Design and Implementation of McKernel
Version 1.0

Masamichi Takagi, Balazs Gerofi, Tomoki Shirasawa, Gou Nakamura
and Yutaka Ishikawa

平成 27 年 7 月 8 日

目 次

図 目 次

表 目 次

第1章　Introduction

1.1　本資料の目的

本資料の目的は、OS カーネル開発者に McKernel の設計と実装を説明することである。このために、課題、目標値、解決策を説明する。また、モジュールとソースコード、機能とソースコードの対応、必要に応じて関数のインターフェイスを説明する。

1.2　背景と課題

2020 年までに登場するスーパーコンピュータは、レイテンシコアを多数搭載したメニーコア型 CPU を計算ノードとする構成、あるいは、GPGPU のような規則演算を高速実行するアーキテクチャを有する演算加速型 CPU とレイテンシコアを有する CPU のヘテロ構成のアーキテクチャを計算ノードとする構成、の 2 つが考えられている。いずれの場合においても、新たなメモリ階層の導入、インターコネクト回路との統合、がおこなわれ、CPU 群とメモリシステムは NUMA アーキテクチャとなる。NUMA アーキテクチャにおいて、コア群と UMA メモリシステムの対を NUMA ノードと呼ぶことにする。富士通製スーパーコンピュータ FX100 では NUMA ノード数は 2、Intel 製 PC サーバ系では NUMA ノード数が 8〜16 のシステムもある。2020 年以降、3 次元実装によりコア群がスタックされるようになると、NUMA ノード数はさらに増えるであろう (図 1.1)。

このような NUMA 構成からなる計算ノードの理論演算性能は数 TFlops から数十 TFlops となり、計算ノード数 $O(1M-100K)$ でエクサフロップスに達する。計算ノード上の OS カーネルは、NUMA 構成における CPU、メモリ、インターコネクト資源を効率良く管理しアプリケーションプログラムの最適実行を提供する必要がある。将来のユーザ環境を考えると以下の課題あるいは要求に対応する必要がある。各項目のカッコ内のワードは以降の議論で参照する時の課題名称である。

1. 最適メモリ領域割り当て (`NUMA`)

   アプリケーションのマルチスレッド化では、スレッドがアクセスするメモリ領域をそのスレッドが動作する NUMA ノード内のメモリ領域に割り当てることによりデータアクセスを局所化する手法が採られている（スレッドアフィニティ）。同様にカーネルで管理されているユーザアプリケーションのスレッド構造体やページテーブルも NUMA ノード内のメモリ領域に割り当てることによりコアがアクセスするメモリ領域の局所性が上がる。これにより、NUMA ノード間でのデータ転送を減らすことが出来る。

   Linux における mbind システムコールや cgroups の cpusets で設定可能なメモリ領域を以下に示す。

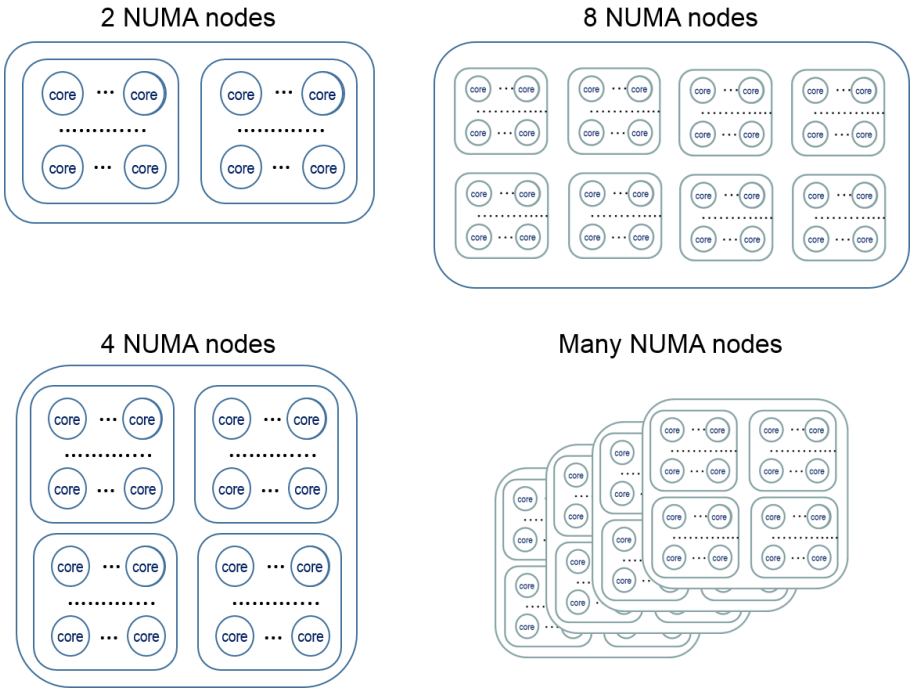


図 1.1: NUMA 構成の可能性

NUMA ノードのコアおよびメモリ量	cpuset.cpus, cpuset.mems
ファイルシステムバッファ領域	cpuset.memory_spread_page
slab メモリ領域	cpuset.memory_spread_slab
仮想メモリ領域制御	mbind
プロセス構造体	-
ページテーブル	-

ここで仮想メモリ領域制御とは、プロセスの仮想メモリ空間内で領域毎にメモリ割り当てポリシを制御することを意味する。プロセスの仮想メモリが使用する物理メモリ領域だけでなく、そのプロセスを管理しているカーネル構造体のメモリ領域の物理メモリ領域も NUMA 局所化できれば、可能な限りの NUMA 局所化が可能となる。

2. キャッシュ汚染 (CACHE)
アプリケーションは各ノードで $O(100)$ のスレッドを同時に実行させて処理を行う。処理の際に参照するデータの多くは個々のコアの持つキャッシュ上に存在する。Linux など従来の OS では、OS サービスのための処理がこのキャッシュ資源をアプリケーションと取り合い、アプリケーション性能低下の原因の一つになる。論文 [6] では、Intel 社の Knights Ferry におけるキャッシュ汚染および性能劣化の実験をしている。現状の Xeon Phi では性能劣化は少なく、本課題は大きな問題とは言えない。しかし、今後、NUMA ノード数の増大に伴いメモリ遅延が増える場合に性能劣化が顕著になる可能性がある。

キャッシュ汚染をなくすためにはアプリケーションが実行しているコア上で、i) システムプロセスなど他のプロセスを実行しない、ii) デバイス割り込み処理をさせない、iii)

アプリケーションからのシステムコール処理をしない、ということでなくすことが出来る。現在の Linux が可能なのは最初の 2 つである。

3. PGAS モデル支援 (`PGAS`)
   PGAS(Partitioned Global Address Space) モデルは並列プログラミング言語環境の一つである。スレッドとスレッド固有のメモリ空間、スレッド間で共有するグローバルメモリ空間から構成される。PGAS モデルを採用しているプログラミング言語は XcalableMP、UPC、X10 などがある。

   Linux など従来の OS で PGAS モデルをプロセス群で実現する場合、グローバルメモリ空間を実現するためにプロセス間で共有するメモリ領域は実行時に割り当てる。グローバルメモリ領域に宣言されている変数群のアクセスは、実行時に割り当てられたメモリ領域へのアクセスとなる。PGAS モデルを有するプログラミング言語を提供するためには、コンパイラはこのようなコードを生成する必要がある。既存コンパイラに修正することなく、PGAS モデルを提供できる OS 支援機能が求められる。

4. メニースレッド支援（`THRD`)
   計算資源を有効利用するためにハードウェアが提供するコア数以上のスレッドを生成する oversubscription により性能向上の研究が行われてきている。スレッド実行切り替えのオーバヘッドを小さくするためにユーザスレッド実装が行われるが、ユーザスレッドだけではシステムコールで待ち状態になると他のスレッドに切り替わらなくなる。カーネルと連携したユーザスレッド実装が必要になるが、現在の Linux ではサポートされていない。

   プロセスに関しても oversubscription による性能向上の研究が行われている。プロセス実行切り替えには TLB のフラッシュが必要でありスレッド実行切り替え以上の時間がかかる。PVAS (Partitioned Virtual Address Space)[4, 5] では、プロセス群を同一仮想アドレス空間で実行しユーザスレッドでプロセス群を実行することによりプロセス実行切り替えにかかる時間を短縮している。PVAS の実現では従来 OS の仮想アドレス空間管理方法を変更しないといけなくなる。

5. プロセスの高速生成 (`FAST`)
   HPC アプリケーションは、入力ファイルは異なるが同じアプリケーションを同じプロセス数で何度も実行することが多い。実行後も同じ計算ノード群を使って再度実行する場合には、プロセスを最初から生成するのではなく計算ノードに残っているメモリイメージならびに通信コネクション情報を再利用できる可能性がある。

   これは、アプリケーション実行時、通信デバイスを初期化した時点の状態をメモリに残しておく（チェックポイント)、あるいは、ローカルストレージに退避しておき、アプリケーション終了時に通信ライブラリおよび通信デバイスの終了処理をせずに、退避してあったデータを用いて再実行する（リスタート）ことにより実現できる。本機能はカーネルおよび実行時環境が協調しないと実現できない。

6. OS ノイズ (`NOISE`)
   大規模計算ノードをを使った並列アプリケーションでは、OS ノイズによって実行性能が低下することがある。OS ノイズとは、デーモンによる定期的なファイル書き出しなど

の処理や割り込み処理などに代表される OS サービスにより CPU 資源が使われる状態を指す。計算ノード毎の OS ノイズのばらつきにより、計算ノード毎にアプリケーションの実行時間が異なる。BSP (Bulk Synchronous Parallel) 型並列アプリケーションにおいては、同期の完了時刻が実行時間の遅いプロセスに律速されアプリケーション実行時間が遅くなる。

アプリケーションを実行するアプリケーション用コア群とデーモン群を実行する OS 実行用コア群を分離し、デバイス割り込みやデーモン群は OS 実行用コアでのみ実行することにより、OS ノイズは減少する。しかし、タイマー割り込みに起因するスケジューラの割り込みだけは残り、OS ノイズがなくなることはない。アプリケーションがアプリケーション用コア以上のスレッドを生成しない、あるいは、アプリケーションがスレッド実行の制御を行うならば、タイマー割り込みを止めることが出来る。

7. Quality of Service (`QOS`)
   大規模な Capacity Computing に伴う大量のデータ処理、実時間で到着する観測データに基づく実時間データ同化処理などが今後のスーパーコンピュータ利用形態として見込まれている。このような利用形態を支援するためには、以下に述べる QOS を提供する必要がある。i) ユーザ毎あるいはプロセスレベルあるいは計算ノードからのファイル I/O 性能を制御する。ii) 一つの計算ノード内に複数のアプリケーションあるいは複数のユーザジョブを同時実行する場合には、アプリケーションあるいはジョブ毎に使用可能コア数やメモリ量を制限する。

8. クラウドサービス機能 (`CLOUD`)
   将来のスーパーコンピュータセンターの運用としてクラウドサービスを提供するということも視野に入れて検討しなければいけない。クラウドサービスには、アプリケーションを提供する SaaS (Software as a Service)、アプリケーションを開発実行する OS 環境を提供する PaaS (Platform as a Service)、仮想計算機を提供する IaaS (Infrastructure as a Service) がある。仮想計算機では、将来のスーパーコンピュータのシステムソフトウェア研究開発に使用できる仮想化技術が提供されている必要がる。

9. Linux 環境の互換性 (`CMPT`)
   現在のスーパーコンピュータ上のアプリケーションの大多数は Linux 上で開発されている。上記課題を解決するために新しいカーネルを開発する場合には、Linux 環境の互換性が必要である。

   Linux 環境の互換性においては、Linux が提供するシステムコール群を実装するだけでは不十分である。Fortran, C, C++などのプログラミング言語で記述されているアプリケーションが使用している Linux システムコール群は限られているが、スクリプト言語系、デバッガなどのツール群は Linux システムコールだけでなく、Linux が提供している/proc ファイルシステム上のファイルをアクセスしているために、これらファイルシステム上のディレクトリやファイル群も互換性がないといけない。

10. カーネルの保守性 (`MAINT`)
    バグのない安全なカーネルを維持していくためには、論理的なバグだけでなく、資源排他制御、実行コンテキスト依存処理が正しく実装されていることを自動的に検証していける仕組みが必要である。

1.3 実現手法候補の比較

第 1.2 節で述べた課題を解決するには大きく 3 つのアプローチがある：Linux を改変せずにカーネルモジュールで実現 (この手法を Linux + Module と呼ぶ)、Linux を改変 (この手法を Modified Linux と呼ぶ)、Linux と協調動作する軽量カーネル (軽量カーネルを Light-Weight Kernel (LWK) と呼び、この手法を Linux + LWK と呼ぶ) を開発する。3 つのアプローチに対して、以降、それぞれの課題の実現可能性について議論する。表 1.1 は議論のまとめである。

表 1.1: 3 アプローチのまとめ

	NUMA	CACHE	PGAS	THRD	FAST	NOISE
Linux + Module						
Modified Linux	?	?	?	?	?	?
Linux + LWK	✓	✓	✓	✓	✓	✓

	QOS	CLOUD	CMPT	MAINT
Linux + Module	✓	✓	✓	✓
Modified Linux	✓	✓	✓	?
Linux + LWK	✓	✓	-	✓

1.3.1 Linux + Module

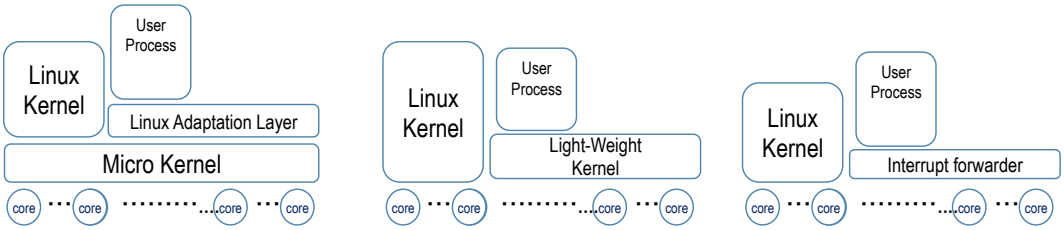
Linux を改変せずにカーネルモジュールで実現するアプローチである。現在の Linux では課題 NUMA、CACHE で述べた仕様はカーネルモジュールレベルで解決できるものではない。課題 FAST は Linux カーネル向けに BLCR のようなチェックポイント機能を追加しないと実現できない。課題 NOISE である OS ノイズはシステムデーモン群およびハードウェア割り込みを OS 用コアにバインドすることにより軽減できる。しかし、タイマー割り込みを止めることは出来ないので、OS ノイズをゼロにすることは出来ない。

1.3.2 Modified Linux

全ての課題を解決するために Linux を最適化、拡張していくアプローチである。以下の問題が生じる。

1. 改変箇所の大きさ
   全ての課題を解決するためには、Linux に対する改変箇所が増大する。アーキテクチャ依存部に留まるのであれば、Linux のバージョンに追随するのは容易であるが、そうでないと、メインストリームに改変が反映されないと Linux のバージョンに追随するコストは膨大になる。

2. Linux の実行セマンティックス維持
   Linux アプリケーションが想定している Linux の実行環境を 100 % 維持して、将来のアーキテクチャに対する最適環境を提供するための改変が可能かどうかを判断することは難しい。Linux 改変コストを考えないで進めると、結果として、Linux のシステム



a) Traditional Micro Kernel Approach

b) Linux + User Process on Light-Weight Kernel

c) Linux + User Process on interrupt forwarder

図 1.2: マイクロカーネル、軽量カーネル、Linux を組み合わせた構成例

コール処理部分、デバイスドライバ部分を除いて、全てのモジュールを変更することにもなりかねないだろう。

1.3.3 Linux Kernel + LWK

軽量カーネルと Linux カーネルの 2 つのカーネルを用意するというアプローチでは、アプリケーションプログラムは軽量カーネル上で実行し、Linux カーネルを実行するコアを別途容易する。アプリケーションが Linux システムコールを発行すると、その処理は Linux カーネル側に委譲する。これにより軽量カーネル上のアプリケーションは Linux の API をアプリケーションに提供できる。本アプローチの利点は次のとおりである。

1. 軽量カーネルは Linux から独立して設計実装できるため、課題解決のための新しい実装を自由に出来る。
2. 軽量カーネル側の OS サービスを最小限とすることによりカーネルソースコードが小さくなる。課題 `MAINT` に対しても、形式的仕様記述言語を導入してカーネルソースコードに直接仕様を記述することにより可読性を上げるとともに将来の自動検証に道を開くことができる。

