

McKernel Specifications for Fujitsu Developers
Version 1.7.0rc1

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

Saturday 30th September, 2017

Contents

1	インターフェイス	11
1.1	プロセス起動コマンド	12
1.2	ダンプ採取・解析	14
1.2.1	ダンプ解析コマンド	14
1.2.2	ダンプ形式変換コマンド	15
1.3	高速プロセス起動ライブラリインターフェイス	15
1.3.1	MPI プロセス開始再開コマンド	17
1.3.2	MPI プロセス終了指示コマンド	18
1.3.3	計算の再開・終了関数 (C 言語)	19
1.3.4	計算の再開・終了関数 (Fortran)	19
1.4	高速プロセス起動カーネルインターフェイス	20
1.4.1	swapout システムコール	20
1.5	Utility Thread Offloading ライブラリインターフェイス	21
1.6	Utility Thread Offloading カーネルインターフェイス	21
1.6.1	McKernel スレッドの Linux へのマイグレートシステムコール	21
1.6.2	スレッド生成先 OS 指定システムコール	23
1.6.3	カーネル種別取得システムコール	23
1.7	XPMEM ライブラリインターフェイス	24
1.7.1	Get Version Number	24
1.7.2	Expose Memory Block	25
1.7.3	Un-Expose Memory Block	25
1.7.4	Get Access Permit	26
1.7.5	Release Access Permit	26
1.7.6	Attach to Memory Block	27
1.7.7	Detach from Memory Block	27
1.8	XPMEM カーネルインターフェイス	28
1.8.1	ioctl システムコール	28
2	実装者向けインターフェイス詳細	31
2.1	概要	31
2.2	プロセス管理	33
2.2.1	Linux からのプロセス起動	34
2.2.2	fork()	35
2.2.3	Files and the File Descriptor Table	35
2.2.4	Signal Handling	36
2.2.5	Process ID	36
2.2.6	Thread ID	37

2.2.7	User ID	37
2.2.8	Process Groups	37
2.3	システムコール	37
2.3.1	System Call Offloading	37
2.3.2	Offloading Strategy	38
2.3.3	gettimeofday()	38
2.3.4	perf_event_open()	39
2.4	Memory Management	39
2.4.1	Unified Address Space	39
2.4.1.1	McKernel Process Virtual Address Mapping	41
2.4.2	Physical Pages requiring Linux Management	42
2.4.3	Handling Different Page Sizes	42
2.4.4	brk()	42
2.4.5	メモリ割り当てにおける NUMA ノード選択	43
2.4.5.1	ユーザメモリ割り当て	43
2.4.5.2	カーネルメモリ割り当て	43
2.4.6	Virtual Dynamic Shared Object (vDSO)	43
2.4.7	ファイルマップ	44
2.4.8	POSIX Shared Memory	45
2.4.9	System V 共有メモリ	45
2.4.9.1	実装の制限	45
2.5	procfs/sysfs	45
2.5.1	ファイルシステムの重ね合わせ	48
2.5.2	アクセス要求の Linux から McKernel への転送	48
2.6	ファイルシステム重ね合わせ	49
2.6.1	詳細	50
2.6.2	実装の制限	55
2.6.3	開発時の留意事項	55
2.7	デバイスドライバ	55
2.7.1	Linux ドライバの利用	55
2.7.2	McKernel 内部での実装	56
2.8	XPMEM ドライバ	56
2.8.1	XPMEM デバイスファイルのオープン	60
2.8.2	XPMEM デバイスファイルの ioctl 制御	60
2.8.3	XPMEM デバイスファイルのクローズ	61
2.8.4	XPMEM の初期化	62
2.8.5	XPMEM の終了	62
2.8.6	xpmem_segment の生成	62
2.8.7	xpmem_segment の破棄	63
2.8.8	xpmem_access_permit の生成	63
2.8.9	xpmem_access_permit の破棄	64
2.8.10	xpmem_attachment の生成	64
2.8.11	xpmem_attachment の破棄	65
2.8.12	vm_range の fault 処理	65
2.8.13	vm_range の削除	66
2.9	ライブラリ切り替え	67
2.10	状態監視	67
2.11	Non-Maskable Interrupt	68

2.11.1	NMI 動作設定	68
2.11.2	NMI 送信	69
2.11.3	NMI ハンドラ	69
2.12	全 CPU 一時停止	69
2.12.1	一時停止指示 (IHK-master core)	72
2.12.2	一時停止からの復帰指示 (IHK-master core)	72
2.12.3	一時停止指示 (IHK-master driver)	73
2.12.4	一時停止からの復帰指示 (IHK-master driver)	73
2.12.5	一時停止および一時停止からの復帰指示	73
2.12.6	NMI ハンドラからの復帰時の指定関数へのジャンプ設定	74
2.12.7	一時停止指示 (ラッパー)	74
2.12.8	一時停止指示	74
2.13	カーネルダンプ	75
2.13.1	全体の処理の流れ	75
2.13.2	ユーザメモリ領域情報取得	79
2.13.3	未使用メモリ領域情報取得	80
2.13.4	ダンプ処理用 ioctl() コマンド	80
2.13.4.1	ダンプファイルの形式	81
2.13.5	ダンプ解析コマンドと gdb コマンドとの連携方法	82
2.13.6	ダンプ形式変換 (crash プラグイン)	83
2.13.7	利用時の留意事項	84
2.14	プロセスダンプ	85
2.14.1	実装の制限	85
2.15	Utility Thread Offloading	86
2.15.1	スレッドマイグレート処理	87
2.15.2	システムコール処理	88
2.15.3	シグナル受信処理	88
2.15.3.1	シグナル送信処理	88
2.15.3.2	mcexec のシグナルハンドラ処理	88
2.15.4	スレッド終了処理	89
2.15.5	実装詳細	89
2.15.5.1	Linux CPU へのスレッド生成の構成	89
2.15.5.2	util_indicate_clone システムコール	91
2.15.5.3	util_migrate_inter_kernel システムコール	91
2.15.5.4	get_system システムコール	92
2.15.5.5	clone システムコール	92
2.15.5.6	schedule の処理	92
2.15.5.7	enter_user_mode の処理	92
2.15.5.8	auto_utilthr_migrate の処理	92
2.15.5.9	do_syscall の処理	93
2.15.5.10	mcexec の処理	93
2.15.5.11	mcctrl の処理	95
2.15.6	実装の制限	99
2.16	高速プロセス起動	99
2.16.1	詳細	100
2.16.2	MPI プロセス起動指示コマンド	104
2.16.3	MPI プロセス終了指示コマンド	106
2.16.4	MPI 実行環境初期化関数 (C 言語)	106

2.16.5	MPI 実行環境初期化関数 (fortran)	107
2.16.6	計算の再開・終了関数 (C 言語)	108
2.16.7	計算の再開・終了関数 (Fortran)	108
2.16.8	初期化関数	108
2.16.9	計算ノードの管理サーバ	109
2.16.10	指示中継コマンド	110
2.16.11	swapout システムコール	111
2.17	Portability	114
2.18	Formal Verification	115
2.18.1	Specification Language	115
2.19	Limitations	117
3	【削除】 運用ガイド	119
3.1	【変更】 インターフェイス	119
3.1.1	カーネル引数	119
3.1.2	ブートスクリプト	120
3.1.3	シャットダウンスクリプト	122
3.1.4	プロセス起動コマンド	122
3.1.5	【削除】 統計情報取得	122
3.1.6	ダンプ解析コマンド	123
3.1.7	ダンプ形式変換コマンド	123
3.2	ブート手順	124
3.3	シャットダウン手順	133
	Bibliography	137

List of Figures

1.1	McKernel software stack	11
2.1	The architecture of McKernel	32
2.2	McKernel Usages	33
2.3	Overview of the IHK/McKernel architecture and the system call delegation mechanism.	34
2.4	シグナル中継処理の動作	36
2.5	Unified Address Space	40
2.6	procfs/sysfs のアクセス要求転送	49
2.7	nocopyupw オプションの有無によるライト処理の違い	50
2.8	Linux ドライバ利用の動作	55
2.9	XPMEM のメモリマッピング	56
2.10	XPMEM の動作フロー	57
2.11	XPMEM のデータ構造を生成・破棄する関数	58
2.12	XPMEM のデータ構造	59
2.13	構成要素関連図	70
2.14	全 CPU 一時停止および一時停止からの復帰のフロー	71
2.15	McKernel 主導ダンプの場合のダンプ採取機能とダンプ形式変換機能の処理の流れ	76
2.16	Linux 主導ダンプの場合のダンプ採取機能とダンプ形式変換機能の処理の流れ	78
2.17	ユーザメモリ領域情報取得処理の流れ	79
2.18	未使用メモリ領域情報取得処理の流れ	80
2.19	ダンプファイルの objdump での出力例	82
2.20	ダンプ形式変換処理の流れ	84
2.21	Linux CPU へのスレッド生成の構成	90
2.22	プロセス構成	101
2.23	関連コマンドと関連プロセスの動作フロー	102
2.24	スワップアウトの処理フロー	112
2.25	スワップインの処理フロー	114

List of Tables

1.1	XPMMEM デバイスに対する ioctl の各コマンドの処理	28
2.1	System calls implemented in McKernel	38
2.2	vDSO pages related to <code>gettimeofday()</code>	39
2.3	/proc files provided by McKernel	46
2.4	/sys files provided by McKernel	47
2.5	<code>mcoverlayfs</code> のマウントオプション	50
2.6	<code>overlayfs</code> の関数に対する修正 (1)	53
2.7	<code>overlayfs</code> の関数に対する修正 (2)	54
2.8	マイグレートされたスレッドが発行するシステムコールの処理	88
2.9	プロセス一覧	100
2.10	Limitations of McKernel	117
3.1	McKernel のカーネル引数	120

Chapter 1

インターフェイス

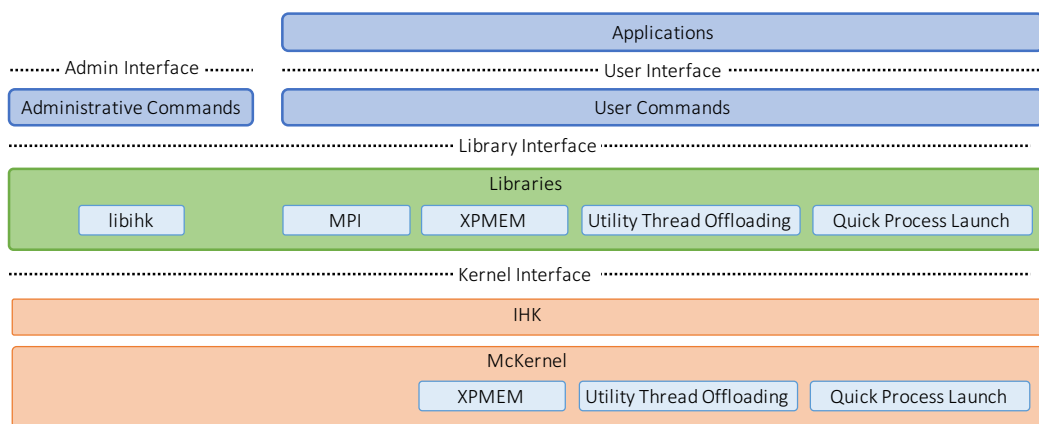


Figure 1.1: McKernel software stack

McKernel のソフトウェアスタックを図 1.1 に示す。本章では、ユーザ向けの User Interface と、アプリ向けの Library Interface と、コマンドやライブラリ向けの Kernel Interface を説明する。

本章の想定読者は、以下の 3 種類のユーザまたは開発者である。

- McKernel のコマンドを用いてアプリを実行するユーザ
- McKernel のライブラリインターフェイスを使用してアプリを開発する開発者
- McKernel のカーネルインターフェイスを使用してライブラリを開発する開発者

ユーザインターフェイス、ライブラリインターフェイスの関連ファイルは以下の通り。なお、インストールディレクトリを<install>とする。

インストール先	インターフェイス	説明
<install>/bin/mcexec	ユーザ	プロセス起動コマンド
<install>/bin/eclair	ダンプ解析ツール	
<install>/bin/vmcore2mckdump	ダンプ形式変換ツール	
<install>/rootfs/usr/lib64/libuti.so	ライブラリ	Utility Thread Offloading ライブラリ
<install>/include/uti.h	ライブラリ	Utility Thread Offloading ライブラリ ヘッダファイル
<install>/include/qlmpilib.h	ライブラリ	高速プロセス起動ヘッダファイル
<install>/lib/libqmpi.so	ライブラリ	高速プロセス起動ライブラリ
<install>/lib/libqfort.so	ライブラリ	高速プロセス起動ライブラリ (Fortran プログラム用)
<install>/lib/libxpmem.so	ライブラリ	XPMEM ライブラリ
<install>/include/xpmem.h	ライブラリ	XPMEM ライブラリヘッダファイル

以下、これら3種のインターフェイスを説明する。

1

1.1 プロセス起動コマンド

2

書式

3

```

mcexec [-c <cpu_id>] [-n <nr_partitions>] [-t <nr_threads>]
[-M (--mpol-threshold=<min>)] [-h (--extend-heap-by=<stride>)]
[-s (--stack-premap=<premap_size>)[,<max>]] [--mpol-no-heap] [--mpol-no-bss]
[--mpol-no-stack] [--mpol-shm-premap] [-m <numa_node>] [--disable-sched-yield]
[-O] [<os_index>]
<program> [<args>...]

```

4

5

6

7

8

9

オプション

10

11

<code>-c <cpu_id></code>	<code>mcexec</code> を実行する CPU の番号を <code><cpu_id></code> に設定する。指定がない場合は 0 が用いられる。
<code>-n <nr_partitions></code>	1 計算ノードの CPU 群を <code><nr_partitions></code> の区画に分割し、第 i 番目に起動された <code>mcexec</code> プロセスから起動される McKernel スレッドが第 i 番目の区画のみを利用するように設定する。分割は物理コア単位で行われる。こうすることで、1 ノード <code><nr_partitions></code> プロセスの MPI+OpenMP 実行において CPU を適切に使い分けることができる。
<code>-t <nr_threads></code>	<code>mcexec</code> のスレッド数を <code><nr_threads></code> に設定する。このオプションが指定されない場合は、 <code>OMP_NUM_THREADS</code> 環境変数が定義されている場合はその値+4 に設定し、存在しない場合は McKernel に割り当てられた CPU 数+4 に設定する。 <code>mcexec</code> スレッドは McKernel からの要求を処理する。同時に多くの要求がなされる可能性があるため、この数は <code><McKernel のスレッド数 + α></code> に設定する必要がある。
<code>-M (--mpol-threshold=<min>)</code>	<code><min></code> 以上のサイズのメモリを要求したときのみ、ユーザが設定したメモリ割り当てポリシーが適用されるようにする。 <code><min></code> は K, M, G (k, m, g でもよい) の単位を付けた場合、それぞれ KiB, MiB, GiB の指定になる。指定がない場合はサイズに関係なくユーザが設定したメモリ割り当てポリシーが適用される。
<code>-h (--extend-heap-by=<step>)</code>	ヒープの拡大時にヒープサイズを少なくとも <code><step></code> バイト拡大する。また、ヒープの終了アドレスをラージページサイズにアラインする。 <code><step></code> は K, M, G (k, m, g でもよい) の単位を付けた場合、それぞれ KiB, MiB, GiB の指定になる。指定がない場合は 4 KB が用いられる。
<code>-s (--stack-premap=<premap_size>,<max>)</code>	プロセス生成時にスタック領域のうち <code><premap_size></code> バイトをプリマップする。また、スタックの最大サイズを <code><max></code> に設定する。 <code><premap_size></code> , <code><max></code> は K, M, G (k, m, g でもよい) の単位を付けた場合、それぞれ KiB, MiB, GiB の指定になる。指定がない場合、 <code><premap_size></code> は 2 MB、 <code><max></code> は <code>ulimit</code> コマンドまたは <code>setrlimit()</code> システムコールで設定された値が用いられる。
<code>--mpol-no-heap</code>	ヒープへのメモリ割り当て時にユーザの設定したメモリ割り当てポリシーに従わない。
<code>--mpol-no-stack</code>	スタックへのメモリ割り当て時にユーザの設定したメモリ割り当てポリシーに従わない。
<code>--mpol-no-bss</code>	bss へのメモリ割り当て時にユーザの設定したメモリ割り当てポリシーに従わない。
<code>--mpol-shm-premap</code>	<code>/dev/shm</code> を用いた共有メモリをプリマップする。
<code>-m <numa_node></code>	メモリを <code><numa_node></code> 番目の NUMA ノードから割り当てる。割り当てが不可能な場合は他の NUMA ノードから割り当てる。
<code>--disable-sched-yield</code>	<code>sched_yield()</code> 関数を何も行わない関数に置き換える。
<code>-0</code>	McKernel に割り当てられた CPU 数より大きい数のスレッドまたはプロセスの生成を許可する。指定がない場合は許可しない。許可されていない場合に、CPU 数より大きい数のスレッドまたはプロセスを <code>clone()</code> , <code>fork()</code> , <code>vfork()</code> など生成しようとすると、当該システムコールが <code>EINVAL</code> エラーを返す。
<code><os_index></code>	プロセス起動先 OS インスタンスを <code><os_index></code> 番に設定する。省略した場合は 0 番の OS インスタンスに起動する。

1 説明

2 `<program>` で指定された実行可能ファイルを `args` で指定された引数で、McKernel 上に起
3 動する。

4 `mcexec` の動作を変える環境変数は以下の通り。

書式	説明
MCEXEC_WL=<path1> [:<path2>...]	<path1>, <path2>, ... 以下に存在する McKernel 用実行ファイルについて、 <code>mcexec</code> の指定を省略する。なお、指定ディレクトリ以下に実行可能ファイルが存在しても、以下のケースでは Linux で実行される。 <ul style="list-style-type: none"> ● McKernel が動作していない場合 ● コマンドが 64 ビット ELF バイナリではない場合 ● コマンド名が <code>mcexec</code>, <code>ihkosctl</code>, <code>ihkconfig</code> である場合
MCEXEC_ALT_ROOT=<path>	<code>ld-linux.so</code> などのローダを探す際に、<path>と実行可能ファイルの <code>.interp</code> セクションに記載されたパスを結合したパスを探す。
MCKERNEL_RLIMIT_STACK= <premap_size>, <max>	(非推奨) プロセス生成時にスタック領域のうち<premap_size>バイトをプリマップする。また、スタックの最大サイズを<max>に設定する。<premap_size>, <max>は K, M, G (k, m, g でもよい) の単位を付けた場合、それぞれ KiB, MiB, GiB の指定になる。指定がない場合、<premap_size>は 2 MB、<max>は <code>ulimit -s</code> コマンドまたは <code>setrlimit()</code> システムコールで設定された値が用いられる。なお、本環境変数の代わりに <code>mcexec</code> の <code>--stack-premap</code> オプションを使用することを推奨する。

使用例は以下の通り。この例では `ls -ls` を McKernel 上で実行する。

```
$ mcexec ls -ls
```

戻り値

<program>の exit status を返す。

1.2 ダンプ採取・解析

カーネルダンプの採取と解析のステップは以下の通り。

- 以下のいずれかの方法でダンプファイルを作成する。
 - IHK の関数 `ihk_os_makedumpfile()` または IHK のコマンド `ihkosctl` を用いて、McKernel 形式のダンプファイルを作成する。
 - Linux の `panic` を契機に `makedumpfile` 形式のダンプファイルを作成する。また、コマンド `vmcore2mckdump` を用いて McKernel 形式に変換する。
- `eclair` と呼ぶコマンドを用いてダンプファイルを解析する。

以下、関連コマンドのインターフェイスを説明する。

1.2.1 ダンプ解析コマンド

書式

```
eclair [-ch] [-d <dump>] [-k <king>] [-o <os_index>] [-l] [-i]
```

オプション

-c	NMI 受付時のコンテキストをスレッドとして扱う。それぞれのコンテキストは 1000000+ (CPU 番号) という TID を持つスレッドとして扱われる。スレッドとして扱うことで、割り込み処理のバックトレースを表示することができる。
-h	利用法を表示する。
-d <dump>	ダンプファイル名を指定する。指定がない場合は mcdump が用いられる。
-k <king>	カーネルイメージファイル名を指定する。指定がない場合は kernel.img が用いられる。
-o <os_index>	OS インスタンスのインデックスを指定する。指定がない場合は 0 が用いられる。
-l	run-queue にユーザスレッドが存在しない CPU について、idle() を実行しているスレッドが存在するように見せかける。
-i	Interactive mode と呼ぶ、デバッグ対象マシンに存在するメモリを直接参照した解析を行う。なお、ダンプ時に interactive mode を指定する必要がある。

1 説明

2 <dump>で指定された eclair 形式のダンプファイルを<os_index>で指定された OS インデックスを持つ OS として、<king>で指定されたカーネルイメージファイルを使って解析する。
3
4 ダンプ解析コマンド内では、gdb が動作しており、gdb と同じコマンドを利用できる。
5 McKernel は、マルチスレッドの単一プロセスに見える。まず、最初に、以下のコマンドを実行して、ダンプ解析コマンドにスレッド一覧を覚えさせる必要がある。

7 (eclair) info threads

8 quit コマンド実行時に、inferior の切り離し許可をユーザに求める。これには、y と応答すること。ダンプ解析コマンドは、gdb のコマンドの、bt コマンドと x コマンドをサポートする。
9

10 1.2.2 ダンプ形式変換コマンド

11 書式

12 vmcore2mckdump <vmcore> <file_name>

13 オプション

<vmcore>	makedumpfile 形式のダンプファイルのファイル名
<file_name>	変換先ダンプファイルのファイル名

14

15 説明

16 <vmcore>で指定された makedumpfile 形式のダンプファイルから McKernel に関連する部分を取り出し<file_name>で指定されたファイルに eclair 形式で出力する。
17

18 1.3 高速プロセス起動ライブラリインターフェイス

19 McKernel は、複数種の MPI プログラムを起動しさらにそれを繰り返すジョブにおいて MPI
20 プログラム起動時間を短縮する機能を提供する。利用例は以下の通り。

- 21 ● アンサンブルシミュレーションとデータ同化を繰り返す気象アプリケーション
22 このアプリではジョブスクリプトでそれぞれの MPI プログラムを交互に起動する。この起動時間を短縮する。
23

本機能を利用するためにはジョブスクリプトとアプリケーションを修正する必要がある。
ジョブスクリプトの修正方法を例を用いて説明する。

修正前

```
/* アンサンブルシミュレーションと同化を 10 回繰り返す */
for i in {1..10}; do

    /* 100 ノードを用いるアンサンブルシミュレーションを 10 個並列に動作させる */
    for j in {1..10}; do
        mpiexec -n 100 -machinefile ./list1_$j p1.out a1 & pids[$i]=$!;
    done

    /* p1.out の終了を待つ */
    for j in {1..10}; do wait ${pids[$j]}; done

    /* アンサンブルシミュレーションで用いたのと同じ 1000 ノードを用いてデータ同化
    を行う */
    mpiexec -n 1000 -machinefile ./list2 p2.out a2
done
```

修正後

```
for i in {1..10}; do
    for j in {1..10}; do
        /* mpiexec を ql_mpiexec_start に置き換える */
        ql_mpiexec_start -n 100 -machinefile ./list1_$j p1.out a1 & pids[$j]=$!;
    done

    for j in {1..10}; do wait ${pids[$j]}; done

    ql_mpiexec_start -n 1000 -machinefile ./list2 p2.out a2
done

/* p1.out と p2.out は常駐しているため、ql_mpiexec_finalize で終了させる。
mpiexec への引数と実行可能ファイル名で MPI プログラムを識別しているため、
実行時と同じものを指定する。 */
for j in {1..10}; do
    ql_mpiexec_finalize -machinefile ./list1_$i p1.out a1;
done
ql_mpiexec_finalize -machinefile ./list2 p2.out a2;
```

アプリケーションの修正方法を擬似コードを用いて説明する。計算を何度も行えるようなループ構造を持たせ、また ql_client() を計算完了後に呼び出すようにする。

```
MPI_Init();
先行・後続 MPI プログラムとの通信準備
loop:
foreach (Fortran の) モジュール
    コマンドライン引数・パラメタファイル・環境変数を用いた初期化処理
先行 MPI プログラムからのデータ受信・スナップショット読み込み
計算
後続 MPI プログラムへのデータ送信・スナップショット書き出し
/* ループボディの終わりに ql_client() を挿入する */
```



```

1   if(ql_client() == QL_CONTINUE) { goto loop; }
2   MPI_Finalize();

```

3 以下、コマンドや関数のインターフェイスを説明する。

4 1.3.1 MPI プロセス開始再開コマンド

5 書式

```

6   ql_mpiexec_start -machinefile <hostfile> [<mpiopts>...] <exe> [<args>...]

```

7 オプション

オプション	内容
-machinefile <hostfile>	ホストファイル
<mpiopts>	mpiexec のオプション
<exe>	実行可能ファイル
<args>	実行可能ファイルの引数

8

9 説明

10 <exe>で指定された MPI プログラムを開始する。または再開指示待ちの状態にある MPI
11 プログラムに次の計算開始を指示する。本コマンドは MPI プログラムの一回の計算の完了
12 と共に終了する。また、MPI プログラムは<hostfile>の内容、<mpiopts>、<exe>とで識別
13 する。

14 ql_mpiexec_{start,finalize} コマンドから MPI プログラムに次の動作、引数、環境変
15 数を渡すために用いるファイルは、環境変数 QL_PARAM_PATH が定義されている場合はその下に、
16 そうでない場合はホームディレクトリ下に作成される。当該ディレクトリは ql_mpiexec_start
17 コマンドを実行するノード、各 MPI プロセスが実行される計算ノードからアクセスできる必
18 要がある。

19 また、環境変数 MPIEXEC_TIMEOUT によるタイムアウトおよび複数の実行可能ファイルの
20 指定はサポートしない。

21 戻り値

戻り値	説明
0	正常終了
0 以外	異常終了

22

23 エラー時出力

24 エラーメッセージは mpiexec が出力するエラーメッセージの他に ql_mpiexec_start 独自
25 に以下のメッセージを出力する。

メッセージ	意味
unknown option: <opt>	未知のオプション <opt>が指定された
bad option: <opt>	オプション <opt>の指定が誤っている
unsupported option: <opt>	オプション <opt>はサポートしていない
'.' is unsupported	'.' はサポートしていない
unable to read hostfile(<hostfile>): <reason>	<hostfile>を<reason>の理由により読み込めない
could not open hostfile(<hostfile>): <reason>	<hostfile><reason>の理由によりオープンできない
<hostfile> not exist	<hostfile>が存在しない
specify -machinefile option	-machinefile オプションが指定されていない
no user program	<exe>が指定されていない
socket directory not exist	ソケット通信用のディレクトリが存在しない
ql_server not execution <reason>	ql_server の起動に<reason>の理由により失敗した
ql_mpiexec_start: socket(<reason>)	ql_mpiexec_start コマンドの socket 操作で<reason>のエラーが発生した
ql_mpiexec_start: bind(<reason>)	ql_mpiexec_start コマンドの bind で<reason>のエラーが発生した
ql_mpiexec_start: listen(<reason>)	ql_mpiexec_start コマンドの listen で<reason>のエラーが発生した
ql_mpiexec_start: connect(<reason>)	ql_mpiexec_start コマンドの connect で<reason>のエラーが発生した

1.3.2 MPI プロセス終了指示コマンド

書式

```
ql_mpiexec_finalize -machinefile <hostfile> [<mpiopts>...] <exe>
```

オプション

オプション	説明
-machinefile <hostfile>	ホストファイル
<mpiopts>	mpiexec のオプション
<exe>	実行可能ファイル

説明

ql_mpiexec_start によって起動された MPI プログラムを終了させる。本コマンドは MPI プログラムの終了と共に終了する。また、MPI プログラムは<hostfile>の内容、<mpiopts>、<exe>とで識別する。

戻り値

戻り値	説明
0	正常終了
0 以外	異常終了

1 エラー時出力

2 エラーメッセージは `mpiexec` が出力するエラーメッセージの他に `ql_mpiexec_finalize` 独自に以下のメッセージを出力する。

メッセージ	意味
unknown option: <opt>	未知のオプション<opt>が指定された
bad option: <opt>	オプション<opt>の指定が誤っている
unsupported option: <opt>	オプション<opt>はサポートしていない
'.' is unsupported	'.' はサポートしていない
unable to read hostfile(<hostfile>): <reason>	<hostfile>を<reason>の理由で読み込めない
could not open hostfile(<hostfile>): <reason>	<hostfile>を<reason>の理由でオープンできない
<hostfile> not exist	ホストファイルが存在しない
specify -machinefile option	-machinefile オプションが指定されていない
no user program	<exe>が指定されていない
socket directory not exist	ソケット通信用のディレクトリが存在しない
not found mpi process	mpiexec プロセスが存在しない

3

4 **1.3.3 計算の再開・終了関数 (C 言語)**

5 書式

6 `ql_client(int *argc, char ***argv)`

7 引数

引数	説明
<code>argc</code>	引数の数へのポインタ
<code>argv</code>	引数文字列の配列へのポインタ

8

9 説明

10 `ql_mpiexec_{start,finalize}` コマンドによる指示を待ち、指示結果を返す。本関数は、
11 MPI プログラム内で、一回の計算の完了後に呼び出す。

12 戻り値

戻り値	説明
<code>QL_CONTINUE</code>	次の計算の開始が指示された
<code>QL_EXIT</code>	MPI プログラムの終了が指示された、あるいは当該プロセスが <code>ql_mpiexec_start</code> コマンドで起動されていない

13

14 **1.3.4 計算の再開・終了関数 (Fortran)**

15 書式

16 `subroutine QL_CLIENT(ierr)`

引数	型	説明
ierr	INT	戻り値

2

説明

3

MPI プログラム内で一回の計算の完了後に呼び出され、`ql_mpiexec_{start,finalize}` コマンドによる指示を待ち、指示結果を返す。なお、本関数を使用するためには `libqlfort.so` を `LD_PRELOAD` でロードする必要がある。また、コンパイラは GNU Fortran Compiler または Intel Fortran Compiler をサポートする。Intel Fortran Compiler 使用時は、コンパイルオプションに `-shared-intel` を指定する必要がある。

4

5

6

7

8

戻り値

9

戻り値	説明
1	次の計算の開始が指示された
0	MPI プログラムの終了が指示された、あるいは当該プロセスが <code>ql_mpiexec_start</code> コマンドで起動されていない

10

1.4 高速プロセス起動カーネルインターフェイス

11

1.4.1 swapout システムコール

12

書式

13

```
int swapout(char *filename, void *workarea, size_t size, int flag)
```

14

引数

15

引数	説明
filename	スワップファイル名へのポインタ
workarea	作業領域へのポインタ
size	作業領域のサイズ
flag	swapout の動作制御用フラグ

16

説明

17

プロセスのメモリ領域のファイルへの待避（スワップアウトと呼ぶ）とファイルからの復元（スワップインと呼ぶ）を行う。

18

19

処理ステップは以下の通り。

20

1 filename が NULL または flag が 1 の場合はステップ 6 に進む。そうでない場合はステップ 2 に進む。

21

22

- 1 2 スワップアウト処理を行う。
- 2 3 flag が2の場合は、ステップ5に進む。そうでない場合は、ステップ4に進む。
- 3 4 mexecへ制御を移し、スワップアウト完了の同期と、`ql_mpiexec_{start,finalize}`
- 4 による指示を待った後、本システムコールに制御を戻す。
- 5 5 スワップイン処理を行う。さらに呼び出し元に戻る。
- 6 6 mexecへ制御を移し、`ql_mpiexec_{start,finalize}`による指示を待った後、本シス
- 7 テムコールに制御を戻す。さらに呼び出し元に戻る。

8 戻り値

戻り値	説明
0	正常終了
-1	エラー
-ENOMEM	メモリが不足
-EINVAL	引数が不正

9

10 1.5 Utility Thread Offloading ライブラリインターフェイス

11 インターフェイスは「McKernel仕様付録（Utility thread offloading ライブラリ編）」に記載
12 する。

13 1.6 Utility Thread Offloading カーネルインターフェイス

14 McKernelは、スレッドをLinuxのCPUにマイグレートする機能（Utility Thread Offloading,
15 UTIと呼ぶ）を提供する。UTIのカーネルインターフェイスは、第1.5節で説明するライブ
16 ラリによって用いられる。

17 以下、関連システムコールのインターフェイスを説明する。

18 1.6.1 McKernelスレッドのLinuxへのマイグレートシステムコール

19 書式

```
20 int util_migrate_inter_kernel(uti_attr_t *attr)
```

21 説明

22 attrと環境変数UTI_CPU_SETで指定された、CPU位置とスレッドの振る舞いの記述に基
23 づき、呼び出し元スレッドをLinux CPUにマイグレートさせる。

24 環境変数UTI_CPU_SETはビットマップ形式でCPU位置を示す。また、uti_attr_tは以
25 下のように定義される。

```
26 #define UTI_MAX_NUMA_DOMAINS (1024)
27
28 typedef struct uti_attr {
29     uint64_t numa_set[(UTI_MAX_NUMA_DOMAINS + 63) / 64];
```

```

/* スレッド配置先 NUMA ノードを表すビットマップ */
uint64_t flags;
/* CPU 位置とスレッドの振る舞いを表すビットマップ */
} uti_attr_t;

```

uti_attr_t の flags はビットマップで、ビット 1 は対応する CPU 位置の指示または振る舞いの記述が有効であることを示す。ビット位置と指示・記述の対応は以下の通り。

```

#define UTI_FLAG_NUMA_SET (1ULL<<1)
/* numa_set フィールドで指定した NUMA ノードへ配置する */
#define UTI_FLAG_SAME_NUMA_DOMAIN (1ULL<<2)
/* 呼び出し元と同一 NUMA ノードへ配置する */
#define UTI_FLAG_DIFFERENT_NUMA_DOMAIN (1ULL<<3)
/* 呼び出し元とは異なる NUMA ノードへ配置する */
#define UTI_FLAG_SAME_L1 (1ULL<<4)
#define UTI_FLAG_SAME_L2 (1ULL<<5)
#define UTI_FLAG_SAME_L3 (1ULL<<6)
/* 呼び出し元とそれぞれのレベルのキャッシュを共有する CPU へ配置する */
#define UTI_FLAG_DIFFERENT_L1 (1ULL<<7)
#define UTI_FLAG_DIFFERENT_L2 (1ULL<<8)
#define UTI_FLAG_DIFFERENT_L3 (1ULL<<9)
/* 呼び出し元とそれぞれのレベルのキャッシュを共有しない CPU へ配置する */
#define UTI_FLAG_EXCLUSIVE_CPU (1ULL<<10)
/* CPU を専有させると効率的に動作する。
   例えば、mwait 命令を用いている。*/
#define UTI_FLAG_CPU_INTENSIVE (1ULL<<11)
/* CPU サイクルを多く使用する。例えば、ネットワーク
   デバイスのイベントキューを繰り返しポーリングする。*/
#define UTI_FLAG_HIGH_PRIORITY (1ULL<<12)
/* スケジューラのプライオリティを上げると効率的に動作する。
   例えば、ネットワークデバイスのイベント待ちをする。*/
#define UTI_FLAG_NON_COOPERATIVE (1ULL<<13)
/* co-operative スケジューリングを行っていない。例えば、
   イベント待ちになった際に sched_yield() を呼ぶ、ということをしなない。*/

```

なお、McKernel から Linux への 1 度のマイグレートのみ可能である。

戻り値

0	正常終了
-1	エラー

エラー時の errno の値

-ENOSYS	util_migrate_inter_kernel がサポートされていない。
-EFAULT	attr にアクセスできない。

1 1.6.2 スレッド生成先 OS 指定システムコール

2 書式

```
3 int util_indicate_clone(int mod, uti_attr_t *attr)
```

4 説明

5 呼び出し元スレッドが発行する clone システムコールの動作を変え、スレッド生成後
6 直ちに mod に指定した OS へマイグレートさせる。CPU 位置と Linux のスケジューラ設
7 定は、attr と環境変数 UTI_CPU_SET で指定されたヒントに基づいて決定される。この関
8 数は、pthread_create() などで Linux へスレッドを生成させるために用いる。本関数も、
9 util_migrate_inter_kernel と同様、McKernel から Linux への 1 度のマイグレートのみ可
10 能である。

mod の取りうる値と意味は以下の通り。

SPAWN_TO_REMOTE	Linux へ生成
SPAWN_TO_LOCAL	McKernel へ生成

11

12 戻り値

0	正常終了
-1	エラー

13

14 エラー時の errno の値

ENOSYS	util_indicate_clone がサポートされていない。
EINVAL	mod に未定義の値を指定した。
EFAULT	attr にアクセスできない。

15

16 1.6.3 カーネル種別取得システムコール

17 書式

```
18 int get_system()
```

19 説明

20 呼び出し元スレッドが動作している OS の種別を返却する。なお、本関数の名称は次バー
21 ジョンにて is_mckernel() 等に変更される予定である。

22 戻り値

0	OS が McKernel
-1	エラー (OS が Linux)

ENOSYS	Linux で呼び出した
--------	--------------

1.7 XPMEM ライブラリインターフェイス

XPMEM を使うことによって、あるプロセスがマップしたメモリ領域を他のプロセスからマップできるようになる。利用方法は以下の通り。第 1 のプロセスのメモリ領域を第 2 のプロセスがマップしようとしているとする。

1. 第 1 のプロセスがメモリ領域を `xpmem_make()` を用いて XPMEM segment として登録する。また、segment id を第 2 のプロセスに渡す。
2. 第 2 のプロセスが `xpmem_get()` で当該 XPMEM segment に対するアクセス許可を得る。
3. 第 2 のプロセスが `xpmem_attach()` で当該 XPMEM segment を自身の仮想アドレス範囲にマップする。
4. 第 2 のプロセスが当該メモリ領域に対する操作を行う。
5. 第 2 のプロセスが `xpmem_detach()` で当該メモリ領域をアンマップする。
6. 第 2 のプロセスが `xpmem_release()` で当該 XPMEM segment に対するアクセス許可が不要になったことをドライバに伝える。
7. 第 1 のプロセスが `xpmem_remove()` を用いて当該 XPMEM segment を破棄する。

以下、関連関数のインターフェイスを説明する。

1.7.1 Get Version Number

書式

```
int xpmem_version (void)
```

説明

This function gets the XPMEM version.

