McKernel Specifications
Version 1.7.1-0.7

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

Monday 18th January, 2021

Contents

List of Figures

List of Tables

Chapter 1

インターフェイス

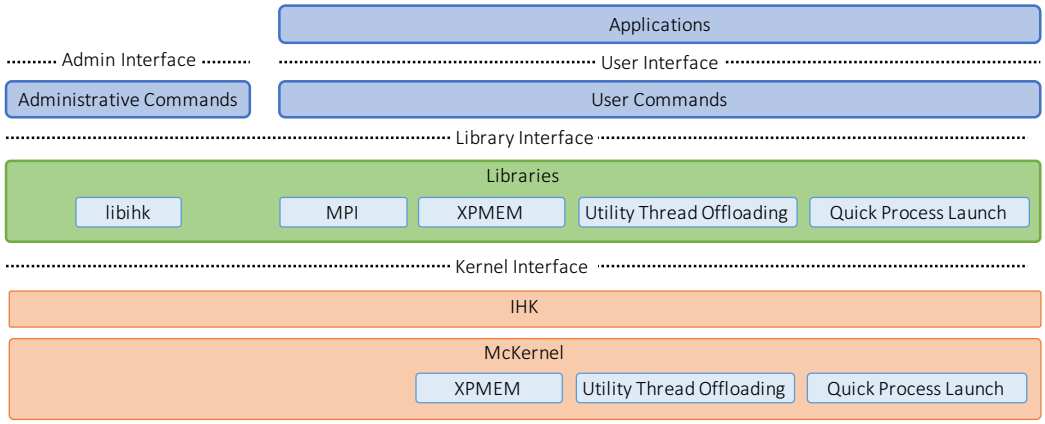


Figure 1.1: McKernel software stack

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 Inteface 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/libqlmpi.so	ライブラリ	高速プロセス起動ライブラリ
<install>/lib/libqlfort.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 プロセス起動コマンド

書式

```
   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>...]
```

オプション

-c <cpu_id>	mcexec を実行する CPU の番号を<cpu_id>に設定する。指定がない場合は 0 が用いられる。
-n <nr_partitions>	1 計算ノードの CPU 群を<nr_partitions>の区画に分割し、第 i 番目に起動された mcexec 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<nr_partitions>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。
-t <nr_threads>	mcexec のスレッド数を<nr_threads>に設定する。このオプションが指定されない場合は、OMP_NUM_THREADS 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。mcexec 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〈McKernel のスレッド数 $+\alpha$〉に設定する必要がある。
-M (--mpol-threshold=)<min>	<min>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が適用されるようにする。<min>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が適用される。
-h (--extend-heap-by=)<step>	ヒープの拡大時にヒープサイズを少なくとも<step>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。<step>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。
-s (--stack-premap=) <premap_size>,<max>	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 ulimit コマンドまたは setrlimit() システムコールで設定された値が用いられる。
--mpol-no-heap	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に従わない。
--mpol-no-stack	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に従わない。
--mpol-no-bss	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に従わない。
--mpol-shm-premap	/dev/shm を用いた共有メモリをプリマップする。
-m <numa_node>	メモリを<numa_node>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。
--disable-sched-yield	sched_yield() 関数を何も行わない関数に置き換える。
-O	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 clone(), fork(), vfork() などで生成しようとすると、当該システムコールが EINVAL エラーを返す。
<os_index>	プロセス起動先 OS インスタンスを<os_index>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。

説明

<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 上に起動する。

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、mcexec 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。 • 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 mcexec, ihkosctl, ihkconfig である場合
MCEXEC_ALT_ROOT=<path>	ld-linux.so などのローダを探す際に、<path>と実行可能ファイルの.interp 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 ulimit -s コマンドまたは setrlimit() 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 mcexec の--stack-premap 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。

1. 以下のいずれかの方法でダンプファイルを作成する。
   (a) 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。
   (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、コマンド vmcore2mckdump を用いて McKernel 形式に変換する。
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。

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 <kimg>] [-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 <kimg>	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。

説明

<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、<kimg>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。

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。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。

```
(eclair) info threads
```

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.

1.2.2 ダンプ形式変換コマンド

書式

```
vmcore2mckdump <vmcore> <file_name>
```

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

説明

<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。

1.3 高速プロセス起動ライブラリインターフェイス

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 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。

- 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
  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。

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 */
```

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

以下、コマンドや関数のインターフェイスを説明する。

1.3.1 MPI プロセス開始再開コマンド

書式

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

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

説明

<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 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。

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、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 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要がある。

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指定はサポートしない。

戻り値

戻り値	説明
0	正常終了
0 以外	異常終了

エラー時出力

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

エラー時出力

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

1.3.3 計算の再開・終了関数 (C 言語)

書式

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

引数

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

説明

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、MPI プログラム内で、一回の計算の完了後に呼び出す。

戻り値

戻り値	説明
QL_CONTINUE	次の計算の開始が指示された
QL_EXIT	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 ql_mpiexec_start コマンドで起動されていない

1.3.4 計算の再開・終了関数 (Fortran)

書式

```
subroutine QL_CLIENT(ierr)
```

引数

引数	型	説明
ierr	INT	戻り値

説明

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。

戻り値

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 ql_mpiexec_start コマンドで起動されていない

1.4 高速プロセス起動カーネルインターフェイス

1.4.1 swapout システムコール

書式

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

引数

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

説明

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。

処理ステップは以下の通り。

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。

2　スワップアウト処理を行う。

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。

4 mcexec へ制御を移し、スワップアウト完了の同期と、ql_mpiexec_{start,finalize} 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 スワップイン処理を行う。さらに呼び出し元に戻る。

6 mcexec へ制御を移し、ql_mpiexec_{start,finalize} による指示を待った後、本システムコールに制御を戻す。さらに呼び出し元に戻る。

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

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

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

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

McKernel は、スレッドを Linux の CPU にマイグレートする機能（Utility Thread Offloading, 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ラリによって用いられる。

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。

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

書式

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

説明

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づき、呼び出し元スレッドを Linux CPU にマイグレートさせる。

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 は以下のように定義される。

