキテクチャを実装するため，図 3 の様にヘテロジニアスマルチコア環境をエミュレート中はホスト OS 上で Gryphin プロセスが 2 つ起動している．そして，各 Gryphin プロセス上では x86 プロセッサを CPU とするハードウェアと，ARM プロセッサを CPU とするハードウェアがエミュレーションされている．さらに，その仮想ハードウェア上ではゲスト OS とそのアプリケーションが動作している．Gryphin はホスト環境のファイルを共有メモリとして各 Gryphin プロセスにマップし，各 Gryphin 上で共有メモリとしてエミュレートされている．各ゲスト OS 上のアプリケーションはゲスト OS にカーネルモジュールとして組み込まれた共有メモリ用のデバイスドライバを使用し，相手のアプリケーションに対して通信を行う．

ここで大きく QEMU と異なるのは，各プロセスがホスト OS 上の同じファイルをマップし，共有メモリを実装している点と，共有メモリへアクセスするために，ゲスト OS にデバイスドライバを組み込んでいる点である．

3.1.1 Gryphin のエミュレート環境

QEMU ではエミュレーション開始時にコマンドラインから実行するハードウェア環境を選択できる．本開発では，各アーキテクチャで動作するハードウェアは共にデフォルトでエミュレートされるものを使用した．x86-Gryphin のデフォルトハードウェアは QEMU PC という QEMU がエミュレーションするハードウェアで構成された仮想 PC である．また，ARM-Gryphin は Integrator/CP というベースボードである．x86-Gryphin では CPU としてデフォルトの QEMU32 を，ARM-Gryphin では CPU として ARM926EJ-S を使用した．また，共に RAM をデフォルトの 128MB とした．

3.2 共有メモリ

この節では，プロセッサ間の共有メモリの実装について述べる．

3.2.1 共有メモリの領域の確保 ー mmap()

QEMU に共有メモリを追加するには，まず QEMU プロセスのメモリ空間上に共有メモリ用のファイルを mmap() システムコールでマップする．各 Gryphin プロセスは，ここで同じファイルをマップすることで，ホスト OS 上の共有メモリ機能を使い，プロセス間で同じメモリ空間を共有する．このように実装することで，valloc() などを呼ばずにメモリを確保することが可能である．

3.2.2 共有メモリの登録 ー cpu_register_physical_memory()

QEMU プロセスのメモリ空間に連続して確保された RAM 用の領域や，デバイスにアクセスするための I/O レジスタは，起動時に QEMU がエミュレートする物理メモリ空間に登録され，使用される．登録せずにその物理アドレス空間を使用しても，何も存在しないため，Linux Kernel は通常，/dev/zero をマップする．

QEMU では物理メモリ空間にエミュレートするメモリ領域を登録する関数がある．それが，cpu_register_physical_memory() 関数である．この関数は 3 つの引数を持っており，それぞれに，登録したい物理メモリの開始アドレス，サイズ，QEMU プロセスのメモリ空間で確保したメモリ領域の先頭からのオフセットを指定する．つまり，ゲスト OS 上から第 1 引数のアドレスを参照すると，実際にはホスト OS 上で確保したメモリ領域を参照することになる．

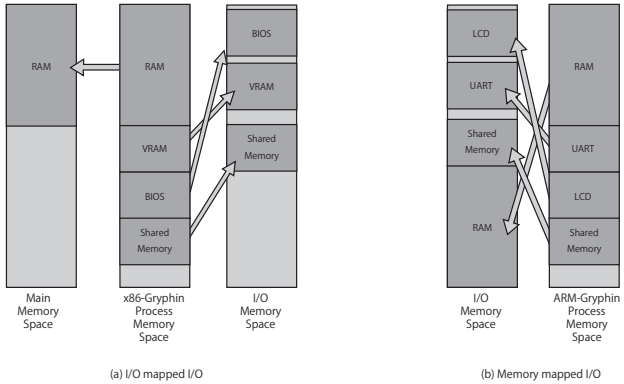

図 4 QEMU が管理する物理メモリ空間

この登録により，プロセス空間の仮想メモリアドレスから，エミュレートされる物理アドレスへのマッピングが行われる．x86 アーキテクチャでは，I/O 空間とメインメモリ空間は独立している I/O マップド I/O 方式であるため，図 4(a) ように別々のメモリ空間に再配置される．ARM アーキテクチャでは，I/O 空間とメインメモリ空間が同じアドレス空間であるメモリマップド I/O 方式であるため，図 4(b) のように，これらのアドレスが重ならないようにデバイスを配置しなければならない．