戻り値

$\neq -1$	XPMEM version number
-1	Failure

1 1.7.2 Expose Memory Block

2 書式

3

```
4 xpmem_segid_t xpmem_make(  
5     void *vaddr,  
6     size_t size,  
7     int permit_type,  
8     void *permit_value)
```

9 説明

10 `xpmem_make()` shares a memory block specified by `vaddr` and `size` by invoking the
11 XPMEM driver. `permit_type` is for the future extension. Use `XPMEM_PERMIT_MODE` for this
12 version. `permit_value` specifies the permissions mode expressed as an octal value.

13 This function is expected to be called by the source process to obtain a segment ID
14 to share with other processes. It is common to call this function with `vaddr = NULL` and
15 `size = XPMEM_MAXADDR_SIZE`. This will share the entire address space of the calling process.

16 戻り値

$\neq -1$	64-bit segment ID (<code>xpmem_segid_t</code>)
-1	Failure

17

18 1.7.3 Un-Expose Memory Block

19 書式

20

```
21 static int xpmem_remove(xpmem_segid_t segid)
```

22 説明

23 The opposite of `xpmem_make()`, this function deletes the mapping specified by `segid`
24 that was created from a previous `xpmem_make()` call. All the attachments created by
25 `xpmem_attach()` are detached and all the permits obtained by `xpmem_get()` are revoked.

26 Optionally, this function is called by the source process, otherwise automatically called
27 by the driver when the source process exits.

28 戻り値

0	Success
-1	Failure

29

1.7.4 Get Access Permit 1

書式 2

3

```
xpmem_apid_t xpmem_get(4  
    xpmem_segid_t segid,5  
    int flags,6  
    int permit_type,7  
    void *permit_value)8
```

説明 9

`xpmem_get()` attempts to get access to a shared memory block specified by `segid`. `flags` specifies access mode, i.e. read-write (`XPMEM_RDWR`) or read-only (`XPMEM_RDONLY`). `permit_type` is for the future extension. Use `XPMEM_PERMIT_MODE` for this version. `permit_value` specifies the permissions mode expressed as an octal value. 10
11
12
13

This function is called by the consumer process to get permission to attach memory from the source virtual address space associated with `segid`. If access is granted, an `apid` will be returned to pass to `xpmem_attach()`. 14
15
16

戻り値 17

<code>≠ -1</code>	64-bit access permit ID (<code>xpmem_apid_t</code>)
<code>-1</code>	Failure

18

1.7.5 Release Access Permit 19

書式 20

21

```
int xpmem_release(xpmem_apid_t apid)22
```

説明 23

The opposite of `xpmem_get()`, this function deletes any mappings associated with `apid` in the consumer's address space. Optionally, this function is called by the consumer process, otherwise automatically called by the driver when the consumer process exits. 24
25
26

戻り値 27

<code>0</code>	Success
<code>-1</code>	Failure

28

1 1.7.6 Attach to Memory Block

2 書式

3

```
4 static int xpmem_attach(  
5     struct xpmem_addr addr,  
6     size_t size,  
7     void *vaddr)
```

8 説明

9 This function attaches a virtual address space range from the source process.
10 struct xpmem_addr is defined as follows.

```
11 struct xpmem_addr {  
12     /** apid that represents memory */  
13     xpmem_apid_t apid;  
14     /** offset into apid's memory region */  
15     off_t offset;  
16 };
```

17 addr.apid is the access permit ID returned from a previous xpmem_get() call. addr.offset
18 is offset into the source memory to begin the mapping. The mapping is created at vaddr
19 with the size of size. Kernel chooses the mapping address if vaddr is NULL.

20 This function is called by the consumer to get a mapping between the shared source
21 address and an address in the consumer process' own address space. If the mapping is
22 successful, then the consumer process can now begin accessing the shared memory.

23 戻り値

≠ -1	Virtual address at which the mapping was created
-1	Failure

24

25 1.7.7 Detach from Memory Block

26 書式

27

```
28 int xpmem_detach(void *vaddr)
```

29 説明

30 This function detach from the virtual address space of the source process.

31 Optionally, this function is called by the consumer process, otherwise automatically
32 called by the driver when the consumer process exits.

33 戻り値

34

0	Success
-1	Failure

1.8 XPMEM カーネルインターフェイス

XPMEM は、あるプロセスがマップしたメモリ領域を他のプロセスからマップできるようにする。XPMEM のカーネルインターフェイスは、第 1.7 節で説明するライブラリによって用いられる。

以下、関連する `ioctl()` のインターフェイスを説明する。

1.8.1 `ioctl` システムコール

書式

```
int ioctl(int fd, int cmd, void* arg)
```

説明

`cmd` で指定された操作を行う。`cmd` ごとの処理を表 1.1 に示す。

Table 1.1: XPMEM デバイスに対する `ioctl` の各コマンドの処理

コマンド	説明
<code>XPMEM_CMD_VERSION</code>	バージョン番号を返す。
<code>XPMEM_CMD_MAKE</code>	<code>arg->vaddr</code> から始まる長さ <code>arg->size</code> のメモリ領域を共有可能にし、segment id を <code>arg->segid</code> に格納する。メモリ領域のパーミッションは <code>arg->permit_value</code> に設定される。
<code>XPMEM_CMD_REMOVE</code>	<code>arg->segid</code> で指定されたメモリ領域の共有を解除する。
<code>XPMEM_CMD_GET</code>	<code>arg->segid</code> で指定されたメモリ領域に対する <code>arg->permit_value</code> で指定されたパーミッションでのアクセス許可取得を試みる。成功した場合、アクセス許可の id が <code>arg->apid</code> に格納される。
<code>XPMEM_CMD_RELEASE</code>	<code>arg->apid</code> で指定されたメモリ領域に対するアクセス許可を返却する。
<code>XPMEM_CMD_ATTACH</code>	<code>arg->apid</code> で指定された共有メモリ領域のうち <code>arg->offset</code> から始まる長さ <code>arg->size</code> の範囲を <code>arg->vaddr</code> から始まるアドレス範囲にマップする。
<code>XPMEM_CMD_DETACH</code>	<code>arg->vaddr</code> から始まる共有マップを解放する。

`XPMEM_CMD_MAKE` コマンドで用いる `xpmem_cmd_make` 構造体は以下のように定義される。

```
struct xpmem_cmd_make {
    __u64 vaddr;
    size_t size;
    int permit_type;
    __u64 permit_value;
    xpmem_segid_t segid;    /* returned on success */
};
```

`xpmem_segid_t` は以下のように定義される。

```
typedef __s64 xpmem_segid_t;    /* segid returned from xpmem_make() */
```

`XPMEM_CMD_REMOVE` コマンドで用いる `xpmem_cmd_remove` 構造体は以下のように定義される。

```

1 struct xpmem_cmd_remove {
2     xpmem_segid_t segid;
3 };

```

4 XPMEM_CMD_GET コマンドで用いる `xpmem_cmd_get` 構造体は以下のように定義される。

```

5 struct xpmem_cmd_get {
6     xpmem_segid_t segid;
7     int flags;
8     int permit_type;
9     __u64 permit_value;
10    xpmem_apid_t apid;    /* returned on success */
11 };

```

12 `xpmem_apid_t` は以下のように定義される。

```

13 typedef __s64 xpmem_apid_t;    /* apid returned from xpmem_get() */

```

14 XPMEM_CMD_RELEASE コマンドで用いる `xpmem_cmd_release` 構造体は以下のように定義される。

```

16 struct xpmem_cmd_release {
17     xpmem_apid_t apid;
18 };

```

19 XPMEM_CMD_ATTACH コマンドで用いる `xpmem_cmd_attach` 構造体は以下のように定義される。

```

21 struct xpmem_cmd_attach {
22     xpmem_apid_t apid;
23     off_t offset;
24     size_t size;
25     __u64 vaddr;
26     int fd;
27     int flags;
28 };

```

29 XPMEM_CMD_DETACH コマンドで用いる `xpmem_cmd_detach` 構造体を以下のように定義される。

```

31 struct xpmem_cmd_detach {
32     __u64 vaddr;
33 };

```

34 XPMEM_CMD_ATTACH コマンドで用いる `xpmem_addr` 構造体は以下のように定義される。

```

35 struct xpmem_addr {
36     xpmem_apid_t apid;    /* apid that represents memory */
37     off_t offset;    /* offset into apid's memory */
38 };

```

39 戻り値

40

0	正常終了
-EFAULT	アドレスが不正である
-EINVAL	引数が無効である

1 Chapter 2

2 実装者向けインターフェイス詳細

3 本章の想定読者は以下の通り。

- 4 • McKernel の、アーキテクチャ移植を含む開発を行う開発者

5 2.1 概要

6 McKernel is a lightweight kernel for HPC with the following features.

- 7 • Quickly adapts to the new hardware techniques to provide scalability and full-control
8 of hardware
- 9 • Supports new programming style such as in-situ data analytics and scientific work-flow
- 10 • Provides a complete set of Linux API

11 McKernel is based on a light-weight kernel developed at the University of Tokyo[2]. It
12 works with systems with Intel Xeon processors and systems with Intel Xeon phi processor.
13 Figure 2.1 shows the architecture of McKernel. Cores and memory of a compute-node are
14 divided into two partitions and Linux runs on one of them and McKernel runs on the other.

15

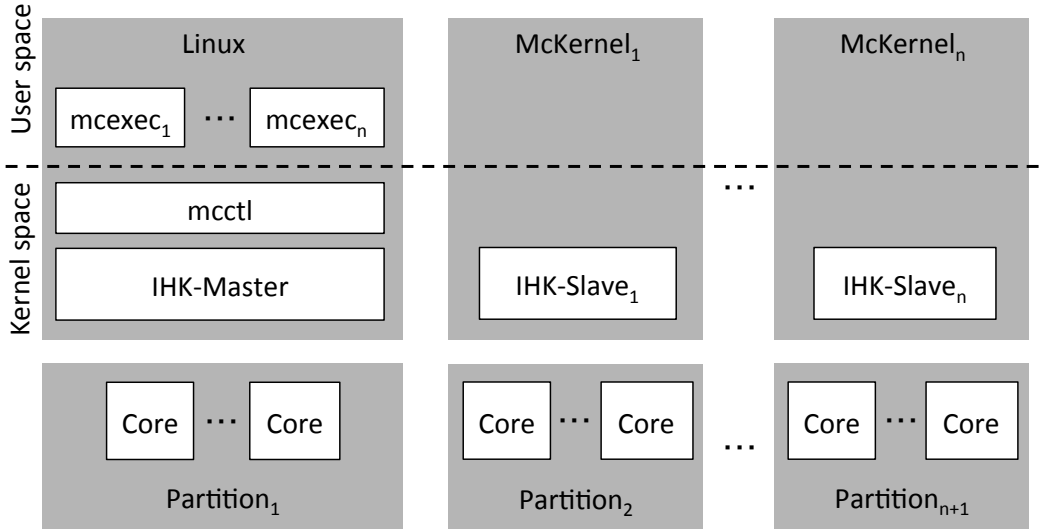


Figure 2.1: The architecture of McKernel

Two kernel modules, `mcctl` and `IHK-Master`, and user processes `mcexec` (`mcexec1`, `mcexec2`, ...) exist in the Linux kernel while McKernel (`McKernel1`, `McKernel2`, ...) and `IHK-Slave` (`IHK-Slave1`, `IHK-Slave2`, ...) reside in each partition.

Linux controls all hardware resources when booting a compute-node. The Interface for Heterogeneous Kernel, formed by both `IHK-Master` and `IHK-Slave`, implements a communication mechanism between Linux and McKernel, called Inter Kernel Communication (IKC). In addition of that, the `IHK-Master` has an important role, allocating cores and memory for McKernel, and booting it. `IHK` is independently designed from McKernel, and it may be used for other kernels with Linux.

The `mcctl` kernel module controls the McKernel. In order to provide Linux API for applications running on McKernel, OS service requests not provided by McKernel is delegated to Linux and performed by Linux. The `mcexec` command requests McKernel to launch an application via `IHK`. After the application's invocation, a `mcexec` process acts as a proxy or ghost process for the McKernel process in the sense that Linux system calls delegated from McKernel via `IHK` are issued by this process.

In the rest of this section, McKernel features, i.e., McKernel usages, process and memory management, system calls, and the `procfs/sysfs` file system will be described.

McKernel を用いたジョブの実行ステップを図 2.2 を用いて説明する。

1. 運用ソフトが計算ノード上に Linux を起動する (図の (1))
2. ユーザがジョブキューを指定することで、McKernel と Linux のどちらを使用するか、また McKernel を使用する場合は様々なチューニングが施されたカーネルイメージのうちどれを使用するかを指定する。例えば、ラージページ化が効果のあるアプリ B を実行しようとしている場合は、その機能を持つイメージを指定するジョブキューにジョブを投入する。
3. 運用ソフトウェアがジョブ投入を受けて、資源のパーティショニング、McKernel の起動、アプリの実行を行う (図の (2))。例では、ラージページ化促進機能を持つ McKernel が起動され、アプリ B がその上で実行される。

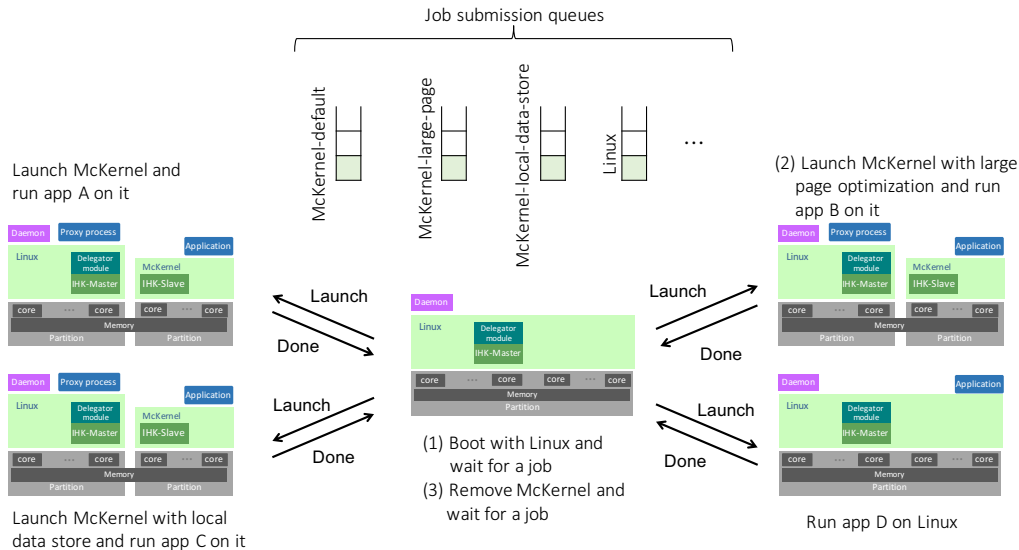


Figure 2.2: McKernel Usages

- 1 4. 運用ソフトウェアが、ジョブ終了時に計算ノード状態を元の状態、すなわち Linux のみ
- 2 が動作する状態に戻す (図の (3))

3 2.2 プロセス管理

4 McKernel has a unique process execution model to realize cooperation with Linux. McKernel
 5 processes are primarily spawn by the Linux command line tool `mcexec`¹. For every single
 6 McKernel process there is a corresponding `mcexec` Linux process that exists throughout the
 7 lifetime of the application. `mcexec` serves the following purposes:

- 8 - It provides an execution context for offloaded system calls (explained in Section 2.3)
- 9 so that they can be invoked directly in Linux
- 10 - It enables transparent access to Linux device drivers through the mechanism of unified
- 11 address-space (discussed in Section 2.4) and the ability to map Linux device files
- 12 directly to McKernel processes
- 13 - It facilitates Linux to maintain certain application associated kernel state that would
- 14 have to be otherwise maintained by McKernel (e.g., open files and the file descriptor
- 15 table (see Section 2.2.3), process specific device driver state, etc.)

16 Due to its role to providing a gateway to specific Linux features, we call `mcexec` the
 17 *proxy-process*. Figure 2.3 provides an overview of IHK/McKernel’s proxy-process architec-
 18 ture as well as the system call offloading mechanism.

¹An alternative way of creating McKernel processes via the `fork()` system call will be discussed in Section 2.2.2.

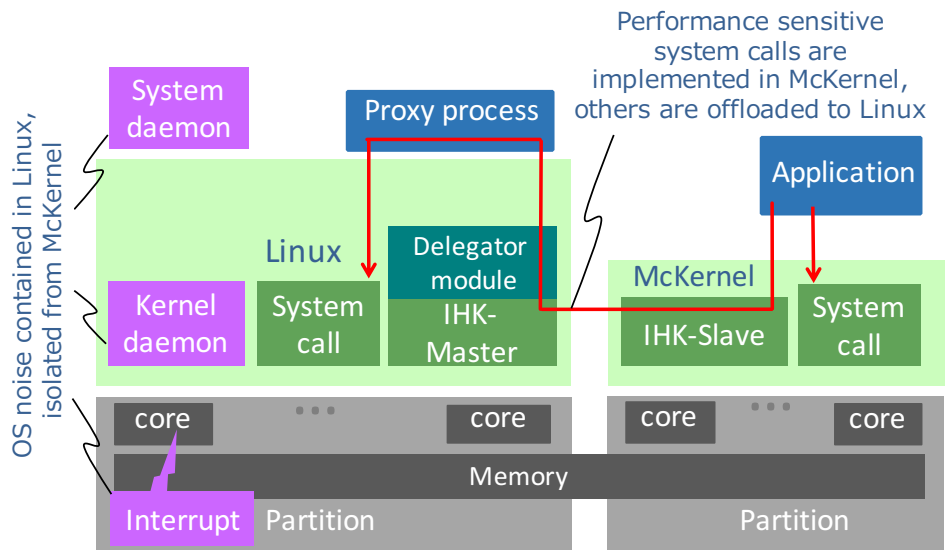


Figure 2.3: Overview of the IHK/McKernel architecture and the system call delegation mechanism.

We emphasize that IHK/McKernel runs HPC applications primarily on the LWK but the full Linux API is available via system call delegation. System call offloading will be detailed in Section 2.3.

Since the user shell process runs on the Linux side, a signal to an McKernel process cannot be delivered directly from Linux. Instead, the shell process issues signals to `mcexec` and `mcexec` forwards the signal to the McKernel process via IKC. For more information on signaling, see Section 2.2.4.

2.2.1 Linux からのプロセス起動

`mcexec` が Linux からプロセスを起動するステップは以下の通り。

1. It opens the device `/dev/mcosn` to communicate with McKernel.
2. It sends the ELF binary description header, the command line and environment variables to the McKernel.
3. It uploads the application binary to McKernel's memory area.
4. It creates a Linux thread pool that will serve system call offloading requests. Additionally, one of the workers is designated for waiting for signals from McKernel.
5. It sends a request for starting the process to McKernel.
6. The main thread waits for termination of all workers.
7. When a worker receives the `exit_group()` system call, it terminates all workers in the thread pool.

なお、環境変数 `MCEXEC_WL` に McKernel 用実行可能ファイルの（親）ディレクトリを指定することで、`mcexec` の指定を省略できる。複数ディレクトリを指定する場合は、コロンを

1 デリミタとして指定する。なお、指定ディレクトリ以下に実行可能ファイルが存在しても、以
2 下のケースでは Linux で実行される。

- 3 • McKernel が動作していない場合
- 4 • コマンドが 64 ビット ELF バイナリではない場合
- 5 • コマンド名が `mcexec`, `ihkosctl`, `ihkconfig` である場合

6 この機能は、`mcctrl` が Linux のローダのリストに特別なローダを挿入することで実現される。

7 2.2.2 fork()

8 The `fork()` system call is supported in McKernel and it is an alternative way for spawning
9 new processes. `fork()` is handled as follows:

- 10 1. McKernel allocates a CPU core and memory for the child process.
- 11 2. McKernel creates information on process and virtual memory, and the user execution
12 context.
- 13 3. McKernel copies the parent memory to the child process. Note that the anonymous
14 memory areas such as text, data, bss, are copied without using copy-on-write technique
15 in the current implementation.
- 16 4. McKernel requests `mcexec` to perform a fork system call (i.e., to create a new proxy
17 process for the child) in Linux. `mcexec` executes the following steps:
 - 18 (a) `mcexec` issues the fork system call to create a new Linux process (call it the child
19 proxy).
 - 20 (b) The child proxy closes the device `/dev/mcosn` and reopens it again in order to
21 communicate with McKernel.
 - 22 (c) The child proxy creates the worker thread pool that serve the same role of the
23 parent process's worker threads.
 - 24 (d) The child proxy sends a reply message to McKernel.
- 25 5. When McKernel receives the reply message, it puts the child process into the run-
26 queue.
- 27 6. McKernel returns to its parent process with the child process ID.

28 2.2.3 Files and the File Descriptor Table

29 McKernel does not maintain file system related information (e.g., file caches) and file de-
30 scriptors are managed by the proxy process on Linux. When an McKernel process opens a
31 file, its file descriptor is created in the `mcexec` process and the number is merely returned
32 to the McKernel process.

33 It is worth noting that `mcexec` keeps the IHK device file open for communication with
34 McKernel. Because a file descriptor is an integer value, the IHK device could theoretically
35 be accessed from application code. In order to avoid such scenario, `mcexec` ensures that the
36 IHK device file cannot be accessed by application code.

2.2.4 Signal Handling

1

Two types of signals are considered: One is signals for the `mcexec` process. An example is the user sends a signal to the process from the shell. Another one is signals for a McKernel process, e.g., page fault signal caused by accessing wrong address in the McKernel process.

2

3

4

When the `mcexec` process receives a signal, that signal is transferred to the McKernel process via McKernel. When McKernel receives a signal for the McKernel process from the `mcexec` process during waiting for completion of a Linux system call, McKernel requests the `mcexec` process for aborting the system call execution.

5

6

7

8

図 2.4 を用いてシグナル中継機能の動作を説明する。ホスト OS の `mcexec` が受け取ったシグナルは、IKC を通じて McKernel に通知され、シグナル登録処理 (`do_kill`) に伝えられる。シグナル登録処理では、シグナルを表す `sig_pending` 構造体を作成し、シグナル送付先の `process` 構造体に登録する。ここで、シグナル送付先がスレッドの場合は `process` 構造体の `sigpending` に登録するが、スレッドを特定しないシグナルの場合は `process` 構造体の中のスレッド共通の `sigshared` の `sigpending` に登録する。他の事象により発生したシグナルも同様にシグナル登録処理 (`do_kill`) によって `process` 構造体にシグナルが登録される。シグナルを受信するプロセスを実行する CPU では、割り込み処理後やシステムコール処理後などのユーザ空間への切り替えのタイミングでプロセスに届いているシグナル (`process` 構造体に登録されている `sig_pending` 構造体) をチェック (`check_signal`) し、シグナルが届いている場合には、その処理を行う。シグナルの処理は、`process` 構造体の `sighandler` に従って行う。`sighandler` のシグナル番号の項目にシグナルハンドラが登録されている場合は、登録されているシグナルハンドラを呼び出す。シグナルを無視する場合は何もしない。それ以外の場合はプロセスを終了 (シグナルによる終了) する (但し、シグナル番号が `SIGCHLD` と `SIGURG` では、シグナルハンドラの登録が無い場合は無視される)。

9

10

11

12

13

14

15

16

17

18

19

20

21

22

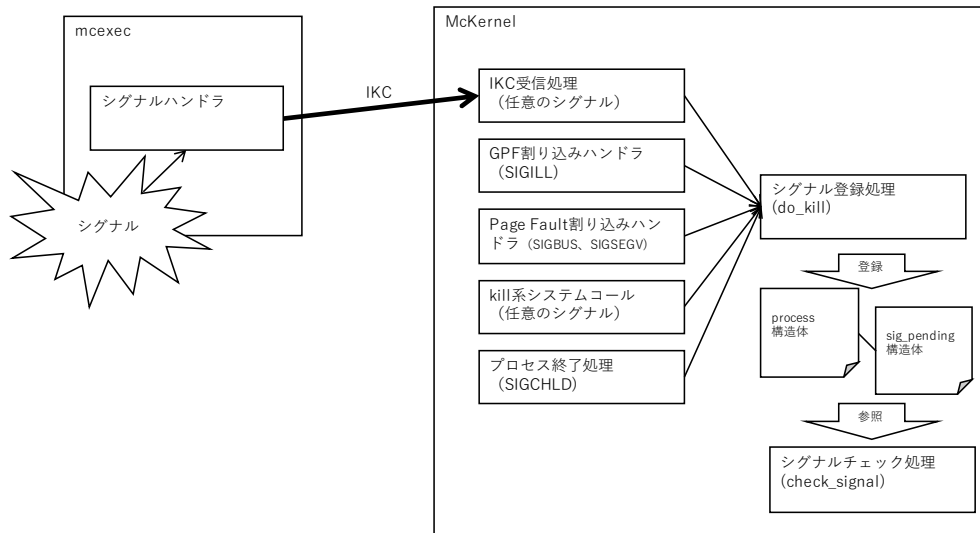


Figure 2.4: シグナル中継処理の動作

23

2.2.5 Process ID

24

The process ID of a McKernel process is held in the corresponding proxy process and it is managed via Linux API.

25

26

1 2.2.6 Thread ID

2 McKernel スレッドのスレッド ID は、対応する proxy process スレッドで管理される。McKernel
3 スレッド生成時の proxy process スレッドとの対応付けステップは以下の通り。

4 C1 proxy process (`mcexec`) は起動時に生成するスレッド数を決定し、その数だけ生成する。

5 C2 McKernel はスレッド生成時に、そのスレッドと対応付ける proxy process のスレッド
6 を proxy process に問い合わせる。

7 C3 McKernel は新しく生成する McKernel スレッドに当該 proxy process スレッドのスレッ
8 ド ID を割り当てる。また、McKernel はスレッド ID をキャッシュすることでスレッド
9 ID 問い合わせを高速化する。

10 `mcexec` は生成するスレッド数を以下の方法で決定する。

11 S1 `-t <nr_threads>` のオプションが指定された場合はその値を用いる。

12 S2 上記オプションが指定されなかった場合は、環境変数 `OMP_NUM_THREADS` が設定されてい
13 る場合は、環境変数の値を用いる。この環境変数が設定されていない場合は McKernel
14 に割り当てられた CPU 数を用いる。

15 McKernel のスレッド数上限は proxy process がステップ C1 で生成するスレッド数で決
16 まる。このためユーザは上記のステップ S2 で決定される数では足りない場合は `mcexec` の `-t`
17 `<nr_threads>` オプションを用いて十分な数を指定する必要がある。

18 2.2.7 User ID

19 UID 情報取得のオーバーヘッドを削減するため、UID は McKernel と Linux の両方で管理す
20 る。変更の際は McKernel 上の値を変更した後、IKC を用いて Linux 上の値を変更する。

21 2.2.8 Process Groups

22 プロセスグループにシグナルを送付する際のシグナル送付対象プロセス調査のオーバーヘッ
23 ドを削減するため、また、`setpgid` システムコールにおいて、対象プロセスが `execve` を実行
24 したか否かのチェックを行えるようにするため、`pgid` は Linux と McKernel の両方で管理す
25 る。変更の際は McKernel 上の値を変更した後、IKC を用いて Linux 上の値を変更する。

26 2.3 システムコール

27 As already mentioned, one of the proxy process' roles is to facilitate system call offloading
28 by providing an execution context on behalf of the application so that offloaded calls can
29 be directly invoked in Linux.

30 2.3.1 System Call Offloading

31 The main steps of system call offloading (also shown in Figure 2.3) are as follows. When
32 McKernel determines that a system call needs to be offloaded it marshalls the system
33 call number along with its arguments and sends a message to Linux via a dedicated IKC
34 channel. The corresponding proxy process running on Linux is by default waiting for system
35 call requests through an `ioctl1()` call into IHK's system call delegator kernel module. The

delegator kernel module’s IKC interrupt handler wakes up the proxy process, which returns to userspace and simply invokes the requested system call. Once it obtains the return value, it instructs the delegator module to send the result back to McKernel, which subsequently passes the value to user-space.

System call offloading internally relies on IHK’s Inter-Kernel Communication (IKC) facility. For more information on IKC, refer to “IHK Specifications”.

2.3.2 Offloading Strategy

There are mainly two categories of system calls that need to be implemented by McKernel:

1. System calls that cannot be offloaded to Linux side, and
2. Performance critical system calls

The first category includes CPU affinity system calls such as `sched_setaffinity()`, signaling system calls such as `sigaction()`, and memory-related system calls such as `mmap()` and `fork()`. The second category includes timer-related system calls such as `gettimeofday()`.

System calls, implemented in McKernel or planned to implement, is listed in Table 2.1. Other system calls are delated the Linux.

Table 2.1: System calls implemented in McKernel

Category	Implemented	Planned
Proess man- agement	arch_prctl (x86.64 specific), clone, execve, exit, exit_group, fork, futex, get_cpu_id, gete{u,g}id, get{g,p,t,u}id, getppid, getres{g,u}id, {get,set}rlimit, kill, pause, ptrace, rt_sigaction, rt_sigpending, rt_sigprocmask, rt_sigqueueinfo, rt_sigreturn, rt_sigsuspend, set_tid_address, setfs{u,g}id, set{g,u,t}id, setpgid, setre{g,u}id, setres{g,u}id, sigaltstack, tgkill, vfork, wait4, waittid	{get,set}_thread_area, rt_sigtimedwait, signalfd, signalfd4
Memory management	brk, {get,set}_mempolicy, madvise, mincore, mlock, mmap, move_pages, mprotect, mremap, msync, munlock, munmap, process_vm_{readv,writev}, remap_file_pages, shmat, shmctl, shmdt, shmget	{get,set}_robust_list, mbind, migrate_pages, mlockall, modify_ldt, munlockall
Schedule	getcpu, {get,set}itimer, {get,set}timeofday, nanosleep, sched_{get,set}affinity, sched_yield, times	
Performance counter	perf_event_open	

2.3.3 gettimeofday()

`gettimeofday()` is implemented in user-space by using Virtual Dynamic Shared Object (vDSO) mechanism (see Section 2.4.6 for vDSO).

Table 2.2 shows the related vDSO pages.

Table 2.2: vDSO pages related to `gettimeofday()`

Name	Description
<code>vdso</code>	System call code and data
<code>vvar</code>	Kernel variables
<code>hpet</code>	Register of the High Precision Event Timer
<code>pvti</code>	Virtual clock updated by virtual machine, such as Xen and KVM

1 2.3.4 `perf_event_open()`

2 `perf_event_open()` is implemented in McKernel by using the technique mentioned in Sec-
 3 tion 2.7.2.

4 2.4 Memory Management

5 We already described how system call offloading works in the IHK/McKernel architecture.
 6 Notice, however, that certain system call arguments may be pointers (e.g., the buffer argu-
 7 ment of a `read()` system call) and the actual operation takes place on the contents of the
 8 referred memory. Thus, the main problem is how the proxy process on Linux can resolve
 9 virtual addresses in arguments so that it can access the memory of the application running
 10 on McKernel.

11 In order to overcome this problem McKernel deploys a mechanism called *unified ad-*
 12 *dress space*, which essentially ensures that the proxy process can transparently access the
 13 same mappings as its corresponding McKernel process. This mechanism is detailed in the
 14 following sections.

15 2.4.1 Unified Address Space

16 The unified address space model in IHK/McKernel ensures that offloaded system calls can
 17 seamlessly resolve arguments even in case of pointers. This mechanism is depicted in Figure
 18 2.5 and it is implemented as follows. First, the proxy process is compiled as a position
 19 independent binary, which enables us to map the code and data segments specific to the
 20 proxy process to an address range which is explicitly excluded from McKernel’s user space.
 21 The box on the right side of the figure with label "Not used" demonstrates the excluded
 22 region. Second, the entire valid virtual address range of McKernel’s application user-space
 23 is covered by a special mapping in the proxy process for which we use a pseudo file mapping
 24 in Linux. This mapping is indicated by the yellow box on the left side of the figure.

25 Note, that the proxy process does not need to fill in any virtual to physical mappings
 26 at the time of creating the pseudo mapping and it remains empty unless an address is
 27 referenced. Every time an unmapped address is accessed, however, the page fault handler
 28 of the pseudo mapping consults the page tables corresponding to the application on the
 29 LWK and maps it to the exact same physical page. Such mappings are demonstrated in the
 30 figure by the small boxes on the left labeled as *faulted page*. This mechanism ensures that
 31 the proxy process, while executing system calls, has access to the same memory content
 32 as the application. Needless to say, Linux’ page table entries in the pseudo mapping have
 33 to be occasionally synchronized with McKernel, for instance, when the application calls
 34 `munmap()` or modifies certain mappings.

35 A more detailed sequence of resolving a page fault in Linux for an address in the
 36 McKernel process is as follows:

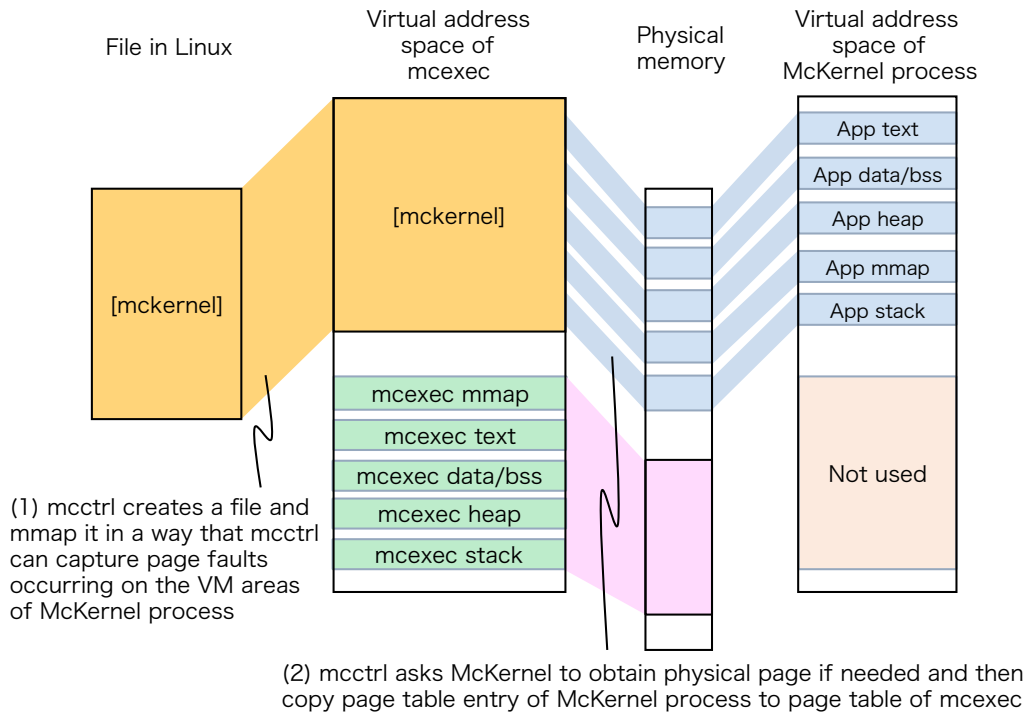


Figure 2.5: Unified Address Space

1. When `mcexec` accesses a memory area pointed by a pointer variable stored in a system call request a Linux page fault occurs. 1
2
2. The `mcctrl` kernel module captures this page fault. It looks up the page table of the McKernel process to find out the page table entry (PTE) of the physical memory. 3
4
3. In case that PTE is not found, the following sequences of issuing remote page fault are performed as follows. 5
6
 - (a) The `mcctrl` module interrupts the system call service. It reports return code `STATUS_PAGE_FAULT` and the faulting address to McKernel. 7
8
 - (b) When McKernel receives the return code `STATUS_PAGE_FAULT`, it resolves the page fault. 9
10
 - (c) After McKernel finishes page fault processing, it requests resuming the previous system call process by sending an IKC message `SCD_MSG_SYSCALL_ONESIDE` to `mcctrl`. 11
12
13
 - (d) When `mcctrl` receives the request of resuming the previous system call at the IKC message `SCD_MSG_SYSCALL_ONESIDE`, it looks up the page table entry again. 14
15
4. `mcctrl` maps the physical memory pointed by the PTE to the virtual address where the page fault occurred. 16
17
5. `mcctrl` requests resuming the execution of the `mcexec` process. 18
6. The `mcexec` process now can access the virtual address requested in the system call. 19

1 As mentioned above when an McKernel process releases physical pages by issuing sys-
2 tem calls such as `munmap()` or `madvise()` with the option `MADV_REMOVE`, the `mcexec` process
3 clears its page tables to make sure future requests will not resolve an invalid mapping.

4 When the `mcexec` process establishes the pseudo mapping covering the McKernel pro-
5 cess's user space the mapping is read/write enabled except for the text area of the McKernel
6 process. When the McKernel process allocates a read-only memory mapping, e.g., when
7 mapping a shared library, the `mcctrl` kernel module remaps this area with the same access
8 permissions in the Linux side. This remap operation is required because the virtual address
9 space for the McKernel process has been created as one contiguous region whose access per-
10 mission is homogeneous. Most of memory mappings created by the McKernel process are
11 read/write permission, and thus such remap operation happens relatively rarely.

12 **2.4.1.1 McKernel Process Virtual Address Mapping**

13 Theoretically all virtual addresses used in the McKernel process must be mapped to the
14 `mcexec` process's virtual address. There are two issues as follows:

- 15 1. The `mcexec` process has its own text, data and BSS area whose addresses are also
16 used in the McKernel process if those execution binaries have been created in the
17 same way.
- 18 2. If the huge stack area is allocated to `mcexec` via shell environment variable `RLIMIT_STACK`,
19 the virtual address space for the McKernel process cannot be assigned.