本アプローチの欠点は、ユーザプロセスを Linux カーネルか軽量カーネルのどちらが管理するのか、ユーザプロセスが発行する Linux システムコールおよびユーザプロセスがアクセスできる/proc ファイルシステムをどのように処理するかに依存する。

1.3.3.1 ユーザプロセス管理

マイクロカーネルまたは軽量カーネルと Linux を組み合わせる構成によるユーザプロセス管理方法の違いを図 1.2 を基に以下説明する。

- a) はマイクロカーネル上に Linux およびユーザプロセス実行環境を実装する方式である。1990 年代の OSF/1 MK-AD が本構成にあたる。OSF/1 MK-AD では Mach マイクロカーネルの上に Unix サーバを実装し、マイクロカーネル上でアプリケーションを実行する。日立製 SR2201 スーパーコンピュータの OS である HI-UX/MPP は OSF/1MK-AD

を採用した OS である。HI-UX/MPP では、計算ノード上でマイクロカーネル、I/O ノード上で UNIX サーバが動作し、OS 機能は計算ノード上で動作する。L4Linux[2] も同様のアプローチであり、L4 マイクロカーネルの上に Linux カーネルが実装される。マイクロカーネルを使った仮想計算機と捉えることも出来る。

本アプローチでは、マイクロカーネルがメモリやデバイスなどの計算機資源を管理し、Linux などのフル OS 機能を提供するカーネル（フル OS カーネル）はユーザプロセスとして実現される。第 1.2 節で述べた課題群のうち CMPT を除き解決することが出来る一方、フル OS カーネルの実行はネイティブ実行に比べてオーバヘッドを生じる。アプリケーションプログラムはマイクロカーネル上で動作し、フル OS サービスを利用するためにフル OS カーネルとプロセス間交信する。Linux 環境の互換性のために、全ての Linux システムコールをプロセス間交信に変換するコストが生じる。また、/proc ファイルシステムをアプリケーションプログラムからアクセスできるようにしなければならないが、/proc にはプロセス・スレッド情報が含まれるため、マイクロカーネル側でも Linux の/proc と同様のインターフェイスを提供する必要がある。

信頼できないフル OS カーネルあるいは資源管理を制御できないフル OS カーネルであれば、このようなアプローチを採用する意義がある。フル OS カーネルが信頼でき、また、Linux2.6.24 以降に取り込まれた cgroups のようにプロセス群の資源管理が可能であるようなフル OS では、敢えてマイクロカーネル上でフル OS を動作させる意義はない。

- b) は Linux カーネルが動作するコアとアプリケーション用軽量カーネルが動作するコア群に分離する実装方式である。アプリケーションコア上ではアプリケーション実行に必要な最低限のカーネル機能（プロセス・スレッド生成・消滅など）を実装し、それ以外のシステムコールは Linux カーネル側で実現する。このアプローチは、アプリケーション用コアをフル OS カーネル用コアと分離し、アプリケーション用コア上では独自の軽量マイクロカーネルを実現するものであり、第 1.2 節で述べた課題のうち、課題 CMPT を除く課題に対応できる。課題 CMPT に関しては次節で議論する。

- c) はアプリケーション用コア上で b) のように軽量マイクロカーネルではなく、割り込みを受けてフル OS に処理を委譲する機能だけを実現する最低限の特権モードで動くプログラムを常駐させるアプローチである。アプリケーションプロセスの生成消滅も含めてフル OS 側が行う。課題 NOISE である OS ノイズを削減できるが、その他の課題を解決するためにはフル OS を修正する必要がある。

このように 3 つの方式の中では、Linux と LWK によるプロセス管理が一番良い実現方式といえる。

1.3.3.2 Linux 環境の互換性実現

課題 CMPT で述べた通り、Linux 環境の互換性のためには Linux と同じシステムコールの実現と/proc ファイルシステムの提供がある。Linux システムコールに関しては、どのシステムコールを軽量カーネルで実現し、どのシステムコールを Linux 側に委譲するかは以下のポリシとするのが良い。

1. 課題 NUMA を解決するために、プロセスが必要とするメモリ領域管理は軽量カーネル側で実現すべきである。すなわち物理メモリ管理、仮想メモリ管理関係のシステムコール（mmap,brk）、プロセス生成・消滅（シグナル処理含む）は軽量カーネル側で提供すべきである。

2. アプリケーション実行性能に影響する機能は軽量カーネル側で実現すべきである。

/proc ファイルシステム下のファイルは Linux カーネルのバージョンが変わると変更されることがあることを想定しておく必要がある。このためには、Linux カーネルのソースコードから自動的に/proc ファイルシステム下のファイルの変更が抽出できるツールを開発すべきである。

第2章　Design Overview

この章では McKernel の構成と機能の概要を説明する。

2.1　構成と機能

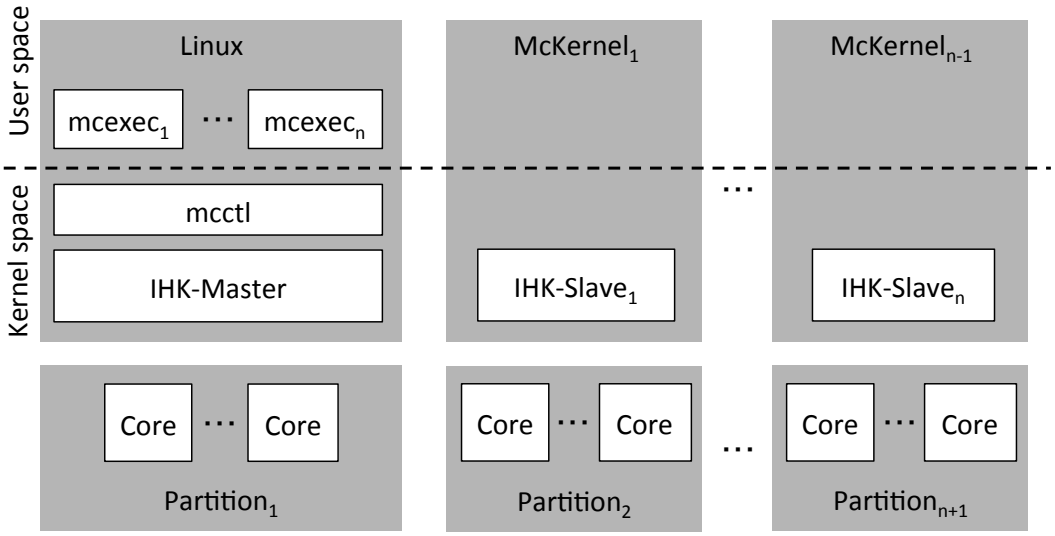


図 2.1: McKernel の構成

McKernel は東京大学で開発された軽量カーネルが基になっている [7]。McKernel では、ノード資源を 2 つのパーティションに分割し、片方で Linux カーネルを動作させ、もう片方で McKernel を動作させる（**図 2.1**）。McKernel 上で稼働するアプリケーションに対して Linux API を提供するために、McKernel が提供しない Linux API は、Linux カーネルに処理を移譲している。Linux 側には、コア群を管理する機能、McKernel を起動する機能、Linux と McKernel 間の通信機能を提供するカーネルドライバが実現されている。これを IHK (Interface for Heterogeneous Kernel) と呼んでいる。IHK は、McKernel だけでなく、他のカーネルの実装にも使えるように McKernel コードと独立した実装としている。現在、McKernel はマルチコアアーキテクチャである Intel 製 Xeon およびメニーコアアーキテクチャである Intel 製 Xeon Phi 上で開発を行っている。

ソフトウェアの構成を**図 2.1** を用いて説明する。McKernel は Linux、Linux のカーネルモジュールとして動作する `mcctl`、n 個 $(n \geq 1)$ の Linux のユーザプロセスとして動作する `mcexec` ($\mathrm{mcexec}_1, \cdots, \mathrm{mcexec}_{n-1}$)、Linux のカーネルモジュールとして動作する Interface for Heterogeneous Kernel Master (IHK-Master)、n 個の McKernel ($\mathrm{McKernel}_1, \cdots, \mathrm{McKernel}_{n-1}$)、McKernel のライブラリとして存在する n 個の IHK-Slave ($\mathrm{IHK\text{-}Slave}_1, \cdots, \mathrm{IHK\text{-}Slave}_{n-1}$) か

らなる。また、Linux と McKernel はそれぞれ割り当てられたコア資源とメモリ資源からなるパーティションで実行される。これら構成要素の機能の概要を、OS 起動、プロセス管理、メモリ管理、システムコール、`procfs/sysfs` に分けて説明する。

2.1.1 McKernel 起動

まず前提となる、McKernel の利用イメージを図 2.2 を用いて説明する。Linux カーネルは計算ノード上で常駐し、アプリケーションが必要とするカーネル機能を有する McKernel が稼働する。例えば、App A はコア以上のスレッドを生成しないので、スケジューラなしの McKernel を使う。これにより OS ノイズのない環境上で App A が動作する。また、App B はプロセスをコア数以上に生成するので、スケジューラをもつ McKernel を使う。

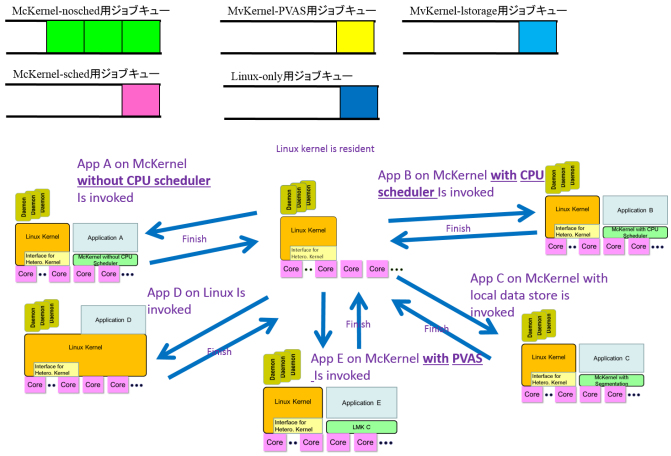


図 2.2: McKernel 利用イメージ

次に、ノード起動から McKernel 起動までのステップを説明する。まず Linux が起動シーケンスを BIOS から引き継ぎノードを起動する。次に IHK-Master が Linux と McKernel のためのパーティションを作成し、また McKernel を起動する。起動中とアプリ実行中に、IHK ー Master と IHK-Slave は連携して、Linux と McKernel にカーネル間通信機能、Inter-Kernel Communication (IKC) を提供する。`mcctl` は McKernel の起動停止を Linux 上から行えるようにする。

2.1.2 プロセス管理

McKernel 上のプロセスは、`mcexec` によって起動される。`mcexec` は IKC を用いて McKernel にプロセス起動を指示する。また、`mcexec` は、McKernel 上のプロセスのシステムコールのうち一部を代理実行するために、McKernel 上のプロセスと仮想アドレス空間を共有する。これは、以下のステップで実現される（図 2.3）。

1. `mcexec` は、stack、text、data、bss、mmap 各領域が上位アドレス側にかたまって配置されるように作成されており、アドレスゼロから始まり、McKernel 上のプロセスの仮想アドレス領域全てを収めるのに十分なサイズを持った仮想アドレスレンジを空けてあ

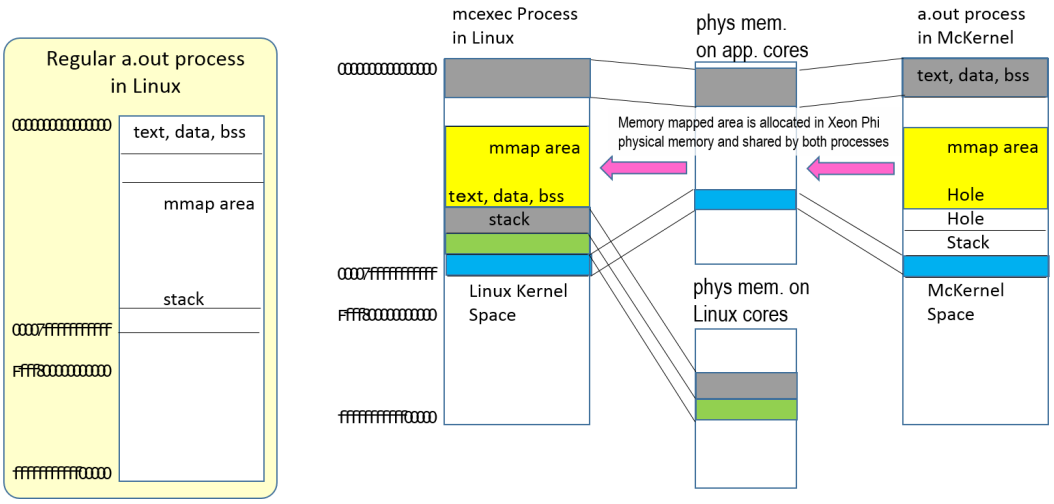


図 2.3: mcexec と McKernel 上のプロセスの仮想アドレス空間の共有

る。McKernel 上のプロセスの仮想アドレス領域は、この空いている仮想アドレスレンジに収まるように設定される。

2. mcexec は、McKernel 上のプロセスにおけるページフォールトを捕捉するため、mcctrl を用い、擬似ファイルを作成し、そのファイルを上記仮想アドレスレンジを指定して mmap() する。

3. McKernel は、上記仮想アドレスレンジ内に McKernel 上のプロセスの仮想アドレス領域を割り当てる。

4. mcexec のページテーブルエントリと McKernel 上のプロセスのページテーブルエントリとの同期を、mcexec がそのエントリを必要とした際に行う。すなわち、mcctrl が、mcexec が起こした、McKernel 上のプロセスの仮想アドレス領域でのページフォールトを捕捉し、McKernel 上のプロセスのページテーブルを参照し、これを mcexec のページテーブルにコピーする。

また、mcexec は McKernel 上のプロセス ID といったプロセスの状態を Linux 上で管理する。プロセス ID については以下のステップで管理する。

1. mcexec が初めて McKernel 上にプロセスを起動する際には mcexec のプロセス ID を McKernel に渡しこれを McKernel に使用させる。

2. McKernel 上のプロセスが子プロセスを生成する際には mcexec においても子プロセスを生成させ、そのプロセス ID を McKernel に通知しこれを McKernel に使用させる。

2.1.3 メモリ管理

McKernel は McKernel 上のプロセスのメモリ管理を行う他に、プロセス管理を説明する際に説明した方法で、`mcexec` に、McKernel 上のプロセスと同じ仮想アドレス空間を持たせる。こうすることで、McKernel 上のプロセスがシステムコールを Linux に代理実行させる際に、`mcexec` が McKernel 上のプロセスのメモリ領域を、ページテーブルの入れ替えやメモリコピーの必要なく参照できる。

2.1.4 システムコール

McKernel はプロセスに対し Linux API 互換のシステムコール機能を提供する。システムコールは以下のステップで実行される。

1. McKernel 上で動作するプロセスがシステムコールを発行する。
2. McKernel は上記システムコールを McKernel 内で実行するか、Linux で実行するか判断する。McKernel 内で実行する場合はシステムコールを実行しプロセスに制御を戻す。Linux で実行する場合には IKC を用いて `mcexec` と通信を行い、システムコール実行指示を出す。
3. `mcexec` はシステムコール実行指示を受け、システムコールを実行し、結果を IKC を用いて McKernel に返す。

2.1.5 `procfs/sysfs`

McKernel はプロセスへ Linux API 互換の `procfs/sysfs` 機能を提供する。`procfs/sysfs` は以下のステップで実現される。

- `procfs/sysfs` のエントリのうち、Linux のエントリを流用できず、McKernel がアクセス要求に対応する必要があるものについては、`mcctrl` が Linux の `procfs/sysfs` インターフェイスを用いてファイルを作成しておく。
- McKernel 上のプロセスの `procfs/sysfs` のエントリへのアクセスをシステムコールをフックすることで捕捉する。`mcexec` において該当エントリへのアクセス要求を Linux が対応するか McKernel が対応するか判断し、Linux が対応する場合は Linux の提供するファイルをアクセスさせる。McKernel が対応する場合はパスを変更してシステムコールの実行を続けることにより、McKernel が提供するファイルをアクセスさせる。

2.2 ソースコードとの対応関係

IHK-master, IHK-slave とソースコードとの対応関係を**図 2.4** に示す。プロセッサアーキテクチャに依存する実装についてはアーキテクチャ非依存部分を抽出し、アーキテクチャ非依存部、アーキテクチャ依存部に分けて実装し、ソースコードもアーキテクチャ非依存部、アーキテクチャ依存部に分けている。