```
#define UTI_MAX_NUMA_DOMAINS (1024)

typedef struct uti_attr {
    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。

1.6.2 スレッド生成先 OS 指定システムコール

書式

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

説明

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、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

戻り値

0	正常終了
-1	エラー

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。

1.6.3 カーネル種別取得システムコール

書式

```
int get_system()
```

説明

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ジョンにて is_mckernel() 等に変更される予定である。

戻り値

0	OS が McKernel
-1	エラー（OS が Linux）

エラー時の errno の値

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

1.7.2 Expose Memory Block

書式

```
xpmem_segid_t xpmem_make(
        void *vaddr,
        size_t size,
        int permit_type,
        void *permit_value)
```

説明

`xpmem_make()` shares a memory block specified by `vaddr` and `size` by invoking the XPMEM driver. `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.

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

戻り値

$\neq -1$	64-bit segment ID (`xpmem_segid_t`)
-1	Failure

1.7.3 Un-Expose Memory Block

書式

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

説明

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

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

戻り値

0	Success
-1	Failure

1.7.4 Get Access Permit

書式

```
xpmem_apid_t xpmem_get(
        xpmem_segid_t segid,
        int flags,
        int permit_type,
        void *permit_value)
```

説明

`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.

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()`.

戻り値

$\neq -1$	64-bit access permit ID (`xpmem_apid_t`)
-1	Failure

1.7.5 Release Access Permit

書式

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

説明

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.

戻り値

0	Success
-1	Failure

1.7.6 Attach to Memory Block

書式

```
static int xpmem_attach(
        struct xpmem_addr addr,
        size_t size,
        void *vaddr)
```

説明

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

```
struct xpmem_addr {
    /** apid that represents memory */
    xpmem_apid_t apid;
    /** offset into apid's memory region */
    off_t offset;
};
```

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

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

戻り値

$\neq -1$	Virtual address at which the mapping was created
-1	Failure

1.7.7 Detach from Memory Block

書式

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

説明

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

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

戻り値

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
XPMEM_CMD_VERSION	バージョン番号を返す。
XPMEM_CMD_MAKE	`arg->vaddr` から始まる長さ `arg->size` のメモリ領域を共有可能にし、segment id を `arg->segid` に格納する。メモリ領域のパーミッションは `arg->permit_value` に設定される。
XPMEM_CMD_REMOVE	`arg->segid` で指定されたメモリ領域の共有を解除する。
XPMEM_CMD_GET	`arg->segid` で指定されたメモリ領域に対する `arg->permit_value` 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 `arg->apid` に格納される。
XPMEM_CMD_RELEASE	`arg->apid` 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。
XPMEM_CMD_ATTACH	`arg->apid` で指定された共有メモリ領域のうち `arg->offset` から始まる長さ `arg->size` の範囲を `arg->vaddr` から始まるアドレス範囲にマップする。
XPMEM_CMD_DETACH	`arg->vaddr` 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。

```
struct xpmem_cmd_remove {
        xpmem_segid_t segid;
};
```

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。

```
struct xpmem_cmd_get {
        xpmem_segid_t segid;
        int flags;
        int permit_type;
        __u64 permit_value;
        xpmem_apid_t apid;      /* returned on success */
};
```

xpmem_apid_t は以下のように定義される。

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

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。

```
struct xpmem_cmd_release {
        xpmem_apid_t apid;
};
```

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。

```
struct xpmem_cmd_attach {
        xpmem_apid_t apid;
        off_t offset;
        size_t size;
        __u64 vaddr;
        int fd;
        int flags;
};
```

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。

```
struct xpmem_cmd_detach {
        __u64 vaddr;
};
```

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。

```
struct xpmem_addr {
        xpmem_apid_t apid;      /* apid that represents memory */
        off_t offset;           /* offset into apid's memory */
};
```