20 The solution of those issues on Linux for `x86_64` architectures is described as follows.

21 **2.4.1.1.1 Avoiding Conflict of text, data, and BSS**

22 In the Linux convention for `x86_64` architectures, the text segment starts from virtual ad-
23 dress `0x400000` and the data segment starts from 2 MiB upper address than the text seg-
24 ment. If both an McKernel application and `mcexec` are compiled and linked, those addresses
25 are conflict.

26 As we briefly mentioned above, the `mcexec` binary is created as position independent
27 binary so that each segment's address can be dynamically decided by the runtime. In
28 Linux convention for `x86_64` architectures, by issuing `mmap`, the map address will be the
29 next to the address of the stack area whose address is the highest address in the user address
30 space.

31 **2.4.1.1.2 Huge Stack Size**

32 The virtual address space plan of the McKernel process follows Linux address plan, i.e.,
33 the user space is contiguous and starts from virtual address 0. That is, in order to keep the
34 same address space of the McKernel process in the `mcexec`, the same address space must
35 not be occupied by the `mcexec` process. There is one problem to do so. In Linux for `x86_64`
36 architectures, the start address of a stack area is randomly decided and its size is the lesser
37 of $\frac{5}{6}$ total memory size and size specified by the `RLIMIT_STACK` environment variable. If the
38 huge stack occupies the virtual memory in the `mcexec`, there is no chance to reserve the address
39 space for the McKernel process. In order to eliminate this problem, the `RLIMIT_STACK`
40 environmental variable for `mcexec` and the McKernel process is separated. That is, the

mcexec checks if RLIMIT_STACK is larger than some amount of size (currently 1 GiB), it saves RLIMIT_STACK to a temporal environmental variable (MCKERNEL_RLIMIT_STACK) and exec() itself again with a small stack (10 MiB). The new mcexec process restores the original value to RLIMIT_STACK so that this environment variable is used for the McKernel process.

2.4.2 Physical Pages requiring Linux Management

The physical pages of a McKernel process must be under Linux management for I/O related operation (e.g. pin-down). This is because the driver running on the Linux side performs I/O operation and the operation relies on the Linux paging mechanism. For example, when a McKernel process tries to send data in a buffer to a remote host, it calls the Linux driver code and the driver code in turn pins down the physical pages for the buffer using the Linux kernel function. The kernel function in turn assumes that the pages are under Linux management (i.e. managed by struct page).

Thus, IHK takes the physical pages from physical pages managed by the Linux. That is, IHK reserves physical pages for the co-kernels by using `_get_free_pages()` Linux API. Since the `_get_free_pages()` allocates up to 1024 pages at a time, IHK repeatedly calls this function to get contiguous pages more than 1024 pages.

2.4.3 Handling Different Page Sizes

There are several implementation options to support different page sizes in Linux:

1. Linux Transparent Huge Pages (THP)
2. Hugepage option in System V IPC shared memory
3. Linux HugeTLBfs
4. Hugepage option in `mmap()` flags

McKernel implments a similar technique to Linux THP, i.e., it automatically maps physical memory with large pages whenever it is possible.

2.4.4 brk()

McKernel の `brk()` システムコールには、ページフォルトオーバーヘッドを削減し、またラージページ化を促進する機能が追加されている。

`brk()` の動作は以下の通り。なお、ヒープ終端アドレスを b 、`brk()` の引数をページ境界で丸め上げたアドレスを r で表す。また、プロセス起動コマンド `mcexec` (第 1.1 参照) のオプション (`--extend-heap-by=<step>`) で指定されたパラメタを S で表す。

1. ヒープの縮小が要求された場合、何もせずに戻る。
2. $r - b < S$ の場合、ヒープ終端アドレスを以下の x に設定する。

$$r + S \leq x < r + S + a, x \bmod a = 0, a = \begin{cases} 2^{12} & \text{if } S \leq 4096; \\ 2^{21} & \text{if } S > 4096. \end{cases}$$

3. $r - b \geq S$ の場合、ヒープの最終アドレスを r に設定する。

1 4. 拡張された部分をプリページングする。

2 なお、この機能は、同一計算ノード上に他ユーザのジョブが存在することはないので、物
3 理ページ利用のフェアネスを考慮する必要がないため、不要になった物理ページを OS に返
4 す必要がない、という HPC アプリの特性を用いている。

5 2.4.5 メモリ割り当てにおける NUMA ノード選択

6 2.4.5.1 ユーザメモリ割り当て

7 ユーザメモリ割り当てにおける NUMA ノード選択については、Linux の機能に対し以下の機
8 能が追加されている。

- 9 1. ヒープ、anonymous mmap 領域だけではなく、text, data, bss, stack の各領域に対しても
10 プロセスのメモリポリシーを用いる。また、これらの領域ごとにプロセスのメモリポ
11 リシーを用いるか否かを指定できる。この指定は、プロセス起動コマンド `mcexec` (第
12 1.1 参照) のオプション (`--mpol-no-{heap,stack,bss}`) によって行う。
- 13 2. ユーザ指定のメモリポリシーを用いるメモリ要求サイズの閾値 (この値と同じか大きい
14 場合のみユーザ指定のメモリポリシーを用いる) を指定できる。この指定は、プロセス
15 起動コマンド `mcexec` (第 1.1 参照) のオプション (`--mpol-threshold=<min>`) によっ
16 て行う。

17 2.4.5.2 カーネルメモリ割り当て

18 カーネルメモリ割り当てにおける NUMA ノード選択は Linux と同様の方法で行う。すなわ
19 ち、NUMA ノード間の距離行列を用いて、要求元が存在する NUMA ノードから最も距離の
20 短い NUMA ノードからメモリを取得する。

21 2.4.6 Virtual Dynamic Shared Object (vDSO)

22 Mckernel provides the vDSO mechanism, which eliminates the need for switching to kernel-
23 mode when performing some system calls.

24 The steps are in the followings.

- 25 1. The physical addresses of the Linux vDSO pages are compiled by looking into `System.map`
26 when configuring McKernel. They are kept in `mcctrl`.
- 27 2. McKernel adds mappings of the Linux vDSO pages to a process when creating the
28 process. McKernel asks `mcctrl` for their physical addresses.
- 29 3. McKernel passes their virtual addresses to the process via the Auxiliary Vector in the
30 stack.
- 31 4. When a system call is called, first the control is transferred to the `glibc` wrapper
32 function. And then the control is transferred to the function in the vDSO pages
33 without switching processor mode.
- 34 5. The function performs required processing using the data in the vDSO pages.

2.4.7 ファイルマップ

ファイルマップはファイルと一対一対応する `fileobj` と呼ぶ構造体で管理する。ファイルマップに伴うファイル I/O は、`fileobj` と一対一対応する、Linux 側に存在する `pager` と呼ぶ構造体で管理する。ファイルマップの動作を例を用いて説明する。

1. 第 1 のプロセスが `open()` でファイルディスクリプタを取得する。
2. 第 1 のプロセスが前記ファイルディスクリプタを引数とした `mmap()` で McKernel にファイルマップ作成を要求する。
3. McKernel は `mcctrl` に `pager` を要求する。
4. `mcctrl` は `pager` のリストをファイルの `inode` で検索する。リストにないため新たな `pager` を作成しリストに挿入し、その `pager` を返す。
5. McKernel は `fileobj` のリストを `pager` のアドレスで検索する。リストにないため新たに `fileobj` を作成して、取得した `pager` と紐付けた上で、`fileobj` のリストに挿入する。また、`VM_range` 構造体の `memobj` フィールドにポインタを格納する。
6. 第 1 のプロセスがページフォールトを起こす。読み込みのページフォールトを起こしたとする。
7. McKernel が `fileobj` の `get_page()` を呼んで、以下のステップで物理ページを取得する。
 - (a) 割り当て済み物理ページを管理するハッシュリストをオフセットで検索する。ハッシュリストにないためアロケータを呼ぶことで新たな物理ページを取得し、ハッシュリストに挿入する。
 - (b) `pager` に依頼して、当該物理ページにファイルの対応部分の内容を書き込む。
 - (c) 取得した物理ページのアドレスを返す。
8. McKernel は取得した物理ページに対応するページテーブルエントリを作成し挿入する。
9. 第 1 のプロセスが当該物理ページに対する操作を行う。
10. 第 2 のプロセスが `open()` でファイルディスクリプタを取得する。
11. 第 2 のプロセスが前記ファイルディスクリプタを引数とする `mmap()` で McKernel にファイルマップ作成を要求する。
12. McKernel は `mcctrl` に `pager` を要求する。
13. `mcctrl` は `pager` のリストを `inode` で検索し、第 1 のプロセスからの依頼によって作成された `pager` を返す。
14. McKernel は `fileobj` のリストを `pager` のアドレスで検索し、第 1 のプロセスによって作成された `fileobj` を取得し、`VM_range` に記録する。
15. 第 2 のプロセスがページフォールトを起こす。読み込みのページフォールトを起こしたとする。
16. McKernel が `fileobj` の `get_page()` を呼ぶ。

- 1 17. McKernel は割り当て済み物理ページを管理するハッシュリストをオフセットで検索し、
2 第 1 のプロセスによって取得された物理ページを取得する。
- 3 18. McKernel は取得した物理ページに対応するページテーブルエントリを作成し挿入する。
- 4 19. 第 2 のプロセスが当該物理ページに対する操作を行う。

5 2.4.8 POSIX Shared Memory

6 McKernel の POSIX Shared Memory 機能 (`/dev/shm/*` ファイルのマッピングによる共有メモリ
7 機能) にはプリマップ機能が追加されている。この機能は `mcexec` (第 1.1 参照) のオプション
8 (`--mpol-shm-premap`) によって有効にできる。

9 2.4.9 System V 共有メモリ

10 System V 共有メモリ機能によるメモリ領域は、`shmobj` と呼ぶ構造体を用いて、ファイルマッ
11 プと同様に管理する。

12 共有メモリセグメントの属性は、`shmctl` システムコールで要求されたときにそのまま
13 ユーザに渡せるように、カーネル内でも `shmid_ds` 構造体の形で保持する。`shmid_ds` 構造体
14 は、対応する `shmobj` に内包させる。

15 共有メモリのマッピングが `munmap()` で部分解放されたときの共有メモリマッピングの
16 分断や、`fork()` による共有メモリマップ数の増加、プロセス終了による共有メモリマップ数
17 の減少といったマップ数の管理は、共有メモリのアタッチ数 `shm_nattch` で行う。

18 2.4.9.1 実装の制限

19 Linux の `shmget` システムコール仕様のうち、引数 `shmflg` に `SHM_NORESERVE` を指定した、ス
20 ヲップ領域予約なし共有セグメントの作成はサポートしない (指定は無視される)。

21 また、Linux の `shmctl` システムコールの仕様のうち、以下のものをサポートしない。

- 22 1. 引数 `cmd` に `SHM_LOCK` を指定した、共有メモリセグメントのページロック
- 23 2. 引数 `cmd` に `SHM_UNLOCK` を指定した、共有メモリセグメントのページロック解除

24 2.5 procfs/sysfs

25 The `procfs/sysfs` files provided by McKernel are listed in Table 2.3 and Table 2.4.

Table 2.3: /proc files provided by McKernel

Full path	Description
/proc/stat	Kernel statistics
/proc/[PID]	Directory containing information of [PID]
/proc/[PID]/auxv	Additional information to ELF loader
/proc/[PID]/cgroup	cgroup it belongs to
/proc/[PID]/cmdline	Command line
/proc/[PID]/cpuset	CPU set
/proc/[PID]/maps	List of memory maps
/proc/[PID]/mem	Memory held by this process
/proc/[PID]/pagemap	Flat page table
/proc/[PID]/smaps	An extension based on <code>maps</code> , showing the memory consumption of each mapping and flags associated with it
/proc/[PID]/stat	Process status
/proc/[PID]/status	Process status in human readable form
/proc/[PID]/task/[THID]	Directory containing information of [THID]
/proc/[PID]/task/[THID]/mem	Memory held by this thread
/proc/[PID]/task/[THID]/stat	Thread status

Table 2.4: /sys files provided by McKernel

Full path	Description
/sys/bus/cpu/devices/cpu*	Symbolic link to /sys/devices/system/cpu/cpu*
/sys/devices/system/cpu/offline	CPUs that are not online because they have been HOT-PLUGGED off or exceed the limit of cpus allowed by the kernel configuration
/sys/devices/system/cpu/online	CPUs that are online and being scheduled
/sys/devices/system/cpu/possible	CPUs that have been allocated resources and can be brought online if they are present
/sys/devices/system/cpu/present	CPUs that have been identified as being present in the system
/sys/devices/system/cpu/cpu*/online	1: Online, 0: Offline
/sys/devices/system/cpu/cpu*/cache/index*/level	Represents the hierarchy in the multi-level cache
/sys/devices/system/cpu/cpu*/cache/index*/type	Type of the cache - data, inst or unified
/sys/devices/system/cpu/cpu*/cache/index*/size	Total size of the cache
/sys/devices/system/cpu/cpu*/cache/index*/coherency_line_size	Size of each cache line usually representing the minimum amount of data that gets transferred from memory
/sys/devices/system/cpu/cpu*/cache/index*/number_of_sets	total number of sets, a set is a collection of cache lines sharing the same index
/sys/devices/system/cpu/cpu*/cache/index*/physical_line_partition	number of physical cache lines sharing the same cachetag
/sys/devices/system/cpu/cpu*/cache/index*/ways_of_associativity	Number of ways in which a particular memory block can be placed in the cache
/sys/devices/system/cpu/cpu*/cache/index*/shared_cpu_map	Set of CPUs sharing this cache in bitmap form
/sys/devices/system/cpu/cpu*/cache/index*/shared_cpu_list	Set of CPUs sharing this cache in human readable form
/sys/devices/system/cpu/cpu*/node*	Symbolic link to /sys/devices/system/node/node*
/sys/devices/system/cpu/cpu*/topology/physical_package_id	Physical package (e.g. socket) ID
/sys/devices/system/cpu/cpu*/topology/core_id	Core ID within a physical package
/sys/devices/system/cpu/cpu*/topology/core_siblings	Logical core set within a physical package in bitmap form. Logical cores include Hyperthreading cores.
/sys/devices/system/cpu/cpu*/topology/core_siblings_list	Logical core set within a physical package in human readable form
/sys/devices/system/cpu/cpu*/topology/thread_siblings	Logical core set within a physical core in bitmap form
/sys/devices/system/cpu/cpu*/topology/thread_siblings_list	Logical core set within a physical core in human readable form
/sys/devices/system/node/online	Numa nodes that are online
/sys/devices/system/node/possible	Nodes that could be possibly become online at some point
/sys/devices/system/node/node*/distance	Distance between the node and all the other nodes in the system
/sys/devices/system/node/node*/cpumap	Logical core set in the node in bitmap form
/sys/devices/system/node/node*/cpu*	Symbolic link to /sys/devices/system/cpu/cpu*
/sys/devices/pci<dom>:/bus/<dom>:<bus>:<slot>.<func>/local_cpus	Nearby CPU mask (logical core set in bitmap form)
/sys/devices/pci<dom>:/bus/<dom>:<bus>:<slot>.<func>/local_cpulist	Nearby CPU mask (logical core set in human readable form)
/sys/devices/system/cpu/num_processors	Number of logical cores (McKernel extension)

procfs/sysfs 機能は、以下の 3 機能で実現する。

- McKernel がその内容を提供する procfs/sysfs と、Linux のそれとを優先度付きで重ね合わせ、さらに重ね合わせたファイルシステムを /proc や /sys で始まる標準パスで mcexec に見せる機能 (mcoverlayfs)
- コールバック関数を mcctrl と McKernel の両方から登録できるようにし、またアクセス要求を Linux から McKernel へ転送する機能

以下、それぞれの機能を説明する。

2.5.1 ファイルシステムの重ね合わせ

ファイルシステムの重ね合わせのステップは以下の通り。

1. McKernel が /proc/mcos0 を作成する。
2. mcoverlayfs を用いて、/proc/mcos0/と /proc を重ね合わせ /tmp/mcos/mcos0_proc にマウントする。また、mcoverlayfs の機能を用いて、前者と後者に同一ファイルが存在する際には、前者がアクセスされるように設定する。さらに、/tmp/mcos/mcos0_proc を /proc に bind mount する。こうすることで、/proc に存在するファイルであって、McKernel プロセスに Linux プロセスとは異なる内容を見せる必要のないものについては /proc/mcos0/ に当該ファイルを準備しないことで元々の /proc のファイルを見せることができる。また、異なる内容を見せる必要のあるものについては、/proc/mcos0/ に当該ファイルを準備することでそれを見せることができる。
3. McKernel が同様に、/sys/devices/virtual/mcos/mcos0/sys を作成し、/sys/devices/virtual/mcos/mcos0/sys と /sys を重ね合わせ /tmp/mcos/mcos0_sys にマウントし、/tmp/mcos/mcos0_sys を /sys に bind mount する。
4. mcctrl と McKernel が /proc/mcos0 および /sys/devices/virtual/mcos/mcos0/sys のファイル・ディレクトリを作成する。なお、ファイル・ディレクトリの内容は作成時に登録するアクセスコールバック関数によって提供される。
5. McKernel プロセスが /proc または /sys ファイルにアクセスする。アクセス要求は必要に応じて Linux から McKernel に転送される。

2.5.2 アクセス要求の Linux から McKernel への転送

アクセス要求の Linux から McKernel への転送の動作を図 2.6 を用いて説明する。

1. アプリは open() などのシステムコールを用いて procfs/sysfs のファイルへのアクセスを試みる。このシステムコールの処理は Linux 側に転送される。(図の (1))
2. mcexec がシステムコールを代理実行する。mcoverlayfs が優先度に基づいて McKernel が提供するファイルまたは Linux が提供するファイルへのアクセス振り分けを行う。この場合は前者に振り分けられたとする。(図の (2))
3. McKernel が提供するファイルに登録されたコールバック関数が呼び出される。(図の (3))
4. Linux 側 procfs/sysfs フレームワークがアクセス要求を McKernel 側フレームワークに転送する。McKernel 側フレームワークはアクセスに応じた処理を行う。例えば、ファイルの内容を返却する。(図の (4))

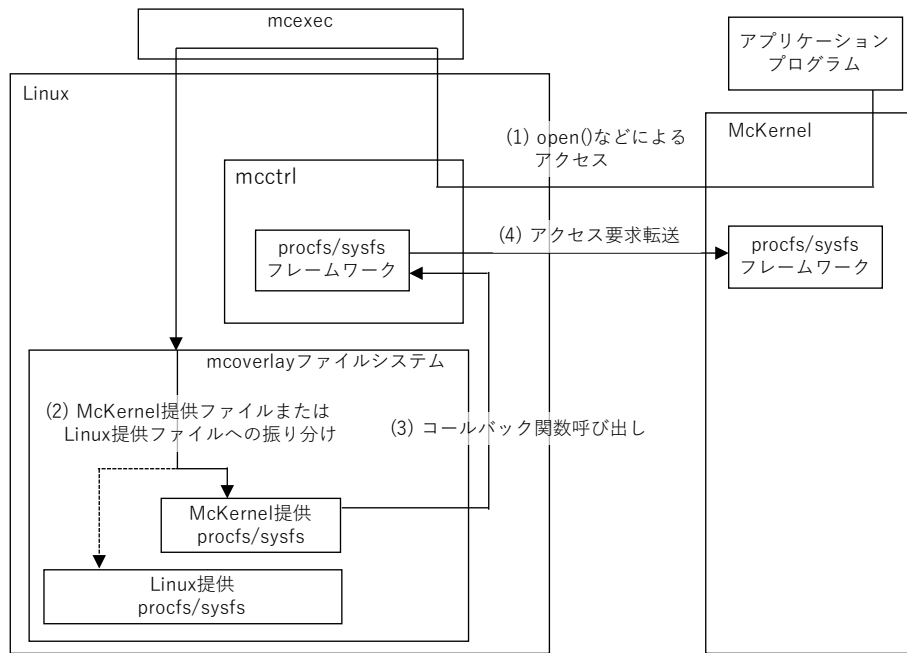


Figure 2.6: procfs/sysfs のアクセス要求転送

2.6 ファイルシステム重ね合わせ

McKernel はカーネルモジュール `mcoverlayfs` によって 2 つのファイルシステムを優先度付きで重ね合わせることができる。本機能は `procfs/sysfs` 機能のために用いられる。

`mcoverlayfs` は `overlayfs` に以下の機能を追加することで実装されている。

1. copyup 処理無効化

`mcoverlayfs` は `lowerdir` と `upperdir` に重ね合わせたいファイルシステムを指定する。McKernel では、`lowerdir` に、McKernel がその内容を提供する `procfs/sysfs` と Linux のそれとを指定して用いる。`overlayfs` では、ライト対象のファイルが `lowerdir` 上のファイルの場合、`copyup` 処理を行い `upperdir` に対象ファイルを作成し、そのファイルをオープンすることで、ライト処理を可能とする。このようにすると、アクセス要求は `procfs/sysfs` に届かない。そのため、`copyup` 処理を無効化し、直接対象ファイルにライト処理する機能を追加する。本機能はオプションに `nocopyupw` を指定することで有効となる。

`nocopyupw` オプションの有無によるライト処理の違いを図 2.7 に示す。

- `nocopyupw` オプションなしで、ライト対象のファイルが `lowerdir` 上のファイルの場合、`copyup` 処理を行い `upperdir` に対象ファイルを作成し、そのファイルをオープンすることで、ライト処理を可能とする。
- `nocopyupw` オプションありの場合、ライト対象のファイルが `lowerdir` 上のファイルの場合でも、`copyup` 処理せず、そのファイルをオープンすることで、ライト処理を可能とする。

2. procfs/sysfs サポート

`mcoverlayfs` では `overlayfs` に対して `procfs/sysfs` のディレクトリのマウント機能を追加している。本機能はオプションに `nofscheck` を指定することで有効となる。

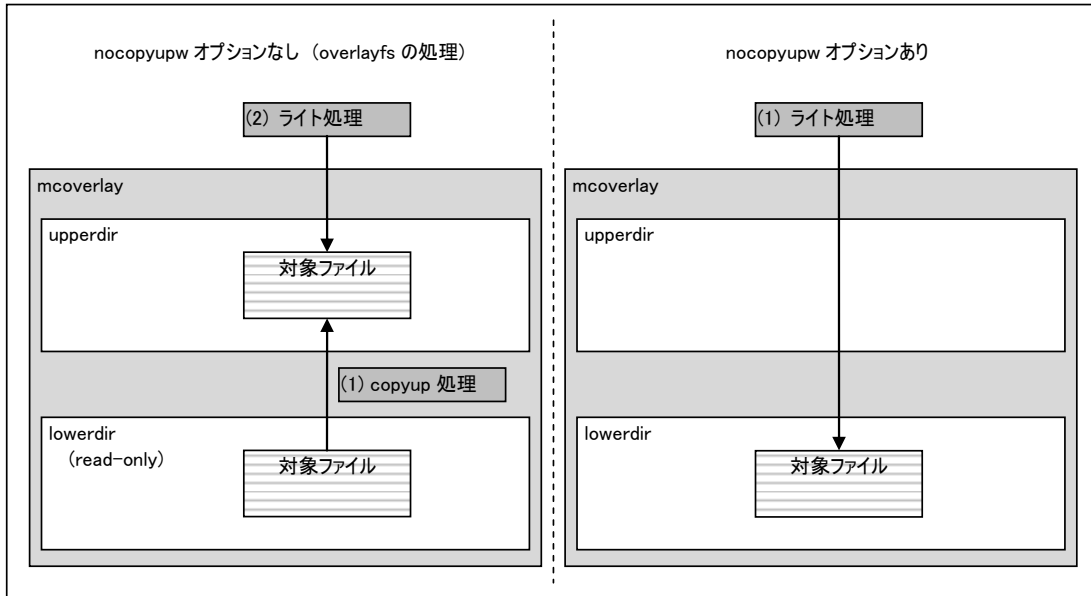


Figure 2.7: nocopyupw オプションの有無によるライト処理の違い

mcoverlayfs のマウントオプションを表 2.5 に示す。nocopyupw, nofscheck が overlaysfs に対して追加されたオプションである。

Table 2.5: mcoverlayfs のマウントオプション

オプション	説明
lowerdir=<dirs>	lowerdir を指定する。 ';' で区切り複数指定可能 (最大 500)
upperdir=<dir>	upperdir を指定する。
workdir=<dir>	workdir を指定する。 workdir は、upperdir と同じマウント下のディレクトリでなければならない。
default_permissions	デフォルトパーミッションを設定する。
nocopyupw	書き込み時に upperdir にファイルを作成し、それに対して書き込みを行う処理 (copyup 処理) を無効にする。 procfs/sysfs を lowerdir に指定する際は本オプションを指定する必要がある。
nofscheck	procfs/sysfs を lowerdir に指定可能にする。

2

2.6.1 詳細

3

overlaysfs のデータ構造に対する修正は以下の通り。

4

1. ovl_opt_bit
マウントオプションを追加するために、以下の enum 及び、マクロを追加する。

5

6

```
enum ovl_opt_bit {
    __OVL_OPT_DEFAULT    = 0,
```

7

8

```

1         __OVL_OPT_NOCOPYUPW    = (1 << 0),
2         __OVL_OPT_NOFSCHECK   = (1 << 1),
3     };
4
5     #define OVL_OPT_NOCOPYUPW(opt) ((opt) & __OVL_OPT_NOCOPYUPW)
6     #define OVL_OPT_NOFSCHECK(opt) ((opt) & __OVL_OPT_NOFSCHECK)

```

- 7 2. ovl_d_fsdata
8 d_fsdata を格納するために、以下の構造体を追加する。

```

9     struct ovl_d_fsdata {
10         struct list_head list;
11         struct dentry *d;
12         struct ovl_entry *oe;
13     };

```

- 14 3. ovl_config
15 マウントオプションを追加するために、opt を追加する。

```

16     struct ovl_config {
17         char *lowerdir;
18         char *upperdir;
19         char *workdir;
20         bool default_permissions;
21         unsigned opt;          <-- 追加
22     };

```

- 23 4. ovl_fs
24 d_fsdata を格納するために、d_fsdata_list を追加する。

```

25     struct ovl_fs {
26         struct vfsmount *upper_mnt;
27         unsigned numlower;
28         struct vfsmount **lower_mnt;
29         struct dentry *workdir;
30         long lower_namelen;
31         /* pathnames of lower and upper dirs, for show_options */
32         struct ovl_config config;
33         struct list_head d_fsdata_list; <-- 追加
34     };

```

- 35 5. ovl_tokens
36 マウントオプションを追加するために、OPT_NOCOPYUPW 及び、OPT_NOFSCHECK
37 を追加する。

```

38     enum {
39         OPT_LOWERDIR,
40         OPT_UPPERDIR,
41         OPT_WORKDIR,
42         OPT_DEFAULT_PERMISSIONS,
43         OPT_NOCOPYUPW,          <-- 追加
44         OPT_NOFSCHECK,         <-- 追加
45         OPT_ERR,
46     };
47
48     static const match_table_t ovl_tokens = {
49         {OPT_LOWERDIR,          "lowerdir=%s"},
50         {OPT_UPPERDIR,         "upperdir=%s"},

```

```

        {OPT_WORKDIR,          "workdir=%s"},          1
        {OPT_DEFAULT_PERMISSIONS, "default_permissions"}, 2
        {OPT_NOCOPYUPW,       "nocopyupw"},          <-- 追加 3
        {OPT_NOFSCHECK,      "nofscheck"},          <-- 追加 4
        {OPT_ERR,            NULL}                   5
};                                                  6

```

6. `ovl_fs_type` 7
`name` の値を”`mcoverlay`”に変更する。 8

```

static struct file_system_type ovl_fs_type = {          9
    .owner          = THIS_MODULE,                    10
    .name           = "mcoverlay",                    <-- 変更 11
    .mount          = ovl_mount,                      12
    .kill_sb        = kill_anon_super,                13
};                                                    14
MODULE_ALIAS_FS("mcoverlay");                        <-- 変更 15

```

`overlayfs` に対する関数の修正を表 2.6、表 2.7 に示す。 16

Table 2.6: overlayfs の関数に対する修正 (1)

関数	修正内容
<code>ovl_copy_xattr()</code>	<code>OVL_OPT_NOFSCHECK(opt)</code> が有効の場合、 <code>vfs_getxattr()</code> のエラーを無視する。
<code>ovl_copy_up_locked()</code>	<code>ovl_copy_xattr()</code> 呼び出し時に <code>ovl_get_config_opt()</code> で取得した <code>opt</code> 値を渡す。
<code>ovl_clear_empty()</code>	<code>ovl_copy_xattr()</code> 呼び出し時に <code>ovl_get_config_opt()</code> で取得した <code>opt</code> 値を渡す。
<code>ovl_setattr()</code>	<code>ovl_get_config_opt()</code> で <code>opt</code> 値を取得する。 <code>OVL_OPT_NOCOPYUPW(opt)</code> の場合、処理しない。
<code>ovl_permission()</code>	<code>ovl_reset_ovl_entry()</code> を呼び出してから処理する。
<code>ovl_setxattr()</code>	<code>ovl_get_config_opt()</code> で <code>opt</code> 値を取得する。 <code>OVL_OPT_NOCOPYUPW(opt)</code> の場合、処理しない。
<code>ovl_removexattr()</code>	<code>ovl_get_config_opt()</code> で <code>opt</code> 値を取得する。 <code>OVL_OPT_NOCOPYUPW(opt)</code> の場合、処理しない。
<code>ovl_d_select_inode()</code>	<ol style="list-style-type: none"> <code>ovl_get_config_opt()</code> で <code>opt</code> 値を取得する。 <code>OVL_OPT_NOCOPYUPW(opt)</code> の場合、<code>ovl_open_need_copy_up()</code> を呼び出さない。 <code>OVL_OPT_NOFSCHECK(opt)</code> で対象ファイルが <code>sysfs</code> の場合、<code>ovl_find_d_fsdata()</code> を呼び出して <code>dentry</code> が登録されているか確認する。登録されていない場合には <code>ovl_add_d_fsdata()</code> を呼び出して登録し、<code>dentry->d_fsdata</code> に <code>realpath.dentry->d_fsdata</code> の値を設定する。
<code>ovl_get_config_opt()</code>	<code>opt</code> 値を返す。
<code>ovl_reset_ovl_entry()</code>	<ol style="list-style-type: none"> <code>ovl_get_config_opt()</code> で <code>opt</code> 値を取得する。 <code>OVL_OPT_NOFSCHECK(opt)</code> の場合、<code>ovl_find_d_fsdata()</code> を呼び出して、<code>dentry</code> が登録されている場合には取得した <code>d_fsdata</code> を <code>oe</code> に設定する。
<code>ovl_find_d_fsdata()</code>	<code>dentry->d_sb->s_fs_info</code> の <code>d_fsdata_list</code> に登録されている <code>d_fsdata</code> を検索して、 <code>dentry</code> が登録されていた場合、 <code>dentry</code> の <code>ovl_entry</code> を戻す。
<code>ovl_add_d_fsdata()</code>	<ol style="list-style-type: none"> <code>struct ovl_d_fsdata</code> のメモリ領域を確保して、<code>dentry</code> の登録データを設定する。 <code>dentry->d_sb->s_fs_info</code> の <code>d_fsdata_list</code> に登録する。
<code>ovl_clear_d_fsdata()</code>	<code>d_fsdata_list</code> に登録されている全ての <code>d_fsdata</code> を削除して、 <code>struct ovl_d_fsdata</code> のメモリ領域を解放する。
<code>ovl_path_type()</code>	<code>ovl_reset_ovl_entry()</code> を呼び出してから処理する。
<code>ovl_path_upper()</code>	
<code>ovl_dentry_upper()</code>	
<code>ovl_dentry_lower()</code>	
<code>ovl_dentry_real()</code>	
<code>ovl_dir_cache()</code>	
<code>ovl_set_dir_cache()</code>	
<code>ovl_path_lower()</code>	
<code>ovl_dentry_is_opaque()</code>	
<code>ovl_dentry_set_opaque()</code>	
<code>ovl_dentry_update()</code>	
<code>ovl_dentry_version_inc()</code>	
<code>ovl_dentry_version_get()</code>	
<code>ovl_dentry_release()</code>	
<code>ovl_dentry_revalidate()</code>	
<code>ovl_dentry_weak_revalidate()</code>	

Table 2.7: overlayfs の関数に対する修正 (2)

関数	修正内容
<code>ovl_lookup_real()</code>	<code>OVL_OPT_NOFSCHECK(opt)</code> の場合、 <code>ovl_dentry_weird()</code> を呼び出さない。
<code>ovl_path_next()</code>	<code>ovl_reset_ovl_entry()</code> を呼び出してから処理する。
<code>ovl_lookup()</code>	<ol style="list-style-type: none"> 1. <code>ovl_get_config_opt()</code> で <code>opt</code> 値を取得する。 2. <code>ovl_reset_ovl_entry()</code> を呼び出してから処理する。 3. <code>ovl_lookup_real()</code> を呼び出す際、<code>opt</code> 値を渡す。
<code>ovl_put_super()</code>	<code>ovl_clear_dfsdata()</code> を呼び出してから処理する。
<code>ovl_statfs()</code>	<code>struct kstatfs</code> の <code>f_type</code> に <code>MCOVERLAYFS_SUPER_MAGIC</code> を設定する。
<code>ovl_show_options()</code>	<code>nocopyupw</code> , <code>nofscheck</code> オプションの説明を追加する。
<code>ovl_parse_opt()</code>	<code>nocopyupw</code> , <code>nofscheck</code> オプションの設定を追加する。
<code>ovl_mount_dir_noesc()</code>	<code>OVL_OPT_NOFSCHECK(opt)</code> の場合、 <code>ovl_dentry_weird()</code> を呼び出さない。
<code>ovl_mount_dir()</code>	<code>ovl_mount_dir_noesc()</code> を呼び出す際、 <code>opt</code> を渡す。
<code>ovl_lower_dir()</code>	<code>ovl_mount_dir_noesc()</code> を呼び出す際、 <code>opt</code> を渡す。
<code>ovl_fill_super()</code>	<ol style="list-style-type: none"> 1. <code>struct ovl_fs</code> の <code>dfsdata_list</code> を初期化する。 2. <code>ovl_mount_dir()</code> を呼び出す際、<code>opt</code> を渡す。 3. <code>ovl_lower_dir()</code> を呼び出す際、<code>opt</code> を渡す。 4. <code>OVL_OPT_NOCOPYUPW(opt)</code> の場合、以下の設定を行わない。 <ul style="list-style-type: none"> • <code>mnt->mnt_flags = MNT_READONLY;</code> • <code>sb->s_flags = MS_RDONLY;</code>

1 2.6.2 実装の制限

2 McKernel が生成する `/proc/[pid]/` 下のファイルを open して、close せずに open した状態
3 で exec して、exec したプロセスで同一ファイルを open するとエラー (ENOENT) となる。原因
4 は、exec() 時には、新たなプロセスの情報を返せるようにするため `/proc/[pid]/` 下のファ
5 イルを作成し直す。overlayfs は lower に指定されるディレクトリ下のファイルの inode 番
6 号が変わった場合、エラーを返すためである。

7 2.6.3 開発時の留意事項

8 Linux-4.0 から Linux-4.6 への移行に際する仮想ファイルシステムの以下の仕様変更に従
9 する必要があった。

- 10 1. `struct inode_operations` の `dentry_open()` が削除されて、`struct dentry_operations`
11 の `d_select_inode()` が追加された。
- 12 2. VFS の `vfs_open()` では、`dentry_open()` が呼ばれずに、`d_select_inode()` が呼び出
13 されるようになった。

14 また、以下のバージョンの Linux カーネルでのみ動作する。

- 15 ● 3.10.0-327 から 3.10.0-693 (RHEL-7.2 から 7.4)
- 16 ● 4.0.0 から 4.1.0
- 17 ● 4.6.0 から 4.7.0

18 2.7 デバイスドライバ

19 McKernel では、Linux で動作するドライバをそのまま利用可能であるが、システムコール移
20 譲のオーバーヘッドを削減するために、McKernel 内部で実装することもできる。

21 以下、それぞれの方法を説明する。

22 2.7.1 Linux ドライバの利用

23 Linux ドライバ経由でメモリマップされたデバイスのレジスタを McKernel プロセスからアク
24 セス可能にすることで、Linux ドライバをそのまま利用できるようにする。

動作を図 2.8 を用いて説明する。

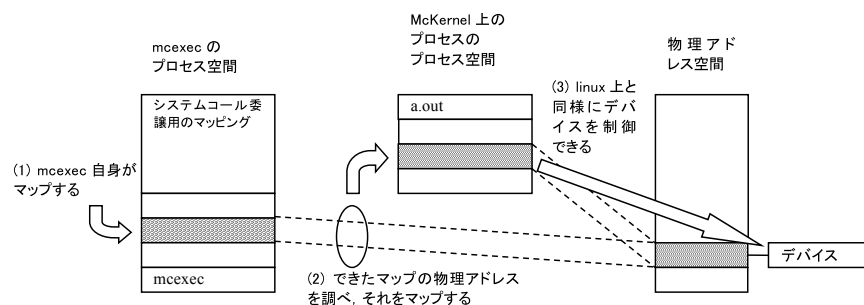


Figure 2.8: Linux ドライバ利用の動作

1. システムコール移譲の仕組みを用いてデバイスファイルの `open()`, `ioctl()` を行う。 1
2. レジスタのマップについてはシステムコール移譲の仕組みを用いて、`mcexec` 空間へのマップと、McKernel の仮想メモリ領域構造体への特別なマップであることの記録を行う。(図の (1)) 2
3
4
3. McKernel でのページフォールトの際に Linux に物理ページを問い合わせ、同じ物理ページを参照するマップを McKernel 上のプロセス空間に作成する。(図の (2)) 5
6

2.7.2 McKernel 内部での実装 7

特定のデバイスファイルに対して、`open()` 時に McKernel 内で処理を行うことをプロセス構造体に記録しておき、ファイル操作のシステムコールの際にその記録を参照することで、これらのファイルに対する操作を McKernel 内で行う。動作は以下の通り。 8
9
10