今回の共有メモリは，両アーキテクチャ上で共に空いている空間を使用することで，同じ物理アドレスを指定することができた．

Vol.2009-ARC-183 No.15
Vol.2009-OS-111 No.15
2009/4/23

3.3 デバイスドライバ

この節ではゲスト OS が共有メモリへアクセスする際に使用するデバイスドライバの実装について述べる．

3.3.1 共有メモリの直接参照の実装

マシンに接続されたデバイスの I/O レジスタや，バッファをユーザープログラムからアクセスするため，mmap() システムコールを提供する．3.2 節で述べたように，Gryphin でエミュレートされている共有メモリは，Gryphin 上の物理メモリ空間に存在する．まず，デバイスドライバはデバイスにアクセスがあった際には，Gryphin 上で共有メモリを登録した物理メモリアドレスとサイズを指定し，これをマップする．マップする際は，remap_pfn_range() 関数を使用した．remap_pfn_range() 関数は指定したプロセスの仮想メモリ空間に，物理メモリ空間をそのままマップする関数である．これにより，図 5 のように，ユーザープログラムは Gryphin デバイスに対し，mmap() システムコールを呼ぶことによって，簡単にそのプログラムの仮想メモリ空間に共有メモリをマップすることが可能である．

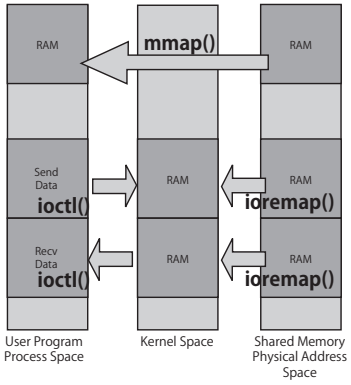

図 5 デバイスドライバの機能

3.3.2 通信用リングバッファの実装

OS 間でデータを通信する際には，データを確実に届けるため，何らかの方法でバッファリングを行う必要がある．本研究では，各アプリケーションごとに，受信用と送信用のリングバッファを用いて 2 プロセッサ間の通信を行うよう実装した．またバッファリングの際，データの損失をなすため，一方のプロセッサの読み込み処理が何らかの理由により滞り，受信データがリングバッファにいっぱいになっても，もう一方の書き込みデータがあふれないように書き込み処理はブロックされる．

リングバッファは今回，1 対 1 の 2 者間のみの通信と限定し，お互いの読み込み，書き込み位置を保存するレジスタを 2 組割り当てた．レジスタは，共有メモリが存在するメモリ領域の先頭，1 ページフレームに割り当てた．

3.3.3 プロセッサ間接続と OS 間通信の実装

OS 間通信のインターフェイスとして，今回は実装が簡単な ioctl() 関数を使用した．ioctl() 関数の引数は，デバイスに対するファイルディスクリプタと，そのデバイスに対する操作用コマンド，デバイスに対してのデータ用引数で構成されている．つまり ioctl() 関数はデバイスに対し，使用したい操作用コマンドを任意に定義できる関数である．

今回は読み込み用のコマンドとして GRYPHIN_IOC_READ を，書き込み用のコマンドとして GRYPHINI_IOC_WRITE を実装した．ただし，ioctl() 関数はデータのサイズを指定できないため，今回は 4KB の固定長データのみ送受信できる用に実装した．

さて，これらのコマンドが発行されたとして，ユーザプログラムと共有メモリ間でデータを送受信するには，まずデバイスドライバが動作するカーネル空間に共有メモリの物理メモリ空間をマップする必要がある．これは ioremap() 関数を使うことで実装した．ioremap() 関数はマップしたい物理メモリアドレスとサイズを指定すると，カーネルの仮想メモリ空間にマップし，その仮想メモリアドレスを返す関数である．これにより，共有メモリをマップした仮想メモリアドレスと，ユーザプログラムから指定されたデータ領域から，copy_to_user(), copy_from_user() 関数を使用してデータを送受信できる．

3.4 プロセッサ間割り込みの実装

プロセッサ間割り込みには，自分からの割り込み要求送信，相手側からの割り込み要求受信の 2 つの経路がある．

まず，自分からの割り込みを送信する経路を実装する．共有メモリ用コントローラを各プロセッサの I/O 空間に接続する．I/O 空間への接続には QEMU で提供されている cpu_register_io_memory() 関数を使用する．次に，割り込み要求をシグナルで相手のプロセスへ送信するコマンドを実装する．コマンドが実行されると，共有メモリ用コントローラはシグナルを相手のプロセスに送る．このようにすることで，自分からの割り込みを送ることができるようになる．