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

Chapter 2

実装者向けインターフェイス詳細

本章の想定読者は以下の通り。

- McKernel の、アーキテクチャ移植を含む開発を行う開発者

2.1 概要

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

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

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

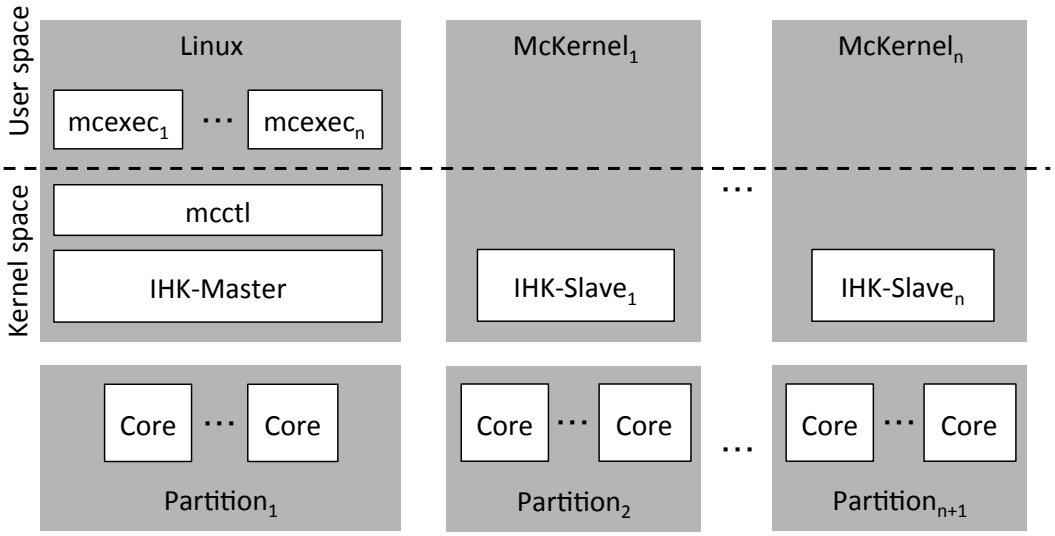


Figure 2.1: The architecture of McKernel

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 hadware 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 addtion 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 descried.

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。

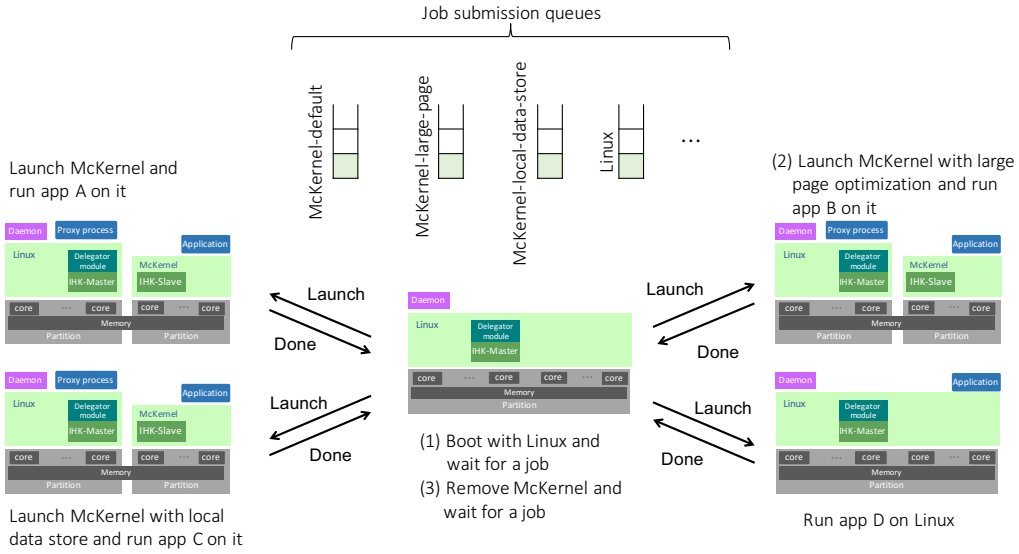


Figure 2.2: McKernel Usages

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 のみが動作する状態に戻す（図の（３））

2.2 プロセス管理

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

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

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

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

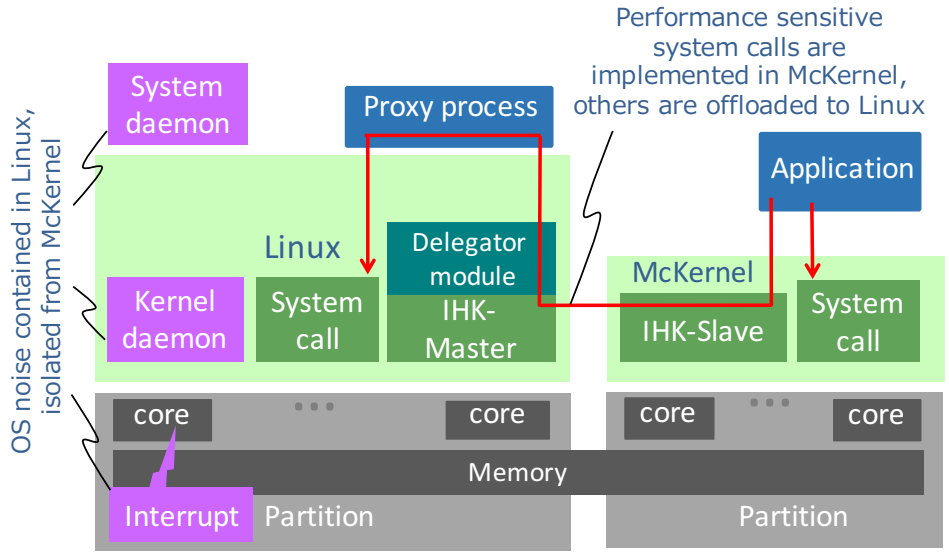


Figure 2.3: Overview of the IHK/McKernel architecture and the system call delegation mechanism.

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 singnaling, 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/mcos`*n* to communicate with McKernel.
2. It sends the ELF binary description header, the commmand 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

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、以下のケースでは Linux で実行される。

- 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 `mcexec`, `ihkosctl`, `ihkconfig` である場合

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。

2.2.2 `fork()`

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

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

2.2.3 Files and the File Descriptor Table

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

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

2.2.4 Signal Handling

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.

When the `mcexec` process receives a signal, that signal is transfered 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.