1. プロセスが `open()` を呼び出した際に対象が McKernel 内で処理を行うデバイスファイルであるかをパスにより調べる。そうであった場合は、ダミーの `fd` を取得し、`struct process` の `struct mckfd` のリストに `fd` に対応するエントリを挿入する。また、そのエントリにファイル操作のコールバック関数を登録する。 11
12
13
14
2. プロセスがファイル操作のシステムコールを呼び出した際に、`struct process` の `struct mckfd` のリストに `fd` に対応するエントリが存在するかを調べる。存在する場合は、当該エントリに登録されているコールバック関数を呼び出す。 15
16
17

2.8 XPMEM ドライバ 18

XPMEM は、あるプロセスがマップしたメモリ領域を他のプロセスからマップできるようにする。XPMEM はユーザライブラリ部分とドライバ部分に分かれており、ドライバ部分は McKernel 内部で実装されている。 19
20
21

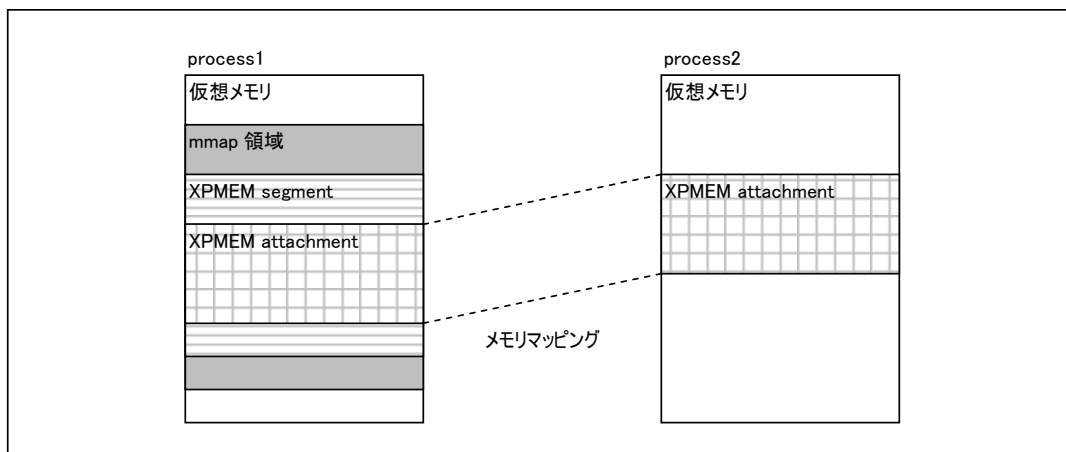


Figure 2.9: XPMEM のメモリマッピング



Figure 2.10: XPMEM の動作フロー

XPMEMのメモリマッピングを図 2.9、動作フローを図 2.10 に示す。XPMEMでは、プロセス (process1) が mmap したメモリ領域から、xpmem_make() で指定された領域を XPMMEM segment として管理して他のプロセスからマップできるようにする。マップしたいプロセス (process2) は、xpmem_get() でアクセスパーミッションを得て、xpmem_attach() で指定された XPMMEM segment のメモリ領域を XPMMEM attachment として管理して、マップする。マップは、プロセス (process2) が XPMMEM attachment 領域にアクセスして、ページフォルトが発生した際、ページテーブルエントリが示す物理アドレスを、プロセス (process1) の XPMMEM segment 領域の物理アドレスに置き換えることで実現する。

XPMEM のデータ構造を生成・破棄する関数を図 2.11 に示す。

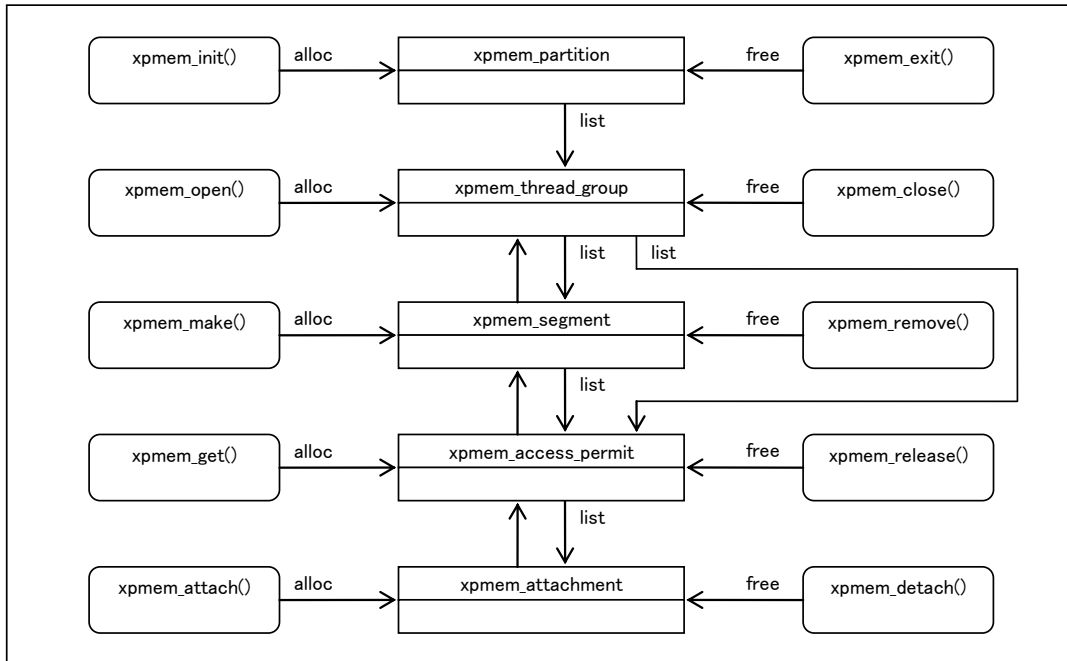


Figure 2.11: XPMMEM のデータ構造を生成・破棄する関数

XPMMEM では、以下のデータ構造を管理して機能を実現する。

1. xpmem_partition
2. xpmem_thread_group
3. xpmem_segment
4. xpmem_access_permit
5. xpmem_attachment

XPMMEM のデータ構造を図 2.12 に示す。

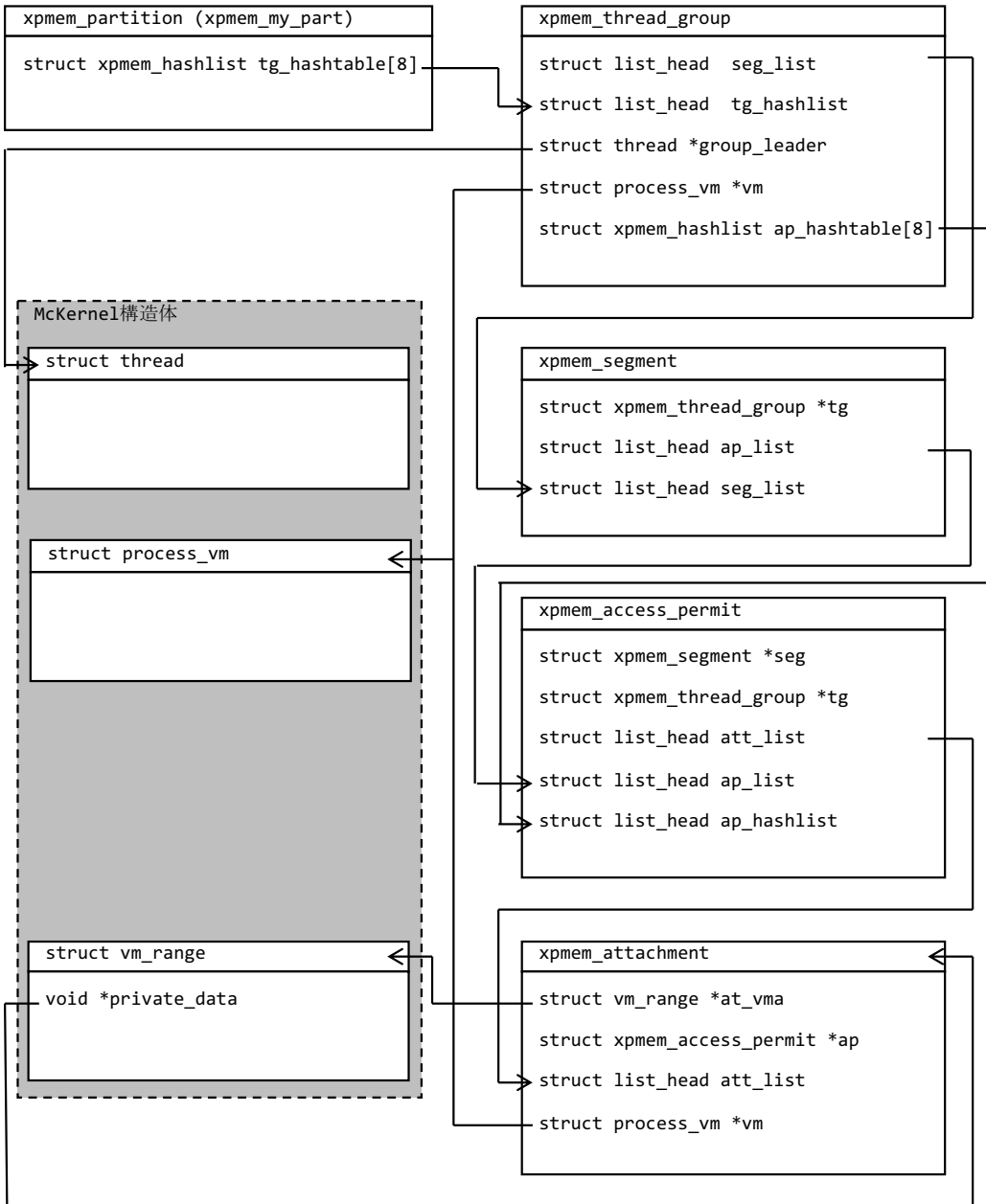


Figure 2.12: XPMEM のデータ構造

以下、各関数のインターフェイスと動作を説明する。

2.8.1 XPMEM デバイスファイルのオープン

書式

```
int xpmem_open(ihk_mc_user_context_t *ctx)
```

説明

xpmem_open は、以下の処理を行う。

1. XPMEM を初期化していない場合には、xpmem_init() を呼び出して XPMEM を初期化する。
2. fd を取得する必要があるが、/dev/xpmem デバイスファイルには影響を与えないように、do_syscall() を呼び出して、/dev/null デバイスファイルを開いて、その fd を使用する。fd がマイナス値の場合にはエラー値を返す。
3. _xpmem_open() を呼び出して、自プロセスの xpmem_thread_group が生成されていなければ生成する。
4. mckfd を生成して、初期設定する。

戻り値

0 以上	ファイルディスクリプタ (正常終了)
-EINVAL	引数が無効である
-ENOMEM	十分な空きメモリ領域が無い

2.8.2 XPMEM デバイスファイルの ioctl 制御

書式

```
static int xpmem_ioctl(struct mckfd *mckfd, ihk_mc_user_context_t *ctx)
```

説明

xpmem_ioctl は、以下の処理を行う。

1. cmd を処理する関数を呼び出す。
 - (a) XPMEM_CMD_VERSION
XPMEM_CURRENT_VERSION を返す。
 - (b) XPMEM_CMD_MAKE
xpmem_cmd_make データを取得する。
xpmem_make() を呼び出す。
xpmem_cmd_make データの segid を設定する。

- 1 (c) XPMEM_CMD_REMOVE
 2 xpmem_cmd_remove データを取得する。
 3 xpmem_remove() を呼び出す。
- 4 (d) XPMEM_CMD_GET
 5 xpmem_cmd_get データを取得する。
 6 xpmem_get() を呼び出す。xpmem_cmd_get データの apid を設定する。
- 7 (e) XPMEM_CMD_RELEASE
 8 xpmem_cmd_release データを取得する。
 9 xpmem_release() を呼び出す。
- 10 (f) XPMEM_CMD_ATTACH
 11 xpmem_cmd_attach データを取得する。
 12 xpmem_attach() を呼び出す。xpmem_cmd_attach データの vaddr を設定する。
- 13 (g) XPMEM_CMD_DETACH
 14 xpmem_cmd_detach データを取得する。
 15 xpmem_detach() を呼び出す。

16 戻り値

0	正常終了
-EFAULT	アドレスが不正である
-EINVAL	引数が無効である

17

18 2.8.3 XPMEM デバイスファイルのクローズ

19 書式

20 `static int xpmem_close(struct mckfd *mckfd, ihk_mc_user_context_t *ctx)`

21 説明

22 xpmem_close は、以下の処理を行う。

- 23 1. pid から xpmem_thread_group(tg) を取得する。
- 24 2. xpmem_release_aps_of_tg() を呼び出して、xpmem_access_permit、xpmem_attachment
25 を破棄する。
- 26 3. xpmem_remove_segs_of_tg() を呼び出して、xpmem_segment を破棄する。
- 27 4. xpmem_destroy_tg() を呼び出して、xpmem_thread_group を破棄する。
- 28 5. /dev/xpmem をオープンしているプロセスが存在しない場合には、xpmem_exit() を呼
29 び出して XPMEM を終了する。
- 30 6. xpmem_open() でオープンした/dev/null デバイスファイルについては、sys_close() で
31 クローズする。

32 戻り値

33

0	正常終了
---	------

2.8.4 XPMEM の初期化 1

書式 2

```
static int xpmem_init(void) 3
```

説明 4

xpmem_init は、以下の処理を行う。 5

1. xpmem_partition を生成して、初期設定する。 6

戻り値 7

0	正常終了
-ENOMEM	十分な空きメモリ領域が無い

8

2.8.5 XPMEM の終了 9

書式 10

```
static void xpmem_exit(void) 11
```

説明 12

xpmem_exit は、以下の処理を行う。 13

1. xpmem_partition を破棄する。 14

戻り値 15

なし。 16

2.8.6 xpmem_segment の生成 17

書式 18

```
static int xpmem_make(unsigned long vaddr, size_t size, int permit_type, void *
*permit_value, xpmem_segid_t *segid_p) 20
```

説明 21

xpmem_make は、以下の処理を行う。 22

1. 自プロセスの xpmem_thread_group を取得する。 23
2. segid を算出する。 24
3. xpmem_segment を生成して、初期設定する。 25
4. segid_p に segid を設定する。 26

1 戻り値

0	正常終了
-EINVAL	引数が無効である
-ENOMEM	十分な空きメモリ領域が無い
-XPMEM_ERRNO_NOPROC	対象プロセスの情報が無い

2

3 2.8.7 xpmem_segment の破棄

4 書式

```
5 static int xpmem_remove(xpmem_segid_t segid)
```

6 説明

7 xpmem_remove は、以下の処理を行う。

- 8 1. segid から xpmem_thread_group(seg_tg) を取得する。
- 9 2. seg_tg、segid から xpmem_segment を取得する。
- 10 3. 取得した xpmem_segment を破棄する。

11 戻り値

0	正常終了
-EACCES	許可がない
-EINVAL	引数が無効である

12

13 2.8.8 xpmem_access_permit の生成

14 書式

```
15 static int xpmem_get(xpmem_segid_t segid, int flags, int permit_type, void  
16 *permit_value, xpmem_apid_t *apid_p)
```

17 説明

18 xpmem_get は、以下の処理を行う。

- 19 1. segid から xpmem_thread_group(seg_tg) を取得する。
- 20 2. seg_tg、sigid から xpmem_segment を取得する。
- 21 3. 自プロセスの xpmem_thread_group(ap_tg) を取得する。
- 22 4. ap_tg から apid を算出する。apid がマイナス値の場合にはエラー値を戻す。
- 23 5. xpmem_access_permit を生成して、初期設定する。
- 24 6. apid_p に apid を設定する。

戻り値

1

0	正常終了
-EACCES	許可がない
-EINVAL	引数が無効である
-ENOMEM	十分な空きメモリ領域が無い
-XPMEM_ERRNO_NOPROC	対象プロセスの情報が無い

2

2.8.9 xpmem_access_permit の破棄

3

書式

4

```
static int xpmem_release(xpmem_apid_t apid)
```

5

説明

6

xpmem_release は、以下の処理を行う。

7

1. apid から xpmem_thread_group(ap_tg) を取得する。
2. ap_tg、apid から xpmem_access_permit を取得する。
3. 取得した xpmem_access_permit を破棄する。

8

9

10

戻り値

11

0	正常終了
-EACCES	許可がない
-EINVAL	引数が無効である

12

2.8.10 xpmem_attachment の生成

13

書式

14

```
static int xpmem_attach(struct mckfd *mckfd, xpmem_apid_t apid, off_t offset, size_t size, unsigned long vaddr, int fd, int att_flags, unsigned long *at_vaddr_p)
```

17

説明

18

xpmem_attach は、以下の処理を行う。

19

1. apid から xpmem_thread_group(ap_tg) を取得する。
2. ap_tg、apid から xpmem_access_permit(ap) を取得する。
3. ap から xpmem_thread_group(seg_tg)、xpmem_segment(seg) を取得する。
4. xpmem_attachment を生成して、初期設定する。

20

21

22

23

- 1 5. do_mmap() を呼び出して、メモリ領域 (at_vaddr) を確保する。
- 2 6. at_vaddr から vm_range(range) を取得する。
- 3 7. range->private_data に xpmem_attachment を設定する。

4 戻り値

0	正常終了
-EINVAL	引数が無効である
-ENOENT	そのようなファイルやディレクトリは無い
-ENOMEM	十分な空きメモリ領域が無い

5

6 2.8.11 xpmem_attachment の破棄

7 書式

```
8 static int xpmem_detach(unsigned long at_vaddr)
```

9 説明

10 xpmem_detach は、以下の処理を行う。

- 11 1. at_vaddr から vm_range(range) を取得する。
- 12 2. range->private_data から xpmem_attachment を取得する。
- 13 3. xpmem_vm_munmap() を呼び出して、以下の処理を行う。
 - 14 (a) ihk_mc_clear_range() を呼び出して、メモリ領域を解放する。
 - 15 (b) range->memobj を解放する。
 - 16 (c) range を解放する。
- 17 4. 取得した xpmem_attachment を破棄する。

18 戻り値

0	正常終了
-EACCES	許可がない
-EINVAL	引数が無効である

19

20 2.8.12 vm_range の fault 処理

21 書式

```
22 int xpmem_fault_process_memory_range(struct process_vm *vm, struct vm_range
23 *vmr, unsigned long vaddr, uint64_t reason)
```

説明

xpmem_fault_process_memory_range は、以下の処理を行う。

1. vmr->private_data から xpmem_attachment(att) を取得する。att が NULL の場合にはエラー値 (-EFAULT) を返す。
2. att から xpmem_access_permit(ap) を取得する。
3. ap から xpmem_thread_group(ap_tg) を取得する。ap->flags または ap_tg->flags が XPMEMi_FLAG_DESTROYING の場合にはエラー値 (-EFAULT) を返す。
4. ap から xpmem_segment(seg) を取得する。
5. seg から xpmem_thread_group(seg_tg) を取得する。seg->flags または seg_tg->flags が XPMEM_FLAG_DESTROYING の場合にはエラー値 (-ENOENT) を返す。
6. xpmem_remap_pte() を呼び出して、以下の処理を行う。
 - (a) ihk_mc_pt_lookup_pte() を呼び出して、seg の vaddr から pte_t(seg_pte) を取得する。
 - (b) ihk_mc_pt_lookup_pte() を呼び出して、vaddr から pte_t(att_pte) を取得する。
 - (c) ihk_mc_pt_set_pte() を呼び出して、att_pte の物理アドレスを seg_pte の物理アドレスに置き換える。

戻り値

0	正常終了
-EFAULT	アドレスが不正である
-ENOENT	そのようなファイルやディレクトリは無い

2.8.13 vm_range の削除

書式

```
int xpmem_remove_process_memory_range(struct process_vm *vm, struct vm_range *vmr)
```

説明

xpmem_remove_process_memory_range は、以下の処理を行う。

1. vmr->private_data から xpmem_attachment(att) を取得する。
2. att が指定されていた場合には、以下の処理を行う。
 - (a) att を解放する。
 - (b) vmr->private_data に NULL を設定する。

戻り値

2.9 ライブラリ切り替え

McKernel は、特定のパスについて、McKernel 上に起動されたプロセスと Linux 上に起動されたプロセスとに対して異なるファイルを見せる機能を提供する。これは、McKernel での実行と Linux での実行とで異なるライブラリファイルをリンクせねばならない例外的なケース（例えば、第 2.15 節で説明する Utility Thread Offloading のライブラリ）で、ローダ/リンカに異なるファイルをリンクさせることを目的とする。

動作は以下の通り。IHK/McKernel のインストールディレクトリを<install>とする。

1. unshare コマンドを用いて mcexec の mount name space の設定を変更し、mcexec が mcctrl に依頼する bind mount が他プロセスからは見えないようにする。
2. mcexec が mcctrl に制御を移す。
3. mcctrl がユーザ id を root に変更し、<install>/rootfs/以下のファイルのそれぞれを/に bind mount する。
4. mcctrl がユーザ id を元に戻し mcexec に制御を戻す。

2.10 状態監視

McKernel のハングアップ検知は以下のステップで実施される。

1. 運用ソフトウェアが IHK の関数を用いて通知のための eventfd を取得する。
2. McKernel が CPU ごとの状態と状態遷移回数を記録する。
3. Linux 上で動作するスレッド (ihkmond) が上記の状態を監視し、2 度同じ状態にあった場合、ハングアップと判断し、上記 eventfd を用いて運用ソフトウェアに通知する。

監視スレッドとハングアップ通知のインターフェイスは”IHK Specifications“に記載する。本節では第 2 のステップを説明する。

状態と状態遷移回数の記録には struct ihk_os_monitor 型の変数を用いる。以下の説明ではこの型を持つ監視用の変数を monitor と呼ぶ。struct ihk_os_monitor の関連部分は以下のように定義される。

```
struct ihk_os_monitor {
    ...
    int status;          /* OS 状態 */
    unsigned long counter; /* OS 状態が変化した回数 */
};
```

状態と状態遷移回数の記録の動作は以下の通り。

1. McKernel が以下のようにイベントに応じて状態と状態遷移回数を更新する。
 - カーネルモードからユーザモードへの移行時: monitor.status を IHK_OS_MONITOR_USER に設定する。

- ユーザモードからカーネルモードへの移行時: `monitor.status` を `IHK_OS_MONITOR_KERNEL` に設定し、`monitor.counter` をインクリメントする。 1 2
- システムコール移譲時: 移譲開始直前に `monitor.status` の値を保存し、`IHK_OS_MONITOR_KERNEL_OFFLOAD` に設定する。また、移譲完了後に `monitor.status` の値を保存しておいた値に戻し、`monitor.counter` をインクリメントする。 3 4 5
- `rt_sigtimedwait()`、`do_sigsuspend()`、`futex()`、`nanosleep()` 呼び出し時: 関数に入った直後に `monitor.status` を `IHK_OS_MONITOR_KERNEL_HEAVY` に設定する。なお、この状態に長時間滞在してもハングアップとは判定しない。 6 7 8
- `idle()` 呼び出し時: 関数に入った直後に `monitor.status` を `IHK_OS_MONITOR_IDLE` に設定する。なお、この状態に長時間滞在してもハングアップとは判定しない。その後、`cpu_safe_halt()` から復帰したタイミングで `monitor.status` を `IHK_OS_MONITOR_KERNEL` に設定し、`monitor.counter` をインクリメントする。 9 10 11 12

2.11 Non-Maskable Interrupt 13

Non-Maskable Interrupt (NMI) は対象 CPU に以下の動作をさせるために用いられる。 14

- カーネルダンプの準備 15
- 一時停止状態への遷移 16
- 一時停止状態からの復帰 17

なお、カーネルダンプの準備については第 2.13 節に、一時停止状態への遷移及びそこからの復帰については第 2.12 節に記載する。 18 19

NMI の利用ステップは以下の通り。 20

1. McKernel がブート時に `ihk_set_nmi_mode_addr()` で NMI の動作を指定する McKernel の変数 `nmi_mode` の物理アドレスを IHK-master に伝える。 21 22
2. IHK-master driver が `nmi_mode` の値を上記の動作のいずれかを示す値に設定し、`smp_ihk_os_send_nmi()` を呼び、各 CPU に NMI を送る。 23 24
3. 各 CPU が以下を実行する。 25
 - (a) NMI を受けて、NMI ハンドラ `nmi()` に制御を移す。 26
 - (b) `nmi()` で `nmi_mode` の値に応じた処理を行う。 27

以下、関連関数の動作を説明する。 28

2.11.1 NMI 動作設定 29

書式 30

```
int ihk_set_nmi_mode_addr(unsigned long addr) 31
```

説明 32

`addr` で指定される物理アドレスを NMI の動作を規定する McKernel の変数 `nmi_mode` の物理アドレスとして IHK に登録する。こうすることで、IHK から McKernel の NMI ハンドラの動作を切り替えることができるようになる。`nmi_mode` の値と NMI ハンドラの動作の対応は以下の通り。 33 34 35 36

値	動作
0	NMI ハンドラで各 CPU のカーネルダンプの準備を行う。
1	NMI ハンドラで各 CPU の状態を一時停止状態へ遷移させる。
2	NMI ハンドラで各 CPU の状態を一時停止状態から復帰させる。

1 戻り値

0	正常終了
---	------

2 **2.11.2 NMI 送信**

3 書式

4 `static int smp_ikh_os_send_nmi(ikh_os_t ikh_os, void *priv, int mode)`

5 説明

6 `nmi_mode` を `mode` に設定した上で各 CPU に NMI を発行する。

7 戻り値

0	正常終了
-EINVAL	エラー

8

9 **2.11.3 NMI ハンドラ**

10 書式

11 `void nmi()`

12 説明

13 `nmi_mode` に指定された値に従った動作を行う。`nmi_mode` の値と動作の対応は第 2.11.1 に
14 示す。

15 **2.12 全 CPU 一時停止**

16 McKernel は全 CPU を FROZEN と呼ぶ一時停止状態に遷移させる機能および FROZEN から
17 復帰させる機能を提供する。この機能と全 CPU を低電力状態に遷移させる機能とを組み合
18 わせることで、ジョブ単位での低電力状態への遷移とそこからの復帰を実現する。

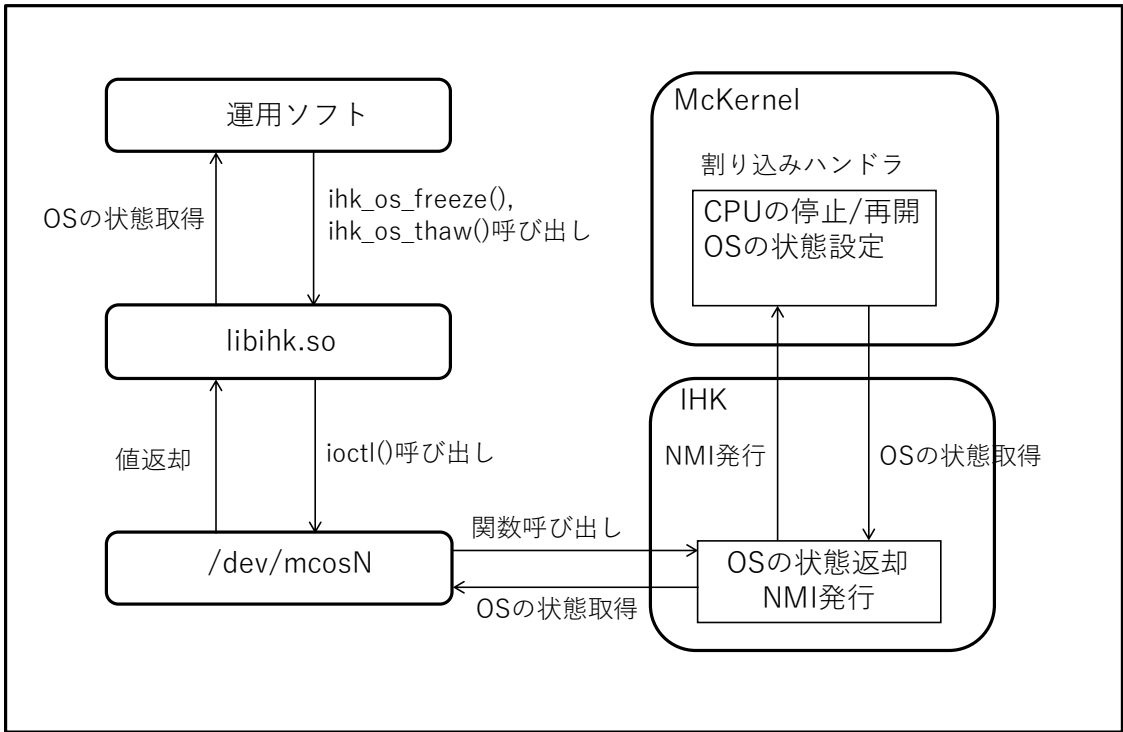


Figure 2.13: 構成要素関連図

全 CPU 一時停止機能の構成を図 2.13 に示す。

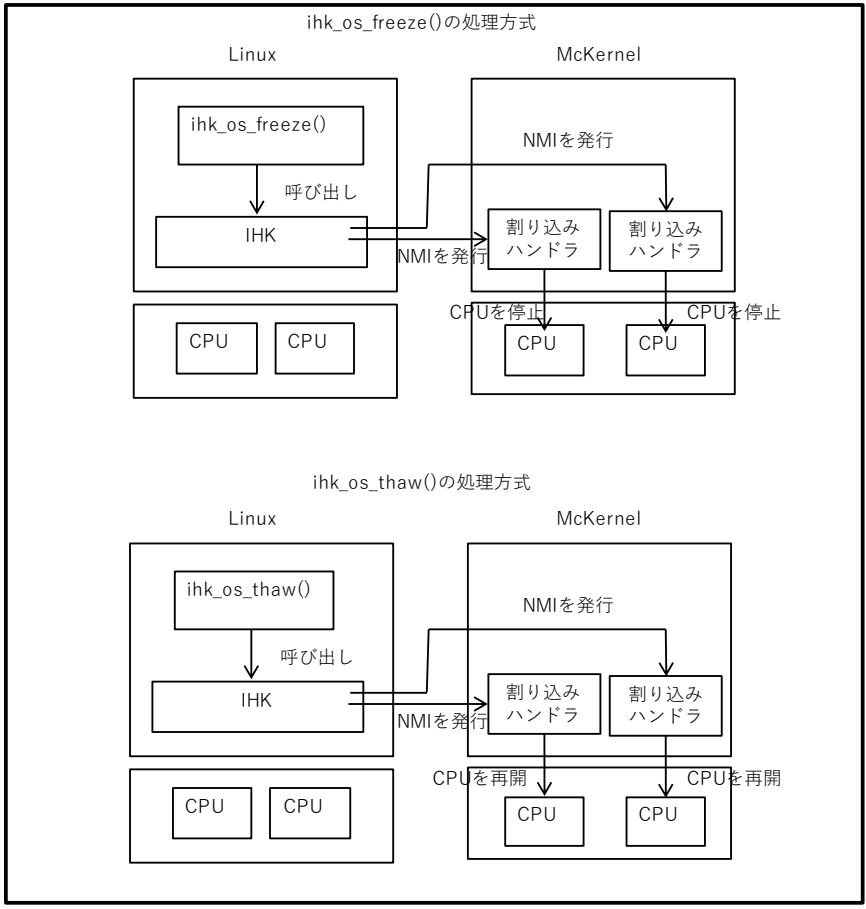


Figure 2.14: 全 CPU 一時停止および一時停止からの復帰のフロー

- 1 全 CPU 一時停止の動作を図 2.14 を用いて説明する。
- 2 1. バッチジョブスケジューラが `ihk_os_freeze()` 経由で `IHK_OS_FREEZE` コマンドを指定
- 3 して `ioctl()` を呼ぶ。
- 4 2. IHK-master core が `_ihk_os_freeze()` 経由で IHK-master driver の `smp_ihk_os_freeze()`
- 5 を呼ぶ。
- 6 3. IHK-master driver が `nmi_mode` に一時停止状態への遷移を示す値を設定し、`smp_ihk_`
- 7 `os_send_nmi()` を呼び、各 CPU に NMI を送る。
- 8 4. 各 CPU が以下を実行する。
- 9 (a) NMI を受けて、NMI ハンドラ `nmi()` に制御を移す。
- 10 (b) `nmi()` で `nmi_mode` に設定された指示に従った処理を行う。この場合は一時停止状
- 11 態への遷移であるため、`freeze_thaw()` を呼ぶ。
- 12 (c) `freeze_thaw()` で `nmi_mode` に設定された指示に従った処理を行う。この場合は
- 13 一時停止状態への遷移であるため、`mod_nmi_ctx()` を用いて `iret` 命令後のジャンプ
- 14 先を `_freeze()` にする。

(d) `__freeze()` は `freeze()` を呼び出す。`freeze()` は以下を実行する。 1

i. CPU の状態をバックアップ用変数に保持する。 2

ii. CPU の状態を一時停止状態に設定する。 3

iii. CPU を停止させる。`x86_64` アーキでは `hlt` 命令を実行する。 4

一時停止からの復帰の動作を図 2.14 を用いて説明する。 5

1. バッチジョブスケジューラが `ihk_os_thaw()` 経由で `IHK_OS_THAW` コマンドを指定して `ioctl()` を呼ぶ。 6 7

2. `IHK-master core` が `__ihk_os_thaw()` 経由で `IHK-master driver` の `smp_ihk_os_thaw()` を呼ぶ。 8 9

3. `IHK-master driver` が `nmi_mode` に一時停止状態からの復帰を示す値を設定し、`smp_ihk_os_send_nmi()` を呼び、各 CPU に NMI を送る。 10 11

4. 各 CPU が以下を実行する。 12

(a) NMI を受けて、NMI ハンドラ `nmi()` に制御を移す。 13

(b) `nmi()` で `nmi_mode` に設定された指示に従った処理を行う。この場合は一時停止状態からの復帰であるため、`freeze_thaw()` を呼ぶ。 14 15

(c) `freeze_thaw()` で `nmi_mode` に設定された指示に従った処理を行う。この場合は一時停止状態からの復帰であるため、CPU の状態をバックアップ用変数を用いて復元する。 16 17 18

以下、関連関数のインターフェイスと動作を説明する。 19

2.12.1 一時停止指示 (IHK-master core) 20

書式 21

```
static int __ihk_os_freeze(struct ihk_host_linux_os_data *data) 22
```

説明 23

アーキ依存の一時停止指示関数を呼ぶ。`smp-x86` では `smp_ihk_os_freeze()` を呼び出す。 24

戻り値 25

0	正常終了
---	------

2.12.2 一時停止からの復帰指示 (IHK-master core) 27

書式 28

```
static int __ihk_os_thaw(struct ihk_host_linux_os_data *data) 29
```


1 説明

2 アーキ依存の一時停止からの復帰指示関数を呼ぶ。smp-x86 では smp_ihk_os_thaw() を呼
3 び出す。

4 戻り値

0	正常終了
---	------

5

6 **2.12.3 一時停止指示 (IHK-master driver)**

7 書式

8 `static int smp_ihk_os_freeze(ihk_os_t ihk_os, void *priv)`

9 説明

10 smp_ihk_os_send_nmi() を呼び出して各 CPU に NMI を送り、CPU の状態を一時停止状
11 態へ遷移させ、また CPU を NMI を受けるまで停止させる。

12 戻り値

0	正常終了
---	------

13

14 **2.12.4 一時停止からの復帰指示 (IHK-master driver)**

15 書式

16 `static int smp_ihk_os_thaw(ihk_os_t ihk_os, void *priv)`

17 説明

18 smp_ihk_os_send_nmi() を呼び出して各 CPU に NMI をを送り、NMI 待ちで停止している
19 CPU の処理を再開させ、また CPU の状態を元の状態に戻す。

20 戻り値

0	正常終了
---	------

21

22 **2.12.5 一時停止および一時停止からの復帰指示**

23 書式

24 `long freeze_thaw(void *nmi_ctx)`

説明

1

2

1. 変数 `nmi_mode` が一時停止状態への遷移を意味する場合、`mod_nmi_ctx()` を呼び出すことで `__freeze()` を呼び出し、CPU の状態を一時停止状態に遷移させ、また CPU を NMI を受けるまで停止させる。
2. 変数 `nmi_mode` が一時停止状態からの復帰を意味する場合、CPU の状態を一時停止前の状態に戻す。

3

4

5

6

7

戻り値

8

0	一時停止を行った
1	一時停止からの復帰を行った

9

2.12.6 NMI ハンドラからの復帰時の指定関数へのジャンプ設定

10

書式

11

```
void mod_nmi_ctx(void *nmi_ctx, void (*func)())
```

12

説明

13

NMI ハンドラからの復帰時 (x86_64 アーキテクチャでは `iret` 命令実行時) に割り込み発生命令に戻らず、`func` で指定した、NMI 受け付けが必要な関数にジャンプするようにスタックの内容を変更する。このような処理が必要なのは、NMI ハンドラ内では NMI を受け付けないためである。`func` に `__freeze()` を指定することで、CPU を NMI 待ちの状態に停止させることができる。

14

15

16

17

18

2.12.7 一時停止指示 (ラッパー)

19

書式

20

```
void __freeze()
```

21

説明

22

`freeze()` を呼び出して CPU を一時停止させ、その後割り込みハンドラから復帰する。x86_64 アーキでは割り込みハンドラからの復帰には `iret` 命令を用いる。

23

24

2.12.8 一時停止指示

25

書式

26

```
void freeze()
```

27

1 説明

2 ステップは以下の通り。

- 3 1. CPU 状態を保存する。
- 4 2. CPU 状態を `IHK_OS_MONITOR_KERNEL_FROZEN` に遷移させる。
- 5 3. `cpu_halt()` を呼び CPU を停止させる。なお、CPU は NMI を受けると処理を再開する。
- 6 4. CPU が処理を再開した後、CPU 状態を保存しておいた値に戻す。