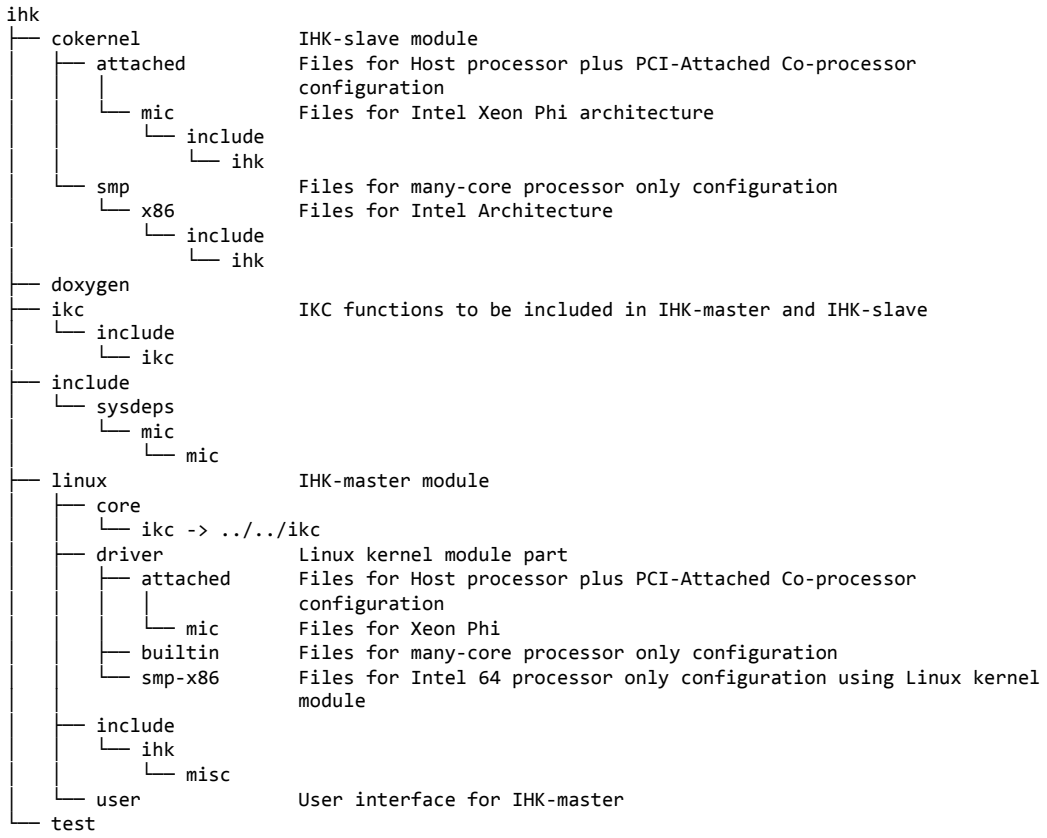


図 2.4: IHK-master, IHK-slave とソースコードとの対応関係

また、McKernel を動作させるノードの構成としては、プロセッサの 1 パーティションで Linux、他のパーティションで McKernel を動作させる構成（builtin 構成と呼ぶ）、プロセッサで Linux を動作させ PCI バスなどの I/O バスに接続されたデバイスで McKernel を動作させる構成（attached 構成と呼ぶ）の 2 つをサポートする。これに対応してソースコードを builtin と attached という名のディレクトリで分けている。

`mcctl`、`mcexec`、McKernel とソースコードとの対応関係を**図 2.5** に示す。

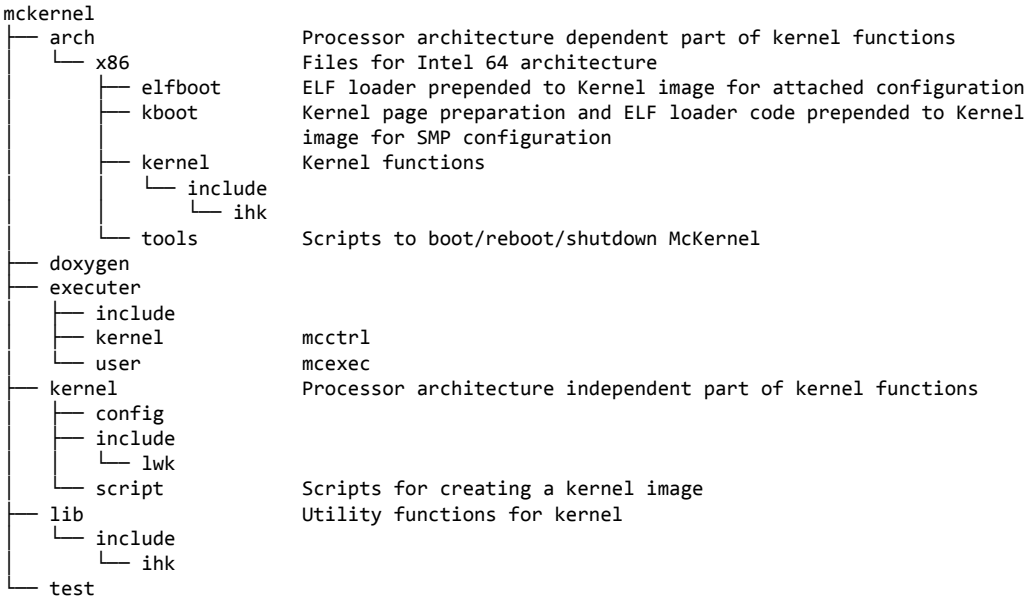


図 2.5: mcctl、mcexec、McKernel とソースコードとの対応関係

第3章　Interface for Heterogeneous Kernels (IHK)

McKernel は、並列アプリケーションのスケーラビリティを確保するために、ノード資源をパーティショニングして一つのパーティションで Linux を動作させ、他のパーティションで軽量カーネルを動作させる手法を取っている [7]。ここで、この手法で共通に必要となる、最初にブートしたカーネルから他のカーネルを起動する機能や複数カーネルが互いに通信する機能を、Interface for Heterogeneous Kernel (IHK) と呼ぶ、フレームワークとして機能するモジュール群として実装している。この節では、この IHK の機能を説明する。

3.1　Introduction

Interface for Heterogeneous Kernels (IHK) is a low-level software infrastructure, which enables partitioning node resources and the management of lightweight kernels on subsets of the resources. This section introduces the basic architecture of IHK and gives a brief overview of its main components. An overview of the IHK architecture is shown in Figure 3.1.

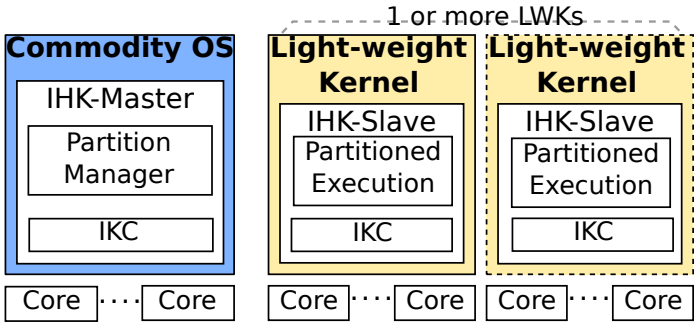


図 3.1: **Architectural overview of IHK components.**

IHK categorizes kernels in two types: a master kernel and the slave kernels (i.e., lightweight kernels). The master kernel is a kernel that is booted in the node first through the normal booting process, for example, booted from BIOS or UEFI, and is typically a commodity operating system, it is Linux in the rest of this document. Slave kernels are kernels that are booted from the master kernel. IHK's components in the master and slave kernels are called IHK-master and IHK-slave, respectively.

Resource partitioning, the management and bootstrapping of lightweight slave kernels are implemented in IHK-master, while support for executing over a partition of resources

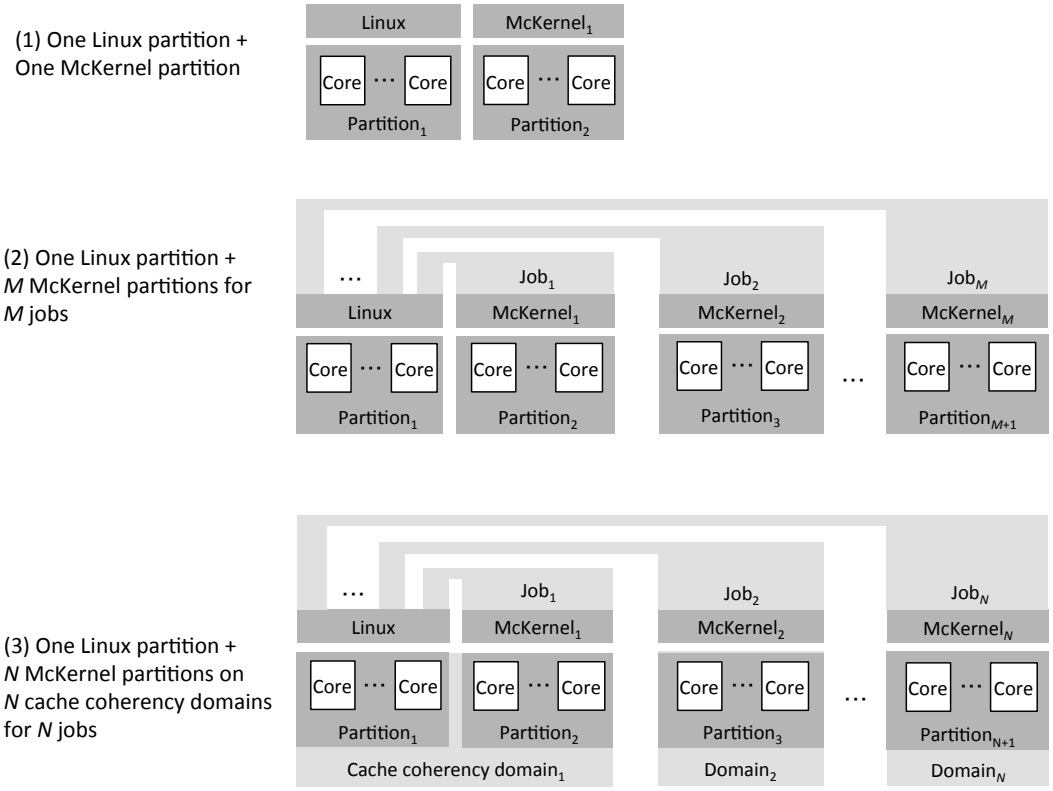


図 3.2: Supported configurations of dividing resources into partitions and binding them to Linux and McKernel instances.

is implemented in IHK-slave. A low-level communication facility called IHK-IKC is present both in IHK-master and IHK-Slave.

3.1.1 パーティションの構成

IHK は、以下の 3 つのパーティション構成をサポートする。

第 1 の構成は、キャッシュコヒーレンシドメインが 1 つで、1 ノードで 1 ジョブが動作する構成である。この例を **Fig. 3.2(1)** に示す。この例では、ノードを 2 つのパーティションに分け、Linux と McKernel を動作させる。

第 2 の構成は、キャッシュコヒーレンシドメインが 1 つで、1 ノードで複数ジョブが動作する構成である。この例を **Fig. 3.2(2)** に示す。この例では、ノードをジョブの個数プラス 1 個のパーティションに分け、最初のパーティションで Linux を動作させ、残りのパーティションではそれぞれ 1 つの McKernel インスタンスを動作させる。McKernel インスタンスはそれぞれ 1 つのジョブを動作させる。1 つの Linux がすべての McKernel インスタンスに対してシステムコールなどの OS サービスを提供する。

第 3 の構成は、キャッシュコヒーレンシドメインが複数で、1 ノードで複数ジョブが動作する構成である。この例を **Fig. 3.2(3)** に示す。この例では、最初のコヒーレンシドメインを 2 パーティションに分割し、片方で Linux を動作させ、もう片方で McKernel インスタンスを動作させる。残りのコヒーレンシドメインについてはコヒーレンシドメインごとにパー

ティションを作成し、それぞれ一つの McKernel インスタンスを動作させる。McKernel インスタンスはそれぞれ 1 つのジョブを動作させる。 1 つの Linux がすべての McKernel インスタンスに対してシステムコールなどの OS サービスを提供する。

3.1.2 Quick Usage Example

This section shows a quick usage example of IHK's resource partitioning functionality and demonstrates the interaction with LWK instances in case of using the IHK-SMP x86 driver module. The IHK resource management modules are intended to be loaded at boot time. There are two modules required during initialization, `ihk.ko` is the core IHK driver which provides the basic management interfaces through device files (discussed below). While `ihk_smp_x86.ko` is the actual node resource partitioner.

```
$ insmod kmod/ihk.ko
$ insmod kmod/ihk_smp_x86.ko
```

The above command line instructions load the master IHK core infrastructure module followed by the SMP partitioner. The manycore device representing resources on an SMP node can be then manipulated via ioctl() calls to the device file `/dev/mcd0`, for which the `ihkconfig` command line tool is provided. The modules do not reserve any resources at load time (except for a couple of small datastructures).

IHK-SMP x86 uses the Linux kernel's hotplug system to detach CPU cores from Linux. In order to reserve CPU cores for the IHK framework, one needs to issue the following command:

```
$ ihkconfig 0 reserve cpu=2-4,10
```

The example above reserves CPUs with logical IDs 2,3,4 and 10. For detailed explanation on the CPU ID format see Section 3.2.1. Note that CPU 0 is not permitted to be reserved for IHK and it always remains for use by Linux.

Similarly, memory can be reserved for the IHK framework using the following command:

```
$ ihkconfig 0 reserve mem=2048M
```

The number specified indicates the number of bytes, but it can be simplified with the addition of standard metric prefixes (i.e., `k`, `M`, `G`, etc..). The above example reserves 2GBs of physical memory. For simplicity, we have not specified NUMA related information here, see Section 3.2.1.1 for NUMA specific memory allocation. In summary, at this point IHK holds 4 CPU cores and 2 GBs of memory.

In order to actually load and boot a kernel image first an OS instance needs to be created executing the following command:

```
$ ihkconfig 0 create
```

After this command executed, a new device file (e.g., `/dev/mcos0`) is created that represents the new OS. The `ihkosctl` tool enables us to configure a particular OS device file, such as to assign node resources to it, upload a kernel image, or to boot the specified OS kernel, etc:

```
$ ihkosctl 0 alloc cpu=2,3
$ ihkosctl 0 alloc mem=512M
```

For instance, the example above allocates CPU cores with ID 2 and 3 and a physically contiguous memory region of 512MB to the OS device file with index 0. Specification for CPU cores and memory follow the same format as used for IHK reservations.

```
$ ihkosctl 0 load lightweight-kernel.img
```

Using `ihkosctl`, the above instruction loads the specified kernel image. An IHK compatible kernel image is a standard ELF binary linked against the IHK slave provided library so that it can interact with other components in the system. The method of building kernel images will be discussed later. Finally, the kernel can be booted with the following command:

```
$ ihkosctl 0 boot
```

`ihkosctl` provides a wide range of other functionalities, for instance to display the kernel message log of the corresponding OS device one could invoke:

```
$ ihkosctl 0 kmsg
```

As it has been shown above, using `ihkconfig` and `ihkosctl` one can create OS instances and assign resources to them. In case of multiple lightweight kernel instances, resources need to be assigned for each instance separately:

```
$ ihkconfig 0 create
$ ihkosctl 0 alloc cpu=2,3
$ ihkosctl 0 alloc mem=512M
$ ihkconfig 0 create
$ ihkosctl 1 alloc cpu=4,10
$ ihkosctl 1 alloc mem=512M
```

Following our example for the previous section, the above example shows how to create two OS instances and assign part of the resources to each. As seen, the second OS instance can be referred to as index `1` in the `ihkosctl` command.

In continuation to the example related to multiple kernels above, the following commands show how one could specify different kernel images for two separate partitions:

```
$ ihkosctl 0 load lightweight-kernel.img
$ ihkosctl 1 load lightweight-kernel-with-NVM-support.img
```

Specifically, this example loads a kernel image to the partition denoted by index `1`, which has support for non-volatile memories.

3.2 LWK Management in Linux

This section discusses the functionalities and components of IHK-master and details its interface. The resource partitioning mechanism provided by the implementation in the Linux kernel is also explained.

IHK-master consists of two types of modules. *IHK-master core* provides the basic IHK framework and management infrastructure. It is required for registering/removing the so called *IHK-master drivers* (discussed below) and provides administration interface through device files and ioctl() APIs for:

- Managing devices.
- Managing OS kernel instances.