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)。

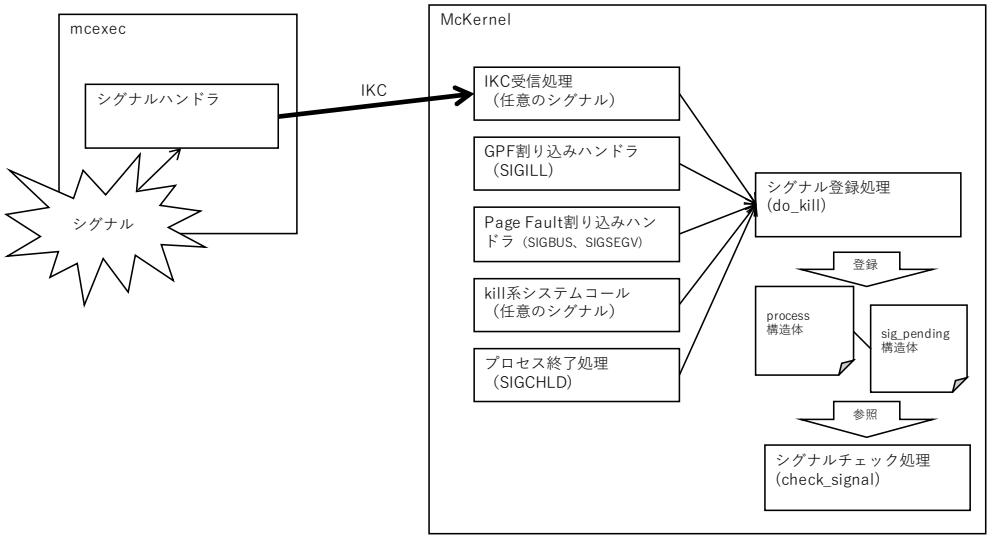


Figure 2.4: シグナル中継処理の動作

2.2.5 Process ID

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

2.2.6 Thread ID

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 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。

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。

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 のスレッドを proxy process に問い合わせる。

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 ID 問い合わせを高速化する。

`mcexec` は生成するスレッド数を以下の方法で決定する。

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。

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 に割り当てられた CPU 数を用いる。

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 <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。

2.2.7 User ID

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。

2.2.8 Process Groups

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。

2.3 システムコール

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

2.3.1 System Call Offloading

The main steps of system call offloading (also shown in Figure 2.3) are as follows. When McKernel determines that a system call needs to be offloaded it marshalls the system call number along with its arguments and sends a message to Linux via a dedicated IKC channel. The corresponding proxy process running on Linux is by default waiting for system call requests through an `ioctl()` 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 management	`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
`vdso`	System call code and data
`vvar`	Kernel variables
`hpet`	Rregister of the High Precision Event Timer
`pvti`	Virtual clock updated by virtual machine, such as Xen and KVM

2.3.4 `perf_event_open()`

`perf_event_open()` is implemented in McKernel by using the technique mentioned in Section 2.7.2.

2.4 Memory Management

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

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

2.4.1 Unified Address Space

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

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

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

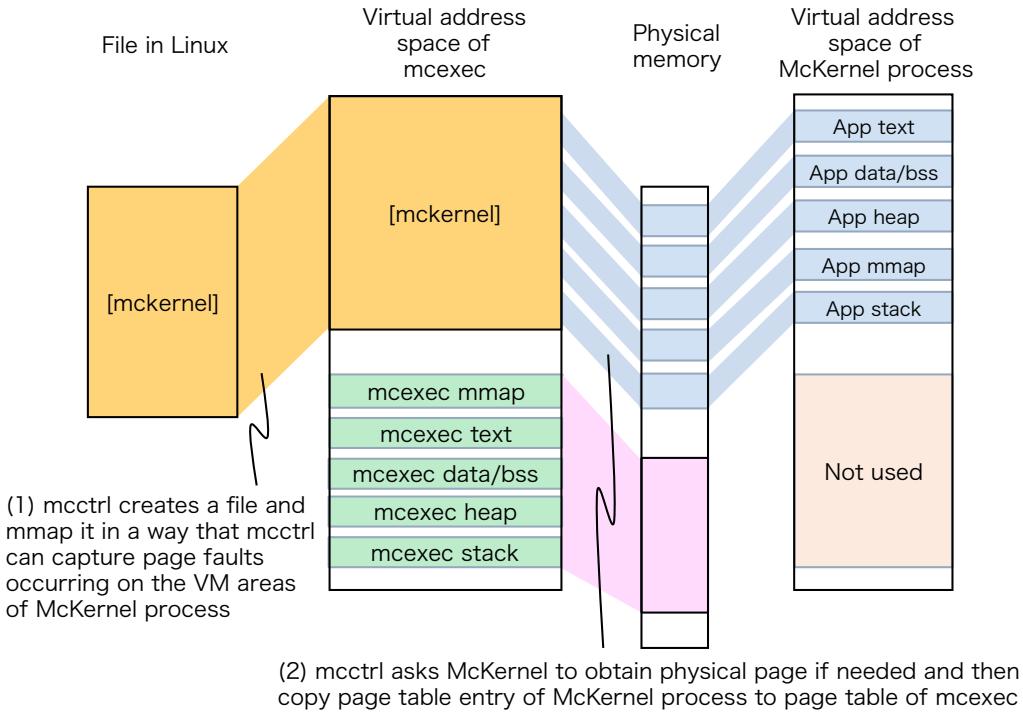


Figure 2.5: Unified Address Space

1. When `mcexec` accesses a memory area pointed by a pointer variable stored in a system call request a Linux page fault occurs.

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. In case that PTE is not found, the following sequences of issuing remote page fault are performed as follows.

   (a) The `mcctrl` module interrupts the sytem call service. It reports return code `STATUS_PAGE_FAULT` and the faulting address to McKernel.

   (b) When McKernel receives the return code `STATUS_PAGE_FAULT`, it resolves the page fault.

   (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`.

   (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.

4. `mcctrl` maps the physical memory pointed by the PTE to the virtual address where the page fault occured.

5. `mcctrl` requests resuming the execution of the `mcexec` process.