7 2.13 カーネルダンプ

8 カーネルダンプの採取と解析のステップは以下の通り。

- 9 1. 以下のいずれかの方法でダンプファイルを作成する。
 - 10 (a) IHK の関数 `ihk_os_makedumpfile()` または IHK のコマンド `ihkosctl` を用いて、
 - 11 McKernel 形式のダンプファイルを作成する（以降、McKernel 主導ダンプと呼ぶ）。
 - 12 (b) Linux の `panic` を契機に `makedumpfile` 形式のダンプファイルを作成する。また、
 - 13 コマンド `vmcore2mckdump` を用いて McKernel 形式に変換する（以降、Linux 主導
 - 14 ダンプと呼ぶ）。
- 15 2. `eclair` と呼ぶコマンドを用いてダンプファイルを解析する。

16 以下、詳細を説明する。

17 2.13.1 全体の処理の流れ

18 McKernel 主導ダンプの場合のダンプ採取機能とダンプ形式変換機能の処理の流れを図 2.15
19 を用いて説明する。

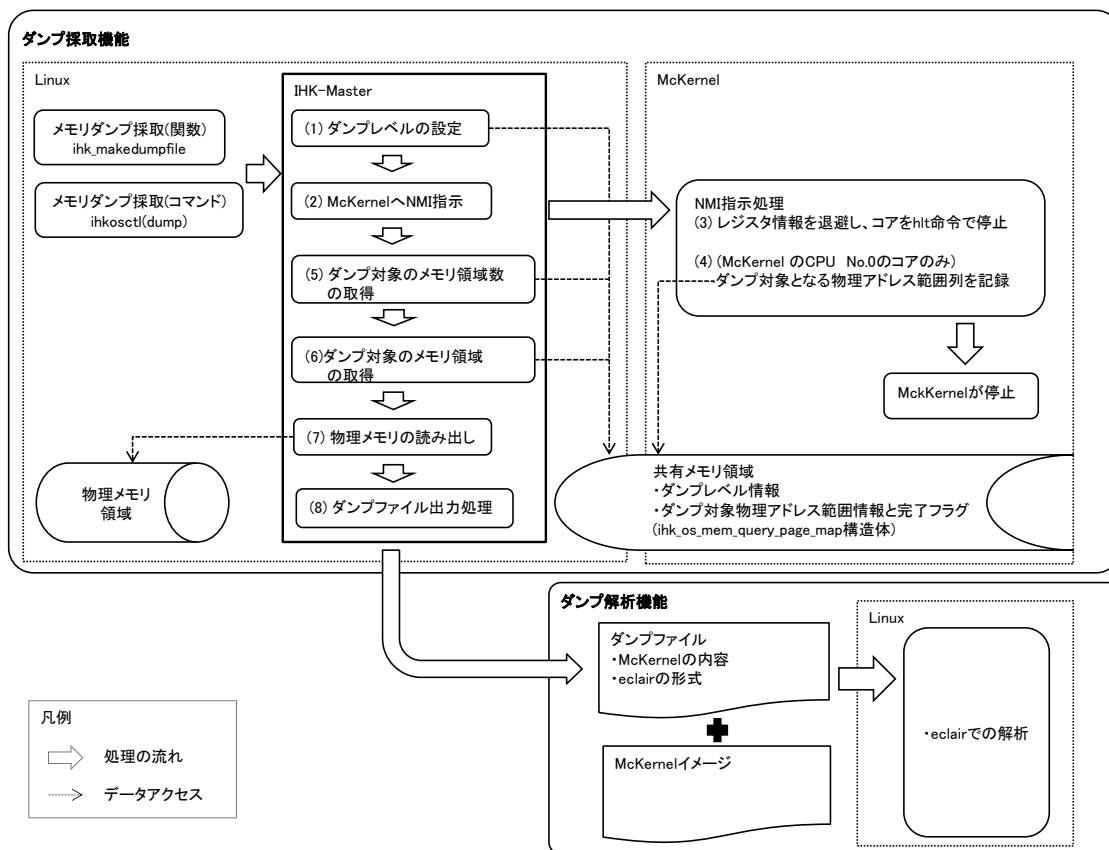


Figure 2.15: McKernel 主導ダンプの場合のダンプ採取機能とダンプ形式変換機能の処理の流れ

1. IHK が OS ブート時に、各物理ページがダンプ対象であることを示す情報（以下、ダンプ対象ページリストと呼ぶ）を Linux と McKernel とで共有しているメモリ領域（以降、共有メモリと呼ぶ）に確保する。また、IHK が McKernel に割り当てた物理アドレス範囲をダンプ対象とするように初期化する。 1
2
3
4
2. 管理者が `ihk_os_makedumpfile()` でダンプを指示する。 5
3. IHK が共有メモリにダンプレベルを記録する。また、共有メモリ上の、ダンプ対象ページリストの設定完了を表すフラグ（以降、完了フラグと呼ぶ）を 0 に設定する。（図の (1)） 6
7
8
4. IHK が McKernel の各コアへ NMI を送る。（図の (2)） 9
5. McKernel の第 0 CPU 以外の CPU はレジスタ情報を退避した後 `hlt` 命令で停止する（図の (3)）。McKernel の第 0 CPU は以下を実行する。（図の (3)、(4)） 10
11
 - (a) レジスタ情報を退避する。 12
 - (b) 共有メモリを参照してダンプレベルを取得する。 13
 - (c) ダンプからユーザ領域を除外する指定がされている場合は、ユーザメモリ領域情報を取得し、ダンプ対象ページリストの対応ビットを 0 にする。 14
15

- 1 (d) ダンプから未使用領域を除外する指定がされている場合は、未使用メモリ領域情報
2 報を取得し、ダンプ対象ページリストの対応ビットを0にする。
- 3 (e) 完了フラグに1をセットする。
- 4 (f) hlt 命令で停止する。
- 5 6. IHK が完了フラグが1になるまで待ち、`ioctl()` でダンプ対象のメモリ領域数を取得
6 し、領域情報を格納するメモリ領域を確保し、さらに `ioctl()` でダンプ対象の領域情報
7 を前記メモリ領域に記録する。(図の (5)、(6))
- 8 7. IHK がダンプ対象のメモリ領域を `ioctl()` で読み出し、ファイルに書き込む。(図の
9 (7)、(8))
- 10 8. 管理者は `eclair` を用いてダンプファイルの解析を行う。

11 ダンプ対象は `ihk_dump_page_set` で表現する。定義は以下の通り。

```
12 struct ihk_dump_page_set {  
13     unsigned int completion_flag; /* 書き込み完了フラグ */  
14     unsigned int count;          /* ダンプ対象のページ情報数 */  
15     unsigned long page_size;     /* ダンプ対象のページ情報の全体サイズ */  
16     unsigned long phy_page;     /* ダンプ対象のページ情報の物理アドレス  
17                                 (struct ihk_dump_page の配列) */  
18 }  
19  
20 struct ihk_dump_page {  
21     unsigned long start;        /* マップ情報の開始物理アドレス */  
22     unsigned long map_count;    /* マップ情報の領域数 (map[] の配列数) */  
23     unsigned long map[];       /* マップ情報 (ビットマップ形式) */  
24 };
```

25 Linux 主導ダンプの場合のダンプ採取機能とダンプ形式変換機能の処理の流れを図 2.16
26 を用いて説明する。

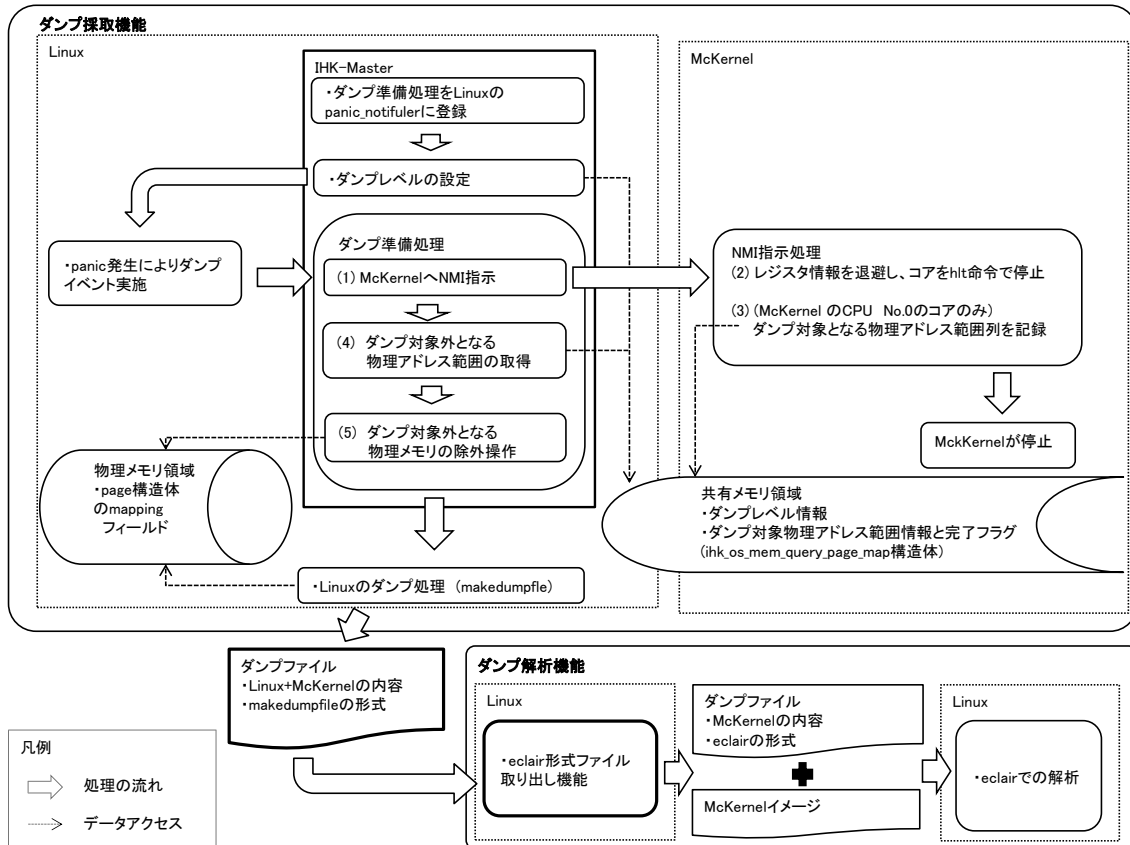


Figure 2.16: Linux 主導ダンプの場合のダンプ採取機能とダンプ形式変換機能の処理の流れ

1. IHK が OS ブート時に、ダンプ対象ページリストを共有メモリに確保する。また、IHK 1
- 2 が McKernel に割り当てた物理アドレス範囲をダンプ対象とするように初期化する。 2
3. IHK が McKernel 起動時にダンプレベル設定オプション (-d) でダンプレベルを指定す 3
- 4 る。IHK は共有メモリにこのダンプレベルを記録する。また、完了フラグを 0 に設定 4
- 5 する。 5
6. IHK がダンプ準備処理関数を Linux の `panic_notifier` に登録する。 6
7. Linux で `panic` が発生し、登録されているダンプ準備処理関数が呼び出される。 7
8. IHK が McKernel の各コアへ NMI を送る。(図の (1)) 8
9. McKernel の第 0 CPU 以外の CPU は、レジスタ情報を退避し、`hlt` 命令で停止する (図 9
- 10 の (3))。McKernel の第 0 CPU は以下を実行する。(図の (2)、(3)) 10
- 11 (a) レジスタ情報を退避する。 11
- 12 (b) 共有メモリを参照してダンプレベルを取得する。 12
- 13 (c) ダンプからユーザ領域を除外する指定がされている場合は、ユーザメモリ領域情 13
- 14 報を取得し、ダンプ対象ページリストにダンプからの除外を記録する。 14
- 15 (d) ダンプから未使用領域を除外する指定がされている場合は、未使用メモリ領域情 15
- 16 報を取得し、ダンプ対象ページリストのダンプからの除外を記録する。 16

- 1 (e) 完了フラグに1をセットする。
- 2 (f) hlt 命令で停止する。
- 3 7. IHK が完了フラグが1になるまで待ち、`ioctl()` でダンプ対象外の物理アドレス範囲
- 4 に該当する Linux の page 構造体の `mapping` フィールドを操作し `anonymous` に設定す
- 5 る。(図の (4)、(5))
- 6 8. Linux が `makedumpfile` コマンドを実行する。
- 7 9. Linux が Linux と McKernel の両方の情報を含むダンプファイルを作成する。
- 8 10. 管理者が `ldump2mcdump` コマンドで、`makedumpfile` 形式のダンプファイルを `eclair` 形
- 9 式に変換する。
- 10 11. 管理者は `eclair` を用いてダンプファイルの解析を行う。

11 2.13.2 ユーザメモリ領域情報取得

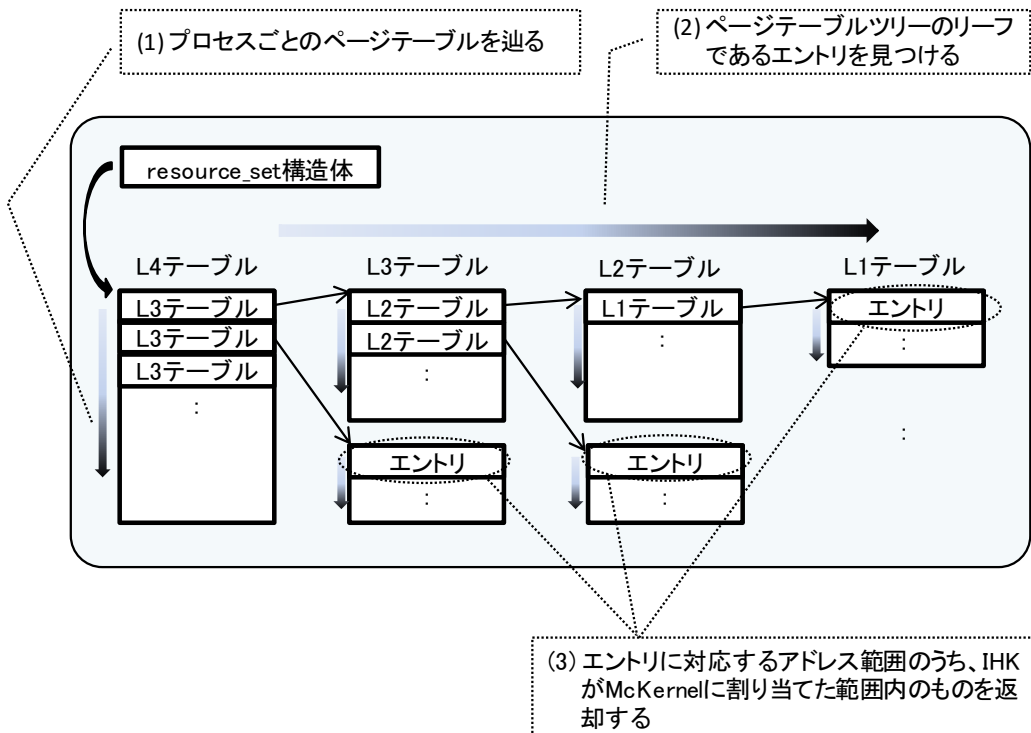


Figure 2.17: ユーザメモリ領域情報取得処理の流れ

- 12 ユーザメモリ領域情報取得処理の流れを図 2.17 を用いて説明する。
- 13 1. `resource_set` を参照して全プロセスを辿り、プロセスごとのページテーブルについて
- 14 以下を行う。(図の (1))
- 15 (a) ページテーブルツリーのリーフであるエントリを見つける。(図の (2)) なお、4
- 16 段目のエントリは 4 KB ページのエントリ、3 段目かつ `PageSize` フラグが1のエ
- 17 ントリは 2 MB、2 段めかつ `PageSize` フラグが1のエントリは 1 GB ページのエ
- 18 ントリである。

- (b) エントリに対応するアドレス範囲のうち、IHK が McKernel に割り当てた範囲に
 収まるものをユーザメモリ領域として返却する。収まらないものはエントリが破
 壊されているとみなし破棄する。(図の (3))

2.13.3 未使用メモリ領域情報取得

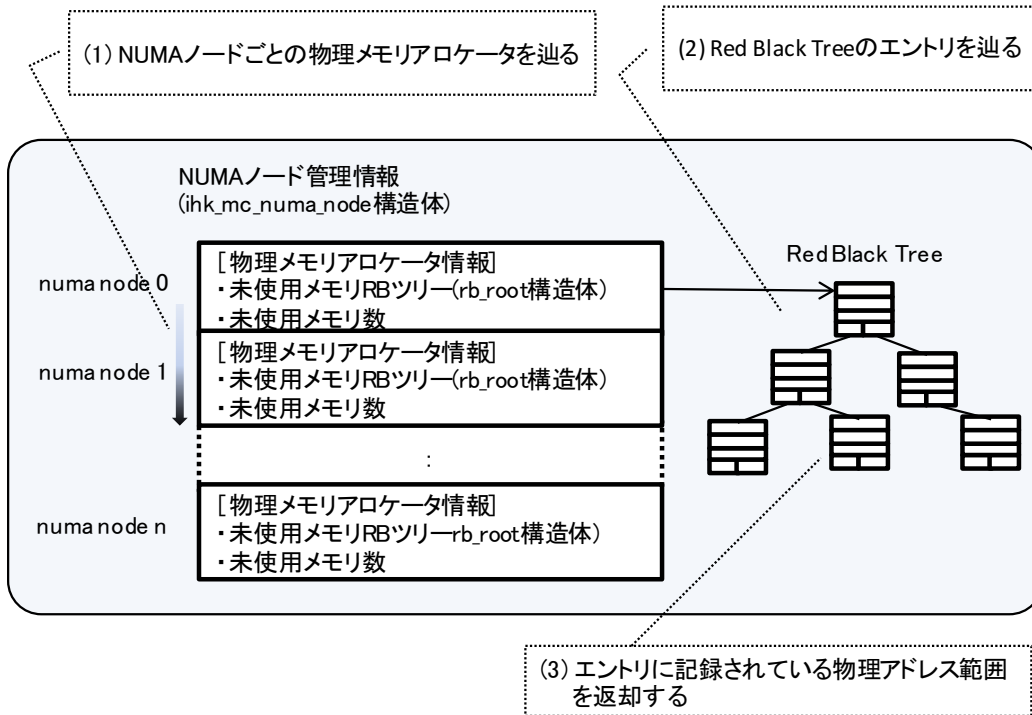


Figure 2.18: 未使用メモリ領域情報取得処理の流れ

未使用メモリ領域情報取得の処理の流れを図 2.18 を用いて説明する。

1. NUMA ノード管理情報 (ihk_mc_numa_node 構造体) を参照して NUMA ノードごとの物理メモリアロケータについて以下を行う。(図の (1))
 - (a) 未使用メモリを管理する Red Black tree(rb_root 構造体) のエントリを辿る。(図の (2))
 - (b) エントリ (free_chunk 構造体) に記録されている物理アドレス範囲を返却する。(図の (3))

2.13.4 ダンプ処理用 ioctl() コマンド

書式

```
int ioctl(int fd, IHK_OS_DUMP, struct ihk_dump_args *args)
```

説明

fd で指定された OS インスタンスに対して、args->cmd に指定されたダンプ関連処理を行う。

dumpargs_t は以下のように定義される。


```

1 struct ihk_dump_args {
2     int cmd; /* コマンド */
3     unsigned int level; /* ダンプレベル */
4     long start; /* 開始物理アドレス */
5     long size; /* サイズ */
6     void *buf; /* メモリ内容 */
7     int num_mem_chunks; /* メモリ領域数 */
8     struct ihk_dump_mem_chunk *mem_chunks; /* メモリ領域情報 */
9 };

```

10 struct ihk_dump_mem_chunk は以下のように定義される。

```

11 struct ihk_dump_mem_chunk {
12     unsigned long addr;
13     unsigned long size;
14 };

```

15 args->cmd ごとの処理は以下の通り。

args->cmd	動作				
DUMP_QUERY_NUM_MEM_AREAS	ダンプ対象メモリ領域数を返す。				
DUMP_QUERY_MEM_AREAS	ダンプ対象メモリ領域の情報を args->mem_chunks に格納する。呼び出し元が args->mem_chunks の領域を用意する。				
DUMP_READ	args->start, args->size で指定された物理メモリ領域の内容を args->buf で指定されたバッファにコピーする。				
DUMP_SET_LEVEL	<p>ダンプ対象とするメモリ領域の種類を args->level に設定する。設定可能な値は以下の通り。</p> <table border="1"> <tr> <td>0</td> <td>IHK が McKernel に割り当てたメモリ領域を出力する。</td> </tr> <tr> <td>24</td> <td>カーネルが使用しているメモリ領域を出力する。</td> </tr> </table> <p>なお、args->level が設定可能でない値であった場合は-EINVAL を返却する。</p>	0	IHK が McKernel に割り当てたメモリ領域を出力する。	24	カーネルが使用しているメモリ領域を出力する。
0	IHK が McKernel に割り当てたメモリ領域を出力する。				
24	カーネルが使用しているメモリ領域を出力する。				
DUMP_NMI	全 CPU に NMI を発行し、ダンプの準備を指示する。				
DUMP_SET_ANONYMOUS	(IHK が McKernel に割り当てたメモリ領域) から (args->mem_chunks, args->num_mem_chunks で指定したメモリ領域) を除いた領域に対し、Linux の struct page の mapping フィールドの最下位ビットをセットし anonymous テーブルに見せかける。こうすることで、Linux の makedumpfile が該当領域をダンプ対象から除外できるようになる。				
DUMP_QUERY	IHK によって割り当てられた物理メモリ領域の情報を args->start, args->size に格納する。本機能は、IHK が McKernel に割り当てたメモリ領域の全てをダンプする際に使用する。				

16 戻り値

0	正常終了
-EFAULT	アドレスが不正である
-EINVAL	引数が無効である

17 2.13.4.1 ダンプファイルの形式

18 ダンプファイルは ELF 形式を採用している。ダンプファイルで使用しているセクションは以下
19 の通り。

セクション名	説明
Date	ダンプ採取日時 例： Thu Mar 3 21:42:35 2016
hostname	ダンプ採取ホスト名 例： kncc08
User	ダンプ採取 実ユーザ名 例： nakamura
physmem	物理メモリダンプ

なお、レジスタの値はダンプファイルには格納しない。その代わりに、スレッドを表現する構造体に格納されている退避コンテキストから値を取得する。スレッドを表現する構造体の位置は、まず各コアの run queue の位置をシンボル情報から取得し、そこに挿入されているエントリを見つけることで取得する。objdump での出力例を図 2.19 に示す。

eclair形式のファイルフォーマット

```

mcdump_20160303_214235:   file format elf64-little

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
  0 date           00000018  0000000000000000  0000000000000000  00000040  2**0
    CONTENTS
  1 hostname       00000006  0000000000000000  0000000000000000  00000058  2**0
    CONTENTS
  2 user           00000008  0000000000000000  0000000000000000  0000005e  2**0
    CONTENTS
  3 physmem        20000000  0000000771800000  0000000771800000  00000066  2**0
    CONTENTS、ALLOCF、LOAD、DATA

Contents of section date:
0000 54687520 4d617220 20332032 313a3432  Thu Mar  3 21:42
0010 3a333520 32303136                :35 2016

Contents of section hostname:
0000 6b6e6363 3038                kncc08

Contents of section user:
0000 6e616b61 6d757261                username

Contents of section physmem:
77180000 00000000 00f0ffff 00000000 00400000  .....@..
77180010 21000000 0c000000 48004800 00000000  !.....H.H....
77180020 ffffffff ffffffff ffffffff ffffffff  .....
(以下略)

```

Figure 2.19: ダンプファイルの objdump での出力例

2.13.5 ダンプ解析コマンドと gdb コマンドとの連携方法

ダンプ解析コマンドと gdb コマンドとの間の remote serial protocol は、IPv4 を使った TCP 通信でやり取りする。unix ドメインソケットなども利用可能とは思いますが、異常終了時にごみファイルが残ることを回避するために TCP/IP 通信を選択した。ユーザが直接 gdb を起動して毎回リモートデバッグの設定を行うことは、難しくはないが面倒である。そこで、ユーザには、gdb ではなくダンプ解析コマンドを起動してもらおう。ダンプ解析コマンドが gdb コマ

1 ンドの起動とリモートデバッグの設定を行う。ダンプ解析コマンドによるリモートデバッグ
2 の設定から、実際にユーザからの解析コマンドを受け取る gdb に、スムーズに端末を受け渡
3 すため、以下の手順で動作する。

- 4 1. ユーザから起動されたダンプ解析コマンドは、コマンドラインオプションを解析して
5 gdb エージェントとしての初期化をする。
- 6 2. ダンプ解析コマンドは、remote serial protocol 通信用の TCP ソケットを作成する。
- 7 3. ダンプ解析コマンドは、gdb を fork() と exec() で起動する。この時、以下のようなコマ
8 ンドライン引数としてリモートデバッグの設定に必要なコマンドを与える。
- 9

```
10 -q -ex set prompt (eclair) -ex target remote :< TCP ポート番号> <カーネルイメージファイ  
11 ル名>
```

- 12 4. ダンプ解析コマンドは、TCP ソケットに gdb が接続してくるのを待つ。
- 13 5. ダンプ解析コマンドは、TCP ソケット接続後、端末からの入力をせずに gdb エージェ
14 ントとしての動作に専念する。

15 上記の手順によって、ダンプ解析コマンドと gdb とが同じ端末を共有した状態になる。共有
16 していても、標準入力の読み出しをダンプ解析コマンドが一切実行しなければ、gdb が標準
17 入力を占有しているのと同じ動作をさせることができる。

18 2.13.6 ダンプ形式変換 (crash プラグイン)

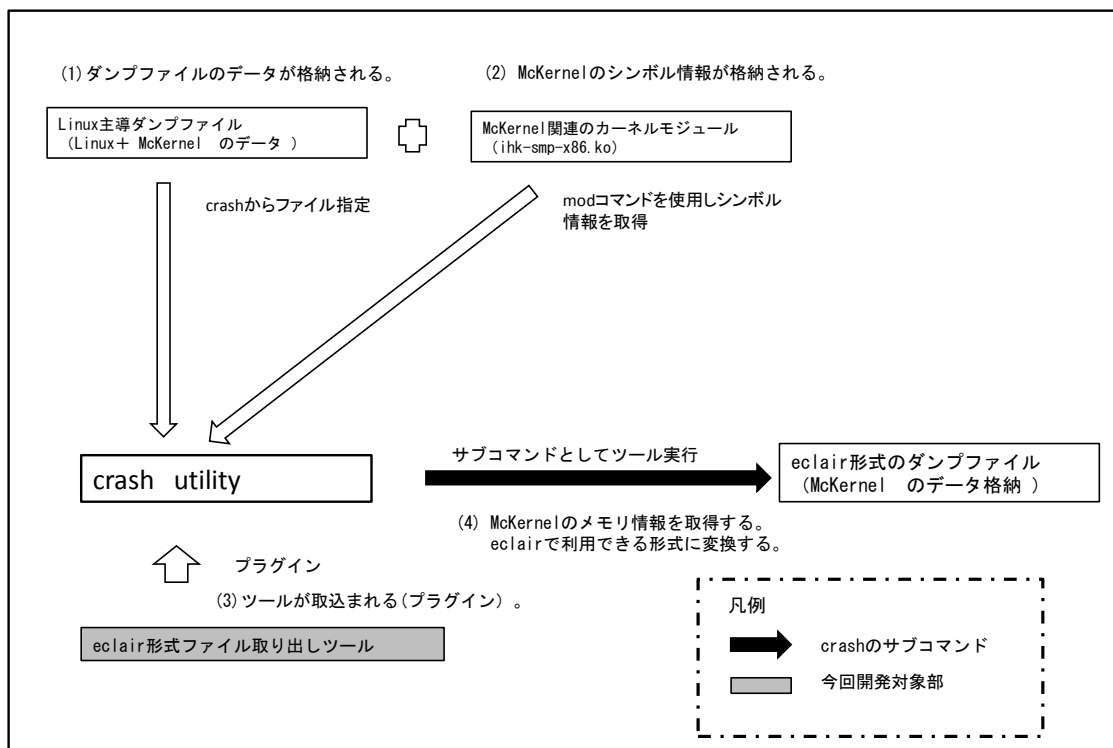


Figure 2.20: ダンプ形式変換処理の流れ

ダンプ形式変換処理の流れを図 2.20 を用いて説明する。

- (1) crash 内領域にダンプファイルのデータが格納される。
コマンド : `crash <vmlinux のパス> <ダンプファイル (vmcore) のパス>`
- (2) crash 内領域に IHK および McKernel のカーネルモジュールのシンボル情報が格納される。
コマンド : `mod -s ihk-smp-x86 <ihk-smp-x86.o のパス>`
シンボル情報 : `dump_page_set_addr` (ダンプ対象ページリストのアドレス情報)
- (3) crash 内にプラグインが取込まれる。
コマンド : `extend <crash utility extension のパス (例 : dump2mcdump.so)>`
- (4) ダンプ形式変換ツール (`ldump2mcdump`) を実行し、(2) のシンボル情報を用いて、McKernel に割り当てられた物理アドレス領域の情報を取得する。
取得した情報を用いて、(1) から McKernel 関連情報を取り出し、eclair 形式に整形して出力する。

2.13.7 利用時の留意事項

Linux 主导ダンプは RHEL-7.4 以降のバージョンの Linux カーネルでのみ動作する。また、Linux のブートパラメタに `crash_kexec_post_notifiers` を指定して、Linux に、ファイルを生成する前に `panic_notifier` に登録された関数を呼ばせる必要がある。

1 2.14 プロセスダンプ

2 McKernel では Linux と同様の方法でプロセスのダンプファイルを生成できる。

3 2.14.1 実装の制限

4 出力される情報についての制限は以下の通り。

- 5 1. 複数スレッドが存在しても、親プロセスの情報しか出力しない。
- 6 2. NOTE セグメントに格納されるプロセス状態 (`struct elf_prstatus64` 型) のうち、以
7 下のフィールドに対応する情報は格納しない。

```
8
9     struct elf_siginfo pr_info;    /* シグナル情報 */
10
11     short int pr_cursig;          /* 現在のシグナル */
12
13     a8_uint64_t pr_sigpend;      /* ペンディングされているシグナル */
14
15     a8_uint64_t pr_sighold;      /* hold されているシグナル */
16
17     pid_t pr_pid;
18
19     pid_t pr_ppid;
20
21     pid_t pr_pgrp;
22
23     pid_t pr_sid;
24
25     struct prstatus64_timeval pr_utime; /* ユーザ時間 */
26
27     struct prstatus64_timeval pr_stime; /* システム時間 */
28
29     struct prstatus64_timeval pr_cutime; /* 累積ユーザ時間 */
30
31     struct prstatus64_timeval pr_cstime; /* 累積システム時間 */
32
```

33 なお、`struct elf_siginfo` は以下のように定義される。

```
34
35     struct elf_siginfo {
36
37         int si_signo; /* signal number */
38
39         int si_code; /* extra code */
40
```

```

    int si_errno; /* errno */
};

```

3. NOTE セグメントに格納されるプロセス情報 (struct elf_prpsinfo64 型) のうち、以下のフィールドに対応する情報は格納しない。

```

char pr_sname;      /* プロセス状態 (文字列) */
char pr_zomb;      /* Zombie か否か */
char pr_nice;      /* Nice 値 */
a8_uint64_t pr_flag; /* フラグ */
unsigned int pr_uid;
unsigned int pr_gid;
int pr_ppid, pr_pgrp, pr_sid;
char pr_fname[16]; /* 実行可能ファイル名 */
char pr_psargs[ELF_PRARGSZ]; /* 引数リスト先頭部分 */

```

2.15 Utility Thread Offloading

McKernel は、スレッドを Linux の CPU にマイグレートする機能を提供する。

この機能により、通信のプログレススレッドなどのヘルパースレッド (utility thread と呼ぶ) を、計算用 CPU 資源を利用することなく実行することができる。

Linux CPU へのマイグレートは、以下の処理の組み合わせによって実現する。

1. スレッドマイグレート処理
McKernel で実行しているスレッドを Linux CPU にマイグレートする処理
2. システムコール処理
Linux にマイグレートしたスレッドの発行するシステムコールと McKernel スレッドの発行するシステムコールとの一貫性を担保する処理
3. シグナル受信処理
Linux にマイグレートしたスレッドへシグナルを中継する処理
4. スレッド終了処理
Linux にマイグレートしたスレッドを正しく終了させる処理

以下、それぞれの処理の概要を説明する。

1 2.15.1 スレッドマイグレート処理

2 スレッドマイグレートの処理のうち、McKernel 側の処理は以下の通り。

- 3 1. McKernel で実行中のユーザスレッドが自スレッドを Linux にマイグレートすることを
4 システムコールを用いて指示する。
- 5 2. McKernel はシステムコールを受けて以下の処理を行う。
 - 6 (a) マイグレート対象のユーザスレッドのコンテキストを取得する。
 - 7 (b) McKernel から Linux へのシステムコール委譲を用いて、`mcexec` に対してスレ
8 ッドのマイグレート指示を行う。このとき、取得したコンテキストを引き渡す。
- 9 3. McKernel のスレッドは Linux へマイグレートされたスレッドが終了するまでシステム
10 コール完了を待ってスリープする。
- 11 4. McKernel はシステムコールが完了すると当該スレッドを起床する。
- 12 5. McKernel のスレッドはシステムコールの戻り値を引数として `_exit` を呼び出し、スレ
13 ッドを終了する。

14 スレッドマイグレートの処理のうち、`mcexec` 側の処理は以下の通り。

- 15 1. McKernel からスレッドマイグレート指示を受ける。
- 16 2. システムコールワーカースレッドを新規に生成する。これは、自スレッドで
17 マイグレートしたコンテキストを処理するため、システムコールワーカースレッドが
18 不足するためである。生成したスレッドはシステムコール委譲待ちとなる。なお、マイ
19 グレートは一回のみ可能であるため、システムコールワーカースレッドが必要以上に生
20 成されることはない。
- 21 3. 当該スレッドの孫プロセスを生成する。当該孫プロセスは当該スレッドに `ptrace` シス
22 テムコールを用いて接続し、当該スレッドが発行するシステムコールを捕捉する（次節
23 で説明する）。
- 24 4. 自スレッドにおいて、コンテキストをマイグレート対象スレッドの
25 コンテキストに切り替える。このとき、切り替え前のコンテキストを保存しておく。
26 切り替え前コンテキストを保存するのは、シグナル受信時やスレッド終了時に一時的
27 に `mcexec` のコンテキストに復帰する必要があるためである。
- 28 5. コンテキスト切り替え後、スレッドはマイグレート対象のスレッドと
29 して、マイグレート指示のシステムコールからの戻りアドレスから処理を再開する。

2.15.2 システムコール処理

Linuxにマイグレートしたスレッドが発行するシステムコールは捕捉し、必要に応じて McKernel に処理を依頼する。

これは、システムコールの中には、`futex()` や `mmap()` など、McKernel の状態を操作するものがあるためである。

システムコールは `ptrace` を用いて捕捉する。具体的には、マイグレート時にマイグレートしたスレッド (`tracee`) を監視する `tracer` プロセスを Linux 上で生成し、`tracee` にシステムコールの発行を報告させる。また、`tracer` が `tracee` のレジスタを操作することで必要に応じて Linux 上でのシステムコール発行をスキップさせる。

システムコールごとの処理を 2.8 に示す。

Table 2.8: マイグレートされたスレッドが発行するシステムコールの処理

システムコール	処理
<code>mmap, mprotect, munmap, brk, futex</code>	IKC を用いて McKernel に処理を依頼する
<code>getpid, gettid</code>	<code>mcexec</code> に記録しておいた id を返す
<code>open, read, write</code> など McKernel からはシステムコール移譲を行うもの	Linux 上でシステムコールを発行する
<code>exit_group, _exit</code>	<code>mcexec</code> で処理する
それ以外	エラー (ENOSYS) とする

2.15.3 シグナル受信処理

2.15.3.1 シグナル送信処理

既存の McKernel に、Linux にシステムコールマイグレートしているプロセスに対して McKernel からシグナルを送信し、処理を中断する処理が存在する。この処理を応用して、Linux にマイグレートしたスレッドへのシグナル配送を実現する。具体的には以下の処理を行う。

- シグナル送信処理において、シグナル配送先スレッドが Linux にマイグレートされている場合、システムコールオフロード中スレッドへのシグナル送信と同様に IKC を通じて `mcexec` にシグナル送信を依頼する。
- `mcexec` は McKernel のスレッド ID (リモートスレッド ID) から Linux 上のスレッド ID (ローカルスレッド ID) への変換を行い、ローカルスレッド ID に対してシグナルを送信する。

2.15.3.2 `mcexec` のシグナルハンドラ処理

`mcexec` は Linux から McKernel のスレッドに送られたシグナルを McKernel へ中継するために、特別なシグナルハンドラを登録している。このため、McKernel のスレッドから Linux にマイグレートされたスレッドに送られるシグナルに対応するシグナルハンドラの呼び出しは、直接行うことができず、この特別なシグナルハンドラから行う必要がある。`mcexec` のシグナルハンドラの入り口と出口では、TLS を切り替える必要がある。TLS を切り替えない場合、Linux にマイグレートされたスレッドの `errno` やその他の TLS 領域

1 を破壊する可能性があるためである。
2 TLS の切り替えは、libc の `arch_prctl` や `syscall` を使用できない (これらは `errno` を更新
3 する)。TLS の切り替えはシステム依存の手段でシステムコールを呼び出す必要がある
4 (例えば、x86_64 では `syscall` 命令の発行)。
5 TLS を切り替えるため、Linux にマイグレートしているスレッドに対して
6 `mcctrl` において `mcexec` の TLS と McKernel スレッドの TLS を保持しておく。
7 TLS 切り替え要求に対して通常の `mcexec` のスレッドは何もしないが、Linux に
8 マイグレートされたスレッドでは、シグナルハンドラの入り口では `mcexec` の TLS に
9 切り替え、出口では McKernel スレッドの TLS に切り替える。