In particular, the IHK-master core module enables in-kernel interfaces (by means of exporting a set of IHK specific Linux kernel functions) which allow registration and de-registration of IHK-master drivers.

IHK-master drivers represent resources, such as CPU cores of an SMP chip along with the physical memory of the given node or PCI-Express attached co-processors. Specifically, the current IHK implementation in Linux provides two types of IHK-master drivers:

- *IHK-attached:* Enables configuration and interaction with a co-processor attached through PCI Express, at this moment only the Intel Xeon Phi is supported.
- *IHK-SMP x86:* Represents a virtual device that enables partitioning CPU cores of an x86 (Xeon) SMP chip as well as the physical memory attached to the node among OS instances.

Note, that neither the IHK-master core module, nor the IHK-attached/SMP x86 drivers require any modifications to the Linux kernel.

IHK-master drivers support the abstraction of *IHK devices*, which essentially represent resources. On top of IHK devices one can create *IHK OS instances* and use the framework to assign a set of the underlying resources to the particular OS instance. As we mentioned earlier, IHK exposes its management interface via device files which in turn can be controlled with specific command line tools. To demonstrate the actual execution steps of an IHK device registration and the creation of an OS instance consider Figure 3.3.

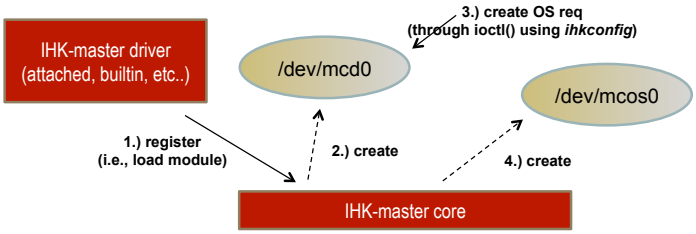


図 3.3: **Execution steps of IHK-master driver registration and device file creation.**

The initial state of the figure is right after the core IHK modules has been loaded. The following four steps are then highlighted:

1. Load an IHK-master driver module, which automatically registers itself into the IHK framework.
2. The IHK framework creates the `/dev/mcd0` device file, which will represent the resources accessible through the inserted IHK master driver.
3. Use the `ihkconfig` command line tool (discussed in detail below) to request creation of an OS instance, which in turn does an ioctl() call on the specified IHK device.
4. The IHK framework creates `/dev/mcos0` device file, which represents an OS instance on top of IHK device `/dev/mcd0`.

Note the index `0` in the file names of `/dev/mcd0` and `/dev/mcos0`. The IHK framework allows registration of multiple IHK-master drivers as well as the creation of multiple OS instances over a specific IHK device. The index of the corresponding device file is assigned by the framework automatically.

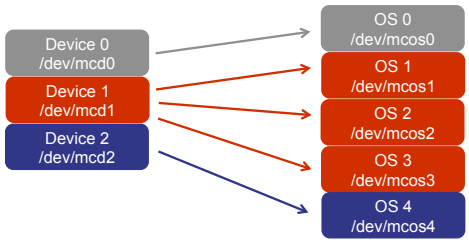


図 3.4: **Relation between IHK devices and OS instances.**

To emphasize the relation between OS instances and IHK devices see Figure 3.4. As shown, `/dev/mcd1` has multiple OS instances on top of it.

3.2.1 Command Line Tools for Resource Partitioning and LWK Management

This section discusses the two main command line tools provided by the IHK framework. Both of these tools interact with specific device files via ioctl() calls, but since users are advised (although not restricted) to use these tools instead of the ioctl() interface directly, we discuss the usage of the tools themselves.

3.2.1.1 `ihkconfig`: Resource Partitioning Tool

`ihkconfig` is responsible of providing a simple interface for interacting with IHK device files, i.e., those named as `/dev/mcdX`. Note that although the in-kernel IHK device structure has various operations, such as initialization/un-initialization and specific memory mapping APIs used for mapping remote memory of a device to the Linux kernel, there are only a few operations exposed to the user-level ihkconfig. These operations are the following:

3.2.1.1.1 Reserve CPU cores

This operation enables reservation of specific CPU cores for the IHK framework and it has the following format:

`ihkconfig <dev index> reserve cpu=<CPU id list>`, where `<dev index>` identifies the IHK device file that appears as the result of the insertion of the IHK-master driver module, and `<CPU id list>` is the following format: `<CPU logical id>`,...,`<CPU logical id>` or `<CPU logical id>` - `<CPU logical id>` (must be a positive range in ascending order) or a mixture of the two: `<CPU logical id>`,...,`<CPU logical id>` - `<CPU logical id>`. CPU logical ID begins at 0 and the maximum value is "number of CPUs in system - 1". An actual example of usage would be:

```
$ ihkconfig 0 reserve cpu=24-31
```

The reserve operation may be executed multiple times adding CPU logical ID cores as required.

3.2.1.1.2 Query CPU cores

This operation enables querying which CPU cores the IHK framework has reserved and it has the following format:

`ihkconfig <dev index> query cpu`, where `<dev index>` identifies the IHK device file that appears as the result of the insertion of the IHK-master driver module.

The command returns the list of CPUs in the same format as the above reservation code.

3.2.1.1.3 Release CPU cores

This operation releases the specific CPU cores from the IHK framework and it has the following format:

`ihkconfig <dev index> release cpu=<CPU id list>`, where `<dev index>` identifies the IHK device file that appears as the result of the insertion of the IHK-master driver module, and `<CPU id list>` is the following format: `<CPU logical id>`,...,`<CPU logical id>` or `<CPU logical id>` - `<CPU logical id>` (must be a positive range in ascending order) or a mixture of the two: `<CPU logical id>`,...,`<CPU logical id>` - `<CPU logical id>`. CPU logical ID begins at 0 and the maximum value is "number of CPUs in system - 1". An actual example of usage would be:

```
$ ihkconfig 0 release cpu=24-31
```

The release operation may be executed multiple times removing CPU logical ID cores from IHK as required.

3.2.1.1.4 Reserve Memory

This operation enables reservation of memory for the IHK framework and it has the following format:

`ihkconfig <dev index> reserve mem=<amount of memory>`, where `<dev index>` identifies the IHK device file that appears as the result of the insertion of the IHK-master driver module, and `<amount of memory>` is given in the following format: `X[M|G|T][@Y]`, where `X` is a decimal number denoting the number of bytes requested, unless one of the standard metric prefixes is attached (i.e., `M` as Mega, `G` as Giga, or `T` as Terra), in which case it stands for the specified metric. Moreover, the optional `@` symbol that can be followed by a decimal number denotes the targeted NUMA node, where the default NUMA node is 0. An actual example of usage of allocating 2 Gigabytes from NUMA node 1 would be:

```
$ ihkconfig 0 reserve mem=2G@1
```

The reserve operation may be executed multiple times adding physical memory as required. The operation may fail in case the system wide available memory is less than the amount requested.

3.2.1.1.5 Query Memory

This operation enables querying which CPU cores the IHK framework has reserved and it has the following format:

`ihkconfig <dev index> query mem`, where `<dev index>` identifies the IHK device file that appears as the result of the insertion of the IHK-master driver module.

The command returns the list of memory regions in the same format as the above reservation code.

3.2.1.1.6 Release Memory

This operation releases memory from the IHK framework and it has the following format:

`ihkconfig <dev index> release mem=<amount of memory>`, where `<dev index>` identifies the IHK device file that appears as the result of the insertion of the IHK-master driver module, and `<amount of memory>` is given in the following format: `X[M|G|T][@Y]`, where `X` is a decimal number denoting the number of bytes, unless one of the standard metric prefixes is attached (i.e., `M` as Mega, `G` as Giga, or `T` as Terra), in which case it stands for the specified metric. Moreover, the optional `@` symbol that can be followed by a decimal number denotes the targeted NUMA node, where the default NUMA node is 0. An actual example of usage would be:

```
$ ihkconfig 0 release mem=1G@1
```

The release operation may be executed multiple times freeing physical memory as required. The operation may fail in case the IHK reserved memory is less than the amount requested.

3.2.1.1.7 Create OS instance

This operation enables the creation of an OS instance over the specific IHK device and it has the following format:

`ihkconfig <dev index> create`, where `<dev index>` identifies the IHK device file that appears as the result of the insertion of the IHK-master driver module. An actual example of usage would be:

```
$ ihkconfig 0 create
```

Unless an error occurs, the command returns an index `X` which will denote the specific OS device file with path name of `/dev/mcosX`.

3.2.1.1.8 Destroy OS instance

This operation enables the destruction of an OS instance over the specific IHK device and it has the following format:

`ihkconfig <dev index> destroy <os index>`, where `<dev index>` identifies the IHK device file that appears as the result of the insertion of the IHK-master driver module and `<os index>` identifies the OS index that has been returned after the OS creation operation. An actual example of usage would be:

```
$ ihkconfig 0 destroy 2
```

TODO: explain when failure may occur.

3.2.1.2 `ihkosctl`: LWK Management Tool

`ihkosctl` is responsible of providing a simple interface for interacting with IHK OS instance device files, i.e., those named as `/dev/mcosX`.

3.2.1.2.1 Assign CPU cores

This operation enables the assignment of CPU cores to an OS instance and it has the following format:

`ihkosctl <os index> assign cpu=<CPU id list>`, where `<os index>` identifies the OS index that has been returned by the OS creation operation, and `<CPU id list>` is the following format: `<CPU logical id>`,...,`<CPU logical id>` or `<CPU logical id>` - `<CPU logical id>` (must be a positive range in ascending order) or a mixture of the two: `<CPU logical id>`,...,`<CPU logical id>` - `<CPU logical id>`. CPU logical ID begins at 0 and the maximum value is "number of CPUs in system - 1". Note that only CPU logical IDs which have been reserved for the IHK framework are available. An actual example of usage would be:

```
$ ihkosctl 0 assign cpu=2-8
```

In which example, CPU cores 2,..,8 are assigned to OS instance 0.

3.2.1.2.2 Assign Memory

This operation enables the allocation of physical memory to an OS instance and it has the following format:

`ihkosctl <os index> assign mem=<amount of memory>`, where `<os index>` identifies the OS index that has been returned by the OS creation operation, the IHK OS instance's index that has been returned as the result of the creation operation, and `<amount of memory>` is given in the following format: `X[M|G|T][@Y]`, where `X` is a decimal number denoting the number of bytes requested, unless one of the standard metric prefixes is attached (i.e., `M` as Mega, `G` as Giga, or `T` as Terra), in which case it stands for the specified metric. Moreover, the optional `@` symbol that can be followed by a decimal number denotes the targeted NUMA node, where the default NUMA node is 0. Note that only memory which have been reserved for the IHK framework is available. An actual example of usage would be:

```
$ ihkosctl 0 assign mem=512M@1
```

In which example, 512MBs memory from NUMA node 1 is assigned to OS instance 0.

3.2.1.2.3 Load kernel image

This operation enables loading a specific kernel image into an OS instance and it has the following format:

`ihkosctl <os index> load <filename>`, where `<os index>` identifies the OS index that has been returned by the OS creation operation, `<filename>` specifies the path to the kernel image intended to be loaded for the OS instance. An actual example of usage would be:

```
$ ihkosctl 0 load /home/example/lwk/kernel.elf.img
```

In which example, /home/example/lwk/kernel.elf.img is loaded. As mentioned earlier, an IHK compatible kernel image is a standard ELF binary linked against the IHK-slave provided library so that it can interact with the other components in the system. The method of building kernel images will be discussed in Section 3.3.

3.2.1.2.4 Set kernel arguments

This operation enables assigning kernel command line parameters to an OS instance, which will be passed to the kernel during boot. It has the following format:

`ihkosctl <os index> kargs <kernel arguments>`, where `<os index>` identifies the OS index that has been returned after the OS creation operation and `<kernel arguments>` is a list of comma separated values. An actual example of usage would be:

```
$ ihkosctl 0 kargs foo=bar,foo2=bar2
```

In which example, foo=bar and foo2=bar2 are the boot time arguments.

3.2.1.2.5 Boot kernel

This operation instructs the OS instance to boot the kernel image specified earlier. It has the following format:

`ihkosctl <os index> boot`, where `<os index>` identifies the OS index that has been returned after the OS creation operation. An actual example of usage would be:

```
$ ihkosctl 0 boot
```

3.2.1.2.6 Display kernel message

This operation obtains the kernel message buffer from the OS instance. It has the following format:

`ihkosctl <os index> kmsg`, where `<os index>` identifies the OS index that has been returned after the OS creation operation. An actual example of usage would be:

```
$ ihkosctl 0 kmsg
```

3.2.1.2.7 Shutdown kernel

This operation instructs the OS instance to shut down. It has the following format:

`ihkosctl <os index> shutdown`, where `<os index>` identifies the OS index that has been returned after the OS creation operation. An actual example of usage would be:

```
$ ihkosctl 0 shutdown
```

3.3 System Programming Interface for LWK

This Section discusses the system programming interface (SPI) for LWK which makes it easy to perform tasks needed to run your own LWK. This chapter guides the user through by explaining the tasks needed and which of them are performed by IHK and which tasks should be implemented by the user. They are explained by going through the tasks needed, that are grouped into those needed when booting the LWK and those needed after LWK enters its main routine.

3.3.1 Booting LWK

Fig. 3.5 explains the steps for Linux to boot an LWK using IHK. All of them are performed by IHK. A special care is needed for the location of the trampoline code. That is, the trampoline code must fit in 20-bit address space because IPI is used to make the first LWK core jump to the trampoline code and 20-bit representation is used for the address in the IPI. A patch to Linux reserves the memory area at boot time in the current implementation.

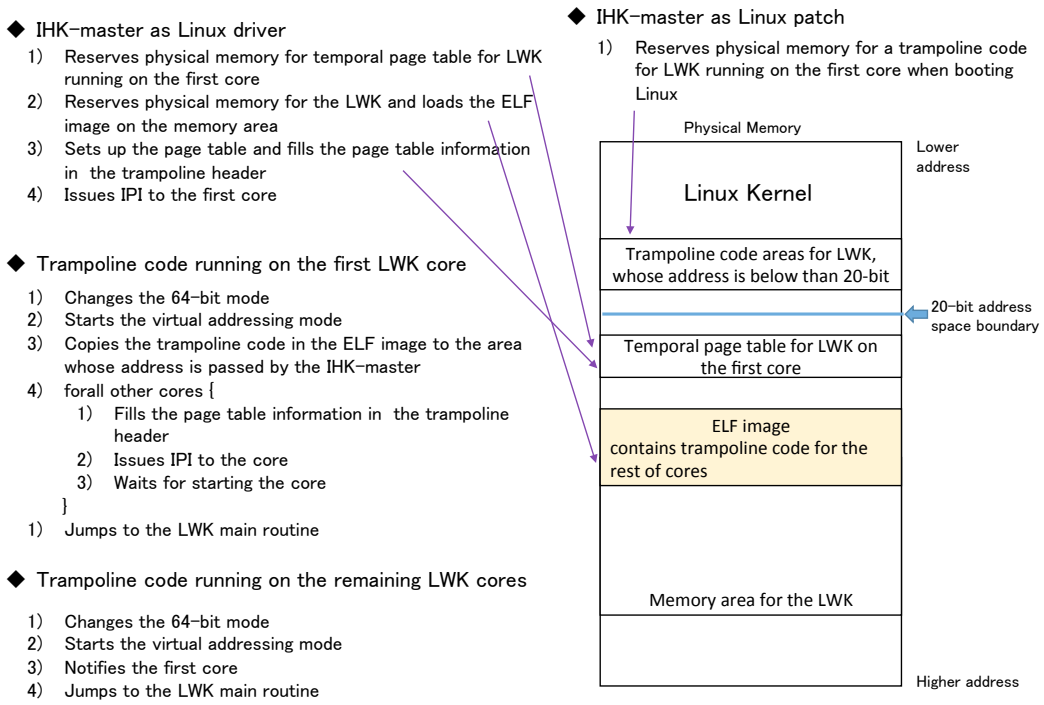


図 3.5: **Booting sequence of cores for LWK.**

Another special care is needed for the location of the temporal page table. That is, the page table must fit in 32-bit address space because the control register (CR3) has 32-bit width when a core is in 32-bit mode in the early phase of the trampoline code.