次に，相手側からの割り込みを受信する場合は各環境の割り込みコントローラにプロセッサ間割り込み用の IRQ を設定する．具体的には，x86 環境では i8259 に，ARM 環境では

Vol.2009-ARC-183 No.15
Vol.2009-OS-111 No.15
2009/4/23

PL192 に共有メモリ用の IRQ を登録する．この時点で，各プロセッサは割り込みコントローラを経由して共有メモリと接続されたことになる．次にプロセス間の割り込みを実装するために，シグナルハンドラを登録する．シグナルハンドラは割り込みコントローラを操作し，接続されたプロセッサに割り込み信号を送る．このようにすることで，相手から発行された割り込み要求を受け取ることができるようになる．

ゲスト OS より上位のレイヤーでは割り込みが発生すると，ゲスト OS へ通知が届き，割り込みハンドラが実行される．割り込みハンドラ中で，相手側の write() 関数などのブロックを検知した場合は，プロセスをアンブロックするため，シグナル送信する．

3.5 アトミック命令の実装

Gryphin における共有メモリ上で何らかのアトミック命令を実装するには，ホスト環境において共有メモリへのアクセスを排他制御し，アトミックに命令がエミュレーションされるようにしなければならない．

QEMU 上でこういった動作を可能にするには，デバイス I/O 用の関数に手を加えればよい．QEMU がエミュレートするデバイスに対して I/O 処理が行われる際は，デバイスごとにプログラマが定義した I/O 用の関数が呼び出される．そこで，共有メモリのロックを指示する専用レジスタを新たに定義し，このレジスタへのアクセスに用いられる I/O 用関数で，ホスト OS のロック機構を使い共有メモリをロックする．

このようにデバイスの I/O 用関数に，アトミックな処理を実装することで，共有メモリへのアクセス時におけるアトミック性を保証する．

3.6 リモートコマンド実行プログラムの実装

今回は，共有メモリとデバイスドライバを使用した実験を行うため，リモートコマンド実行プログラムを実装した．これは，指定したシェルコマンドを相手側のコアで実行し，その結果を表示する，アプリケーションプログラムである．図 6 に処理の流れを図示する．

このプログラムは，サーバクライアント型のプログラムであり，ioctl() 関数を使用してデータ送受信する．送受信の際は，データを NULL 文字で区切り，データの終端に連続した NULL 文字を設けることで，データを受け渡す．

クライアント側ではプログラムの引数に相手側で実行したいコマンドとその引数を指定して実行する．クライアントプログラムは実行された環境をサーバへ知らせるため，UID，GID，カレントディレクトリを取得し，これらの情報とコマンドライン引数を 4KB のバッファに格納し，それを ioctl() 関数で共有メモリに書き込む．その後 ioctl() 関数を呼び，サーバが結果を書き込みブロックを解除するまでブロックする．

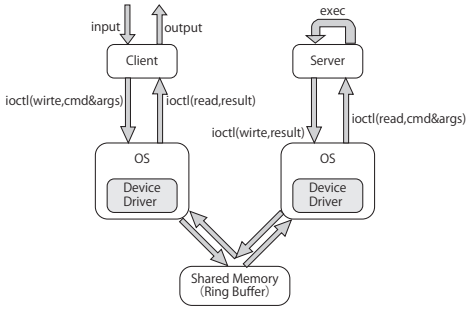

図 6 リモートコマンド実行プログラム

そして，ioctl() 関数が返ってくると，クライアントプログラムは結果を標準出力へ出力し，終了する．

サーバ側ではサーバプログラムを実行するとまず，クライアントからのリクエストが来るまで ioctl() 関数でブロックする．リクエストが来たら，データを UID，GID，カレントディレクトリ，そしてコマンド名と引数に分ける．その後パイプを作成し，fork() システムコールにより子プロセスを生成する．親プロセスは子プロセスが終了するまで wait() システムコールを呼んで待機する．子プロセスは UID，GID，カレントディレクトリをセットし，標準出力をパイプに変更する．最後に execvp() 関数を呼び，コマンドを実行し，終了する．子プロセスが終了したことを知った親プロセスは，パイプから結果を読み取り，送信用バッファにコピーする．これを ioctl() 関数で書き込み，次のリクエストに備え，ioctl() 関数を呼び出しブロックする．

4. 実　　験

4.1 実 装 環 境

本システムの実装環境はホスト環境と二つのゲスト環境から構成されている．ホスト環境は，PC DELL Precision 490，CPU Xeon 5130 (2GHz, 4MB L2Cache, 1333MHz FSB)，RAM DDR2-SDRAM 2GB，OS CentOS5.2 （Linux Kernel 2.6.18）である．

x86 マシンゲスト環境は，PC QEMU PC，CPU QEMU32，RAM 128MB，OS FedoraCore10（Linux Kernel 2.6.27.7）ARM マシンゲスト環境は，BASEBORAD Integrator/CP，CPU ARM926EJ-S，RAM 128MB，OS Linux Kernel 2.6.24.7 + BusyBox という構成である．