6. The `mcexec` process now can access the virtual address requested in the system call.

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

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

2.4.1.1 McKernel Process Virtual Address Mapping

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

1. The `mcexec` process has its own text, data and BSS area whose addresses are also used in the McKernel process if those execution binaries have been created in the same way.

2. If the huge stack area is allocated to `mcexec` via shell environment variable `RLIMIT_STACK`, the virtual address space for the McKernel process cannot be assigned.

The solution of those issues on Linux for x86_64 architectues is described as follows.

2.4.1.1.1 Avoiding Conflict of text, data, and BSS

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

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

2.4.1.1.2 Huge Stack Size

The virtual address space plan of the McKernel process follows Linux address plan, i.e., the user space is contiguos and starts from virtual address 0. That is, in order to keep the same address space of the McKernel process in the `mcexec`, the same address space must not be occupied by the `mcexec` process. There is one problem to do so. In Linux for x86_64 architectures, the start address of a stack area is randomly decided and its size is the lesser of $\frac{5}{6}$ total memory size and size specified by the `RLIMIT_STACK` environment variable. If the huge stack occupies the virtual meory in the `mcexec`, there is no chance to reserve the address space for the McKernel process. In order to eliminate this problem, the `RLIMIT_STACK` environmental variable for `mcexec` and the McKernel process is separeted. 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 repeaditly 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、ページフォー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。

4. 拡張された部分をプリページングする。

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 に返す必要がない、という HPC アプリの特性を用いている。

2.4.5 メモリ割り当てにおける NUMA ノード選択

2.4.5.1 ユーザメモリ割り当て

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能が追加されている。

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 1.1 参照）のオプション (`--mpol-no-{heap,stack,bss}`) によって行う。

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、プロセス起動コマンド `mcexec`（第 1.1 参照）のオプション (`--mpol-threshold=<min>`) によって行う。

2.4.5.2 カーネルメモリ割り当て

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短い NUMA ノードからメモリを取得する。

2.4.6 Virtual Dynamic Shared Object (vDSO)

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

The steps are in the followings.

1. The physical addresses of the Linux vDSO pages are compiled by looking into `System.map` when configureing McKernel. They are kept in `mcctrl`.

2. McKernel adds mappings of the Linux vDSO pages to a process when creating the process. McKernel asks `mcctrl` for their physical addresses.

3. McKernel passes their virtual addresses to the process via the Auxiliary Vector in the stack.

4. When a system call is called, first the control is transferred to the `glibc` wrapper function. And then the control is transferred to the function in the vDSO pages without switching processor mode.

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。

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、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。

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。

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。

2.4.8 POSIX Shared Memory

McKernel の POSIX Shared Memory 機能（/dev/shm/*ファイルのマップによる共有メモリ機能）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 (--mpol-shm-premap) によって有効にできる。

2.4.9 System V 共有メモリ

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、ファイルマップと同様に管理する。

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 構造体は、対応する shmobj に内包させる。

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。

2.4.9.1 実装の制限

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、スワ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）。

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。

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
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

2.5 procfs/sysfs

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 maps, 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 shareing this cache in bitmap form
/sys/devices/system/cpu/cpu*/cache/index*/ shared_cpu_list	Set of CPUs shareing this cache in human readble 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))