3.3.2 After Entering LWK Main Routine

When the IHK-slave passes the control to the LWK main routine, it is given the physical address of the kernel arguments as the first argument, the physical address of the kernel text as the second address. One page is allocated as its stack area and the stack pointer points to it. Fig. 3.6 shows the memory map set at this timing. The virtual address range of `[ffff ffff 8000 0000, ffff ffff ffff ffff]` points the LWK kernel image in physical address space and the virtual address range of `[0000 0000 0000 0000, 0000 ff80 0000 0000]` points to the same address range in physical address space. LWK developer is recommended to create its own memory map based on this mapping.

The user implementation can retrieve the resource partition information and kernel argument from Linux, passes kernel-message buffer to Linux and boots another LWK core using the following IHK functions. These functions assume that the virtual address range of `[ffff ffff 8000 0000, ffff ffff ffff ffff]` points to the LWK kernel image.

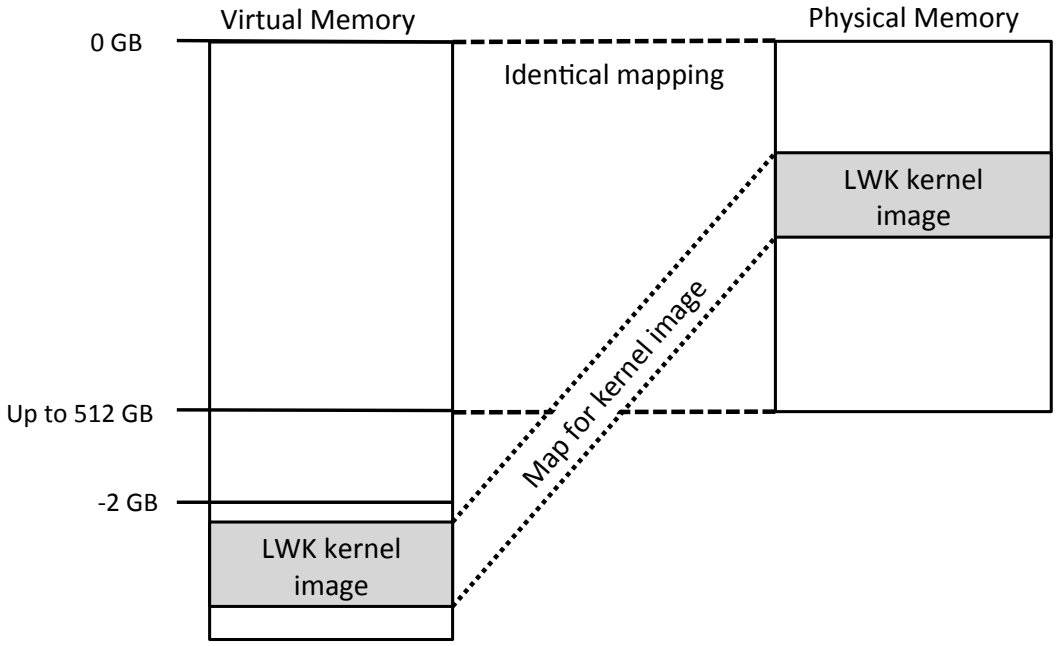


図 3.6: **Memory map when the LWK core enters LWK main routine.**

3.3.2.1 ihk_get_physical_memory_ranges

`ihk_get_physical_memory_ranges` is called by LWK and obtains the information of memory partition assigned by the IHK-master.

Synopsis

```
struct ihk_memory_range *ihk_get_physical_memory_ranges(int *nranges);
```

nranges	**output**	Number of memory ranges

Description

The partition is represented by a set of the physical memory ranges. The function returns a set of ⟨address, size⟩ to support the case where available memory range is split because there are unusable ranges reserved by BIOS.

ihk_memory_range

`ihk_memory_range` represents memory range.

```
struct ihk_memory_range {
    unsigned long addr;
    unsigned long size;
};
```

`addr` is the start address of a memory range. ⟨addr:zero, size:zero⟩ is used for the terminator. `size` is the size of the memory range.

Return Value

⟨zero, zero⟩-terminated array of ⟨address, size⟩

3.3.2.2 ihk_get_core_ids

`ihk_get_core_ids` is called by LWK and obtains the set of cores which IHK-master assigned to the LWK.

Synopsis

```
uint64_t *ihk_get_core_ids(int *nelems);
```

nelems	**output**	Size of the bitmap measured in 64-bit unit

Return Value

A pointer to a bitmap array which represents core-set available to IHK-slave

3.3.2.3 ihk_get_kargs

`ihk_get_kargs` is called by LWK and obtains the kernel arguments given by the IHK master.

Synopsis

```
char *ihk_get_kargs();
```

Return Value

A pointer to the kernel argument string

3.3.2.4 ihk_set_kmsg

`ihk_set_kmsg` is called by LWK and tells the IHK-master kernel message buffer.

Synopsis

```
int ihk_set_kmsg(unsigned long addr, unsigned long size);
```

addr	**input**	Address of kernel message buffer in physical address space on the caller side
size	**input**	Size of kernel message buffer

Return Value

Zero on success or error number on error

3.3.2.5 ihk_remote_physical_to_local_physical

`ihk_remote_physical_to_local_physical` is called by Linux and translates a physical address on the processor on which LWK is running into a physical address on the processor on which Linux is running.

Synopsis

```
unsigned long ihk_remote_physical_to_local_physical(ihk_os_t os, unsigned
              long rphys);
```

os	**input**	OS on the remote side. Zero means Linux.
rphys	**input**	Physical address on the remote side
size	**input**	Size of the memory area

Description

This function is required only in attached mode. For example, Linux running on Xeon obtains a physical address of LWK running on Xeon Phi and translate it into the physical address of Xeon using this function.

ihk_os_t

`ihk_os_t` represents an OS. An IHK user can treat it as an opaque type.

Return Value

Address in physical address space on the local (caller) side

3.3.2.6 ihk_boot_cpu

`ihk_boot_cpu` is called by LWK and boots another core.

Synopsis

```
int ihk_boot_cpu(int cpu_id, unsigned long start, unsigned long pt, unsigned
            long sp);
```

cpu_id	**input**	Physical APIC CPU ID
start	**input**	Virtual address of entry point
pt	**input**	Physical address of page table
sp	**input**	Virtual address of stack pointer

Return Value

Zero on success, error number on error

3.4 Inter-Kernel Communication (IKC)

This chapter discusses Inter-Kernel Communication (IKC) which provides communication facilities among the Linux and LWKs. IKC uses peer-to-peer message based communication. Two kinds of channels, which we call the master channel and the regular channel, are used for the communication. A master channel connects Linux and an LWK and serves as a control channel for creating regular channels and destroying channels. An regular channel is created by connection request from LWK to Linux through the master channel and serves as an general communication channel. The reception is perfomed either by explicitly calling non-blocking receive function or by notification (IRQ) and call-back mechanism.

3.4.1 Creating Master Channel

The master channel should be created before its use by using the following function.

3.4.1.1 ihk_ikc_master_init on IHK-master

`ihk_ikc_master_init` is called by Linux and initializes the master channel.

1 **Synopsis**

2 `int ihk_ikc_master_init(ihk_os_t os);`

3

4 `os` **input** Structure representing remote OS

5 **Return Value**

6 Zero on success, error number on error

3.4.1.2 ihk_ikc_master_init on IHK-slave

`ihk_ikc_master_init` is called by LWK and initializes the master channel.

Synopsis

```
void ihk_ikc_master_init(void);
```

3.4.2 Creating Regular Channels

A regular channel is created by making Linux listen to connection requests and making LWK send connection request to Linux through the master channel using the following functions.

3.4.2.1 ihk_ikc_listen_port

`ihk_ikc_listen_port` is called by Linux or LWK and make it listen to connection requests through the master channel.

Synopsis

```
int ihk_ikc_listen_port(struct ihk_ikc_listen_param *param);
```

param	**inout**	**in**	Channel parameters
		out	CPU ID of the caller (`recv_cpu`)

Description

It allocates receive queue if rq is NULL.

ihk_ikc_listen_param

`ihk_ikc_listen_param` specifies parameters for a channel to be created.

```
struct ihk_ikc_listen_param {
    int (*handler)(struct ihk_ikc_channel_info *);
    int port;
    int pkt_size;
    int queue_size;
    int magic;
    int recv_cpu;
};
```

`handler`	A function called when accepting an incoming connection request and prepares `ihk_ikc_channel_info`.
`port`	Port number
`pkt_size`	Packet size
`queue_size`	Queue size
`magic`	Magic number for identification of the communication initiator
`recv_cpu`	CPU ID of the listener

An IHK user must set the first four fields before passing it to `ihk_ikc_listen_port`. The IHK user must define function which is set to `handler` field of this structure. The function is called when accepting an incoming connection request. The function must sets `packet_handler` field of `ihk_ikc_channel_info`. The value of the field is then copied to `handler` field of `ihk_ikc_channel_desc` and becomes the call-back function which is called when detecting an arrival of a packet. This accept-time call-back mechanism is used to create a table which is indexed by a CPU ID and returns the channel bound to the CPU.

ihk_ikc_channel_info

`ihk_ikc_channel_info` is a intermediate object used by the accept-time call-back function to pass the packet-arrival-time call-back function to the channel.

```
struct ihk_ikc_channel_info {
    struct ihk_ikc_channel_desc *channel;
    ihk_ikc_ph_t packet_handler;
};
```

`channel` is only used internally. `packet_handler` is a intermediate pointer which points to the packet-arrival-time call-back function. An IHK user must define a function which manipulates the `packet_handler` field as described in 3.4.2.1.

Return Value

Zero on success, error number on error

3.4.2.2 ihk_ikc_connect

This function is called by LWK and sends a connection request via the master channel.

Synopsis

```
int ihk_ikc_connect(ihk_os_t os, struct ihk_ikc_connect_param *p);
```

os	**input**	Structure representing remote OS
p	**inout** **in**	Channel parameters
	out	New channel (`channel`)

Description

ihk_ikc_connect_param

`ihk_ikc_connect_param` specifies the parameters for the channel to be created.

```
struct ihk_ikc_connect_param {
    int port;
    int pkt_size;
    int queue_size;
    int magic;
    ihk_ikc_ph_t                handler;
    struct ihk_ikc_channel_desc *channel;
};
```

`port`	Port number
`pkt_size`	Packet size
`queue_size`	Queue size
`magic`	Magic number for identification of the communication initiator
`handler`	Packet handler called when calling `ihk_ikc_recv_handler`
`channel`	Channel descriptor which is set when connected

An IHK user must set `port, pkt_size, queue_size, magic, handler` fields. `channel` field is set to the descriptor of the channel.

ihk_ikc_channel_desc

`ihk_ikc_channel_desc` represents an IKC channel. A IHK user can treat it as an opaque type.

Return Value

Zero on success, error number on error

3.4.3 Send and Receive

A kernel can send packets to the peer kernel and detect an arrival of an incoming packet by call-back function mechanism using IRQ. A kernel can choose whether or not to copy out the received packet from its internal buffer. That is, it can select to copy out the packet to a specified buffer and then manipulate the packet or to manipulate the packet in the internal buffer. This send and receive are performed using the following functions.

3.4.3.1 ihk_ikc_recv_handler

`ihk_ikc_recv_handler` is called by Linux or LWK and registers a call-back function and an argument passed to it. The call-back function is called when detecting an arrival of a packet.

Synopsis

```
int ihk_ikc_recv_handler(struct ihk_ikc_channel_desc *channel, ihk_ikc_ph_t
             h, void *harg, int opt);
```

channel	**input**	Pointer to the structure representing an IKC channel
h	**input**	Function pointer to a handler
harg	**input**	Pointer passed to the handler

Description

This function sends one notification message to the remote side (e.g. sends an interrupt) after handling all of the arrived packets. It is unsafe to read memory area for the packet outside the handler when setting NO_COPY. It is safe when not setting NO_COPY because the arrived packet is memory-copied to user buffer for that case.

Return Value

Zero on success, error number on error

3.4.3.2 ihk_ikc_ph_t

`ihk_ikc_ph_t` represents the call-back function which is called when detecting an arrival of an incoming packet.

```
typedef int (*ihk_ikc_ph_t)(struct ihk_ikc_channel_desc *, void *, void *);
```

It takes the descriptor of IKC channel as the first argument, the address of the incoming packet as the second argument and `harg` passed by `ihk_ikc_recv_handler` as the third argument.

`harg` supports the use case where an IHK user can bind an abstracted channel structure used in the IHK user module to the IKC channel so that the handler can identify the abstracted channel through which the packet has arrived. A reverse search table which returns the abstracted channel given the IKC channel ID is needed if `harg` is not passed down to the call-back function.

3.4.3.3 ihk_ikc_send

`ihk_ikc_send` is called by Linux or LWK and sends a packet through an IKC channel.

Synopsis

```
int ihk_ikc_send(struct ihk_ikc_channel_desc *channel, void *p, int opt);
```

channel	**input**	IKC channel
p	**input**	Address of the outgoing packet
opt	**input**	Option

Description

This function sends a notification message to the remote side (e.g. sends an interrupt) unless `IKC_NO_NOTIFY` bit of `opt` is set to one. It is safe to overwrite memory area pointed by p after calling `ihk_ikc_send` because the packet is memory-copied before sending. It is the IHK user's responsibility to perform flow control.

Return Value

Zero on success, error number on error

3.4.4 Disconnecting Channels

An IKC channel is disconnected by making LWK send a disconnection request to Linux through the master channel. It can be destroyed after it is disconnected. They are performed by using the following functions.

3.4.4.1 ihk_ikc_disconnect

`ihk_ikc_disconnect` is called by Linux or LWK and disconnects an IKC channel.

Synopsis

```
int ihk_ikc_disconnect(struct ihk_ikc_channel_desc *c);
```

`c`	**input**	IKC channel

Return Value

Zero on success, error number on error

3.4.4.2 ihk_ikc_destroy_channel

`ihk_ikc_destroy_channel` is called by Linux or LWK and destroys the instance of the master IKC channel or a regular IKC channel.

Synopsis

```
void ihk_ikc_destroy_channel( struct ihk_ikc_channel_desc *c);
```

`c`	**input**	IKC channel

第4章　McKernel

4.1　Process Management

McKernel は OS カーネルの通常のプロセス管理に加え、Linux と McKernel を同時協調動作させるというために特殊な管理を必要とする。本節ではこれを説明する。

McKernel は、McKernel 上で動作するプロセスを生成する際に、そのプロセスに対応する、Linux 上で動作するプロセス（ghost process、あるいは proxy process、あるいは `mcexec` と呼ぶ）を用意する。そして、McKernel は IKC を用いて `mcexec` と通信して一部のシステムコールの処理を `mcexec` に依頼する。この依頼（システムコールの代理実行、あるいは移譲と呼ぶ）のために、一部のプロセスの状態を `mcexec` に管理させ、また `mcexec` が McKernel 上で動作するプロセスの仮想アドレス空間を参照できるようにする。またユーザやバッチジョブ実行環境のプロセス操作のうち、シグナルの送信は `mcexec` に向けて行うこととし、この場合は `mcexec` は IKC を用いて McKernel と通信しシグナルを McKernel で動作するプロセスに中継する。

4.1.1　McKernel のプロセス起動

McKernel のプロセス起動は、利用者が Linux 上で `mcexec` コマンドを起動することによって行う。`mcexec` の処理概要は以下の通りである。

1. McKernel のデバイスファイル (`/dev/mcos`***n***) を開き、McKernel と通信できるようにする。
2. `mcexec` から McKernel にコマンドラインと環境変数を送付し、コマンド実行の準備をする (MCEXEC_UP_PREPARE_IMAGE)。
3. McKernel のメモリにアプリイメージを展開する (MCEXEC_UP_TRANSFER)。
4. `mcexec` にて、McKernel のコア数+1 のワーカースレッドを起動する。ワーカースレッドの内１スレッドはシグナルハンドリングなどを受け持つ。他のスレッドは McKernel のコアに対応し、システムコールの処理を受け持つ。予めコア数分のワーカースレッドを起動するのは、McKernel 側でのスレッド起動を効率化するためである。
5. `mcexec` から McKernel にアプリをプロセスとして起動することを指示する (MCEXEC_UP_START_IMAGE)。
6. メインスレッドは全てのワーカースレッドの終了を待つ。

7. ワーカースレッドは McKernel からのシステムコール要求を受け付け (MCEXEC_UP_WAIT_SYSCALL)、要求されたシステムコールを処理する。exit_group システムコールに対しては、全てのワーカースレッドを終了させる。