Vol.2009-ARC-183 No.15
Vol.2009-OS-111 No.15
2009/4/23

4.2 構築結果

本研究で開発した Gryphin は QEMU に対し 3 章で述べた実装を行い，共有メモリとプロセッサ間の接続に関しては C 言語で 76 行の少ないコード追加で実現することができた，また，プロセッサ間割り込みや，共有メモリに対するアトミックな処理に関しては，300 行程度のコードで実装することができた．また，共有メモリへのインターフェイスに関してはデバイスドラバを用いることで，カーネルソースコード自体を直接変更することを防ぎ，デバイスドライバは 512 行，リモートコマンド実行プログラムのクライアントは 93 行，サーバは 134 行で実装できた．

QEMU に対する主な変更は，共有メモリをプロセスのメモリ上にマップし，エミュレートするハードウェアを初期化する段階で，その共有メモリを新しいデバイスとして追加することである．これは，QEMU の組み込み関数を使用しているため，変更による QEMU への影響も少ないと考えられる．

4.3 リモートコマンド実行プログラムによる通信実験

以下の実験はクライアント側に ARM，サーバ側に x86 コアを選択した．

4.3.1 /proc/cpuinfo の表示

リモートコマンド実行プログラムの基本的な通信が行えるかをテストするため，相手側の/proc/cpuinfo を cat する実験を行った．まず，クライアント側は，ローカルマシンの/proc/cpuinfo を cat し CPU が ARM であることを確認した．次に，リモートコマンド実行プログラムを使用し，リモートマシンの/proc/cpuinfo を cat，CPU が x86 であることを確認した．

4.3.2 UID，GID，カレントディレクトリの受け渡し

リモート側のコマンド実効 ID とカレントディレクトリが，クライアント側のもので実行されたか確認するため，相手側に touch コマンドをリクエストし，ファイルを作成する実験を行った．クライアント側は UID，GID ともに 500，カレントディレクトリを/home/aoki とし，リモートコマンド実行プログラムでファイル名を testfile として touch コマンドを実行した．結果，サーバ側の/home/aoki ディレクトリに testfile が UID，GID ともに 500 で生成されたことを確認した．

5. おわりに

本研究では，今後発展が見込まれるヘテロジニアスマルチコアのボトルネックなどの発見や，新しいマルチコア高速化手法の提案を目的に，QEMU をベースにしたヘテロジニアスマルチコアエミュレータ，Gryphin を開発した．Gryphin では，x86，および ARM プロセッサをサポートし，各プロセッサをそれぞれ QEMU プロセスとしてホスト OS 上で動作させ，プロセッサ間を共有メモリで接続することで，異なったプロセッサを仮想的に接続したハードウェア環境をエミュレートすることができた．また，共有メモリをホスト OS のファイルマップで実装することで，容易にプロセス間の共有メモリ空間を実現できた．さらに，共有メモリを用いたアーキテクチャにすることで，QEMU の変更を最小限にし，かつカーネルを変更せずに済んだ．そして，共有メモリへの操作はカーネルモジュール（デバイスドライバ）として実装することで，ゲスト OS 上からの共有メモリへのアクセスも実現できた．共有メモリのアクセスの際にはプロセッサ間割り込みによってブロックしたプロセスをアンブロックされるようにし，また，2 プロセス間のアトミック性を保証したことによって共有メモリへのアクセスを安全に行うことができるように実装できた．これらの共有メモリとデバイスドライバを使用したリモートコマンド実行プログラムを実装し，OS 間でデータ送受信することができることを確認した．

現状では，ヘテロジニアスマルチコア環境のエミュレーションが可能になった段階である．今後は，この仮想ハードウェア上で動作する新しいアプリケーションサービスの考察や，x86，ARM 以外のプロセッサとの接続，多数の異なるプロセッサによるヘテロジニアスマルチコア化などを行っていきたいと考えている．

参考文献

1) Riley, M.W., Warnock, J.D. and Wendel, D.F.: Cell Broadband Engine processor: Design and implementation, *IBM*, Vol.51, No.5, pp.545–557 (2007).
2) Bellard, F.: QEMU, a Fast and Portable DynamicTranslator, *USENIX* (2005).
3) Bochs: site http://bochs.sourceforge.net/. the cross platform IA-32 emulator.