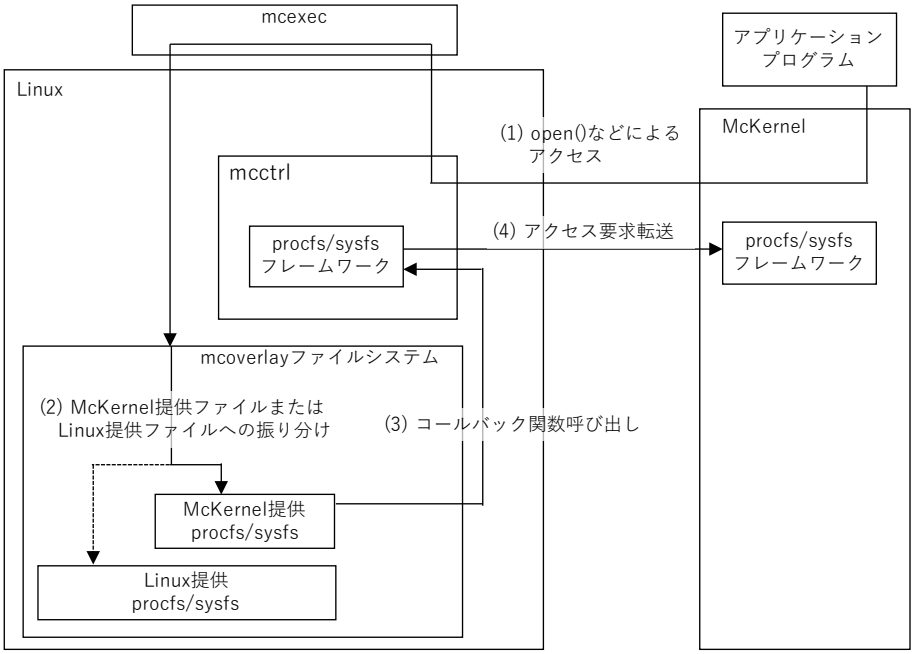


Figure 2.6: `procfs/sysfs` のアクセス要求転送

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。

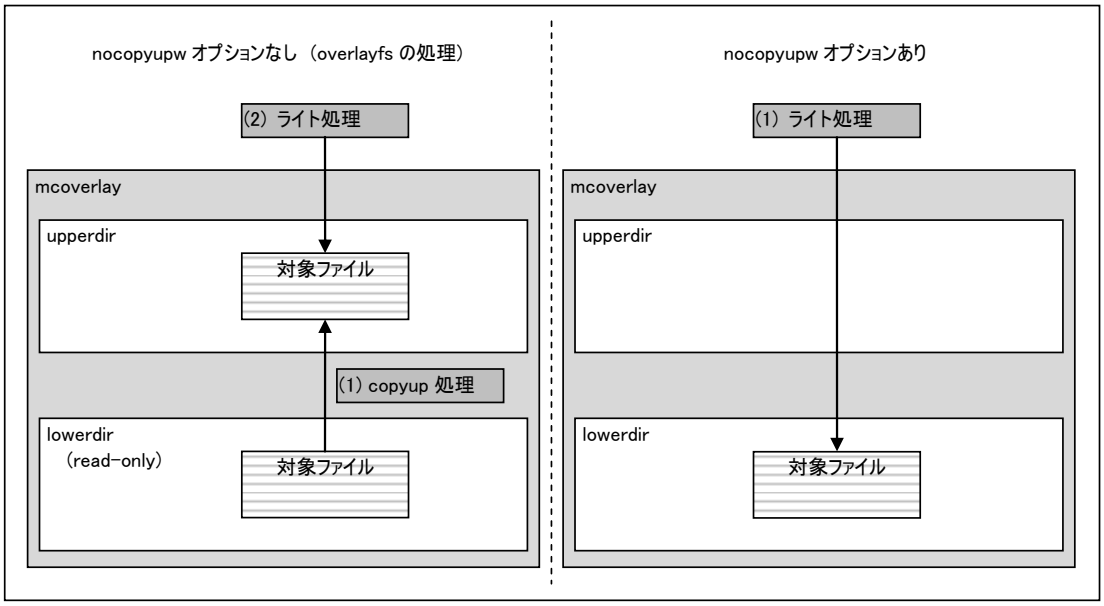


Figure 2.7: nocopyupw オプションの有無によるライト処理の違い

mcoverlayfs のマウントオプションを表 2.5 に示す。nocopyupw, nofscheck が overlayfs 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。

2.6.1 詳細

overlayfs のデータ構造に対する修正は以下の通り。

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。

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

```
        __OVL_OPT_NOCOPYUPW     = (1 << 0),
        __OVL_OPT_NOFSCHECK     = (1 << 1),
};

#define OVL_OPT_NOCOPYUPW(opt)  ((opt) & __OVL_OPT_NOCOPYUPW)
#define OVL_OPT_NOFSCHECK(opt)  ((opt) & __OVL_OPT_NOFSCHECK)
```

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

```
struct ovl_d_fsdata {
        struct list_head list;
        struct dentry *d;
        struct ovl_entry *oe;
};
```

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

```
struct ovl_config {
        char *lowerdir;
        char *upperdir;
        char *workdir;
        bool default_permissions;
        unsigned opt;              <-- 追加
};
```

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

```
struct ovl_fs {
        struct vfsmount *upper_mnt;
        unsigned numlower;
        struct vfsmount **lower_mnt;
        struct dentry *workdir;
        long lower_namelen;
        /* pathnames of lower and upper dirs, for show_options */
        struct ovl_config config;
        struct list_head d_fsdata_list;  <-- 追加
};
```

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

```
enum {
        OPT_LOWERDIR,
        OPT_UPPERDIR,
        OPT_WORKDIR,
        OPT_DEFAULT_PERMISSIONS,
        OPT_NOCOPYUPW,            <-- 追加
        OPT_NOFSCHECK,            <-- 追加
        OPT_ERR,
};

static const match_table_t ovl_tokens = {
        {OPT_LOWERDIR,                  "lowerdir=%s"},
        {OPT_UPPERDIR,                  "upperdir=%s"},
```

```
        {OPT_WORKDIR,                   "workdir=%s"},
        {OPT_DEFAULT_PERMISSIONS,       "default_permissions"},
        {OPT_NOCOPYUPW,                 "nocopyupw"},           <-- 追加
        {OPT_NOFSCHECK,                 "nofscheck"},           <-- 追加
        {OPT_ERR,                       NULL}
};
```

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

```
static struct file_system_type ovl_fs_type = {
        .owner          = THIS_MODULE,
        .name           = "mcoverlay",      <-- 変更
        .mount          = ovl_mount,
        .kill_sb        = kill_anon_super,
};
MODULE_ALIAS_FS("mcoverlay");               <-- 変更
```