4.1.2 fork

McKernel で実行中のプロセスで fork システムコールが発行された際の動作を説明する。

1. 新しいプロセス用に実行するコアを割り当てる。
2. 新しいプロセスのプロセス情報と仮想記憶情報、ユーザコンテキストを作成する。
3. 親プロセスのメモリ内容をコピーして子プロセスのメモリ内容とする。将来的にはコピーオンライト実装とする。
4. `mcexec` に fork を要求する。`mcexec` は以下の処理を行う。
   (a) `mcexec` を fork し、新しいプロセスを生成する。
   (b) 子プロセスは McKernel のデバイスファイル (`/dev/mcos`*n*) を開き直し (close と open)、子プロセスとして McKernel と通信できるようにする。
   (c) 子プロセス用に McKernel のコア数+1 のワーカースレッドを起動する (McKernel のプロセス起動と同様)。
   (d) 子プロセスのメインスレッドは全てのワーカースレッドの終了を待って終了する。
   (e) 親プロセスの fork 処理は子プロセスの準備ができた後に McKernel に復帰する。
5. `mcexec` の子プロセスは McKernel の親プロセスのメモリマッピングを持った状態になっているため、これを破棄 (munmap) する。
6. 新しいプロセスのシステムコール戻り値を 0 とする。
7. 新しいプロセスに割り当てたコアでプロセスの実行を開始する。
8. 親プロセスは新しいプロセスのプロセス ID を返却する。

4.1.3 スレッド生成

McKernel で実行中のプロセスで clone システムコールの発行によってスレッドを生成する動作を説明する。

1. 新しいスレッド用に実行するコアを割り当てる。
2. 新しいスレッドのプロセス情報と仮想記憶情報、ユーザコンテキストを作成する。
3. 新しいスレッドのシステムコール戻り値を 0 とする。
4. 新しいスレッドに割り当てたコアでスレッドの実行を開始する。
5. 親プロセスは新しいプロセスのスレッド ID を返却する。

4.1.4 ファイルディスクリプタ

McKernel では原則ファイルディスクリプタを管理しない。ファイルディスクリプタは全て mcexec で管理される。

スレッドの生成では、親スレッドと子スレッドはファイルディスクリプタを共有する。一方、プロセスの生成では親プロセスと子プロセスは独立したファイルディスクリプタを使用する。これを実現するために、mcexec はプロセス単位に存在しなければならない。

また、mcexec は McKernel と通信するためにデバイスファイルをオープンする。即ち、デバイスファイルのファイルディスクリプタを使用するが、このファイルディスクリプタをユーザプログラムからアクセスされると予期せぬ不具合を来す恐れがある (例えばデバイスファイルをクローズされると以降の処理が継続不可能になる)。このため、McKernel から委譲されたシステムコールでデバイスファイルのファイルディスクリプタの操作がある場合は、エラーとする。

4.1.5 シグナル

シグナルは、mcexec が受けるシグナル (例えば shell 環境で^C の押下) と、McKernel で発生するシグナル (例えばユーザプログラムでの不正領域アクセス) がある。

mcexec が受けたシグナルは McKernel に伝えて、McKernel 側で処理する必要がある。

また、McKernel でシグナルを処理する際に、mcexec がシステムコール処理中の場合にはシステムコールを中断する必要がある場合がある (シグナルを無視する場合やシグナルがマスクされている場合はこの限りではない)。mcexec が実行中のシステムコールを中断するには、McKernel から mcexec に対してシステムコール処理中断を要求しなければならない。

4.1.6 プロセス ID とスレッド ID

McKernel のプロセスのプロセス ID は対応する mcexec のプロセス ID と同一にする。

McKernel のスレッドのスレッド ID は原則として対応する mcexec のスレッド ID と同一にするが、以下の例外がある。

1. メインスレッドはプロセス ID と同一とする。McKernel のメインスレッドに対応する mcexec のスレッドは、mcexec のメインスレッドではなくワーカースレッドの 1 つである。

2. McKernel のスレッドを実行するコアが変更になった場合 (migrate)、対応する mcexec のスレッドが変更になるが、McKernel のスレッド ID は変更を受けない。また、同一のスレッド ID が二重に存在することがないように制御しなければならない。

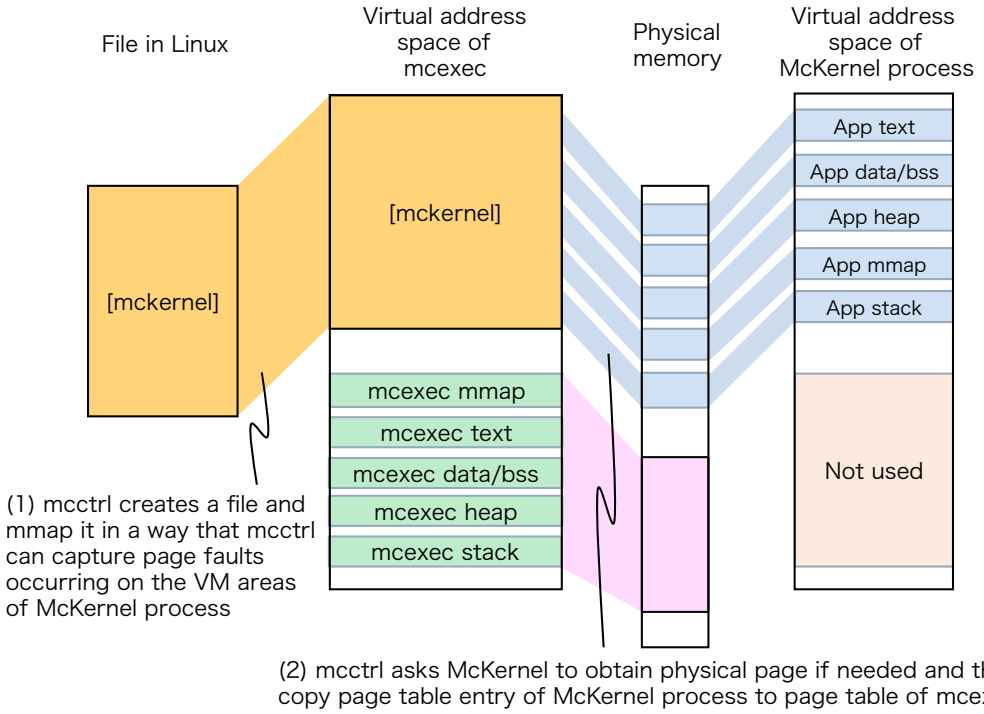


図 4.1: How `mcexec` hooks page-faults occuring on VM areas of McKernel process and maintains the same page table entries in the page table of `mcexec` as in the page table of McKernel process.

4.2 Memory Management

4.2.1 Ghost Process (mcexec) のプロセス空間管理

4.2.1.1 委譲用マップとそのページフォルト処理

`mcexec` は、システムコール委譲のため、`mcexec` のプロセス空間上に McKernel 上のプロセスと同じ仮想アドレスで同じ物理ページを参照できる仮想メモリ領域（委譲用マップと呼ぶ）を作成する。

通常、ある空間で発行されたシステムコール要求を別の空間で実行するためには、システムコールの引数を解釈する必要がある。なぜなら、引数がポインタのとき、引数が示す領域をも別空間上に再現する必要があるからである。しかし、委譲用マップを使用することで、McKernel 上のプロセスの仮想アドレスをそのまま使って `mcexec` プロセスが同じ物理ページを参照できるようになるので、個別にシステムコールの引数が示す領域を再現する必要がなくなる。

図 4.1 を用いて委譲用マップの作成方法を説明する。まず、`anon_inode_getfile()` を用いて `[mckernel]` という名を持つ擬似ファイルを作成し、McKernel 上のプロセスの仮想メモリ領域全体をカバーするアドレスレンジを指定してこのファイルをメモリマップする（図の（1））。こうすることで、委譲されたシステムコール処理が McKernel 上のプロセスのメモリにアクセスしようとしたことを、この委譲用マップ上のページフォルトとして `mcctrl` が捕捉することができる。`mcctrl` は、このページフォルトを McKernel 上のプロセスのページ

テーブルを直接参照して同じ物理ページをマップすることで解決する（図の（２））。デマンドページングなどで McKernel 上のプロセスにも物理ページが割り当てられていないことがある。この場合、ページフォルトの発生を McKernel に伝えて McKernel 上のプロセスに物理ページを割り当てさせる。これを、リモートページフォルトと呼ぶ。その後、mcctrl が、McKernel 上のプロセスのページテーブルを直接参照して割り当てられた物理ページを調べ、それを委譲用マップにマップする。

委譲用マップ上のページフォルトが、リモートページフォルト処理を実行して解決される場合の処理手順を以下に示す。

1. 委譲されたシステムコール処理が委譲用マップにアクセスして、ページフォルトを発生させる。
2. mcctrl は、委譲用マップ上のページフォルトを捕捉すると、対応する McKernel 上のプロセスのページテーブルを直接参照して PTE を検索する。
3. PTE が見つからない場合、つまりリモートページフォルトが必要な場合、
   (a) mcctrl は、委譲されたシステムコールの response page にフォルトアドレスを格納し、特別な終了状態 `STATUS_PAGE_FAULT` を報告してシステムコール委譲を中断させる。
   (b) McKernel は、`STATUS_PAGE_FAULT` でシステムコール委譲が中断されると、response page 上のフォルトアドレスを使ってページフォルト処理を実行する。
   (c) McKernel は、ページフォルト処理を終えると、request page にシステムコール委譲の続行要求を格納し、IKC メッセージ `SCD_MSG_SYSCALL_ONESIDE` をホスト OS に送る。
   (d) mcctrl は、IKC メッセージ `SCD_MSG_SYSCALL_ONESIDE` を受け取り、request page にシステムコール委譲続行要求を見つけると、2 の PTE 検索からやり直す。
4. mcctrl は、見つけた PTE が示す物理ページをフォルトが発生した仮想ページにマップする。
5. mcctrl は、Linux にページフォルト元の再実行を要求する。
6. 委譲されたシステムコール処理は、再実行されると、委譲用マップへのアクセスで、McKernel 上のプロセスと同じ物理ページにアクセスする。

McKernel 上のプロセスが、`munmap()` などマップを解放するシステムコールあるいは `MADV_REMOVE` を指定した `madvise()` などの物理ページを解放するシステムコールを発行したとき、同期的に特別なシステムコール委譲要求を発行し、`mcexec` 上の対応する領域の物理ページのマップを解除する。McKernel は、プロセス空間にマップされた物理ページを、そのプロセスがシステムコールで明示的に要求することなくアンマップすることはないので、それらのシステムコールの延長でのみ `mcexec` に物理ページのマップ解除を要求すれば十分である。

委譲用マップは Linux の制約から、読み出し専用のページと読み書き両用のページを混在させることができない。そこで、McKernel 上での読み出し専用マップの作成に同期して、

特別なシステムコール委譲要求を発行し、mcexec 上の対応する領域のマップを分割、縮小することで読み出し専用の委譲用マップの仮想メモリ領域を空け、そして読み出し専用マップを作成をする。読み出し専用マップの削除についても同期して削除を行う。読み出し専用マップの作成は、共有ライブラリのロードを除けばまれなので、同期的に mcexec 上のマップを操作しても性能上の影響は小さい。

委譲用マップは Linux の制約から、ページ保護例外を捕捉することができない。このため、copy-on-write 動作を実現できない。また逆に、McKernel 上のプロセスが、copy-on-write によるページ置き換えが発生する都度 mcexec に通知することも、アプリケーション性能に与える影響を考えるとできない。そこで、mcctrl によるページテーブル検索では copy-on-write ページはページ不在扱いとしてリモートページフォルト処理をさせ、McKernel でのリモートページフォルト処理では常に eager copy をさせる。

4.2.1.2 委譲用マップ配置

委譲用マップを使ってシステムコールを委譲するには、McKernel 上のプロセスの仮想アドレス領域を mcexec 上の委譲用マップの中に完全に収めなければならない。なぜなら、mmap() で作成した領域に限らず、テキスト、データ、スタックの全てがシステムコールの引数からポイントされる可能性があるからである。しかし、委譲用マップの位置とサイズの設定について以下の 2 つの課題がある。

1. mcexec 自身のテキストセグメントとデータセグメントによる委譲用マップの位置の制限
2. mcexec の main thread のスタック用予約領域による委譲用マップのサイズの制限

これらについて、Intel アーキテクチャでの回避策を説明する。

4.2.1.2.1 mcexec 自身のテキストセグメントとデータセグメントによる委譲用マップの位置の制限

Intel アーキテクチャでは、慣習的に、実行ファイルのテキストセグメントの開始アドレスは仮想アドレス 0x400000 に固定されている。また、データセグメントはテキストセグメントの終端から 2 MiB ほど上位側に配置される。したがって、このままでは、mcexec のテキストセグメントと McKernel 上のプロセスのテキストセグメントが重なることを回避できない。

そこで、mcexec を実行ファイルではなく、実行可能な動的リンクライブラリとして作成した。ライブラリであれば、各セグメントは mmap() のマップアドレス決定方法にしたがって動的に配置される。Intel アーキテクチャLinux のデフォルトである flexible mmap layout 属性のプロセスならば、マップアドレスは、スタック用の予約領域のアドレスが小さい側に接するように決定される。これを利用して、mcexec のテキスト、データ、スタック、mmap 領域の全てを、mcexec プロセス空間の最上位アドレス側にまとめた。

mcexec の main thread のスタック用予約領域による委譲用マップのサイズの制限 McKernel は、McKernel のプロセスに割り当てられるのは、仮想アドレス空間のうちゼロから始まる連続領域であるという前提で動作する。このため、mcexec の委譲用マップも、アドレスゼロから始まる連続領域である必要がある。一方で、Intel アーキテクチャLinux では、mcexec に対

して、ランダムに決められたアドレスから始まりプロセス空間の $\frac{5}{6}$ と RLIMIT_STACK の小さい方のサイズを持つ仮想メモリ領域をスタック用予約領域として確保し、それより小さいアドレス位置にヒープ領域を設定し、それより小さいアドレス位置にテキスト領域を設定し、それより小さいアドレス位置に mmap 領域を設定する。このため、委譲用マップのサイズは、この mmap 領域の先頭アドレス以下になるという制限が生じる。。このため、スタックサイズを無制限として mcexec を起動するとスタック予約領域のサイズがプロセス空間の $\frac{5}{6}$ となるため、委譲用マップのサイズはプロセス空間の $\frac{1}{6}$ 未満になり、十分なサイズを確保できなくなる。

そこで、mcexec 自身の RLIMIT_STACK と McKernel 上のプロセスの RLIMIT_STACK とを分離した。まず、mcexec に自身の RLIMIT_STACK をチェックさせる。1 GiB を超える場合は 10 MiB に設定して自身を exec*() させてスタック用予約領域を縮小させる。次に、mcexec がスタック用予約領域の縮小をした場合でも、McKernel 上のプロセスには、最初に mcexec が起動されたときの RLIMIT_STACK を設定した。これらによって、McKernel 上のプロセスの RLIMIT_STACK の設定は最初に mcexec が起動されたときの RLIMIT_STACK を引き継ぐことができる。

mcexec 自身に 1 GiB までのスタックサイズを許すのは、スタックオーバフローが疑われる事象が発生したときに、オーバフローが発生しているかどうかをスタックサイズを大きくして観察できるようにするためである。

4.2.2 IHK-Master による IHK-Slave 用メモリの管理

4.2.2.1 委譲用マップ上のページ固定サポート

ユーザプロセスが libibverbs などのライブラリを利用してデバイスと直接 I/O することを、Linux 同様に McKernel でも実現する。これらのライブラリは、ioctl() などのシステムコールを利用して直接 I/O の準備をデバイスドライバに依頼し、依頼を受けたデバイスドライバは、システムコールの引数が示す領域をページ固定した上でその物理アドレスをデバイスに設定するものと想定している。Linux でページ固定できる物理ページは、Linux が RAM として管理している物理ページのみである。そこで、Linux が RAM として管理している物理メモリで McKernel を動作させることで、委譲用マップ上でのページ固定を可能にする。

これは、IHK-Master が IHK-Slave に割り当てるメモリを Linux から __get_free_pages() で確保することで実現する。__get_free_pages() は Linux の制約から、通常、最大 1024 ページしか割り当てられないので、繰り返し呼び出して物理連続となったものをマージすることで、IHK-Slave への割り当てに必要な物理連続な単一のブロックを作り出す。

__get_free_pages() を選択したのは、最大割り当てサイズに制限があるものの、Boot Memory Allocator や Contiguous Memory Allocator に比べて、広く利用可能なインタフェースであるためである。