10 2.15.4 スレッド終了処理

11 マイグレートされたスレッドの終了処理のステップは以下の通り。

12 1. スレッド終了の捕捉

13
14 マイグレートしたスレッドが `_exit()` を呼び出した場合は `tracer` が `ptrace` で補足し、
15 シグナルによってスレッドが終了する場合は `mcctrl` が Linux のスレッド終了フック
16 (`trace_sched_process_exit`) を用いて捕捉する。

17 2. McKernel 側でのスレッド終了

18
19 `mcexec` が IKC を用いて McKernel にスレッドの終了ステータスを通知し、McKernel
20 側のスレッドを終了させる。

21 3. Linux 側でのスレッド終了

22
23 以下のステップで Linux 側のスレッドを終了させる。

- 24 (a) `mcexec` のスレッドのコンテキストを、マイグレートした McKernel スレッドのもの
25 のから、`mcctrl` に記録しておいたマイグレート処理前のものへ切り替える。コン
26 テキストの切り替えによって、`mcexec` の `ioctl()` の直後から実行が再開される。
- 27 (b) `tracer` プロセスが終了する。
- 28 (c) `mcexec` のスレッドが `_exit()` を呼び出すことで終了する。

29 2.15.5 実装詳細

30 2.15.5.1 Linux CPU へのスレッド生成の構成

31 Linux CPU へのスレッド生成の構成を図 2.21 に示す。

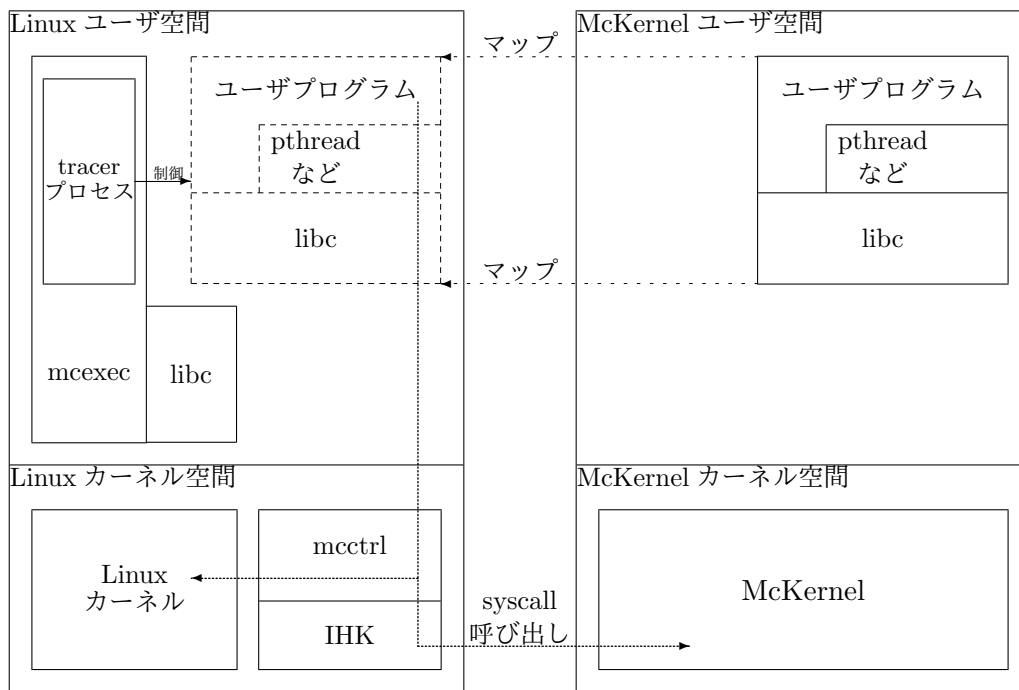


Figure 2.21: Linux CPU へのスレッド生成の構成

1 ユーザプログラムのスレッドを Linux CPU に生成 (実際はマイグレート) すると、ユー
2 ザプログラムが占めるメモリが Linux ユーザ空間にマップされ、Linux ユーザ空間の `mcexec`
3 から参照可能となる。このとき、Linux ユーザ空間内には、`mcexec` の `libc` とユーザプログラ
4 ムの `libc` が異なる実体として配置されている。

5 McKernel のスレッドを Linux に生成するとき、`mcexec` の子プロセスとして `tracer` プロ
6 セスを生成する。`tracer` プロセスは Linux に生成したスレッドのシステムコールを監視し、
7 Linux CPU のユーザプログラムが発行したシステムコールの一部 (`mmap` など) を `mcctrl` 経
8 由で McKernel 上で処理するように制御する。

9 2.15.5.2 `util_indicate_clone` システムコール

10 `util_indicate_clone` システムコールは自スレッドの `thread` 構造体に `util_indicate_clone`
11 システムコールの引数 `mod` と `arg` (カーネル空間にコピー済の `arg`) を設定する。

12 `thread` 構造体の関連フィールドは以下のように定義される。

```
13  
14 struct thread {  
15     ... 略 ...  
16  
17     int mod_clone;           // 生成対象 OS  
18  
19     void *mod_clone_arg;     // CPU 位置の指示、スレッドの振る舞いの記述  
20  
21 };  
22  
23
```

24 2.15.5.3 `util_migrate_inter_kernel` システムコール

25 `util_migrate_inter_kernel` システムコールは以下の処理を行う。

- 26 1. `arg` が非 NULL の場合、`arg` の内容をユーザ空間からカーネル空間にコピーする。コピー
27 に失敗した場合、`EFAULT` を返却する。
- 28 2. コンテキスト退避用ページを確保する。
- 29 3. コンテキスト退避用ページにユーザコンテキストを退避する。
- 30 4. コンテキスト退避用ページの物理アドレスを引数として、
31 `sched_setaffinity` をオフロードする²。
- 32 5. コンテキスト退避用ページを解放する。
- 33 6. `sched_setaffinity` のオフロードの戻り値が正 (成功) の場合、以下を行う。

- 34 (a) 戻り値の `0x100000000` ビットが立っている場合、プロセスの終了を表すため、`terminate()`
35 でプロセスを終了する。

²他のシステムコール番号と被らない
ように、オフロード対象ではない `sched_setaffinity` のシステムコール番号を
使用している。`sched_setaffinity` をオフロードすると、
`mcexec` にてスレッドオフロード処理を行う

- (b) それ以外の場合、`sched_setaffinity()` の処理をもう一度 McKernel に依頼することで、コンテキスト保存領域を `unmap` し、`do_exit()` でスレッドを終了する。

7. `sched_setaffinity` のオフロードの戻り値が負 (エラー) の場合、
戻り値を返却する。

2.15.5.4 `get_system` システムコール

`get_system` システムコールは McKernel 上で実行すると 0 を返却する。

Linux 上で実行すると、当該システムコールは存在しないため、`ENOSYS` でエラーリターンする。

2.15.5.5 `clone` システムコール

`clone` システムコールにて、子スレッドの `thread` 構造体を `runq` に接続する (`runq_add_thread` 呼び出し) 前に以下の処理を行う。

1. 親スレッドの `mod_clone` に `SPAWN_TO_REMOTE` が設定されている場合、
子スレッドの `thread` の `mod_clone` に `SPAWNING_TO_REMOTE` を設定する。
これにより、子スレッドを `schedule` が処理するとき `util_migrate_inter_kernel` の処理が行われる。

2.15.5.6 `schedule` の処理

`schedule` に対して、以下の変更を行う。

1. `next` を探す処理において、`mod_clone` に `SPAWNING_TO_REMOTE` が設定されているスレッドが `runq` に有る場合、そのスレッドを優先して `next` に設定する。

2.15.5.7 `enter_user_mode` の処理

`enter_user_mode` に対して、以下の変更を行う。

1. `check_signal` を呼び出した後で、`auto_utilthr_migrate` を呼び出す。
この処理は `current` スレッドに `SPAWNING_TO_REMOTE` が設定されている場合、
スレッドを開始する前に Linux にマイグレートする。

2.15.5.8 `auto_utilthr_migrate` の処理

`auto_utilthr_migrate` は以下の処理を行う。

1. `current` スレッドの `mod_clone` に `SPAWNING_TO_REMOTE` が設定されている場合、
`mod_clone` に `SPAWN_TO_LOCAL` を設定し、
`util_migrate_inter_kernel` を呼び出す。
これによって、新しいスレッドの開始前に `util_migrate_inter_kernel` システム
コールの処理が実行される。

1 2.15.5.9 do_syscall の処理

2 do_syscall に対して、以下の変更を行う。

3 1. オフロードしたシステムコールの状態が、STATUS_SYSCALL の場合 (Linux から
4 McKernel へのシステムコール委譲)、以下の処理を行う。

- 5 (a) システムコール番号が rt_sigreturn の場合、シグナルハンドラの内容を
6 返却する。
- 7 (b) システムコール番号が rt_sigreturn 以外の場合、以下を行う。
 - 8 i. syscall_table を検索し、システムコール番号が登録されているか調べ、
9 登録されていない場合は ENOSYS を返却する。
 - 10 ii. システムコールコンテキストを作成し、syscall_table に登録されている
11 システムコール処理を呼び出す。結果を返却する。
- 12 (c) システムコールの結果は、send_syscall 呼び出しによって、IKC を通じて
13 Linux に通知する。この処理は remote page fault と同様である。

14 2.15.5.10 mcexec の処理

15 mcexec は以下の処理を行う。

16 1. sched_set_affinity に対するオフロード処理として、以下を行う。

- 17 (a) create_worker_create を呼び、新しいワーカースレッドを作成する。
18 このスレッドは、不足するシステムコールオフロードスレッドを補完するもので
19 ある。
- 20 (b) mcexec スレッドと McKernel スレッドのコンテキスト退避領域を作成する。
- 21 (c) mcctrl に McKernel スレッドのコンテキストを McKernel スレッドのコンテキ
22 スト退避領域へコピーさせる (MCEXEC_UP_UTIL_THREAD1)。
- 23 (d) tracer プロセスを生成する。tracer プロセスの詳細は 4 に示す。
- 24 (e) uti_attr が指定されている場合、mcctrl に uti_attr 処理を依頼する
25
26 (MCEXEC_UP_UTI_ATTR)。
- 27 (f) 以下に示す switch_ctx を行う。
 - 28 i. mcexec スレッドのコンテキスト退避領域に現在のコンテキストを退避する。
 - 29 ii. mcctrl に Linux にマイグレートされたスレッドの情報を登録する
30 (MCEXEC_UP_UTIL_THREAD2)。
 - 31 iii. コンテキストをマイグレートされたスレッドのコンテキストに切り替える。
- 32 (g) マイグレートしたスレッドが完了した後に元のコンテキストに戻る。
- 33 (h) mcexec スレッドのコンテキスト退避領域を解放する。
- 34 (i) スレッドを終了する。

35 2. シグナルを受信した際、シグナルハンドラにて以下の処理を行う。

- 36 (a) mcctrl に MCEXEC_UP_SIG_THREAD を要求し、mcexec の TLS に切り替える。

- (b) `mexec` のスレッドの場合、McKernel に受信したシグナルを通知する。 1
 - (c) Linux にマイグレートしたスレッドの場合は以下の処理を行う。 2
 - i. `mcctrl` に `rt_sigaction` で `MCEXEC_UP_SYSCALL_THREAD` 要求を行い、 3
シグナルハンドラの設定を取得する。 4
 - ii. シグナルハンドラが `SIG_IGN` の場合、何もしない。 5
 - iii. シグナルハンドラが `SIG_DFL` の場合、シグナルが `SIGCHLD`、`SIGURG`、`SIG-` 6
`CONT` 以外の 7
場合はシグナルハンドラを解除し、自プロセスにシグナルを送付する。これ 8
によって、 9
当該シグナルを受信して自プロセスが終了する。 10
 - iv. シグナルハンドラがアドレスの場合、一時的に TLS を元に戻して 11
アドレスの関数 (シグナルハンドラ) を呼び出す。 12
 - (d) `mcctrl` に `MCEXEC_UP_SIG_THREAD` を要求し、元の TLS に切り替える。 13
3. `tracer` プロセスは以下の処理を行う。 14
- (a) `tracee` にて、`tracer` と待ち合わせに使用するパイプを作成する。 15
 - (b) `tracer` プロセスを `tracee` の孫プロセスとして `fork` する。孫プロセスとするのは 16
`tracee` が `tracer` を `wait` しないまま終了する場合に対応するためである。 17
 - (c) `tracee` は以下の処理を行う。 18
 - i. `tracer` の予期せぬ終了を検知できるように、パイプの出力側を閉じる。 19
 - ii. 子プロセスの終了を待つ。子プロセスはすぐに終了する (孫プロセスが `tracer` 20
になる)。 21
 - iii. パイプの入カイベントの発生を (最大)1 秒待つ (`select`)。 22
 - iv. イベントが発生せずに 1 秒経過した場合、タイムアウトでエラーリターン。 23
 - v. `select` がエラーの場合は、そのエラーコードでエラーリターン。 24
 - vi. パイプから 1 バイト読み込み、パイプを閉じる。 25
 - vii. パイプから 1 バイト読み込めなかった場合 (EOF)、`EAGAIN` でエラーリターン。 26
 - viii. 正常にリターン。(以下、`tracee` スレッドはオフロード処理を継続する。) 27
 - (d) パイプの入力側を閉じる。 28
 - (e) 子プロセスを `fork` し、親プロセスは終了 (`exit`) する。子プロセスが `tracer` となる。 29
 - (f) `/dev/mcos` 以外のファイルディスクリプタを全て閉じる。 30
 - (g) 標準入出力を `/dev/null` に割り当てる。 31
 - (h) `tracee` スレッドに `PTRACE_ATTACH` する。 32
 - (i) `tracee` の停止を待つ (`wait`)。 33
 - (j) `PTRACE_SYSCALL` 後の停止理由がシステムコールかシグナルかの区別を付ける 34
ために、`PTRACE_SETOPTIONS` で `PTRACE_O_TRACESYSGOOD` を指定す 35
る。 36
 - (k) パイプに 1 バイト書き出し、パイプを閉じる。 37
 - (l) 以下、無限ループ。 38
 - i. `PTRACE_SYSCALL` により `tracee` を再開する。 39
 - ii. `tracee` の停止を待つ (`wait`)。 40

- 1 iii. tracee が終了した場合、終了コードを McKernel に通知し、tracer を終了する。
- 2 iv. 停止以外の場合、continue³。
- 3 v. システムコールで停止した場合、以下を行う。
 - 4 A. PTRACE.GETREGS を行い、tracee のレジスタを得る。
 - 5 B. システムコール番号が ioctl で引数に MCEXEC_UP_SYSCALL_THREAD
 - 6 が指定
 - 7 されている場合、戻り値を逆オフロード結果に書き換える。
 - 8 C. システムコール番号が逆オフロード対象で、
 - 9 戻り値 (x86 の場合、rax) が -ENOSYS の場合 (システムコール呼び出し時)、
 - 10 システムコール番号を ioctl に変更し、システムコール逆オフロードの引
 - 11 数を設定
 - 12 する。
 - 13 D. PTRACE.SETREGS を行い、tracee のレジスタを更新する。
- 14 vi. システムコール以外 (つまりシグナル) で停止した場合、次回 PTRACE_SYSCALL
- 15 に指定するシグナルとして、停止シグナルを設定する。

16 2.15.5.11 mcctrl の処理

17 MCEXEC_UP_UTIL_THREAD1 コマンドに対しては、McKernel から渡された物理アドレスで示さ
 18 れる McKernel スレッドのコンテキストを、mcexec のコンテキスト退避領域にコピーする。
 19 MCEXEC_UP_UTIL_THREAD2 コマンドに対しては、host_thread 構造体を作成する。
 20 host_thread 構造体は Linux にマイグレートされたスレッドの情報を保持し、以下のよ
 21 うに定義される。

```

22
23 struct host_thread {
24
25     struct host_thread *next;           // 同一 PID 内のリスト
26
27     struct mcos_handler_info *handler;  // LWK 情報へのハンドラ
28
29     int pid;                            // プロセス ID
30
31     int tid;                             // スレッド ID
32
33     unsigned long usp;                  // mcexec コンテキストの SP
34
35     unsigned long lfs;                  // mcexec の TLS ベース
36
37     unsigned long rfs;                  // ユーザプログラムの TLS ベース
38
39 };
40

```

41 mcos_handler_info は LWK の情報を保持し、以下のように定義される。

```

42
43 struct mcos_handler_info {

```

³停止以外の可能性としては、SIGCONT に
 による処理再開が考えられる。

```

int pid;
int cpu;
struct mcctrl_usrdata *ud;
struct file *file;
};

```

2.15.5.11.1 MCEXEC_UP_UTI_ATTR の処理

MCEXEC_UP_UTI_ATTR の要求に対して、以下の処理を行う。

1. 初回の場合、以下を行う。
 - (a) Linux カーネル内の sched_setaffinity と sched_setscheduler_nocheck のアドレスを解決する。
 - (b) ラウンドロビン管理用配列を確保し、0で初期化する。
2. uti_attr のフラグに、背反する組み合わせが指定されている場合、エラーリターンする (-ENOMEM)。
3. mcctrl_usrdata の cpu_topology_list を辿って、McKernel にて clone を発行したスレッド (親スレッド) の McKernel の CPU ID を持つ CPU を検索する。存在しない場合はエラーリターンする (-EINVAL)。
4. 作業用の cpumask を確保し、cpu_active_mask で初期化する。確保できない場合、エラーリターンする (-ENOMEM)。
5. 以下の処理によって、uti_attr のフラグ指定に従い、割り当てる CPU の候補を求め、作業用 cpumask に設定する。
 - (a) フラグに UTI_FLAG_NUMA_SET が設定されている場合、以下の処理を行う。
 - i. mcctrl_usrdata の node_topology_list を辿って、numa_set に設定されている NUMA ID と一致する NUMA ノードを検索し、見付かった NUMA ノードに属す CPU の和集合を求める。
 - ii. 作業用 cpumask と NUMA ノードに属す CPU 集合の積集合を求め、作業用 cpumask に設定する。
 - (b) フラグに UTI_FLAG_SAME_NUMA_DOMAIN か UTI_FLAG_DIFFERENT_NUMA_DOMAIN が設定されている場合、以下の処理を行う。

(c) フラグに UTI_FLAG_CPU_INTENSIVE が設定されている場合、以下の処理を行う。	1
i. 作業用 cpumask から CPU を 1 つ選んで CPU アフィニティに設定する。 (CPU の選択方法は後述)	2 3 4
(d) フラグに UTI_FLAG_HIGH_PRIORITY が設定されている場合、以下の処理を行う。	5
i. 作業用 cpumask から CPU を 1 つ選んで CPU アフィニティに設定する。 (CPU の選択方法は後述)	6 7 8
ii. スケジューラに SCHED_FIFO を設定する。	9
(e) フラグに UTI_FLAG_NON_COOPERATIVE が設定されている場合、以下の処理を行う。	10
i. 作業用 cpumask から CPU を 1 つ選んで CPU アフィニティに設定する。 (CPU の選択方法は後述)	11 12 13
(f) 以上に該当しない場合、作業用 cpumask の内容を CPU アフィニティに設定する。	14
7. 作業用 cpumask を開放する。	15
作業用 cpumask から CPU を 1 つ選択する手順を以下に示す。	16
1. ラウンドロビン管理用配列は CPU ID ごとのマイグレートスレッド数を記録している。 この配列を用いて、作業用 cpumask に含まれかつマイグレートスレッド数が最小となる CPU ID を求める。	17 18 19
2. compare and swap によってラウンドロビン管理用配列を更新 (1 加算) する。 失敗した場合は、1 から再度行う。	20 21
3. 更新した CPU ID 以外の作業用 cpumask のビットをクリアする。	22
2.15.5.11.2 MCEXEC_UP_SIG_THREAD の処理	23
MCEXEC_UP_SIG_THREAD の要求に対して、以下の処理を行う。	24
1. 要求元スレッドが Linux にマイグレートされたスレッドでない場合、 EINVAL でエラーリターンする。	25 26
2. 引数に従い、スレッドの FS ベースアドレスを切り替える。	27
2.15.5.11.3 MCEXEC_UP_SYSCALL_THREAD の処理	28
MCEXEC_UP_SYSCALL_THREAD の要求に対して、以下の処理を行う。	29
1. システムコール番号とシステムコール引数を syscall_request 構造体に 設定する。	30 31
2. util_migrate_inter_kernel の要求を検索する。存在しない場合は ENOENT でエラーリターンする。	32 33

- 1 3. `wait_queue_head_list_node` を作成し、システムコールの結果待ちに
2 備える。
- 3 4. `util_migrate_inter_kernel` のレスポンスに `syscall_request` の物理アドレス
4 設定する。また、レスポンスの状態をシステムコール要求に設定する。
- 5 5. `_notify_syscall_requester` を呼び出して、McKernel 側に
6 レスポンスの変更を通知する。
- 7 6. `wait_event_interruptible` によって、システムコール完了を待つ。
- 8 7. 結果を返却する。

9 2.15.6 実装の制限

10 制限は以下の通り。

- 11 ● Linux CPU にマイグレートしたスレッドに対して `ptrace` システムコールによるトレー
12 スは行えない。
- 13 ● Linux CPU にマイグレートしたスレッドが発行可能なシステムコールの種類は上記で
14 説明したもののみである。
- 15 ● Linux CPU にマイグレートしたスレッドを、再度 McKernel に移動することはできない。

16 2.16 高速プロセス起動

17 McKernel は、複数種の MPI プログラムを起動しさらにそれを繰り返すジョブにおいて MPI
18 プログラム起動時間を短縮する機能を提供する。

19 利用例としては、アンサンブルシミュレーションとデータ同化を繰り返す気象アプリケー
20 ションが挙げられる。

21 起動時間の短縮は、それぞれの MPI プログラムを常駐させて、起動を停止状態からの復
22 帰で置き換えることで実現する。

23 高速プロセス起動は以下の機能から構成される。

- 24 ● プロセス実行停止および停止からの再開機能

25
26 MPI プログラムが、本来プロセスとして終了するタイミングで終了せずに再開指示待
27 ち状態で停止できるようにする。また、再開指示を受けて停止状態から復帰できるよ
28 うにする。ライブラリ関数として実装する。

- 29 ● MPI プログラムの繰り返し起動指示機能

30
31 MPI プログラムの実行回数を把握し、初回は `mpiexec` を用いて起動し、2 回目以降は
32 停止しているプロセスを再開する。`ql_mpiexec.start` と呼ぶユーザコマンドとして実
33 装する。

- MPI プログラムの繰り返し起動からの終了指示機能

再開指示待ち状態で停止している MPI プロセスを終了させる。 `ql_mpiexec_finalize` と呼ぶユーザコマンドとして実装する。

以下、これらの機能の詳細を説明する。

2.16.1 詳細

関連プロセスを表 2.9 示す。

Table 2.9: プロセス一覧

プロセス	説明
<code>ql_mpiexec.start</code>	ユーザがジョブスクリプトに記述して用いる、各回の計算開始を指示するコマンドである。指示は <code>ql_talker</code> と <code>ql_server</code> を経由して MPI プログラムに送られる。なお、一回の計算が完了すると、本コマンドは終了する。
<code>ql_mpiexec.finalize</code>	ユーザがジョブスクリプトに記述して用いる、MPI プログラムの実行終了を指示するコマンドである。指示は <code>ql_talker</code> と <code>ql_server</code> を経由して MPI プログラムに送られる。なお、MPI プログラムの終了と共に本コマンドは終了する。
mpiexec 監視	<code>mpiexec</code> プロセスの起動、死活監視、標準入出力およびエラー出力のリダイレクトを行う。本プロセスは <code>ql_mpiexec.start</code> コマンドより起動され常駐する。また、 <code>mpiexec</code> プロセス終了と共に終了する。
<code>mpiexec</code>	MPI プロセスを生成する。本プロセスは <code>mpiexec</code> 監視プロセスの子プロセスとして起動される。すべてのランク終了と共に終了する。
<code>mcexec</code>	ホスト Linux 上で McKernel のユーザプログラムプロセスを生成・管理する。
<code>ql_server</code>	高速プロセス起動対象の MPI プログラムを記録し、 <code>ql_mpiexec_{start,finalize}</code> コマンドの指示を MPI プロセスに送る。 <code>ql_server</code> は <code>ql_mpiexec.start</code> コマンドから <code>ssh</code> で起動され常駐する。また、高速プロセス起動対象の MPI プログラムが全て終了した時点で終了する。
<code>ql_talker</code>	<code>ql_mpiexec_{start,finalize}</code> と <code>ql_server</code> との間の通信を仲介する。 <code>ql_mpiexec_{start,finalize}</code> コマンドから <code>ssh</code> で <code>ql_server</code> が実行されている計算ノードに起動される。本プロセスは指示完了と共に終了する。

プロセス構成を図 2.22 に示す。

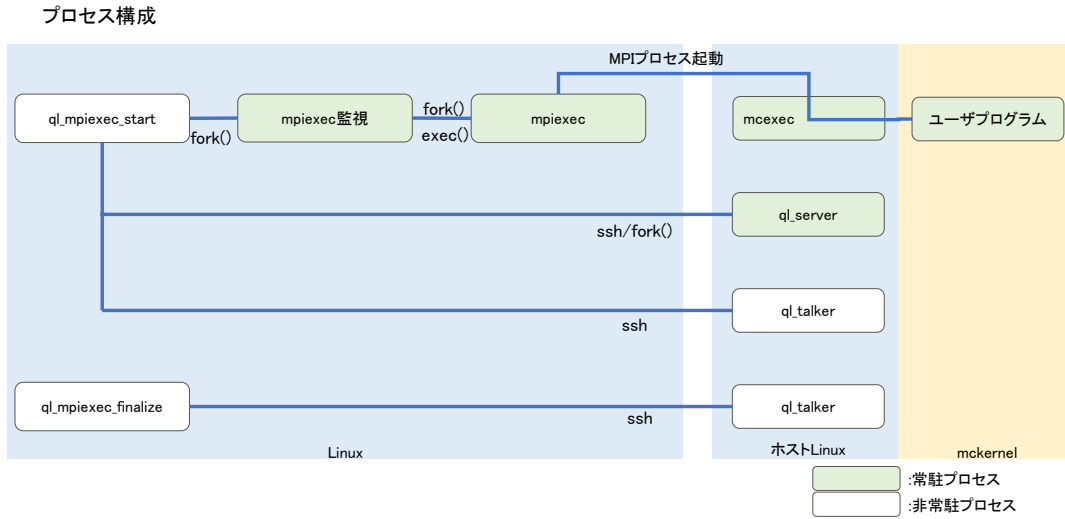


Figure 2.22: プロセス構成

- 1 プロセス間通信で用いるコマンドは以下の通り。

コマンド	マクロ	説明
E コマンド	QL_EXEC_END	mexec が ql_server 経由で ql_mpiexec.start へ各回の計算完了を通知する際に用いる
F コマンド	QL_RET_FINAL	mpiexec 監視プロセスが ql_server へ MPI プログラムの終了を通知する際に用いる
R コマンド	QL_RET_RESUME	ql_server が mexec へ待ち状態からの起床を指示する際に用いる
N コマンド	QL_COM_CONN	ql_mpiexec.start が ql_server に MPI プログラムの登録を依頼する際に用いる
A コマンド	QL_AB_END	ql_server が他プロセスからのコマンドを処理する際に、コマンド転送先プロセスを見つけれなかった場合に返答として用いる

- 2 プロセス間通信の通信電文フォーマットは以下の通り。

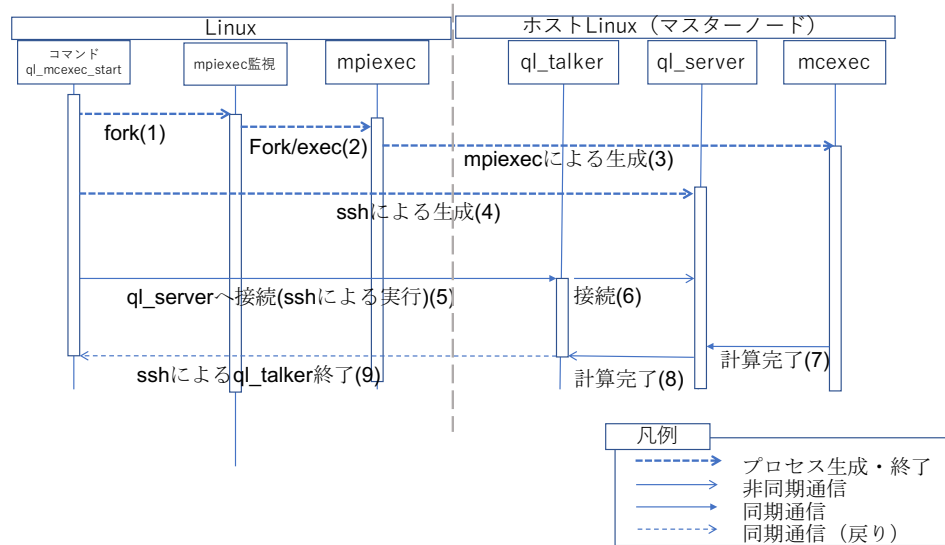
3
4 <コマンド> <データサイズ> <データ>

- 5
6 各フィールドのサイズ、意味は以下の通り。

フィールド	サイズ	説明
<コマンド>	1 byte	コマンド
<データサイズ>	4 byte	byte 単位のデータサイズを表す 16 進数文字列
<データ>	可変	データを表す文字列

- 7 関連コマンドと関連プロセスの動作フローを図 2.23 を用いて説明する。

MPIプログラム初回実行



MPIプログラム再開から終了まで

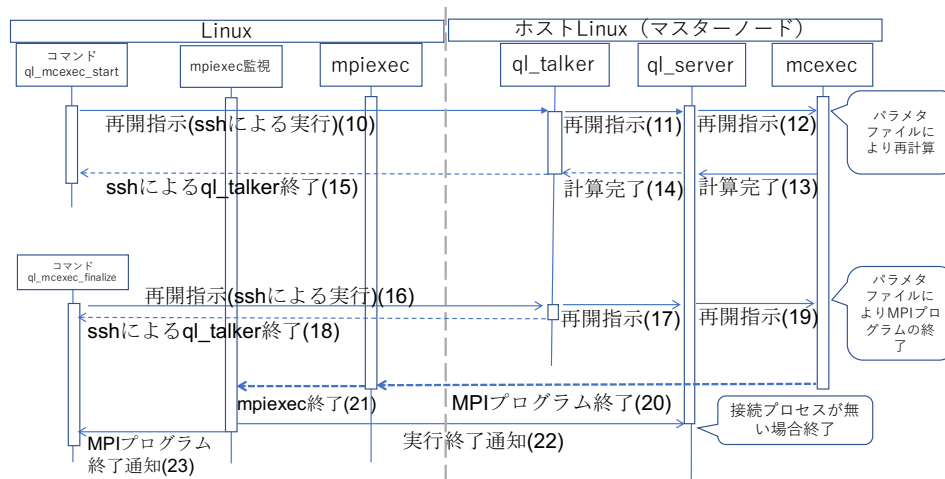


Figure 2.23: 関連コマンドと関連プロセスの動作フロー

- 01 MPIプログラムの初回実行時は、ql_mpiexec_start から、mpiexec 監視プロセスを fork() で常駐プロセスとして起動する。(図の (1)) 1
- 02 mpiexec 監視プロセスは、mpiexec を fork()/exec() で起動する。また、mpiexec の標準入出力およびエラー出力を ql_mpiexec_start へリダイレクトする。(図の (2)) 2
- 03 mpiexec が mexec を用いて MPI プロセスを McKernel 上に起動する。(図の (3)) 3
- 04 ql_mpiexec_start が ssh でホスト Linux 上に ql_server を常駐プロセスとして起動する。(図の (4)) 4
- 05 ql_mpiexec_start が ssh でホスト Linux 上に ql_talker を起動する。(図の (5)) 5
- 06 ql_talker は ql_server に N コマンドを送信し、ql_server から計算完了の返信を待つ。ql_server は当該 MPI プログラムの存在を管理表に記録する。(図の (6)) 6
- 07 7
- 08 8
- 09 9
- 10 10

1 07 `mcexec` は MPI プログラムの一回の計算完了後
2 `ql_server` へ計算完了を意味する E コマンドを送信し、返信を待つ。(図の (7))

3 08 `ql_server` が `ql_talker` へ計算完了を意味する E コマンドを送信する。(図の (8))

4 09 `ql_talker` は E コマンドを受け取り、正常終了する。`ql_mpiexec_start` は `ql_talker`
5 の終了を受けてリダイレクトしている標準入出力およびエラー出力をクローズし終了す
6 る。(図の (9))

7 10 `ql_mpiexec_start` は MPI プログラムの次の計算の開始時、`mpiexec` 監視プロセスに依
8 頼して `mpiexec` の標準入出力およびエラー出力を自身にリダイレクトする。また、`ssh`
9 でホスト Linux 上に `ql_talker` を起動する。(図の (10))

10 11 `ql_talker` は `ql_server` へ再開指示を意味する R コマンドを送信し、`ql_server` から
11 の返信を待つ。(図の (11))

12 12 `ql_server` は `mcexec` へ R コマンドを送信する。
13 MPI プログラムはパラメタファイルを読み、再開指示であることを確認し、引数と環
14 境変数をパラメタファイルに指定されたものに置き換え、次の回の計算を行う。(図の
15 (12))

16 13 `mcexec` は一回の計算完了後
17 `ql_server` へ計算完了 (E コマンド) を送信し、返信を待つ。(図の (13))

18 14 `ql_server` が `ql_talker` へ計算完了 (E コマンド) を送信する。(図の (14))

19 15 `ql_talker` は E コマンドを受け取り、正常終了する。`ql_mpiexec_start` は終了を受けて
20 リダイレクトしている標準入出力およびエラー出力をクローズし終了する。(図の (15))

21 16 `ql_mpiexec_finalize` は `mpiexec` 監視プロセスに依頼して `mpiexec` の標準入出力およ
22 びエラー出力を自身へリダイレクトする。また `ssh` でホスト Linux 上に `ql_talker` を
23 起動する。(図の (16))

24 17 `ql_talker` は `ql_server` へ再開指示を意味する R コマンドを送信し、終了する。(図の
25 (17) (18))

26 18 `ql_server` は `mcexec` へ R コマンドを送信する。
27 MPI プロセスは R コマンドを受けて、パラメタファイルを読み終了指示であることを
28 確認し終了処理を行う。(図の (19))

29 19 `mcexec` は MPI プロセスの終了と共に終了する。(図の (20))

30 20 `mpiexec` は全ランクの終了を待って終了する。`mpiexec` 監視プロセスは `mpiexec` プロ
31 セス終了を検知し、戻り値を取得する。(図の (21))

32 21 `mpiexec` 監視プロセスは `ql_talker` 経由で `ql_server` へ実行終了を意味する F コマン
33 ドを送信する。`ql_server` は当該 MPI プログラムを管理表から削除する。また、管理
34 表が空になった場合は終了する。(図の (22))

35 22 `mpiexec` 監視プロセスは `ql_mpiexec_finalize` へ MPI プログラムの終了を通知し、戻
36 り値を渡し、終了する。(図の (23))

23	<code>ql_mpiexec_finalize</code> は MPI プログラムの終了通知を受けて、リダイレクトしている標準入出力およびエラー出力をクローズし、 <code>mpiexec</code> の戻り値を自身の戻り値として終了する。	1 2 3
2.16.2 MPI プロセス起動指示コマンド		
書式		
	<code>ql_mpiexec_start -machinefile <hostfile_path> [<mpiopts>...] <exe> [<args>..6]</code>	5
説明		
	処理ステップは以下の通り。	7 8 9
1	ホストファイルの内容、 <code>mpiexec</code> への引数、実行可能ファイル名から md5 ハッシュにより MPI プログラム ID を作成する。ID を環境変数 <code>QL_NAME</code> に記録する。	10 11
2	<code>ql_server</code> との通信のためのソケットファイルのパスを環境変数 <code>QL_SOCKET_FILE</code> に記録する。また、 <code>ssh</code> でホストファイルの先頭のホスト（以降、マスターノードと呼ぶ）上に <code>ql_server</code> を起動する。起動失敗した場合は終了コード (-1) で終了する。	12 13 14
3	<code>mpiexec</code> 監視プロセスとの通信のためのソケットファイルが存在しない場合は、ソケットファイルを作成後、 <code>mpiexec</code> 監視プロセスを <code>fork</code> する。 <code>mpiexec</code> 監視プロセスは、 <code>mpiexec</code> を <code>fork/exec</code> で生成する。また、 <code>mpiexec</code> の標準入出力およびエラー出力を無名パイプ（以降、リダイレクト用パイプと呼ぶ）の片方の端に接続する。	15 16 17 18 19
4	再開指示のためパラメタファイルを作成する。	20
5	<code>mpiexec</code> 監視プロセスに、自身の標準入出力、エラー出力のファイルディスクリプタ番号を渡す。 <code>mpiexec</code> 監視プロセスは当該ファイルディスクリプタをリダイレクト用パイプの空いている方の端に接続する。	21 22 23
6	第 1 回の計算開始時は <code>ssh</code> で <code>ql_talker</code> をマスターノード上に起動する。 <code>ql_talker</code> は、 <code>ql_server</code> へ接続を意味する <code>N</code> コマンドを送信し、各回の計算完了を意味する <code>E</code> コマンドを受信するまで待機する。	24 25 26
7	第 2 回目以降の計算開始時は、 <code>ssh</code> で <code>ql_talker</code> をマスターノード上に起動する。 <code>ql_talker</code> は、 <code>ql_server</code> へ再開指示を意味する <code>R</code> コマンドを送信し、各回の計算完了を意味する <code>E</code> コマンド受信まで待機する。	27 28 29
8	<code>ql_talker</code> コマンドがその終了をもって <code>ql_mpiexec_start</code> へ計算完了を通知する。 <code>ql_mpiexec_start</code> は <code>mpiexec</code> 監視プロセスと通信を行って <code>mpiexec</code> が終了していないことを確認する。 <code>mpiexec</code> が終了している場合は、 <code>mpiexec</code> の終了コードを取得する。	30 31 32
9	各回の計算完了の場合はパラメタファイルを削除し 0 を返し終了する。 <code>mpiexec</code> が終了していた場合は、その終了コードを自身の終了コードに設定して終了する。	33 34
	<code>ql_mpiexec_start</code> が使用する環境変数は以下の通り。	35

名前	説明	作成・参照
QL_NAME	MPI プログラム ID	作成
QL_SOCKET_FILE	mcexec と ql_server との接続に用いるソケットファイル名	作成

1 ql_mpiexec_start が使用するファイルは以下の通り。

ファイル名	説明	作成・参照
#{QL_SOCKET_PATH}/ql_sock/<MPI プログラム ID>.s	ql_talker と ql_server との間の通信に用いるソケットファイル	作成/参照
#{QL_PARAM_PATH}/<MPI プログラム ID>.param	ql_mpiexec_start から mcexec への指示と、次回の計算に使用する引数と環境変数を記載するコマンド・パラメタファイル	作成

2 パラメタファイルは、ql_mpiexec_{start,finalize} から mcexec への指示を記載する。
3 内容は、起床後の動作および次の回の計算に必要な引数などのデータである。
4 フォーマットは以下の通り。

5
6 <ヘッダ部>

7
8 <データ部>

9
10 [<データ部>...]