overlayfs に対する関数の修正を表 2.6、表 2.7 に示す。

Table 2.6: overlayfs の関数に対する修正（1）

関数	修正内容
ovl_copy_xattr()	OVL_OPT_NOFSCHECK(opt) が有効の場合、vfs_getxattr() のエラーを無視する。
ovl_copy_up_locked()	ovl_copy_xattr() 呼び出し時に ovl_get_config_opt() で取得した opt 値を渡す。
ovl_clear_empty()	ovl_copy_xattr() 呼び出し時に ovl_get_config_opt() で取得した opt 値を渡す。
ovl_setattr()	ovl_get_config_opt() で opt 値を取得する。 OVL_OPT_NOCOPYUPW(opt) の場合、処理しない。
ovl_permission()	ovl_reset_ovl_entry() を呼び出してから処理する。
ovl_setxattr()	ovl_get_config_opt() で opt 値を取得する。 OVL_OPT_NOCOPYUPW(opt) の場合、処理しない。
ovl_removexattr()	ovl_get_config_opt() で opt 値を取得する。 OVL_OPT_NOCOPYUPW(opt) の場合、処理しない。
ovl_d_select_inode()	1. ovl_get_config_opt() で opt 値を取得する。 2. OVL_OPT_NOCOPYUPW(opt) の場合、ovl_open_need_copy_up() を呼び出さない。 3. OVL_OPT_NOFSCHECK(opt) で対象ファイルが sysfs の場合、ovl_find_d_fsdata() を呼び出して dentry 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 ovl_add_d_fsdata() を呼び出して登録し、dentry->d_fsdata に realpath.dentry->d_fsdata の値を設定する。
ovl_get_config_opt()	opt 値を返す。
ovl_reset_ovl_entry()	1. ovl_get_config_opt() で opt 値を取得する。 2. OVL_OPT_NOFSCHECK(opt) の場合、ovl_find_d_fsdata() を呼び出して、dentry が登録されている場合には取得した d_fsdata を oe に設定する。
ovl_find_d_fsdata()	dentry->d_sb->s_fs_info の d_fsdata_list に登録されている d_fsdata を検索して、dentry が登録されていた場合、dentry の ovl_entry を戻す。
ovl_add_d_fsdata()	1. struct ovl_d_fsdata のメモリ領域を確保して、dentry の登録データを設定する。 2. dentry->d_sb->s_fs_info の d_fsdata_list に登録する。
ovl_clear_d_fsdata()	d_fsdata_list に登録されている全ての d_fsdata を削除して、struct ovl_d_fsdata のメモリ領域を解放する。
ovl_path_type()	ovl_reset_ovl_entry() を呼び出してから処理する。
ovl_path_upper()	
ovl_dentry_upper()	
ovl_dentry_lower()	
ovl_dentry_real()	
ovl_dir_cache()	
ovl_set_dir_cache()	
ovl_path_lower()	
ovl_dentry_is_opaque()	
ovl_dentry_set_opaque()	
ovl_dentry_update()	
ovl_dentry_version_inc()	
ovl_dentry_version_get()	
ovl_dentry_release()	
ovl_dentry_revalidate()	
ovl_dentry_weak_revalidate()	

Table 2.7: overlayfs の関数に対する修正（２）

関数	修正内容
ovl_lookup_real()	OVL_OPT_NOFSCHECK(opt) の場合、ovl_dentry_weird() を呼び出さない。
ovl_path_next()	ovl_reset_ovl_entry() を呼び出してから処理する。
ovl_lookup()	1. ovl_get_config_opt() で opt 値を取得する。 2. ovl_reset_ovl_entry() を呼び出してから処理する。 3. ovl_lookup_real() を呼び出す際、opt 値を渡す。
ovl_put_super()	ovl_clear_d_fsdata() を呼び出してから処理する。
ovl_statfs()	struct kstatfs の f_type に MCOVERLAYFS_SUPER_MAGIC を設定する。
ovl_show_options()	nocopyupw, nofscheck オプションの説明を追加する。
ovl_parse_opt()	nocopyupw, nofscheck オプションの設定を追加する。
ovl_mount_dir_noesc()	OVL_OPT_NOFSCHECK(opt) の場合、ovl_dentry_weird() を呼び出さない。
ovl_mount_dir()	ovl_mount_dir_noesc() を呼び出す際、opt を渡す。
ovl_lower_dir()	ovl_mount_dir_noesc() を呼び出す際、opt を渡す。
ovl_fill_super()	1. struct ovl_fs の d_fsdata_list を初期化する。 2. ovl_mount_dir() を呼び出す際、opt を渡す。 3. ovl_lower_dir() を呼び出す際、opt を渡す。 4. OVL_OPT_NOCOPYUPW(opt) の場合、以下の設定を行わない。 • mnt->mnt_flags |= MNT_READONLY; • sb->s_flags |= MS_RDONLY;

2.6.2 実装の制限

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]/下のファイルを作成し直すが、overlayfs は lower に指定されるディレクトリ下のファイルの inode 番号が変わった場合、エラーを返すためである。

2.6.3 開発時の留意事項

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追従する必要があった。

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

また、以下のバージョンの Linux カーネルでのみ動作する。

- 3.10.0-327 から 3.10.0-693（RHEL-7.2 から 7.4）
- 4.0.0 から 4.1.0
- 4.6.0 から 4.7.0

2.7 デバイスドライバ

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。
以下、それぞれの方法を説明する。

2.7.1 Linux ドライバの利用

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。

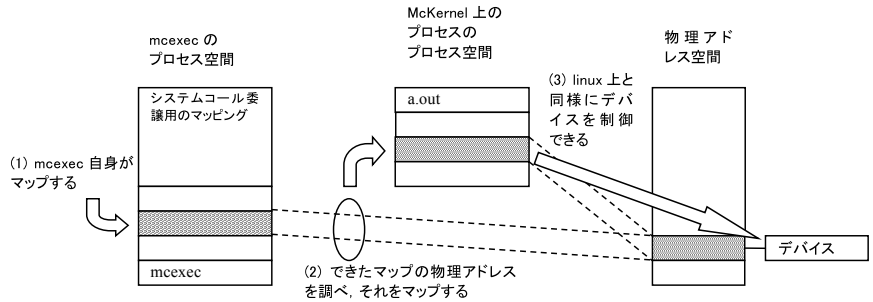


Figure 2.8: Linux ドライバ利用の動作

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。

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））

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））

2.7.2 McKernel 内部での実装

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、そ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。