4.3 System Calls

McKernel は、OS ノイズを削減しつつ、Linux のツールチェインで作成されたプログラムがそのまま McKernel で動くようにシステムコールの機能を提供する。このために、アプ

表 4.1: McKernel のシステムコール実装分類（プロセス管理）

Implemented	Planned
arch_prctl	get_thread_area
clone	getrlimit
execve	ptrace
exit	rt_sigtimedwait
exit_group	set_thread_area
futex	setrlimit
getpid	signalfd
getrlimit	signalfd4
kill	
pause	
ptrace2	
rt_sigaction	
rt_sigpending	
rt_sigprocmask	
rt_sigqueueinfo	
rt_sigreturn	
rt_sigsuspend	
set_tid_address	
setpgid	
sigaltstack	
tgkill	
vfork	
wait4	

リケーションに対して性能クリティカルでないシステムコールについては、McKernel からの指令で Linux で実行する（システムコール委譲と呼ぶ）方法を取る。以下では、まずどのシステムコールを委譲するかを説明し、次にシステムコール委譲方法を説明し、実装が自明でないシステムコールの実装を説明し、最後にシステムコールの制限を説明する。

4.3.1 委譲分類

McKernel は、以下のどちらかに該当するシステムコールを自身で処理し、それ以外をホスト OS に委譲する。

1. McKernel が処理する以外に実現する方法がないもの
2. McKernel が処理することによってアプリケーションの性能が顕著に向上するもの

McKernel が処理する以外に実現する方法がないものには、`sched_setaffinity()` による CPU アフィニティの設定や `sigaction()` によるシグナルハンドラの設定など CPU を操作するものと、`mmap()` や `fork()` などのメモリを操作するものとがある。

McKernel が処理することによってアプリケーションの性能が顕著に向上するものには、`gettimeofday()` による時刻の取得などがある。

本バージョンの McKernel のシステムコール実装分類を **表 4.1**、**表 4.2**、**表 4.3**、**表 4.4** に示す。表にないシステムコールはホスト OS に移譲する。

4.3.2 システムコール委譲方法

システムコールの委譲処理には以下のものが関わる。

表 4.2: McKernel のシステムコール実装分類（メモリ管理）

Implemented	Planned
brk	get_robust_list
gettid	mincore
madvise	mlockall
mlock	modify_ldt
mmap	munlockall
mprotect	set_robust_list
mremap	shmat
munlock	shmctl
munmap	shmdt
remap_file_pages	shmget
	process_vm_readv
	process_vm_writev

表 4.3: McKernel のシステムコール実装分類（スケジュール）

Implemented	Planned
sched_getaffinity	alarm[b]
sched_setaffinity	getitimer[b]
	gettimeofday[b]
	nanosleep[b]
	sched_yield
	setitimer[b]
	settimeofday[b]
	time[b]
	times[b]

[b]These system calls are delegated to Linux for the moment.

1. request page
2. response page
3. システムコール IKC チャネル
   (a) SCD_MSG_SYSCALL_ONESIDE IKC メッセージ
4. mcexec サービススレッド

request page は、McKernel が動作している CPU ごとに存在し、委譲されるシステムコールの引数を McKernel からホスト OS へ伝える。

response page は、McKernel が動作している CPU ごとに存在し、委譲されたシステムコールの戻り値をホスト OS から McKernel に伝える。

表 4.4: McKernel のシステムコール実装分類（パフォーマンスカウンタ）

Implemented	Planned
Original Interface	PAPI Interface
pmc_init	
pmc_reset	
pmc_start	
pmc_stop	

request page および response page がスレッドごとではなく CPU ごとに存在するのは、ある CPU が発行するシステムコール委譲要求が高々1 つだからである。なぜなら、McKernel は、ユーザスレッドが CPU を占有する方式を採用しているため、システムコール委譲中に他のスレッドをスケジュールすることがないからである。

システムコール用 IKC チャネルは、request page ごとに存在する。このチャネル上を McKernel からホスト OS へ送信される `SCD_MSG_SYSCALL_ONESIDE` IKC メッセージは、request page 上に未処理のシステムコール委譲要求があることと委譲要求元プロセスの PID とをホスト OS に伝える。

mcexec サービススレッドは、McKernel 上の対応するプロセスが動作可能な CPU ごとに存在し、委譲されたシステムコール要求を処理する。

システムコール委譲は以下の手順で処理される。

1. ユーザプログラムが、システムコールを発行する。
2. McKernel は、システムコール要求を受けると、システムコールの引数を request page に書き込み、ホスト OS に `SCD_MSG_SYSCALL_ONESIDE` IKC メッセージを送って、request page 上に要求があることと委譲元プロセスの PID とを伝える。
3. mcctrl は、IKC メッセージを受け取ると、チャネルから要求元 CPU を、メッセージから要求元プロセスを特定し、対応する mcexec サービススレッドを起床させる。
4. mcexec サービススレッドは、起床すると、request page に格納された引数でシステムコールを実行し、戻り値を response page に書き込む。
5. McKernel は、response page への書き込みを検出すると、response page 上の戻り値をユーザプログラムに返す。
6. ユーザプログラムは、システムコールの戻り値を受け取ると、処理を続行する。

4.3.3 システムコール機能

この節では、実装が自明でないシステムコールについて説明する。

4.3.3.1 gettimeofday

Intel 64 アーキテクチャの Linux の場合、`gettimeofday()` システムコールとは別に `vsyscall_gettimeofday()` vsyscall が存在し、glibc はこれを利用して高速な `gettimeofday()` を実現している。Intel 64 アーキテクチャ用の McKernel も `vsyscall_gettimeofday()` を実装する。

`vsyscall_gettimeofday()` の McKernel の実装は以下の要素から構成される。

1. gtod ページ
2. `vsyscall_gettimeofday()` 本体
3. TSC 同期機能

gtod ページは、ホスト OS である Linux が所有する物理ページである。TSC の値を `struct timeval` の値に変換するために必要な情報をユーザ空間に提供することが目的である。vsyscall ページと同様に、ユーザ空間の固定位置に読み出し専用ページとしてマップされ、vsyscall ページからポイントされる。gtod ページには、`gettimeofday()` 相当の結果とその時の TSC の値との組を、異なる時間のものを複数格納する。mcctrl が定期的に、前回書き込んだ値を退避したうえで最新の値を書き込むことで、実現する。

gtod ページの内容は以下の通り。

1. `tsc`: 最後の更新時の TSC
2. `timeval`: 最後の更新時の `struct timeval`
3. `prev_tsc`: 最後の更新開始時の `gtod.tsc` の値を退避したもの
4. `prev_timeval`: 最後の更新開始時の `gtod.timeval` の値を退避したもの

`vsyscall_gettimeofday()` 本体は、自 CPU の TSC の値と gtod ページの内容とから、以下のようにして現在の `struct timeval` の値を求める。

$$\begin{aligned} timeval_{current} &= gtod.timeval \\ &+ \frac{gtod.timeval - gtod.prev\_timeval}{gtod.tsc - gtod.prev\_tsc}(TSC_{current} - gtod.tsc) \end{aligned}$$

TSC 同期機能は、Linux のある CPU の TSC に McKernel のすべての CPU の TSC を同期させる機能である。実行する CPU による `vsyscall_gettimeofday()` の結果の違いをできるだけ小さくすることを目的とする。アーキ依存関数として実装される。TSC の同期は、McKernel の起動時に 1 回だけ実行される。そのタイミングは、McKernel としての AP の起動直後を想定している。同期には gtod ページを利用する。その手順は以下の通り。

1. `gtod.tsc` の値が変わるのをスピンしながら待つ。
2. 変更後の `gtod.tsc` の値で `IA32_TIME_STAMP_COUNTER` の値をリセットする。

4.3.3.2 Cross Memory Attach

Linux 3.2 で、 他プロセスのメモリを読み書きする以下のシステムコールが追加された。

`ssize_t process_vm_readv(pid_t` *pid*`, const struct iovec *`*local_iov*`, unsigned long` *liovcnt*`, const struct iovec *`*remote_iov*`, unsigned long` *riovcnt*`, unsigned long` *flags*`);`

`ssize_t process_vm_writev(pid_t` *pid*`, const struct iovec *`*local_iov*`, unsigned long` *liovcnt*`, const struct iovec *`*remote_iov*`, unsigned long` *riovcnt*`, unsigned long` *flags*`);`

McKernel もこれらのシステムコールを実装する。現在想定する実装のステップを述べる。

1. 引数をチェックする。
2. *pid* が示すプロセスを検索し、見つけたプロセスのアドレス空間の解放を抑止する。

3. プロセスの検索でエラーを検出した場合、直ちにエラー終了する。

4. *local_iov* および *remote_iov* の内容チェック。エラーを検出した場合、

   (a) アドレス空間の解放抑止を解除する。

   (b) エラー終了する。

5. *local_iov*、*remote_iov* の両方にコピーすべき領域が存在する間、

   (a) 今回コピーする量を決定する。

   (b) 今回のコピーに使用するローカルバッファをページ固定し、それらのページのページリストを作成する。

   (c) エラーを検出した場合、ループを抜けて (出口 1) で合流する。

   (d) *pid* が示すプロセスのものにアドレス空間を切り替える。

   (e) `copy_from_user()` あるいは `copy_to_user()` を使って、ページリストが示すカーネル仮想アドレスと、*pid* および *remote_iov* が示すユーザバッファとの間でコピーする。

   (f) 呼び出したプロセスのものにアドレス空間を戻す。

   (g) ローカルバッファのページ固定を解除する。

   (h) コピーでエラーを検出した場合、ループを抜けて (出口 1) で合流する。

6. (出口 1)

7. アドレス空間の解放抑止を解除する。

8. エラーを検出した場合はエラーを、そうでない場合は実際にコピーした量を返す。

- 引数チェックのチェック項目は以下の通り。

  1. *pid* は 0 より大きいこと。そうでない場合、`ESRCH` エラー。
  2. *liovcnt* は `IOV_MAX` 以下であること。そうでない場合、`EIVAL` エラー。
  3. *liovcnt* が 0 以外の場合、範囲 [*local_iov* .. *local_iov*+*liovcnt*) は user region 内に収まること。そうでない場合、`EFAULT` エラー。
  4. *riovcnt* は `IOV_MAX` 以下であること。そうでない場合、`EIVAL` エラー。
  5. *riovcnt* が 0 以外の場合、範囲 [*remote_iov* .. *remote_iov*+*riovcnt*) は user region 内に収まること。そうでない場合、`EFAULT` エラー。
  6. *flags* は 0 であること。そうでない場合、`EINVAL` エラー。

- *local_iov* の内容チェックのチェック項目は以下の通り。

  1. 各 `iovec` が示す領域が自プロセスでアクセス可能なこと。そうでない場合、`EFAULT` エラー。
  2. 全 `iovec` の合計長がオーバフローすることなく `ssize_t` に収まること。そうでない場合、`EINVAL` エラー。

- *remote_iov* の内容チェックのチェック項目は以下の通り。
    1. 全 iovec の合計長がオーバフローすることなく ssize_t に収まること。そうでない場合、EINVAL エラー。
- ループ 1 回でコピーする量は、以下のすべてが成立する量のうち最大のものとする。
    1. *local_iov* 上で iovec またがりが発生しない。
    2. *remote_iov* 上で iovec またがりが発生しない。
    3. 今回のコピーに使用するローカルバッファのページリストのデータ量が一定量に収まる。(一定量として 1 ページを想定している。)

4.3.4 システムコールにおける制限事項

バージョン 1.0 での、システムコールの動作の制限を説明する。

4.3.4.1 Cloning a new thread forming a new thread group

The Linux side must track a thread group information for a new thread which is created through `clone()` system call with `CLONE_VM` flag set and `CLONE_THREAD` flag unset. This is because the flag combination means the new thread has its own thread group and the Linux side need to track the information as the Linux side needs to handle thread group related operations. For example, when the new thread calls `exit_group()`, the Linux side sends signals to members of the thread group of the calling thread and the Linux side must refer to the information to do so. But this requirement conflicts with the design decision where the Linux side does not track the information (by making `mcexec` invoke `clone()`) when threads are created to reduce clone overhead for application performance.

However, this combination is rarely used in the applications. Therefore, we decided not to support this combination.

4.3.4.2 NUMA サポート

仮想ページと NUMA-node の対応を管理するシステムコールは、バージョン 1.0 では、１つの NUMA-node しか存在しないとして動作するか、ENOSYS を返す。

4.3.4.3 msync

Only the modified pages mapped by the calling process are written back.

4.4 procfs/sysfs

McKernel は McKernel のプロセスと選択された Linux のプロセスに対し procfs と sysfs を提供する。以下では、ゴール、課題、アプローチ、実現方法を説明する。

4.4.1 ゴール

procfs/sysfs はファイルシステムのインターフェイスを用いてシステム、カーネル、プロセスといった資源の情報を読んだり、それらへの指示を書き込んだりする機構である（以下、情報の読み出しと指示をまとめて情報へのアクセスと呼ぶ）。ゴールは、以下の 2 つである。

1. McKernel で動作するプロセスに対して、McKernel が動作するシステムの情報（すなわち、McKernel が動作するパーティションの情報）、McKernel のカーネルの情報、McKernel で動作するプロセスの情報へのアクセスを提供すること
2. 特定 Linux プログラムの（Linux で動作する）プロセスに対して、McKernel が動作するシステムの情報（すなわち、McKernel が動作するパーティションの情報）、McKernel のカーネルの情報、McKernel で動作するプロセスの情報へのアクセスを提供すること

上記特定 Linux プログラムは、シェルなどを想定し、バージョン 1.1 で詳細検討を行う予定である。

4.4.2 課題

課題は、McKernel のプロセスから見た際に、Linux の提供するファイルと、McKernel が提供するファイルが混在して見えるようにし、また、選択された Linux のプロセスから見た際に、Linux の提供するファイルと、McKernel が提供するファイルのうち必要なものが混在して見えるようにすることである。ファイルシステムをファイルやディレクトリをノードとしたツリーとして捉える。ファイルやディレクトリ参照の場合は、ノードをトラバースする際に適切なノード、すなわち Linux 提供ファイルあるいは McKernel 提供ファイルのいずれか適切な方に導く必要がある。ディレクトリリスティングの場合は、見せるべきものは、Linux 提供ファイルの集合、McKernel 提供ファイルの集合、混在した集合の 3 パターンあり、必要に応じていずれのパターンも見せられるようにする必要がある。また、選択された Linux のプロセスが McKernel の資源情報提示や指示受付を行うファイル／ディレクトリのうちどれを必要としている調査して、対応する必要がある。

4.4.3 アプローチ

Linux は、ファイルシステムを構成するファイルなどのエントリへのアクセス時に起動されるコールバックファンクションを登録する仕組みを提供するため、これを利用する。また、読み込み時に Linux と McKernel とで異なる値を提示する必要があるエントリ、書き込み時に異なる動作をする必要があるエントリを区別して、これらは McKernel で処理し、それ以外を Linux に処理させる。McKernel で処理すべきものには、McKernel のプロセスのメモリマップ、McKernel の動作するパーティションのコア数が挙げられる。McKernel で処理すべきものについては、上記コールバックファンクションは Linux で動作するため、IKC を用いて McKernel と通信をを行って値の提供や指示された動作を行う。本バージョンの McKernel で処理している procfs と sysfs のファイルあるいはディレクトリを**表 4.5** に示す。なお、McKernel で処理すべきものは今後調査して拡充する予定である。

4.4.4 実現方法

まず、McKernel が処理する procfs ファイルと Linux に処理を任せる procfs ファイルとが混在できるようにする。

1. Linux の procfs 上に McKernel 用の procfs ファイルを作成する。このとき、McKernel 上のプロセスに "/proc/..." のように見せたいファイルを"/proc/mcos*OS_number*/..." として作成する。

2. mcexec&mcctrl に open() システムコール要求が委譲されたときに、引数のパス名をチェックする。パス名の先頭が "/proc/" の場合は、"/proc/mcos*OS_number*/" に置き換えて open() を実行する。

3. リダイレクト先にファイルが存在しない場合は、元のパスで open() をやり直す。

Linux の procfs 上への McKernel 用ファイルの作成は、McKernel からの要求によって実行される。McKernel は、以下のタイミングで McKernel 用 procfs ファイルの作成を要求する。

1. McKernel 自身が起動するとき

2. McKernel 上に thread が起動するとき

McKernel 用 procfs ファイルの削除は、2 種類の方法で実行される。第 1 の方法は、McKernel からの要求による削除である。これは、thread 作成時に作成した procfs ファイルを thread 終了時に削除するために使われる。第 2 の方法は、mcctrl による一括削除である。これは、McKernel 起動時に作成した procfs ファイルを McKernel 終了後に削除するために使われる。

McKernel 用 procfs へのアクセスは、以下のように処理される。

1. Linux によって、McKernel 用 procfs へのアクセスが検出され、procfs の作成者である mcctrl に通知される。

2. 通知を受け取った mcctrl は、その procfs ファイルの作成を要求した McKernel にアクセス要求を転送する。

3. 要求を受け取った McKernel は、Linux のメモリに結果を書き込む。

4. McKernel は、処理の結果を mcctrl に送る。

表 4.5: McKernel で処理している procfs と sysfs のファイルあるいはディレクトリ.