11

12 <ヘッダ部>のフォーマットは以下の通り。

13

14 0 COM=<mcexec への指示> <引数の数> <環境変数定義の数>

15

16 それぞれのフィールドの意味及び取りうる値は以下の通り。

フィールド	説明
<mcexec への指示>	R:次の回の計算開始、F:MPI プロセスの終了
<引数の数>	データ部に存在する引数の数
<環境変数定義の数>	データ部に存在する環境変数定義の数

17 <データ部>のフォーマットは以下の通り。

18

19 <種別> <データ長> <データ値>

20

21 それぞれのフィールドの意味及び取りうる値は以下の通り。

フィールド	説明
<種別>	1:引数、2:環境変数定義
<データ長>	データ長
<データ値>	文字列

2.16.3 MPI プロセス終了指示コマンド 1

書式 2

```
ql_mpiexec_finalize -machinefile <hostfile> [<mpiopts>...] <exe> 3
```

説明 4

処理ステップは以下の通り。 5

- 1 ホストファイルの内容、`mpiexec` への引数、実行可能ファイル名から md5 ハッシュにより MPI プログラム ID を作成し、環境変数 `QL_NAME` に記録する。 7
- 2 `mpiexec` 監視プロセスと通信を行うソケットファイルの存在を確認し、存在しない場合は `ql_mpiexec_start` が実行されていないと判断し 1 を返し終了する。 9
- 3 終了指示のためのパラメタファイルを作成する。 11
- 4 `ql_mpiexec_finalize` は自身の標準入出力とエラー出力のファイルディスクリプタ番号を `mpiexec` 監視プロセスに渡す。`mpiexec` 監視プロセスは当該ファイルディスクリプタをリダイレクト用パイプの空いている方の端に接続する。 12
- 5 `ssh` でマスターノード上に `ql_talker` を起動する。`ql_talker` は、`ql_server` へ再開指示を意味する R コマンドを送信する。`ql_mpiexec_start` の場合と異なり `ql_talker` は `ql_server` からの返答を待つことなく終了する。 16
- 6 `mpiexec` 監視プロセスは `mpiexec` の終了時にその終了コードを自身の終了コードに設定し終了する。 18
- 7 `mpiexec` 監視プロセスの終了を受けてパラメタファイルを削除する。また `mpiexec` 監視プロセスから渡された `mpiexec` の終了コードを自身の終了コードに設定して終了する。 21

`ql_mpiexec_finalize` で作成／参照する環境変数は以下の通り。 22

名前	説明	作成・参照
<code>QL_NAME</code>	MPI プログラム ID	作成

`ql_mpiexec_finalize` で作成／参照するファイルは以下の通り。 23

ファイル名	説明	作成・参照
<code>\${QL_SOCKET_PATH}/ql_sock/<MPI プログラム ID>.s</code>	<code>ql_talker</code> と <code>ql_server</code> との間の通信に用いるソケットファイル	参照
<code>\${QL_PARAM_PATH}/<MPI プログラム ID>.param</code>	<code>ql_mpiexec_finalize</code> から <code>mcexec</code> への指示を記載するコマンドファイル。	作成

2.16.4 MPI 実行環境初期化関数 (C 言語) 24

書式 25

```
int MPI_Init(int *argc, char ***argv) 26
```

1 説明

2

3 `argc`, `argv` を用いて高速プロセス起動の初期化を行う。
4 本関数は PMPI インタフェースにより `MPI_Init()` を置き換える。
5 処理のステップは以下の通り。

- 6 1 `PMPI_init()` 関数を呼び出し、MPI 環境を初期化する。また、引数情報を取得する。
- 7 2 `PMPI_init()` が正常終了した場合、`ql_init()` 関数を呼び出し、高速プロセス起動を初
8 期化する。
- 9 3 `PMPI_init()` の戻り値自身の戻り値に設定して戻る。

10 戻り値

11

戻り値	説明
<code>MPI_SUCCESS</code>	正常終了
<code>MPI_ERR_OTHER</code>	<code>MPI_init()</code> が複数回実行された

12 2.16.5 MPI 実行環境初期化関数 (fortran)

13 書式

14 `subroutine MPI_INIT(INT ierr)`

15 説明

16 Fortran 環境において、高速プロセス起動のための初期化を行う。本関数は、PMPI インタ
17 フェースにより `MPI_INIT` を置き換えることで実装される。
18 処理のステップは以下の通り。

- 19 1 `pmpi_init_()` が存在していない場合、`ierr` に `MPI_ERR_OTHER` をセットして戻る。
- 20 2 `pmpi_init_()` を呼び出し、MPI 環境を初期化する。
- 21 3 戻り値 `ierr` が `MPI_SUCCESS` の場合、`ql_init()` 関数を呼び出し、高速プロセス起動を
22 初期化する。

23 なお、Fortran コンパイラは GNU Fortran Compiler もしくは Intel Fortran Compiler をサ
24 ポートする。Intel Fortran Compiler を使用する場合は、コンパイルオプションに `-shared-intel`
25 を指定する必要がある。

26 戻り値

27

戻り値	説明
MPI_SUCCESS	正常終了
MPI_ERR_OTHER	MPI_init() が複数回実行された

2.16.6 計算の再開・終了関数 (C 言語)

書式

```
ql_client(int *argc, char ***argv)
```

説明

処理のステップは以下の通り。

- 1 当該プロセスが `ql_mpiexec_start` により起動されていない場合は、`QL_EXIT` を返す。
- 2 スレッドの停止を行う。また `PMI_Barrier()` で計算完了同期を行う。
- 3 システムコールによりカーネルモードに移行し、`mcexec` に `ql_mpiexec_{start,finalize}` による指示待ちを依頼する。
- 4 指示待ちから復帰し、パラメタファイルを参照して指示を確認する。指示が次の回の計算開始の場合、パラメタファイルを用いて計算のための引数と環境変数を設定する。
- 5 スレッドの再開を行う。
- 6 指示が次の回の計算開始の場合 `QL_CONTINUE`、MPI プロセスの終了の場合 `QL_EXIT` を返す。

2.16.7 計算の再開・終了関数 (Fortran)

書式

```
subroutine QL_CLIENT(ierr)
```

説明

`ql_client()` を呼び、その戻り値を `ierr` に格納して戻る。

2.16.8 初期化関数

書式

```
int ql_init(int argc, char **argv)
```

説明

`MPI_Init()` から呼びされ、高速プロセス起動の初期化を行う。
処理ステップは以下の通り。

- 1 環境変数 QL_NAME から MPI プログラム ID を取得する。
- 2 取得できなかった場合、ql_mpiexec_start から起動されていないと判断し、QL_NORMAL
- 3 を返す。
- 4 3 MPI プログラム ID から、パラメタファイルのパス
- 5 を作成する。
- 6 4 QL_SUCCESS を返す。

7 戻り値

8

戻り値	説明
QL_SUCCESS	高速プロセス起動の初期化成功
QL_NORMAL	当該プロセスが ql_mpiexec_start から起動されていない

9 2.16.9 計算ノードの管理サーバ

10 書式

11 ql_server

12 説明

13

14 ql_server は、ql_mpiexec_start により RANK#0 が存在する計算ノード上に起動され、
15 以下の処理を行う。

- 16 1 既に ql_server が起動されている場合は、-1 を返して終了する。
- 17 2 mcexec、ql_talker との通信に用いるユニックスドメインソケットをオープンする。
- 18 3 select() で当該ソケットを監視する。
- 19 4 電文を読み込み、コマンドとデータを取得する。
- 20 5 ql_talker から N コマンドを受け取った際は、対応する MPI プログラムを管理表に登
21 録する。また、MPI プロセス ID をインデックスとし ql_server に接続しているプロセ
22 スを返すマップ（接続マップと呼ぶ）に ql_talker を登録する。

23

24 接続マップは struct client_fd で実装される。

25 struct client_fd は以下のように定義される。

26

```
27 struct client_fd {
```

28

```
29     int fd;           // 接続元プロセスのファイルディスクリプタ
```

30

```
31     int client;      // 接続元プロセスの種別
```

```

char *name;      // MPI プログラム ID
int status;     // 現在実行中の通信コマンド
};

```

6 mcexec から E コマンドを受けとった際は、`ql_mpiexec_{start,finalize}` の指示があるまで待たせる。また、`mcexec` を接続マップに登録する。さらに、接続マップを用いて対応する `ql_talker` を見つけ、それに対して E コマンドを送信する。

7 `ql_talker` から R コマンドを受けとった際は、`ql_talker` を接続マップに登録する。また、接続マップを用いて対応する `mcexec` プロセスを見つけ、それに対して R コマンドを送信することで `mcexec` を起床する。

8 `mpiexec` 監視プロセスから F コマンドを受け取った際は、対応する MPI プログラムを管理表から削除する。管理表が空になった場合は `ql_server` 自身も終了する。

`ql_talker` や `mcexec` が `ql_server` と通信するために使用するソケットファイルは、環境変数 `QL_SOCKET_PATH` が定義されている場合は `#{QL_SOCKET_PATH}/ql_sock` 下に、定義されていない場合は `/run/user/ユーザ ID/ql_sock` 下に作成される。当該ディレクトリは `ql_mpiexec_start` コマンドが実行されるノードとランク #0 が実行されるノードからアクセスできる必要がある。

2.16.10 指示中継コマンド

書式

```
ql_talker <send_command> <receive_command> <MPI_Program_ID>
```

引数

引数	説明
<send_command>	<code>ql_server</code> へ送信するコマンド (1 文字) を指定する。
<receive_command>	<code>ql_server</code> からの受信を期待するコマンド (1 文字) を指定する。受信を待たずに終了する場合は、"-n" を指定する。
<MPI_Program_ID>	MPI プログラム ID を指定する。

説明

`ql_mpiexec_{start,finalize}` から `ql_server` が動作するノード上に起動され、`ql_server` に `<send_command>` で指定されたコマンドを送り、`<receive_command>` で指定された応答を待つ。

`ql_server` とはユニックスドメインソケットを用いて通信する。処理ステップは以下の通り。

- 1 1 argc の数をチェックし、4 未満の場合は終了コード-1 で終了する。
- 2 2 環境変数を参照して ql_server との接続に用いるユニックスドメインソケットを見つけ、ql_server に接続する。
- 3
- 4 3 <send_command>と<MPI_Program_ID>より電文を作成し、ql_server へ電文を送信する。
- 5 失敗した場合は終了コード-1 で終了する。
- 6 4 <receive_command>に"-n"が指定されていた場合、終了コード 0 で終了する。
- 7 5 <receive_command>を受信した場合、終了コード 0 で終了する。
- 8 <receive_command>以外の文字列を受信した場合終了コード-2 で終了する。

9 戻り値

10

戻り値	説明
0	正常終了
-1	ソケット通信エラー
-2	<receive_command>以外の文字列を受信

11 2.16.11 swapout システムコール

12 書式

13 int swapout(char *filename, void *workarea, size_t size, int flag)

14 引数

15

引数	説明
filename	スワップファイル名へのポインタ
workarea	作業領域へのポインタ
size	作業領域のサイズ
flag	swapout の動作制御用フラグ

16 説明

17

18 A. スワップアウト処理

19

20

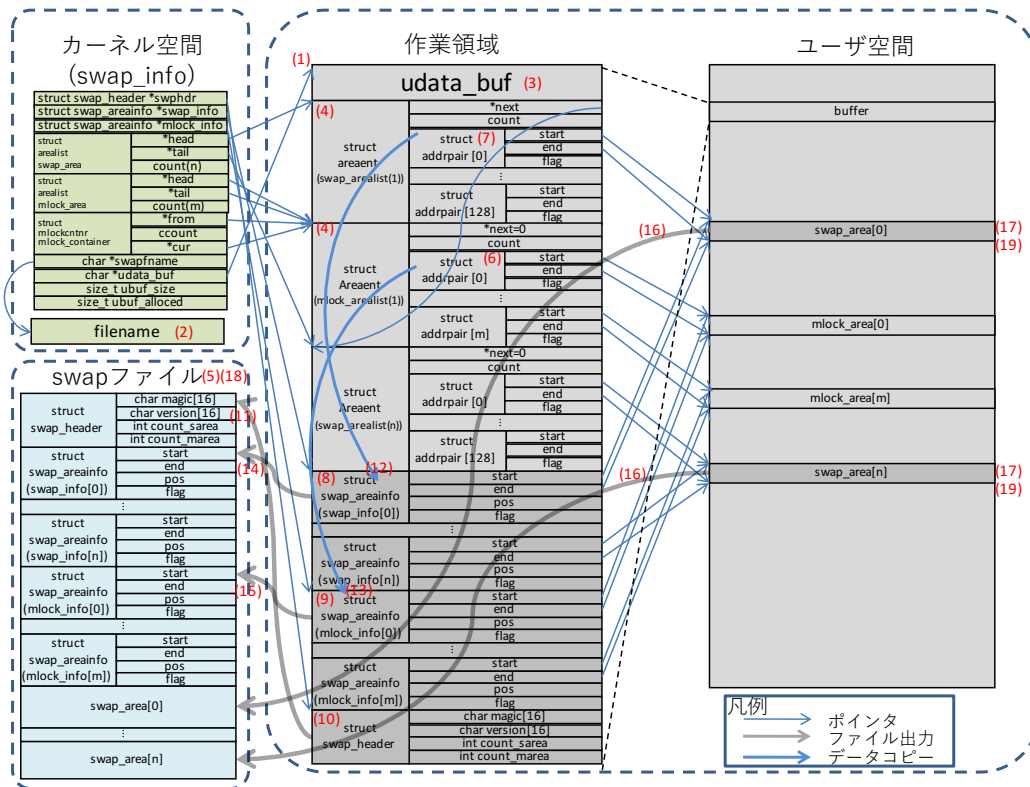


Figure 2.24: スワップアウトの処理フロー

スワップアウトの処理フローを図 2.24 を用いて説明する。

1. McKernel のユーザから渡された作業領域を `mlock()` によりロックする。swapout 情報を管理する `swap_info` 構造体の `udata_buf` メンバに作業領域の先頭アドレスを記録する。(図の (1))
2. 引数で指定されたファイル名を `copy_from_user` でカーネル空間にコピーする。`swap_info` 構造体の `swapfname` メンバにファイル名のアドレスを記録する。(図の (2))
3. 作業領域に汎用バッファ `udata_buf` を割り当てる。(図の (3))
4. 作業領域にスワップエリア管理用リスト `swap_arealist` と `mlock` エリア管理用リスト `mlock_arealist` の領域を割り当てる。(図の (4))
5. swap ファイルを `open()` でオープンする。(図の (5))
6. `lookup_process_memory_range` および `next_process_memory_range` を用いて、ユーザプロセスのメモリ領域を検索し、それぞれについて以下を行う。
 - (a) `mlock()` されている領域の開始アドレス、終了アドレス、`flag` を作業領域の `mlock_arealist` に記録する。(図の (6))
 - (b) `mlock()` されていない領域の開始アドレス、終了アドレス、`flag` を作業領域の `swap_arealist` に記録する。(図の (7))

- 1 7. 作業領域の `swap_arealist` のエントリ数と同数のエントリを持つ `swap_info` 配列を作
2 業領域に割り当てる。カーネル領域の `swap_info` 構造体の `swap_info` メンバに作業領
3 域の `swap_info` 配列の先頭アドレスを記録する。 (図の (8))
- 4 8. 作業領域の `mlock_arealist` のエントリ数と同数のエントリを持つ `mlock_info` 配列を
5 作業領域に割り当てる。カーネル領域の `swap_info` 構造体の `mlock_info` メンバに作業
6 領域の `mlock_info` 配列の先頭アドレスを記録する。 (図の (9))
- 7 9. 作業領域に `swap_header` を割り当てる。カーネル領域の `swap_info` 構造体の `swphdr` メ
8 ンバに先頭アドレスを記録する。 (図の (10))
- 9 10. 作業領域の `swap_header` の `magic` メンバに”McKernel swap”、`version` メンバに”0.9.0”、
10 `count_sarea` メンバに `swap_arealist` のエントリ数、`count_marea` メンバに `mlock_`
11 `arealist` のエントリ数を記録する。
- 12 11. 作業領域の `swap_header` を `write()` を用いてスワップファイルへ書き出す。 (図の (11))
- 13 12. 作業領域の `swap_arealist` のリスト形式データを作業領域の `swap_info` 配列へコピー
14 する。 (図の (12))
- 15 13. 作業領域の `mlock_arealist` のリスト形式データを作業領域の `mlock_info` 配列へコ
16 ピーする。 (図の (13))
- 17 14. 作業領域の `swap_info` 配列を `write()` を用いてスワップファイルへ書き出す。 (図の
18 (14))
- 19 15. 作業領域の `mlock_info` 配列を `write()` を用いてスワップファイルへ書き出す。 (図の
20 (15))
- 21 16. 作業領域の `swap_info` の情報を用いて、ユーザプロセスのメモリ領域のうち、スワップ
22 アウト対象となっているものを `write()` を用いてスワップファイルへ出力する。 (図の
23 (16))
- 24 17. スワップアウト対象となっているメモリ領域のうち、McKernel 側でマップされている
25 ものを `ihk_mc_pt_free_range()` でアンマップする。 (図の (17))
- 26 18. スワップファイルを `close()` を用いてクローズする。 (図の (18))
- 27 19. スワップアウト対象となっているメモリ領域のうち、Linux 側でマップされているもの
28 を `mcexec` に依頼することでアンマップする。 (図の (19))

29 B. スワップイン処理

30
31

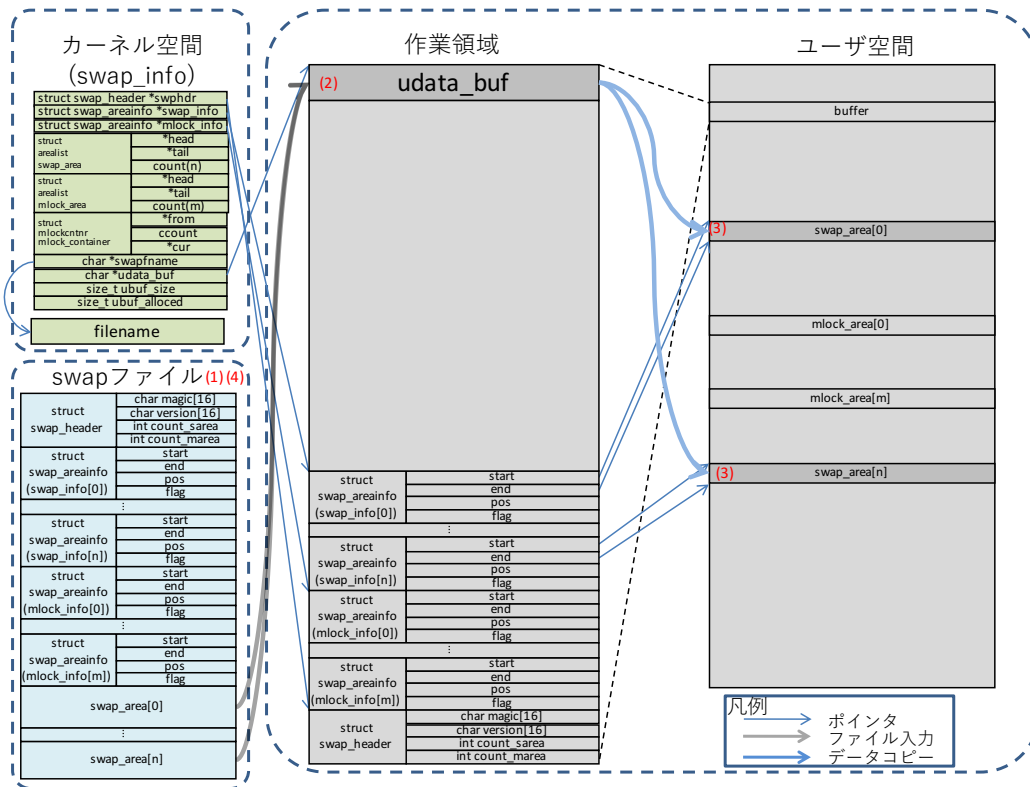


Figure 2.25: スワップインの処理フロー

スワップインの処理フローを図 2.25 を用いて説明する。

1. スワップファイルを `open()` を用いてオープンする。 (図の (1))
2. スワップイン対象アドレス範囲を記録している `swap_info` 配列の各エントリに対して以下を行う。なお、ユーザ空間の作業領域はスワップアウトを経ても残っているため、`swap_info` 配列をファイルから取得する必要はない。
 - (a) `read()` を用いてスワップファイルから作業領域の `udata_buf` へスワップイン対象のメモリ内容をコピーする。 (図の (2))
 - (b) `copy_to_user` を用いて、作業領域の `udata_buf` からユーザプロセスのメモリ領域へ、スワップイン対象のメモリ内容をコピーする。 (図の (3))
3. スワップファイルを `close()` を用いてクローズする。 (図の (4))

2.17 Portability

IHK/McKernel has been designed not only for post K computer but also for other manycore architectures, including Intel Xeon phi. In order to make the source code portable as much as possible. The following is coding convention of IHK/McKernel. The directories for architecture dependent and independent source codes are created and codes are separately stored into those two directories.

1 That is, source codes, including header files, for some specific
2 architecture are located in its architecture dependent directory.
3 The source codes, accessing some hardware registers, are hardware
4 specific, and thus those are machine dependent. Low-level interrupt
5 handlers, some memory management codes, context switch codes, and
6 signaling codes are the examples. Those source codes are located in an
7 architecture dependent directory.
8 Any program code and header files must not include any machine
9 dependent codes including conditional compile macros, such as
10 `#ifdef ARCH`.
11 As much as possible, we define machine independent interfaces so that
12 those interfaces are implemented for each architecture.

13 **2.18 Formal Verification**

14 Some of the behaviors of McKernel is verified in a formal way by embedding behaviors in
15 code and running a verification engine.

16 We employ an extended version of the ANSI/ISO C Specification Language,
17 whose extensions[1] were developed at the project “Dependable Operating
18 Systems for Embedded Systems Aiming at Practical Applications” in the
19 research area named Core Research for Evolutional Science and
20 Technology (CREST), sponsored by Japan Science and Technology Agency
21 (JST).

22 **2.18.1 Specification Language**

23 The following are expressions defined in the formal specification language.

24 The behavior of each function is formally specified by using those
25 expressions that are written as C comments.

26 `\result` specifies return value.

27 `\interrupt_disabled` the CPU is interruptable if 0,
28 the CPU is not interruptable if 1 or more.

29 `\process_env` the execution is under the user context if 1 or more,
30 the execution is under the kernel context if 0.

31 `\atomicity` the execution is not allowed to block if 1 or more,
32 the execution may be suspended if 0.

33 `\dont_call_schedule` the context switch is not allowed if 1 or more,
34 the context switch is allowed if 0.

35 `is_locked(<pointer variable>)` returns true if a memory block pointed by the pointer
36 variable is the
37 lock status, otherwise
38 returns false.

requires \langle **condition expression** \rangle The condition expression must be satisfied at the beginning of the function execution. 1
2
3

ensures \langle **condition expression** \rangle The condition expression must be satisfied at the end of the function execution. 4
5
6

invariant \langle **condition expression** \rangle The condition expression must be satisfied during the function execution. 7
8

Here is a sample code. 9

```

/*@ 10
11
12
    @ behavior valid_vector: 13
14
15    @  assumes 32 <= vector <= 255; 15
16
17    @  requires \valid(h); 17
18
19    @  assigns handlers[vector-32]; 19
20
21    @  ensures \result == 0; 21
22
23    @ behavior invalid_vector: 23
24
25    @  assumes (vector < 32) || (255 < vector); 25
26
27    @  assigns \nothing; 27
28
29    @  ensures \result == -EINVAL; 29
30
31    @*/ 31
32
int ihk_mc_register_interrupt_handler(int vector, 33
34
                                     struct ihk_mc_interrupt_handler *h) 35
36
{ 37
38
    if (vector < 32 || vector > 255) { 39
40
        return -EINVAL; 41
42
    } 43
44

```

```

1     list_add_tail(&h->list, &handlers[vector - 32]);
2
3     return 0;
4
5 }
6

```

2.19 Limitations

Certain system calls are only partially implemented in McKernel or not conforming Linux API. These are either due to design restrictions of the proxy approach or because their support is intentionally omitted. Table 2.10 shows the limitations.

Table 2.10: Limitations of McKernel

Function	Description
<code>arch_prctl</code>	It returns the <code>EOPNOTSUPP</code> error when <code>ARCH_SET_GS</code> is passed.
<code>brk</code>	It extends the heap more than <code>requestd</code> when <code>-h (--extend-heap-by=<step></code> option of <code>mcexec</code> is used with the value larger than 4 KiB.
<code>clone</code>	It supports only the following flags. All other flags cause <code>clone()</code> to return error or are simply ignored. <ul style="list-style-type: none"> • <code>CLONE_CHILD_CLEARTID</code> • <code>CLONE_CHILD_SETTID</code> • <code>CLONE_PARENT_SETTID</code> • <code>CLONE_SETTLS</code> • <code>CLONE_SIGHAND</code> • <code>CLONE_VM</code>
<code>getrusage</code>	The time spent is measured in a different way than Linux for <code>RUSAGE_THREAD</code> . That is, time spent in user-mode and kernel-mode are updated when CPU mode changes (i.e. when switching from user-mode to kernel-mode and vice versa).
<code>mbind</code>	Per-memory-range policy can be set but it is not used when allocating physical pages.
<code>set_mempolicy</code>	<ul style="list-style-type: none"> • <code>MPOL_F_RELATIVE_NODES</code> and <code>MPOL_INTERLEAVE</code> flags are not supported. • <code>MPOL_BIND</code> works in the same way as <code>MPOL_PREFERRED</code>. That is, <code>MPOL_BIND</code> doesn't return an error when there is no space left in the NUMA nodes specified, but continues to search space in the other nodes.
<code>migrate_pages</code>	It returns the <code>ENOSYS</code> error.
<code>msync</code>	Only the modified pages mapped by the calling process are written back.
<code>setpriority, getpriority</code>	They could set/get the priority of a random <code>mcexec</code> thread. This is because there's no fixed correspondence between a McKernel thread which issues the system call and a <code>mcexec</code> thread which handles the offload request.
<code>set_rlimit</code>	It sets the limit values but they are not enforced.
<code>set_robust_list</code>	It returns the <code>ENOSYS</code> error.
<code>signalfd</code>	It returns the <code>EOPNOTSUPP</code> error.
<code>signalfd4</code>	It returns a <code>fd</code> , but signal is not notified through the <code>fd</code> .
<code>setfsuid, setfsgid</code>	It cannot change the id of the calling thread. Instead, it changes that of the <code>mcexec</code> worker thread which takes the system-call offload request.
<code>mmap (hugeTLBfs)</code>	The physical pages corresponding to a map are released when no McKernel process exist. The next map gets fresh physical pages.
Sticky bit on executable file	It has no effect.

Chapter 3

【削除】 運用ガイド

本章の想定読者は以下の通り。

- McKernel を用いたシステムを運用するシステム管理者

SMP プロセッサ向け、x86_64 アーキ向けの関連ファイルの場所は以下の通り。
なお、IHK/McKernel のインストールディレクトリを<install>とする。

インストール先	説明
<install>/kmod/ihk.ko	IHK-master core
<install>/kmod/ihk-smp-x86.ko	IHK-master driver
<install>/kmod/mcctrl.ko	Delegator module
<install>/kmod/mcoverlayfs.ko	/sys, /proc のためのファイルシステム重ね合わせカーネルモジュール
<install>/smp-x86/kernel/mckernel.img	カーネルイメージ
<install>/include/mckernel/ihklib_usage	ihk-os_getrusage() で取得された統計情報を解釈するための構造体および定数マクロ

運用向けコマンド・デーモンのファイルの場所は以下の通り。

なお、IHK/McKernel のインストールディレクトリを<install>とする。

インストール先	説明
<install>/sbin/mcreboot.sh	ブートスクリプト
<install>/sbin/mcstop+release.sh	シャットダウンスクリプト
<install>/bin/mcexec	プロセス起動コマンド
<install>/bin/eclair	ダンプ解析ツール
<install>/bin/vmcore2mckdump	ダンプ形式変換ツール

以下、関連コマンドおよび関連関数のインターフェイスを説明する。

3.1 【変更】 インターフェイス

3.1.1 カーネル引数

McKernel のカーネル引数を表 3.1 に示す。

Table 3.1: McKernel のカーネル引数

引数	説明				
<code>hidos</code>	IKC を有効にする。				
<code>dump_level= <dump_level></code>	Linux の panic ハンドラ経由でダンプを行った場合の、ダンプ対象とするメモリ領域の種類を<dump_level>に設定する。設定可能な値は以下の通り。 <table border="1" data-bbox="516 380 1357 443"> <tr> <td>0</td> <td>IHK が McKernel に割り当てたメモリ領域を出力する。</td> </tr> <tr> <td>24</td> <td>カーネルが使用しているメモリ領域を出力する。</td> </tr> </table> 指定がなかった場合は 24 が用いられる。	0	IHK が McKernel に割り当てたメモリ領域を出力する。	24	カーネルが使用しているメモリ領域を出力する。
0	IHK が McKernel に割り当てたメモリ領域を出力する。				
24	カーネルが使用しているメモリ領域を出力する。				
<code>allow_oversubscribe</code>	McKernel に割り当てられた CPU 数より大きい数のスレッドまたはプロセスの生成を許可する。この引数が指定されない場合に、CPU 数より大きい数のスレッドまたはプロセスを <code>clone()</code> , <code>fork()</code> , <code>vfork()</code> など生成しようとすると、当該システムコールが <code>EINVAL</code> エラーを返す。				

3.1.2 ブートスクリプト

1

書式

2

```
mcreboot.sh [-c <cpulist>] [-r <ikcmap>] [-m <memlist>] [-f <facility>] [-o 3  
<chownopt>] [-i <mon_interval>] [-k <redirct_kmsg>] [-q <irq>] [-t] [-d <dump_level>]  
[-0]
```

5

オプション

6

7

オプション	説明				
-c <cpulist>	McKernel に割り当てる CPU のリストを指定する。フォーマットは以下の通り。 <CPU logical id>[,<CPU logical id>...] または <CPU logical id>-<CPU logical id>[,<CPU logical id>-<CPU logical id>...] または両者の混合。				
-r <ikcmap>	McKernel の CPU が IKC メッセージを送る Linux CPU を指定する。フォーマットは以下の通り。 <CPU list>:<CPU logical id>[+<CPU list>:<CPU logical id>...] <CPU list>のフォーマットは-c オプションにおけるものと同じである。 各<CPU list>:<CPU logical id>は<CPU list>で示される McKernel の CPU が<CPU logical id>で示される Linux の CPU に IKC メッセージを送信することを意味する。				
-m <memlist>	McKernel に割り当てるメモリ領域を指定する。フォーマットは以下の通り。 <サイズ> @ <NUMA-node 番号> [, <サイズ> @ <NUMA-node 番号> ...]。				
-f <facility>	ihkmond が使用する syslog プロトコルの facility を指定する。デフォルトは LOG_LOCAL6。				
-o <chownopt>	IHK のデバイスファイル (/dev/mcd*, /dev/mcos*) のオーナーとグループの値を<user>[:<group>] の形式で指定する。デフォルトは mcreboot.sh を実行したユーザ。				
-i <mon_interval>	ihkmond がハングアップ検知のために OS 状態を確認する時間間隔を秒単位で指定する。-1 が指定された場合はハングアップ検知を行わない。指定がない場合はハングアップ検知を行わない。				
-k <redirect_kmsg>	カーネルメッセージの/dev/log へのリダイレクト有無を指定する。0 が指定された場合はリダイレクトを行わず、0 以外が指定された場合はリダイレクトを行う。指定がない場合はリダイレクトを行わない。				
-q <irq>	IHK が使用する IRQ 番号を指定する。指定がない場合は 64-255 の範囲で空いているものを使用する。				
-t	(x86_64 アーキテクチャ固有) Turbo Boost をオンにする。デフォルトはオフ。				
-d <dump_level>	Linux の panic ハンドラ経由でダンプを行った場合の、ダンプ対象とするメモリ領域の種類を<dump_level>に設定する。設定可能な値は以下の通り。 <table border="1" data-bbox="516 1087 1357 1150"> <tr> <td>0</td> <td>IHK が McKernel に割り当てたメモリ領域を出力する。</td> </tr> <tr> <td>24</td> <td>カーネルが使用しているメモリ領域を出力する。</td> </tr> </table> 指定がなかった場合は 24 が用いられる。	0	IHK が McKernel に割り当てたメモリ領域を出力する。	24	カーネルが使用しているメモリ領域を出力する。
0	IHK が McKernel に割り当てたメモリ領域を出力する。				
24	カーネルが使用しているメモリ領域を出力する。				
-0	McKernel に割り当てられた CPU 数より大きい数のスレッドまたはプロセスの生成を許可する。指定がない場合は許可しない。すなわち、CPU 数より大きい数のスレッドまたはプロセスを生成しようとするとエラーとなる。				

1 説明

2

- 3 McKernel 関連カーネルモジュールを insmod し、<cpulist>で指定された CPU と<memlist>で
- 4 指定されたメモリ領域からなるパーティションを作成し、IKC map を<ikcmap>に設定し、前
- 5 記パーティションに McKernel をブートする。

6 戻り値

0	正常終了
0 以外	エラー

3.1.3 シャットダウンスクリプト

書式

```
mcstop+release.sh
```

オプション

なし

説明

McKernel をシャットダウンし、McKernel 用パーティションを削除し、関連カーネルモジュールを `rmmod` する。

戻り値

0	正常終了
0 以外	エラー

3.1.4 プロセス起動コマンド

インターフェイスは第 1.1 節に記載する。

3.1.5 【削除】統計情報取得

バッチジョブスケジューラは、IHK の関数 `ihk_os_getrusage()` を呼ぶことでジョブの統計情報を取得できる（インターフェイスは”IHK Specifications” 参照）。

`ihk_os_getrusage()` は `void *rusage` という引数で結果を返す。

McKernel では `rusage` の実際の型は `struct mckernel_rusage` 型で、以下のように定義される。

```
struct mckernel_rusage {  
    unsigned long memory_stat_rss[IHK_MAX_NUM_PGIZES];  
    /* ユーザのページサイズごとの anonymous ページ使用量現在値 (バイト単位) */  
    unsigned long memory_stat_mapped_file[IHK_MAX_NUM_PGIZES];  
    /* ユーザのページサイズごとの file-backed ページ使用量現在値 (バイト単位) */  
    unsigned long memory_max_usage;  
    /* ユーザのメモリ使用量最大値 (バイト単位) */  
    unsigned long memory_kmem_usage;  
    /* カーネルのメモリ使用量現在値 (バイト単位) */  
};
```

```

1   unsigned long memory_kmem_max_usage;
2
3   /* カーネルのメモリ使用量最大値 (バイト単位) */
4
5   unsigned long memory_numa_stat[IHK_MAX_NUM_NUMA_NODES];
6
7   /* NUMA ごとのユーザのメモリ使用量現在値 (バイト単位) */
8
9   unsigned long cpuacct_stat_system;
10
11  /* システム時間 (USER_HZ 単位) */
12
13  unsigned long cpuacct_stat_user;}
14
15  /* ユーザ時間 (USER_HZ 単位) */
16
17  unsigned long cpuacct_usage;}
18
19  /* ユーザの CPU 時間 (ナノ秒単位) */
20
21  unsigned long cpuacct_usage_percpu[IHK_MAX_NUM_CPUS];
22
23  /* コアごとのユーザの CPU 時間 (ナノ秒単位) */
24
25  int num_threads;
26
27  /* スレッド数現在値 */
28
29  int max_num_threads;
30
31  /* スレッド数最大値 */
32
33  };
34

```

35 `memory_stat_rss` および `memory_stat_mapped_file` のインデックスはサイズによるページ種であり、x86_64 アーキでは以下のように定義される。

```

37
38  #define IHK_OS_PGFSIZE_4KB 0
39
40  #define IHK_OS_PGFSIZE_2MB 1
41
42  #define IHK_OS_PGFSIZE_1GB 2
43