エントリ	使用が確認されたアプリやライブラリ
/proc/stat	Intel OpenMP run-time
/proc/<pid>/tasks/<tid>/stat	gdb
/proc/<pid>/mem	gdb
/proc/<pid>/aux	gdb
/proc/<pid>/pagemap	LTP
/sys/devices/system/cpu/online	Intel OpenMP run-time (attached configuration)

5. 結果を受け取った mcctrl は、Linux に結果を報告する。

Linux と mcctrl との間は、通常の procfs ファイルと同様に、Linux の `file_operations` インターフェイスを利用する。mcctrl と McKernel との間は、McKernel のシステムコール委譲に使用する syscall channel (IKC) を利用する。

4.4.5 procfs/sysfs における制限事項

バージョン 1.0 での、procfs/sysfs の動作の制限を説明する。

4.4.5.1 `/proc/<PID>/mem`、`/proc/<PID>/task/<TID>/mem`

これらのファイルは、対象プロセスあるいはスレッドがスケジューリングにおける実行状態にある時のみ参照可能である。

4.5 Power Management

McKernel は PowerAPI に規定された OS の API[3] を用いて消費電力を制御する。

4.6 Support for Application Specific Kernels

McKernel の基本形は、Linux のすべての機能を利用できるようにしているため、特定のアプリケーションにとっては不要な機能やオーバヘッドがある。

例えば、現在の Intel 64 アーキテクチャ用 McKernel のメモリ管理は、Linux と同じ copy-on-write や物理ページの動的な共有を実現するために、page 構造体の大きな配列を持っており、利用可能メモリの $\frac{1}{128}$ のサイズを占めている。

そこで、特定のアプリケーションに合わせて、McKernel の機能を取り除いたり、特定のアプリケーション向けの機能を追加したりすることを考えている。

このために、アプリケーションの性能に影響を与えることが予想される機能を選び出し、それらのインターフェイスを定義し、それに従った実装をすることで、異なる実装の開発や交換作業を容易にできるようにする。

この対象となる機能としては、プロセス空間管理、物理ページアロケータ、thread のスケジューラなどが念頭にあるが、これに限らない。

機能の交換や追加は、ソースあるいはオブジェクトファイルの組み合わせを変える、リコンパイルといった方法で行う。

4.7 Debug Facilities

McKernel が提供するカーネルデバッグのための機能、すなわち、カーネルメッセージの McKernel 外部への出力機能、メモリダンプ機能を説明する。

4.7.1 カーネルメッセージの McKernel 外部への出力

McKernel のカーネルメッセージを McKernel 外部に出力する機能についてはバージョン 1.1 で詳細を検討するが、方針を説明する。Linux 上で動作するプロセス（`mcklogd` と呼ぶ）が McKernel のカーネルメッセージが保存されているメモリエリアよりメッセージを取得して、Linux 上で動作する `syslogd` に RFC 5424 で規定されたフォーマットの構造体を作成して送信する。`syslogd` 側ではメッセージを McKernel が動作するノードで区別して分類したり、複数 McKernel インスタンスが 1 つのノードで動作している場合はインスタンスの番号で区別して分類したりできるようにする必要があるため、`mcklogd` はこれらの情報を上記構造体の `HOSTNAME` フィールド、`PRI` フィールド、`STRUCTURED-DATA` フィールドに設定する。具体的には、ホスト名が `host_name` であるノードで動作する McKernel の第 i のインスタンスからの severity が s であるカーネルメッセージに対しては、`HOSTNAME` フィールドを `host_name` に設定し、McKernel を表す facility の番号を f、$p = f \times 8 + s$ として、`PRI` フィールドを<p>に設定し `STRUCTURED-DATA` フィールドに `[mckernel instanceid="`i`"]` を挿入し、メッセージを格納するフィールドに McKernel のカーネルメッセージを設定する。なお、McKernel を表す facility の番号は `local0` から `local7` と呼ばれる 16 から 23 の整数のうちシステムで使われていないものを用いる。

4.7.2 メモリダンプ

McKernel がクラッシュした際には、`core` ファイルを出力し、`gdb` などを用いて解析できるようにする。なお、McKernel が Linux を巻き込んでクラッシュして Linux がメモリダンプを行う際には、バージョン 1.0 では McKernel のメモリ内容がそのメモリダンプに含まれる。これは、McKernel に割り当てたメモリが Linux によっても管理されているためである。

4.8 Portability

McKernel は、ポスト京のアーキテクチャ以外コモディティクラスタでも動作させることを目的としている。このため、新しいプロセッサアーキテクチャのサポートに対して低開発コストで対応できるようにし、また既にサポートしているプロセッサアーキテクチャの機能変更に対して低開発コストで対応できるようにする必要がある。

また、既知のアーキテクチャ依存部分をまとめる。

4.8.1 アーキテクチャ依存部分と非依存部分の分離方針

アーキテクチャ依存部分と非依存部分の分離方針を示す。

1. ディレクトリ構成

   アーキテクチャ依存部分と非依存部分は、ソースファイルを格納するディレクトリを分離する。即ち、アーキテクチャ非依存 (アーキテクチャ共通) のソースファイルは単一のディレクトリに集約し、アーキテクチャ依存のソースファイルは各アーキテクチャ毎に別個のディレクトリを作成して個々のディレクトリに配置する。

2. ヘッダファイル

   アーキテクチャ依存の宣言とアーキテクチャ非依存の宣言は分離して別個のヘッダファイルに記述する。アーキテクチャ依存のヘッダファイルは各アーキテクチャ毎のディレクトリに配置し、アーキテクチャ非依存のヘッダファイルはアーキテクチャ非依存のディレクトリに配置する。

   アーキテクチャ非依存のソースファイルからアーキテクチャ依存のヘッダファイルをインクルード可能とする。このため、アーキテクチャ非依存のソースファイルからインクルードするヘッダファイルは、全てのアーキテクチャが用意しなければならない。但し、予めアーキテクチャ依存のソースファイルからしかインクルードされないことが明らかなヘッダファイルはこの限りではない。

   全てのヘッダファイルにおいて、アーキテクチャによる条件コンパイル (`#ifdef ARCH` など) を原則禁止する。

3. ソースファイル

   アーキテクチャ依存の処理とアーキテクチャ非依存の処理は分離して個別のソースファイルに記述する。アーキテクチャ依存のソースファイルは各アーキテクチャ毎のディレクトリに配置し、アーキテクチャ非依存のソースファイルはアーキテクチャ非依存のディレクトリに配置する。

   全てのソースファイルにおいて、アーキテクチャによる条件コンパイル (`#ifdef ARCH` など) を原則禁止する。

4. アセンブラ

   アセンブラはアーキテクチャ依存なので、アーキテクチャ非依存のソースファイルにアセンブラを記述することはできない。アセンブラを使用する箇所は、別関数やマクロにしてアーキテクチャ依存のソースファイルやヘッダファイルに切り出す必要がある。また、アセンブラソースファイル (`.S`) はアーキテクチャ依存のディレクトリに配置されなければならない。

4.8.2 既知のアーキテクチャ依存部

代表的な既知のアーキテクチャ依存部分を示す。

1. カーネルのブートと初期化

   カーネルの実行を開始する際、CPU のモード変更やカーネル用 TLB の設定、割り込みの初期化などを行う必要がある。また、アプリケーションの実行に備えて、AP の起動と初期化を行う必要がある。

   アーキテクチャ毎に必要な処理が異なる。

2. 割り込み

   割り込み機構はアーキテクチャによって仕様が異なるため、割り込みの設定や割り込みハンドラの入口出口処理、ソフトウェア割り込みの発行処理 (IPI 含む) はアーキテクチャ依存である。

割り込み発生後の処理 (例えばシステムコール) はアーキテクチャ非依存の部分も多いため、全ての処理をアーキテクチャ依存とするのではなく、共通部分はアーキテクチャ非依存としてまとめること。

3. メモリ管理

   仮想アドレスを実現する TLB の構造や、キャッシュコヒーレンシ機構の制御はアーキテクチャ依存である。

   また、デマンドページングを実現するための割り込み処理もアーキテクチャ依存である。

4. 同期排他制御

   SMP 環境で高速に同期排他制御を実現する機構をハードウェアや CPU が備えている場合が多いが、利用方法はアーキテクチャ依存である。

5. レジスタ

   CPU が備えるレジスタ類 (汎用レジスタ、浮動小数点レジスタ、SIMD 系レジスタ、制御レジスタ、デバッグレジスタ、パフォーマンスカウンタ、タイムスタンプカウンタなど) は CPU によって実装されるレジスタの種類や個数が異なるため、アーキテクチャ依存である。

   同一アーキテクチャであっても、CPU の世代によって仕様が異なる場合があるので、必要に応じてレジスタの有無や個数などを調べる処理を行うこと。

6. コンテキスト

   コンテキストにはレジスタ情報やスタックなどのアーキテクチャ依存のリソースを含む。また、カーネルコンテキストからユーザコンテキストへの切り替えなどのもアーキテクチャ依存の方法で実装する必要がある。更に、コンテキストを別な CPU コアに移動するマイグレーションでは、アーキテクチャ依存の IPI を使用する必要がある。

   しかしながら、これらのアーキテクチャ依存の部分を除いたコンテキスト管理処理はアーキテクチャ非依存とすることができるため、コンテキストを抽象化して極力アーキテクチャに依存しない形で実装すること。

7. システムコール

   システムコール番号はアーキテクチャによって異なる。

   また、一部のシステムコールはアーキテクチャによってインタフェースが異なる (`rt_sigaction`、`rt_sigreturn` など)。

8. シグナル

   割り込みをシグナルとしてアプリケーションに通知したり、シグナルの送付に IPI を使用する場合があるので、シグナルの処理はアーキテクチャ依存処理を含む。また、シグナルハンドラの呼び出しでは、シグナルハンドラ処理後にシグナル発生元に復帰可能とするために、シグナル発生時のユーザコンテキスト をユーザスタックに退避する必要があるが、コンテキストやスタックの構造はアーキテクチャ依存である。

しかしながら、これらのアーキテクチャ依存の部分を除いたシグナル処理はアーキテクチャ非依存とすることができるため、シグナルやコンテキストを抽象化して極力アーキテクチャに依存しない形で実装すること。

第5章　Formal Verification for McKernel

McKernel は、アプリ開発者に OS を意識させないという目的、すなわち McKernel 上で動作するアプリは Linux のツールチェインを用いて作成できるようにする、また再コンパイルの必要なく Linux の OS サービスを受けられるようにする、という目的を持つ。このために Linux API 互換性をテストする。また、McKernel 開発のコストを下げるため、ソースコードに修正を加えた際に自動的にコンパイル、機能テスト、性能テストを行うシステムを準備する。以下ではそれぞれについて、テストの範囲と方法、環境、実施計画を説明する。

5.1　形式手法による仕様明確化と形式検証

McKernel を含むオペレーティングシステムはアプリケーションプログラムと比べると、通常のテストで除去しづらい問題を抱える可能性が高い。例えば、spin lock を用いた他 cpu 処理との排他によるデッドロックが起こる。すなわち、あるスレッドがあるロック変数を用いてロックをかけた後、割り込み処理を行うスレッドが同一コアに起動され、同一のロック変数を用いてスピンロックによりロックをかけようとし、デッドロックする。また、割り込みマスク操作による割り込み処理との排他を忘れることによる誤動作が起こる。すなわち、割り込み処理を行うスレッドが同一コアに起動されると共有資源の同時操作などによる誤動作が起こるコードセクションにおいて、割り込みマスクを操作してこれを防ぐべきところを忘れたことによる誤動作が起こる。このような誤動作が露呈する実行パスは、時分割実行の特定の組み合わせでのみ現れるため、LTP のようなブラックボックス的なテストで検出することは難しく、不可能では無いにしてもコスト的にも見合わない。

この課題に対処する方法として仕様言語を用いて関数の動作を記述する方法がある。この方法は、開発者の関数の動作の理解を助けることで開発者がそのような問題を回避する可能性を高めるという効果と、モデルチェッカを用いて問題が存在しないことを検証できるという二つの効果によって、上記課題を解決する。McKernel のバージョン 1.0 では第 1 の効果を実現する。仕様言語には、ANSI/ISO C Specification Language および科学技術振興機構 CREST「実用化を目指した組込みシステム用ディペンダブル・オペレーティングシステム」研究領域（DEOS プロジェクト） で定義された OK カーネル向けアノテーション [1] をベースとし、必要に応じて表現を追加したものを使用する。この言語を用いて、関数の指定したステップにおいて指定した条件が成立する／成立すべき、ということがらを記述する。

5.1.1　表現

McKernel の仕様記述のために用いる表現のうち主なものを説明する。⟨ 引数の変数を表す表現 ⟩、`\result`、`\atomicity`、`requires` ⟨ 条件式 ⟩、`ensures` ⟨ 条件式 ⟩ 以外は仕様言語に対し追加したものである。

⟨**引数の変数を表す表現**⟩ 引数を表す。表現は C 言語の表現を用いる。

\result 返値を表す。

\interrupt_disabled 1 以上のときプロセッサが割り込み禁止状態にあることを表し、0 のときプロセッサが割り込み可能状態にあることを表す。

⟨**グローバル変数を表す表現**⟩ グローバル変数を表す。表現は C 言語の表現を用いる。

\process_env 1 以上のとき実行がユーザコンテキストであることを表し、0 のときそうでないことを表す。

\atomicity 1 以上のとき実行のブロックにつながる動作をしてはいけないことを表し、0 のときそうでないことを表す。

\dont_call_schedule 1 以上のときコンテキストスイッチにつながる動作をしてはいけないことを表し、0 のときそうでないことを表す。

is_locked(⟨**ポインタを表す表現**⟩) ポインタで示されたメモリブロックがロック状態である場合は真を、そうでない場合は偽に評価される。⟨ ポインタを表す表現 ⟩ は C 言語の表現を用いる。

requires ⟨**条件式**⟩ 関数に入った直後に成立すべき条件を表す。

ensures ⟨**条件式**⟩ 関数から出る直前に成立すべき条件を表す。

invariant ⟨**条件式**⟩ 関数に入った直後から関数から出る直前の全てのステップで成立すべき条件を表す。

5.1.2 記述するソースコードの範囲

また、記述するソースコードの範囲は、以下の 2 つとする。

1. アーキ非依存部とアーキ依存部の境目のインターフェイス宣言部
2. アノテーションによって動作がわかりやすくなる以下の関数

   alloc_pages 引数によってエラー時の対処法の切り替えがあるため

関連図書

[1] H. Fujita, M. Matsuda, T. Maeda, S. Miura, and Y. Ishikawa. P-Bus: Programming Interface Layer for Safe OS Kernel Extensions. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 235–236, 2010.

[2] Hermann Härtig et al. L4linux. http://os.inf.tu-dresden.de/L4/LinuxOnL4/.

[3] Sandia National Laboratories. High Performance Computing Power Application Programming Interface (API) Specification. http://powerapi.sandia.gov.

[4] A. Shimada, B. Gerofi, A. Hori, and Y. Ishikawa. Proposing a new task model towards many-core architecture. In *Proceedings of the First International Workshop on Many-core Embedded Systems*. ACM, 2013.

[5] A. Shimada, A. Hori, and Y. Ishikawa. Eliminating costs for crossing process boundary from mpi intra-node communication. In *EuroMPI/ASIA*, 2014.

[6] T. Shimosawa. *Operating System Organization for Manycore Systems*. PhD thesis, Computer Science Department, University of Tokyo, 3 2012.

[7] T. Shimosawa, B. Gerofi, M. Takagi, G. Nakamura, T. Shirasawa, Y. Saeki, M. Shimizu, A. Hori, and Y. Ishikawa. Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures. *In Proc. of IEEE International Conference on High Performance Computing (HiPC)*, 2014.