```

44 3.1.6 ダンプ解析コマンド

45 インターフェイスは第 1.2.1 節に記載する。

46 3.1.7 ダンプ形式変換コマンド

47 インターフェイスは第 1.2.2 節に記載する。

3.2 ブート手順

1

mcreboot.sh を用いてブート手順を説明する。

2

スクリプトは以下の通り。

3

```
1 #!/bin/bash
2
3 # IHK SMP-x86 example boot script.
4 # author: Balazs Gerofi <bgerofi@riken.jp>
5 # Copyright (C) 2014 RIKEN AICS
6 #
7 # This is an example script for loading IHK, configuring a partition and
8 # booting McKernel on it. Unless specific CPUs and memory are requested,
9 # the script reserves half of the CPU cores and 512MB of RAM from
10 # NUMA node 0 when IHK is loaded for the first time.
11 # Otherwise, it destroys the current McKernel instance and reboots it using
12 # the same set of resources as it used previously.
13 # Note that the script does not output anything unless an error occurs.
14
15 prefix="/home/takagi/project/os/install"
16 BINDIR="${prefix}/bin"
17 SBINDIR="${prefix}/sbin"
18 ETCDIR=/home/takagi/project/os/install/etc
19 KMODDIR="${prefix}/kmod"
20 KERNDIR="${prefix}/smp-x86/kernel"
21 ENABLEMCOVERLAYFS="yes"
22
23 mem="512M@0"
24 cpus=""
25 ikc_map=""
26
27 if [ "${BASH_VERSINFO[0]}" -lt 4 ]; then
28     echo "You need at least bash-4.0 to run this script." >&2
29     exit 1
30 fi
31
32 redirect_kmsg=0
33 mon_interval="-1"
34 DUMPLEVEL=24
35 facility="LOG_LOCAL6"
36 chown_option='logname 2> /dev/null '
37
38 if [ "'systemctl status irqbalance_mck.service 2> /dev/null |grep -E 'Active: active'" \
39     != "" -o "'systemctl status irqbalance.service 2> /dev/null |grep -E 'Active: active'" \
40     != "" ]; then
41     irqbalance_used="yes"
42 else
43     irqbalance_used="no"
44 fi
45
46 turbo=""
47 ihk_irq=""
48
49 while getopts :tk:c:m:o:f:r:q:i:d: OPT
50 do
51     case ${OPT} in
52         f) facility=${OPTARG}
53            ;;
54         o) chown_option=${OPTARG}
55            ;;
56         k) redirect_kmsg=${OPTARG}
57            ;;
58         c) cpus=${OPTARG}
59            ;;
60         m) mem=${OPTARG}
61            ;;
62         r) ikc_map=${OPTARG}
```

```

63         ;;
64     q) ihk_irq=${OPTARG}
65         ;;
66     t) turbo="turbo"
67         ;;
68     d) DUMP_LEVEL=${OPTARG}
69         ;;
70     i) mon_interval=${OPTARG}
71         ;;
72     *) echo "invalid option -${OPT}" >&2
73         exit 1
74     esac
75 done
76
77 # Start ihkmond
78 pid=$(pidof ihkmond)
79 if [ "${pid}" != "" ]; then
80     sudo kill -9 ${pid} > /dev/null 2> /dev/null
81 fi
82 if [ "${redirect_kmsg}" != "0" -o "${mon_interval}" != "-1" ]; then
83     ${SBINDIR}/ihkmond -f ${facility} -k ${redirect_kmsg} -i ${mon_interval}
84 fi
85 #
86 # Revert any state that has been initialized before the error occurred.
87 #
88 error_exit() {
89     local status=$1
90
91     case $status in
92     mcos_sys_mounted)
93         if [ "$enable_mcoverlay" == "yes" ]; then
94             umount /tmp/mcos/mcos0_sys
95         fi
96         ;&
97     mcos_proc_mounted)
98         if [ "$enable_mcoverlay" == "yes" ]; then
99             umount /tmp/mcos/mcos0_proc
100        fi
101        ;&
102    mcoverlays_loaded)
103        if [ "$enable_mcoverlay" == "yes" ]; then
104            rmmmod mcoverlay 2>/dev/null
105        fi
106        ;&
107    linux_proc_bind_mounted)
108        if [ "$enable_mcoverlay" == "yes" ]; then
109            umount /tmp/mcos/linux_proc
110        fi
111        ;&
112    tmp_mcos_mounted)
113        if [ "$enable_mcoverlay" == "yes" ]; then
114            umount /tmp/mcos
115        fi
116        ;&
117    tmp_mcos_created)
118        if [ "$enable_mcoverlay" == "yes" ]; then
119            rm -rf /tmp/mcos
120        fi
121        ;&
122    os_created)
123        # Destroy all LWK instances
124        if ls /dev/mcos* 1>/dev/null 2>&1; then
125            for i in /dev/mcos*; do
126                ind=$(echo $i|cut -c10-);
127                if ! ${SBINDIR}/ihkconfig 0 destroy $ind; then
128                    echo "warning: failed to destroy LWK instance $ind" >&2
129                fi
130            done

```

```

131         fi
132         ;&
133     mcctrl_loaded)
134         rmmmod mcctrl 2>/dev/null || echo "warning: failed to remove mcctrl" >&2
135         ;&
136     cpus_reserved)
137         cpus='${SBINDIR}/ihkconfig 0 query cpu'
138         if [ "${cpus}" != "" ]; then
139             if ! ${SBINDIR}/ihkconfig 0 release cpu $cpus > /dev/null; then
140                 echo "warning: failed to release CPUs" >&2
141             fi
142         fi
143         ;&
144     mem_reserved)
145         mem='${SBINDIR}/ihkconfig 0 query mem'
146         if [ "${mem}" != "" ]; then
147             if ! ${SBINDIR}/ihkconfig 0 release mem $mem > /dev/null; then
148                 echo "warning: failed to release memory" >&2
149             fi
150         fi
151         ;&
152     ihk_smp_loaded)
153         rmmmod ihk_smp_x86 2>/dev/null || echo "warning: failed to remove ihk_smp_x86" >&2
154         ;&
155     ihk_loaded)
156         rmmmod ihk 2>/dev/null || echo "warning: failed to remove ihk" >&2
157         ;&
158     irqbalance_stopped)
159         if [ "$(systemctl status irqbalance_mck.service 2> /dev/null | '\
160 'grep -E 'Active: active' " != "" ]); then
161             if ! systemctl stop irqbalance_mck.service 2>/dev/null; then
162                 echo "warning: failed to stop irqbalance_mck" >&2
163             fi
164             if ! systemctl disable irqbalance_mck.service >/dev/null 2>/dev/null; then
165                 echo "warning: failed to disable irqbalance_mck" >&2
166             fi
167             if ! etcdctl=/home/takagi/project/os/install/etc perl -e \
168 '$etcdctl=${ENV{'etcdctl'}}; @files = grep { -f } glob "$etcdctl/proc/irq/*/smp-affinity";'\
169 ' foreach $file (@files) { $dest = substr($file, length($etcdctl));'\
170 ' if(0) {print "cp $file $dest\n";} system("cp $file $dest 2>/dev/null"); }'; then
171                 echo "warning: failed to restore /proc/irq/*/smp-affinity" >&2
172             fi
173             if ! systemctl start irqbalance.service; then
174                 echo "warning: failed to start irqbalance" >&2;
175             fi
176         fi
177         ;&
178     initial)
179         # Nothing more to revert
180         ;;
181     esac
182
183     exit 1
184 }
185
186 ihk_ikc_irq_core=0
187
188 release='uname -r'
189 major='echo ${release} | sed -e 's/^\([0-9]*\).*\/\1/'
190 minor='echo ${release} | sed -e 's/^[0-9]*\.\([0-9]*\).*\/\1/'
191 patch='echo ${release} | sed -e 's/^[0-9]*\.[0-9]*\.\([0-9]*\).*\/\1/'
192 linux_version_code='expr \( ${major} \* 65536 \) + \( ${minor} \* 256 \) + ${patch}'
193 rhel_release='echo ${release} | sed -e 's/^[0-9]*\.[0-9]*\.[0-9]*-\([0-9]*\).*\/\1/'
194 if [ "${release}" = "${rhel_release}" ]; then
195     rhel_release=""
196 fi
197
198 enable_mcoverlay="no"

```

```

199
200 if [ "${ENABLEMCOVERLAYFS}" = "yes" ]; then
201     if [ "${rhel_release}" = "" ]; then
202         if [ ${linux_version_code} -ge 262144 -a ${linux_version_code} -lt 262400 ]; then
203             enable_mcoverlay="yes"
204         fi
205         if [ ${linux_version_code} -ge 263680 -a ${linux_version_code} -lt 263936 ]; then
206             enable_mcoverlay="yes"
207         fi
208     else
209         if [ ${linux_version_code} -eq 199168 -a ${rhel_release} -ge 327 -a ${rhel_release} -le 327 ]; then
210             enable_mcoverlay="yes"
211         fi
212         if [ ${linux_version_code} -ge 262144 -a ${linux_version_code} -lt 262400 ]; then
213             enable_mcoverlay="yes"
214         fi
215     fi
216 fi
217
218 # Figure out CPUs if not requested by user
219 if [ "$cpus" = "" ]; then
220     # Get the number of CPUs on NUMA node 0
221     nr_cpus='lscpu --parse | awk -F", " '{if ($4 == 0) print $4}' | wc -l'
222
223     # Use the second half of the cores
224     let nr_cpus="$nr_cpus / 2"
225     cpus='lscpu --parse | awk -F", " '{if ($4 == 0) print $1}' | tail -n $nr_cpus |\'
226 ' xargs echo -n | sed 's/ /,/g'
227     if [ "$cpus" = "" ]; then
228         echo "error: no available CPUs on NUMA node 0?" >&2
229     fi
230     exit 1
231 fi
232
233 # Remove mcoverlay if loaded
234 if [ "$enable_mcoverlay" = "yes" ]; then
235     if grep mcoverlay /proc/modules &>/dev/null; then
236         if [ "'cat /proc/mounts | grep /tmp/mcos/mcos0_sys'" != "" ]; \
237     then umount -l /tmp/mcos/mcos0_sys; fi
238         if [ "'cat /proc/mounts | grep /tmp/mcos/mcos0_proc'" != "" ]; \
239     then umount -l /tmp/mcos/mcos0_proc; fi
240         if [ "'cat /proc/mounts | grep /tmp/mcos/linux_proc'" != "" ]; \
241     then umount -l /tmp/mcos/linux_proc; fi
242         if [ "'cat /proc/mounts | grep /tmp/mcos'" != "" ]; then umount -l /tmp/mcos; fi
243         if [ -e /tmp/mcos ]; then rm -rf /tmp/mcos; fi
244         if ! rmdir mcoverlay 2>/dev/null; then
245             echo "error: removing mcoverlay" >&2
246             exit 1
247         fi
248     fi
249 fi
250
251 # Stop irqbalance
252 if [ "${irqbalance_used}" = "yes" ]; then
253     systemctl stop irqbalance_mck.service 2>/dev/null
254     if ! systemctl stop irqbalance.service 2>/dev/null ; then
255         echo "error: stopping irqbalance" >&2
256         exit 1
257     fi;
258
259     if ! etcdir=/home/takagi/project/os/install/etc perl -e \
260 'use File::Copy qw(copy); $etcdir=$ENV{'etcdir'}; '\
261 '@files = grep { -f } glob "/proc/irq/*/smp_affinity"; foreach $file (@files) { '\
262 '$rel = substr($file, 1); $dir=substr($rel, 0, length($rel)-length("/smp_affinity")); '\
263 'if(0) { print "cp $file $etcdir/$rel\n"; } if(system("mkdir -p $etcdir/$dir")){ exit 1; } '\
264 'if(!copy($file, "$etcdir/$rel")){ exit 1; } }'; then
265         echo "error: saving /proc/irq/*/smp_affinity" >&2
266         error_exit "mcos_sys_mounted"

```

```

267         fi;
268
269 # Prevent /proc/irq/*/smp_affinity from getting zero after offlining
270 # McKernel CPUs by using the following algorithm.
271 # if (smp_affinity & mck_cores) {
272 #     smp_affinity = (mck_cores ^ -1);
273 # }
274     ncpus='lscpu | grep -E '^CPU\(s\):' | awk '{print $2}'
275     smp_affinity_mask='echo $cpus | ncpus=$ncpus perl -e \
276 'while(<>){@tokens = split /,/;foreach $token (@tokens) \
277 '{@nums = split /-/, $token; for($num = $nums[0]; $num <= $nums[$#nums]; $num++) {\
278 '$ndx=int($num/32); $mask[$ndx] |= (1<<($num % 32))}}'\
279 '$nint32s = int(($ENV{'ncpus'}+31)/32); for($j = $nint32s - 1; $j >= 0; $j--) {\
280 '$ if($j != $nint32s - 1){print ", ";}'\
281 '$ $nblks = ($j != $nint32s - 1) ? 8 : ($ENV{'ncpus'} % 32 != 0) ? '\
282 '$ int(($ENV{'ncpus'} + 3) % 32) / 4) : 8;'\
283 '$ for($i = $nblks - 1; $i >= 0; $i--){ printf("%01x", ($mask[$j] >> ($i*4) & 0xf));}'\
284 #     echo cpus=$cpus ncpus=$ncpus smp_affinity_mask=$smp_affinity_mask
285
286     if ! ncpus=$ncpus smp_affinity_mask=$smp_affinity_mask perl -e \
287 '@dirs = grep { -d } glob "/proc/irq/*"; foreach $dir (@dirs) {\
288 '$ hit = 0; $affinity_str = 'cat $dir/smp_affinity'; chomp $affinity_str;\
289 '$ @int32strs = split /,/, $affinity_str; @int32strs_mask=split /,/, $ENV{'smp_affinity_mask'};\
290 '$ for($i=0; $i <= $#int32strs_mask; $i++) {\
291 '$ $int32strs_inv[$i] = sprintf("%08x", hex($int32strs_mask[$i]^0xffffffff));'\
292 '$ if($i == 0) { $len = int(($ENV{'ncpus'}%32)+3)/4; if($len != 0) {\
293 '$ $int32strs_inv[$i] = substr($int32strs_inv[$i], -$len, $len); } } '\
294 '$ $inv = join(" ", @int32strs_inv); $nint32s = int(($ENV{'ncpus'}+31)/32);\
295 '$ for($j = $nint32s - 1; $j >= 0; $j--) {\
296 '$ if(hex($int32strs[$nint32s - 1 - $j]) & hex($int32strs_mask[$nint32s - 1 - $j])) {\
297 '$ $hit = 1; } } if($hit == 1) {\
298 '$ $cmd = "echo $inv > $dir/smp_affinity 2>/dev/null"; system $cmd;}' ; then
299         echo "error: modifying /proc/irq/*/smp_affinity" >&2
300         error_exit "mcos_sys_mounted"
301     fi
302
303 fi
304
305 # Load IHK if not loaded
306 if ! grep -E 'ihk\s' /proc/modules &>/dev/null; then
307     if ! taskset -c 0 insmod ${KMODDIR}/ihk.ko 2>/dev/null; then
308         echo "error: loading ihk" >&2
309         error_exit "irqbalance_stopped"
310     fi
311 fi
312
313 # Increase swappiness so that we have better chance to allocate memory for IHK
314 echo 100 > /proc/sys/vm/swappiness
315
316 # Drop Linux caches to free memory
317 sync && echo 3 > /proc/sys/vm/drop_caches
318
319 # Merge free memory areas into large, physically contiguous ones
320 echo 1 > /proc/sys/vm/compact_memory 2>/dev/null
321
322 sync
323
324 # Load IHK-SMP if not loaded and reserve CPUs and memory
325 if ! grep ihk_smp_x86 /proc/modules &>/dev/null; then
326     if [ "$ihk_irq" == "" ]; then
327         for i in `seq 64 255`; do
328             if [ ! -d /proc/irq/$i ] && \
329 [ "`cat /proc/interrupts | grep ":" | awk '{print $1}' | grep -o '[0-9]*' | grep -e '^$i$'`" \
330 == "" ]; then
331                 ihk_irq=$i
332                 break
333             fi
334         done

```



```

335         if [ "$ihk_irq" == "" ]; then
336             echo "error: no IRQ available" >&2
337             error_exit "ihk_loaded"
338         fi
339     fi
340     if ! taskset -c 0 insmod ${KMODDIR}/ihk-smp-x86.ko ihk_start_irq=$ihk_irq\
341     ihk_ikc_irq_core=$ihk_ikc_irq_core 2>/dev/null; then
342         echo "error: loading ihk-smp-x86" >&2
343         error_exit "ihk_loaded"
344     fi
345
346     # Offline-reonline RAM (special case for OFP SNC-4 mode)
347     if [ "$(hostname | grep "c[0-9][0-9][0-9].ofp" != "") ] && [ "$(cat /sys/devices/system/node/
348         for i in 0 1 2 3; do
349             find /sys/devices/system/node/node$i/memory*/* -name "online" |\
350     while read f; do
351                 echo 0 > $f 2>&1 > /dev/null;
352             done
353             find /sys/devices/system/node/node$i/memory*/* -name "online" |\
354     while read f; do
355                 echo 1 > $f 2>&1 > /dev/null;
356             done
357         done
358         for i in 4 5 6 7; do
359             find /sys/devices/system/node/node$i/memory*/* -name "online" |\
360     while read f; do
361                 echo 0 > $f 2>&1 > /dev/null;
362             done
363             find /sys/devices/system/node/node$i/memory*/* -name "online" |\
364     while read f; do
365                 echo 1 > $f 2>&1 > /dev/null;
366             done
367         done
368     fi
369
370     if ! ${SBINDIR}/ihkconfig 0 reserve mem ${mem}; then
371         echo "error: reserving memory" >&2
372         error_exit "ihk_smp_loaded"
373     fi
374     if ! ${SBINDIR}/ihkconfig 0 reserve cpu ${cpus}; then
375         echo "error: reserving CPUs" >&2;
376         error_exit "mem_reserved"
377     fi
378 fi
379
380 # Load mcctrl if not loaded
381 if ! grep mcctrl /proc/modules &>/dev/null; then
382     if ! taskset -c 0 insmod ${KMODDIR}/mcctrl.ko 2>/dev/null; then
383         echo "error: inserting mcctrl.ko" >&2
384         error_exit "cpus_reserved"
385     fi
386 fi
387
388 # Destroy all LWK instances
389 if ls /dev/mcos* 1>/dev/null 2>&1; then
390     for i in /dev/mcos*; do
391         ind='echo $i|cut -c10-';
392         # Retry when conflicting with ihkmond
393         nretry=0
394         until ${SBINDIR}/ihkconfig 0 destroy $ind || [ $nretry -lt 4 ]; do
395             sleep 0.25
396             nretry=$(( $nretry + 1 ))
397         done
398         if [ $nretry -eq 4 ]; then
399             echo "error: destroying LWK instance $ind failed" >&2
400             error_exit "mcctrl_loaded"
401         fi
402     done

```

```

403 fi
404
405 # Create OS instance
406 if ! ${SBINDIR}/ihkconfig 0 create; then
407     echo "error: creating OS instance" >&2
408     error_exit "mcctrl_loaded"
409 fi
410
411 # Assign CPUs
412 if ! ${SBINDIR}/ihkosctl 0 assign cpu ${cpus}; then
413     echo "error: assign CPUs" >&2
414     error_exit "os_created"
415 fi
416
417 if [ "$ikc_map" != "" ]; then
418     # Specify IKC map
419     if ! ${SBINDIR}/ihkosctl 0 set ikc_map ${ikc_map}; then
420         echo "error: assign CPUs" >&2
421         error_exit "os_created"
422     fi
423 fi
424
425 # Assign memory
426 if ! ${SBINDIR}/ihkosctl 0 assign mem ${mem}; then
427     echo "error: assign memory" >&2
428     error_exit "os_created"
429 fi
430
431 # Load kernel image
432 if ! ${SBINDIR}/ihkosctl 0 load ${KERNDIR}/mckernel.img; then
433     echo "error: loading kernel image: ${KERNDIR}/mckernel.img" >&2
434     error_exit "os_created"
435 fi
436
437 # Set kernel arguments
438 if ! ${SBINDIR}/ihkosctl 0 kargs "hidos $turbo dump_level=${DUMP_LEVEL}"; then
439     echo "error: setting kernel arguments" >&2
440     error_exit "os_created"
441 fi
442
443 # Boot OS instance
444 if ! ${SBINDIR}/ihkosctl 0 boot; then
445     echo "error: booting" >&2
446     error_exit "os_created"
447 fi
448
449 # Set device file ownership
450 if ! chown ${chown_option} /dev/mcd* /dev/mcos*; then
451     echo "warning: failed to chown device files" >&2
452 fi
453
454 # Overlay /proc, /sys with McKernel specific contents
455 if [ "$enable_mcoverlay" == "yes" ]; then
456     if [ ! -e /tmp/mcos ]; then
457         mkdir -p /tmp/mcos;
458     fi
459     if ! mount -t tmpfs tmpfs /tmp/mcos; then
460         echo "error: mount /tmp/mcos" >&2
461         error_exit "tmp_mcos_created"
462     fi
463     if [ ! -e /tmp/mcos/linux_proc ]; then
464         mkdir -p /tmp/mcos/linux_proc;
465     fi
466     if ! mount --bind /proc /tmp/mcos/linux_proc; then
467         echo "error: mount /tmp/mcos/linux_proc" >&2
468         error_exit "tmp_mcos_mounted"
469     fi
470     if ! taskset -c 0 insmod ${KMODDIR}/mcoverlay.ko 2>/dev/null; then

```

```

471             echo "error: inserting mcoverlay.ko" >&2
472             error_exit "linux_proc_bind_mounted"
473         fi
474     while [ ! -e /proc/mcos0 ]
475     do
476         sleep 0.1
477     done
478     if [ ! -e /tmp/mcos/mcos0_proc ]; then
479         mkdir -p /tmp/mcos/mcos0_proc;
480     fi
481     if [ ! -e /tmp/mcos/mcos0_proc_upper ]; then
482         mkdir -p /tmp/mcos/mcos0_proc_upper;
483     fi
484     if [ ! -e /tmp/mcos/mcos0_proc_work ]; then
485         mkdir -p /tmp/mcos/mcos0_proc_work;
486     fi
487     if ! mount -t mcoverlay mcoverlay -o\
488     lowerdir=/proc/mcos0:/proc , upperdir=/tmp/mcos/mcos0_proc_upper ,\
489     workdir=/tmp/mcos/mcos0_proc_work ,nocopyupw ,nofscheck /tmp/mcos/mcos0_proc; then
490         echo "error: mounting /tmp/mcos/mcos0_proc" >&2
491         error_exit "mcoverlayfs_loaded"
492     fi
493     # TODO: How de we revert this in case of failure??
494     mount --make-rprivate /proc
495
496     while [ ! -e /sys/devices/virtual/mcos/mcos0/sys/setup_complete ]
497     do
498         sleep 0.1
499     done
500     if [ ! -e /tmp/mcos/mcos0_sys ]; then
501         mkdir -p /tmp/mcos/mcos0_sys;
502     fi
503     if [ ! -e /tmp/mcos/mcos0_sys_upper ]; then
504         mkdir -p /tmp/mcos/mcos0_sys_upper;
505     fi
506     if [ ! -e /tmp/mcos/mcos0_sys_work ]; then
507         mkdir -p /tmp/mcos/mcos0_sys_work;
508     fi
509     if ! mount -t mcoverlay mcoverlay -o\
510     lowerdir=/sys/devices/virtual/mcos/mcos0/sys:/sys , upperdir=/tmp/mcos/mcos0_sys_upper ,\
511     workdir=/tmp/mcos/mcos0_sys_work ,nocopyupw ,nofscheck /tmp/mcos/mcos0_sys; then
512         echo "error: mount /tmp/mcos/mcos0_sys" >&2
513         error_exit "mcos_proc_mounted"
514     fi
515     # TODO: How de we revert this in case of failure??
516     mount --make-rprivate /sys
517
518     touch /tmp/mcos/mcos0_proc/mckernel
519
520     rm -rf /tmp/mcos/mcos0_sys/setup_complete
521
522     # Hide NUMA related files which are outside the LWK partition
523     for cpuid in \
524     `find /sys/devices/system/cpu/* -maxdepth 0 -name "cpu[0123456789]*" -printf "%f " `; do
525         if [ ! -e "/sys/devices/virtual/mcos/mcos0/sys/devices/system/cpu/$cpuid" ]; then
526             rm -rf /tmp/mcos/mcos0_sys/devices/system/cpu/$cpuid
527             rm -rf /tmp/mcos/mcos0_sys/bus/cpu/devices/$cpuid
528             rm -rf /tmp/mcos/mcos0_sys/bus/cpu/drivers/processor/$cpuid
529         else
530             for nodeid in \
531             `find /sys/devices/system/cpu/$cpuid/* -maxdepth 0 -name "node[0123456789]*" -printf "%f " `; do
532                 if [ ! -e \
533                 "/sys/devices/virtual/mcos/mcos0/sys/devices/system/cpu/$cpuid/$nodeid" ]; then
534                     rm -f \
535                     /tmp/mcos/mcos0_sys/devices/system/cpu/$cpuid/$nodeid
536                 fi
537             done
538         fi

```

```

539         done
540         for nodeid in \
541 'find /sys/devices/system/node/* -maxdepth 0 -name "node[0123456789]*" -printf "%f "'; do
542             if [ ! -e "/sys/devices/virtual/mcos/mcos0/sys/devices/system/node/$nodeid" ]; \
543 then
544                 rm -rf /tmp/mcos/mcos0_sys/devices/system/node/$nodeid/*
545                 rm -rf /tmp/mcos/mcos0_sys/bus/node/devices/$nodeid
546             else
547                 # Delete non-existent symlinks
548                 for cpuid in \
549 'find /sys/devices/system/node/$nodeid/* -maxdepth 0 -name "cpu[0123456789]*" -printf "%f "'; do
550                     if [ ! -e \
551 "/sys/devices/virtual/mcos/mcos0/sys/devices/system/node/$nodeid/$cpuid" ]; then
552                         rm -f \
553 /tmp/mcos/mcos0_sys/devices/system/node/$nodeid/$cpuid
554                     fi
555                 done
556             fi
557             rm -f /tmp/mcos/mcos0_sys/devices/system/node/$nodeid/memory*
558         fi
559     done
560     rm -f /tmp/mcos/mcos0_sys/devices/system/node/has_*
561     for cpuid in \
562 'find /sys/bus/cpu/devices/* -maxdepth 0 -name "cpu[0123456789]*" -printf "%f "'; do
563         if [ ! -e "/sys/devices/virtual/mcos/mcos0/sys/bus/cpu/devices/$cpuid" ]; then
564             rm -rf /tmp/mcos/mcos0_sys/bus/cpu/devices/$cpuid
565         fi
566     done
567 fi
568
569 # Start irqbalance with CPUs and IRQ for McKernel banned
570 if [ "${irqbalance_used}" == "yes" ]; then
571     banirq='cat /proc/interrupts | \
572 perl -e 'while(<>) { if (/^\s*(\d+).*IHK\--SMP\s*$/) {print $1;}}'
573     sed "s/%mask%/$smp_affinity_mask/g" $ETC_DIR/irqbalance_mck.in | \
574 sed "s/%banirq%/$banirq/g" > /tmp/irqbalance_mck
575     systemctl disable irqbalance_mck.service >/dev/null 2>/dev/null
576     if ! systemctl link $ETC_DIR/irqbalance_mck.service >/dev/null 2>/dev/null; then
577         echo "error: linking irqbalance_mck" >&2
578         error_exit "mcos_sys_mounted"
579     fi
580 fi
581
582 if ! systemctl start irqbalance_mck.service 2>/dev/null ; then
583     echo "error: starting irqbalance_mck" >&2
584     error_exit "mcos_sys_mounted"
585 fi
586 # echo cpus=$cpus ncpus=$ncpus banirq=$banirq
587 fi

```

手順は以下の通り。

1. `ihkmond` を起動する。`ihkmond` は任意のタイミングで起動してよい。これは、`ihkmond` は OS インスタンスの作成を検知して動作を開始するためである。(83 行目) 1
2
3
2. Linux のカーネルバージョンが、`mcoverlayfs` が動作するものであるかを確認する。(200–216 行目) 4
5
3. `irqbalance` を停止する。(251–257 行目) 6
4. `/proc/irq/*/affinity` の設定を保存した上で McKernel CPU を担当から外す。担当 CPU が無くなる場合は、全ての Linux CPU を指定する。(269–303 行目) 7
8
5. `ihk.ko` を `insmod` する。(307 行目) 9

- 1 6. Linux によるメモリフラグメンテーションを緩和するために以下を実施する。(313–320
2 行目)
 - 3 (a) アクティブでないプロセスを積極的にスワップアウトするように設定する
 - 4 (b) クリーンなページキャッシュを無効化し、また `dentries` や `inode` の slab オブジェ
5 クトのうち可能なものを破棄する
 - 6 (c) 連続する空き領域を結合してより大きな空き領域にまとめる
- 7 7. `ihk-smp-x86.ko` を `insmod` する。(340 行目) `ihk-smp-x86.ko` は関数を `ihk.ko` に登
8 録する。このため、`ihk-smp-x86.ko` は `ihk.ko` を `insmod` した後に `insmod` する必要が
9 ある。
- 10 8. メモリを予約する。(370 行目)
- 11 9. CPU を予約する。(374 行目)
- 12 10. McKernel のカーネルモジュール `mcctrl.ko` を `insmod` する。(382 行目) `mcctrl.ko`
13 は McKernel ブート時に呼ばれる関数を `ihk.ko` に登録する。このため、`mcctrl.ko` の
14 `insmod` は `ihk.ko` の `insmod` の後に、またブートの前に行う必要がある。
- 15 11. OS インスタンスを作成する。(406 行目)
- 16 12. OS インスタンスに CPU を割り当てる。(412 行目)
- 17 13. McKernel CPU の IKC メッセージ送信先の Linux CPU を設定する。(419 行目)
- 18 14. OS インスタンスにメモリを割り当てる。(426 行目)
- 19 15. カーネルイメージをロードする。(432 行目)
- 20 16. カーネル引数をカーネルに渡す。(438 行目)
- 21 17. カーネルをブートする。(444 行目)
- 22 18. `/proc`, `/sys` ファイルの準備をする。また、その中で `mcoverlayfs.ko` を `insmod` す
23 る。`mcoverlayfs.ko` は他モジュールとの依存関係を持たない。(454 行目から 567 行
24 目) なお、関数インターフェイスでの対応関数は `ihk_os_create_pseudofs()` である。
- 25 19. `irqbalance` を、Linux CPU のみを対象とする設定で開始する。(569–587 行目)

26 3.3 シャットダウン手順

27 `mcstop+release.sh` を用いてシャットダウン手順を説明する。
28 スクリプトは以下の通り。

```

1 #!/bin/bash
2
3 # IHK SMP-x86 example McKernel unload script.
4 # author: Balazs Gerofi <bgerofi@riken.jp>
5 # Copyright (C) 2015 RIKEN AICS
6 #
7 # This is an example script for destroying McKernel and releasing IHK resources
8 # Note that the script does no output anything unless an error occurs.
9
10 prefix="/home/takagi/project/os/install"
```

```

11 BINDIR="/home/takagi/project/os/install/bin"
12 SBINDIR="/home/takagi/project/os/install/sbin"
13 ETCDIR="/home/takagi/project/os/install/etc"
14 KMODDIR="/home/takagi/project/os/install/kmod"
15 KERNDIR="/home/takagi/project/os/install/smp-x86/kernel"
16
17 mem=""
18 cpus=""
19 irqbalance_used=""
20
21 # No SMP module? Exit.
22 if ! grep ihk-smp-x86 /proc/modules &&>/dev/null; then exit 0; fi
23
24 if [ "'systemctl status irqbalance-mck.service 2> /dev/null |grep -E 'Active: active'" \
25 != "" ]; then
26     irqbalance_used="yes"
27     if ! systemctl stop irqbalance-mck.service 2>/dev/null; then
28         echo "warning: failed to stop irqbalance-mck" >&2
29     fi
30     if ! systemctl disable irqbalance-mck.service >/dev/null 2>/dev/null; then
31         echo "warning: failed to disable irqbalance-mck" >&2
32     fi
33 fi
34
35 # Destroy all LWK instances
36 if ls /dev/mcos* 1>/dev/null 2>&1; then
37     for i in /dev/mcos*; do
38         ind='echo $i|cut -c10-';
39         # Retry when conflicting with ihkmond
40         nretry=0
41         until ${SBINDIR}/ihkconfig 0 destroy $ind || [ $nretry -lt 4 ]; do
42             sleep 0.25
43             nretry=$(( $nretry + 1 ))
44         done
45         if [ $nretry -eq 4 ]; then
46             echo "error: destroying LWK instance $ind failed" >&2
47             exit 1
48         fi
49     done
50 fi
51
52 # Query IHK-SMP resources and release them
53 if ! ${SBINDIR}/ihkconfig 0 query cpu > /dev/null; then
54     echo "error: querying cpus" >&2
55     exit 1
56 fi
57
58 cpus='${SBINDIR}/ihkconfig 0 query cpu'
59 if [ "${cpus}" != "" ]; then
60     if ! ${SBINDIR}/ihkconfig 0 release cpu $cpus > /dev/null; then
61         echo "error: releasing CPUs" >&2
62         exit 1
63     fi
64 fi
65
66 if ! ${SBINDIR}/ihkconfig 0 query mem > /dev/null; then
67     echo "error: querying memory" >&2
68     exit 1
69 fi
70
71 mem='${SBINDIR}/ihkconfig 0 query mem'
72 if [ "${mem}" != "" ]; then
73     if ! ${SBINDIR}/ihkconfig 0 release mem $mem > /dev/null; then
74         echo "error: releasing memory" >&2
75         exit 1
76     fi
77 fi
78

```

```

79 # Remove delegator if loaded
80 if grep mcctrl /proc/modules &>/dev/null; then
81     if ! rmmmod mcctrl 2>/dev/null; then
82         echo "error: removing mcctrl" >&2
83         exit 1
84     fi
85 fi
86
87 # Remove mcoverlay if loaded
88 if grep mcoverlay /proc/modules &>/dev/null; then
89     if [ "`cat /proc/mounts | grep /tmp/mcos/mcos0_sys`" != "" ]; \
90 then umount -l /tmp/mcos/mcos0_sys; fi
91     if [ "`cat /proc/mounts | grep /tmp/mcos/mcos0_proc`" != "" ]; \
92 then umount -l /tmp/mcos/mcos0_proc; fi
93     if [ "`cat /proc/mounts | grep /tmp/mcos/linux_proc`" != "" ]; \
94 then umount -l /tmp/mcos/linux_proc; fi
95     if [ "`cat /proc/mounts | grep /tmp/mcos`" != "" ]; then umount -l /tmp/mcos; fi
96     if [ -e /tmp/mcos ]; then rm -rf /tmp/mcos; fi
97     if ! rmmmod mcoverlay 2>/dev/null; then
98         echo "warning: failed to remove mcoverlay" >&2
99     fi
100 fi
101
102 # Remove SMP module
103 if grep ihk_smp_x86 /proc/modules &>/dev/null; then
104     if ! rmmmod ihk_smp_x86 2>/dev/null; then
105         echo "error: removing ihk_smp_x86" >&2
106         exit 1
107     fi
108 fi
109
110 # Remove core module
111 if grep -E 'ihk\s' /proc/modules &>/dev/null; then
112     if ! rmmmod ihk 2>/dev/null; then
113         echo "error: removing ihk" >&2
114         exit 1
115     fi
116 fi
117
118 # Stop ihkmond
119 pid=`pidof ihkmond`
120 if [ "$pid" != "" ]; then
121     sudo kill -9 ${pid} > /dev/null 2> /dev/null
122 fi
123
124 # Start irqbalance with the original settings
125 if [ "`irqbalance_used`" != "" ]; then
126     if ! etcdir=/home/takagi/project/os/install/etc perl -e \
127 `setcdire=$ENV{`etcdire`}; @files = grep { -f } glob "$etcdire/proc/irq/*/smp_affinity";` \
128 `foreach $file (@files) { $dest = substr($file, length($etcdire));` \
129 `if(0) {print "cp $file $dest\n";} system("cp $file $dest 2>/dev/null"); }`; then
130         echo "warning: failed to restore /proc/irq/*/smp_affinity" >&2
131     fi
132     if ! systemctl start irqbalance.service; then
133         echo "warning: failed to start irqbalance" >&2;
134     fi
135 fi
136
137 # Set back default swappiness
138 echo 60 > /proc/sys/vm/swappiness

```

1 手順は以下の通り。

- 2 1. ブート時に Linux CPU のみを対象とする設定で開始された irqbalance を停止する。
- 3 (24-33 行目)
- 4 2. 全ての OS インスタンスを破壊する。OS インスタンスに割り当てられていた資源は IHK

- が LWK のために予約した状態に移行する。(35-50 行目) 1
3. IHK が LWK のために予約していた資源を開放する。(52-77 行目) 2
4. mcctrl.ko を rmmod する。(81 行目) 3
5. /proc, /sys ファイルの準備をする。また、その中で mcoverlayfs.ko を rmmod する。
(87-100 行目) なお、関数インターフェイスでの対応関数は `ihk_os_destroy_pseudofs()`
である。 4
5
6
6. `ihk-smp-x86.ko` を rmmod する。(104 行目) 7
7. `ihk.ko` を rmmod する。(112 行目) 8
8. `ihkmond` を停止する。(121 行目) 9
9. `/proc/irq/*/affinity` の設定をブート時に保存しておいたものに戻し、ブート前の設
定で `irqbalance` を開始する。(124-135 行目) 10
11
10. Linux カーネルのスワップアウト積極度の設定をデフォルトの値に戻す。(138 行目) 12

1 Bibliography

- 2 [1] H. Fujita, M. Matsuda, T. Maeda, S. Miura, and Y. Ishikawa. P-Bus: Programming
3 Interface Layer for Safe OS Kernel Extensions. In *Pacific Rim International Symposium*
4 *on Dependable Computing (PRDC)*, pages 235–236, 2010.
- 5 [2] T. Shimosawa, B. Gerofi, M. Takagi, G. Nakamura, T. Shirasawa, Y. Saeki, M. Shimizu,
6 A. Hori, and Y. Ishikawa. Interface for Heterogeneous Kernels: A Framework to Enable
7 Hybrid OS Designs targeting High Performance Computing on Manycore Architectures.
8 *In Proc. of IEEE International Conference on High Performance Computing (HiPC)*,
9 2014.