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。

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。

2.8 XPMEM ドライバ

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。

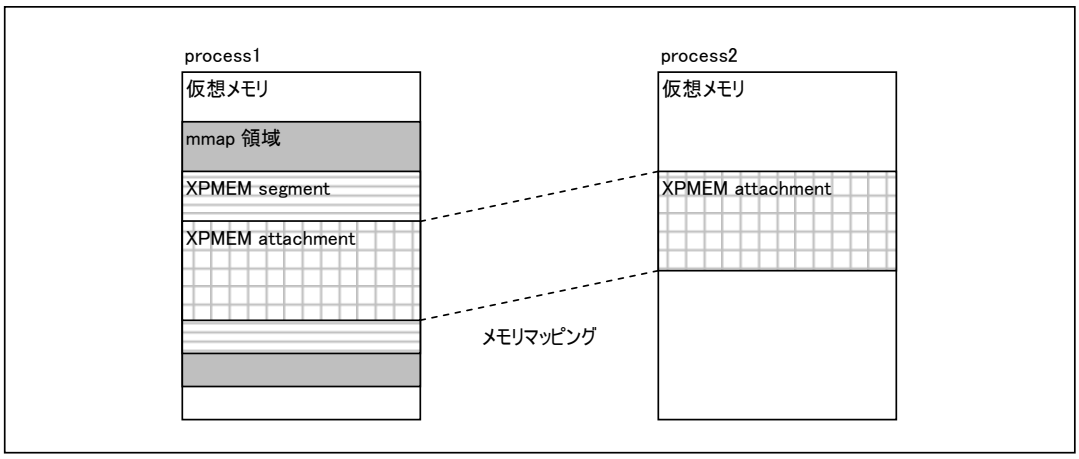


Figure 2.9: XPMEM のメモリマッピング

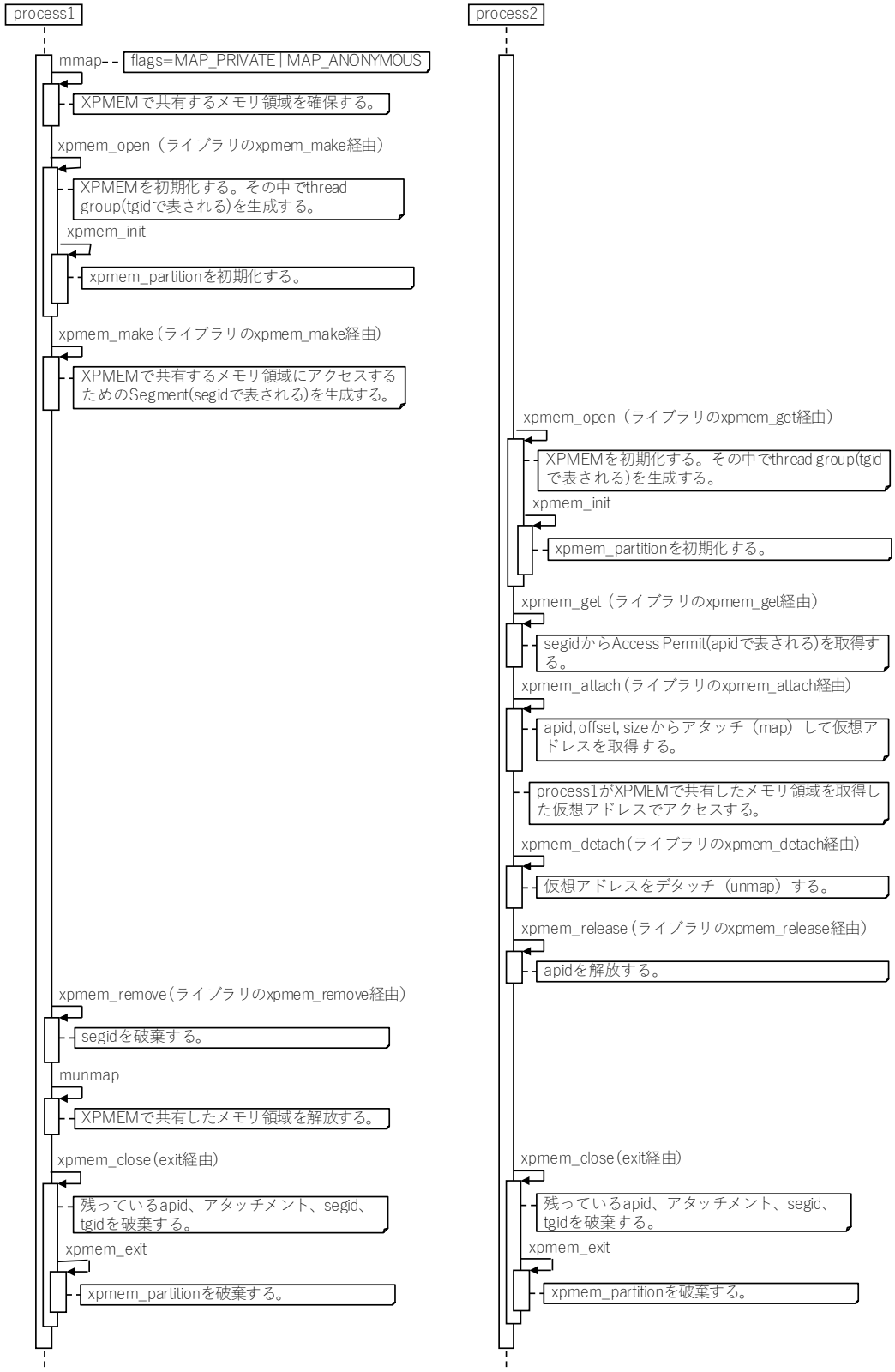


Figure 2.10: XPMEM の動作フロー

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 XPMEM 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 XPMEM segment のメモリ領域を XPMEM attachment として管理して、マップする。マップは、プロセス (process2) が XPMEM 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 XPMEM 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。

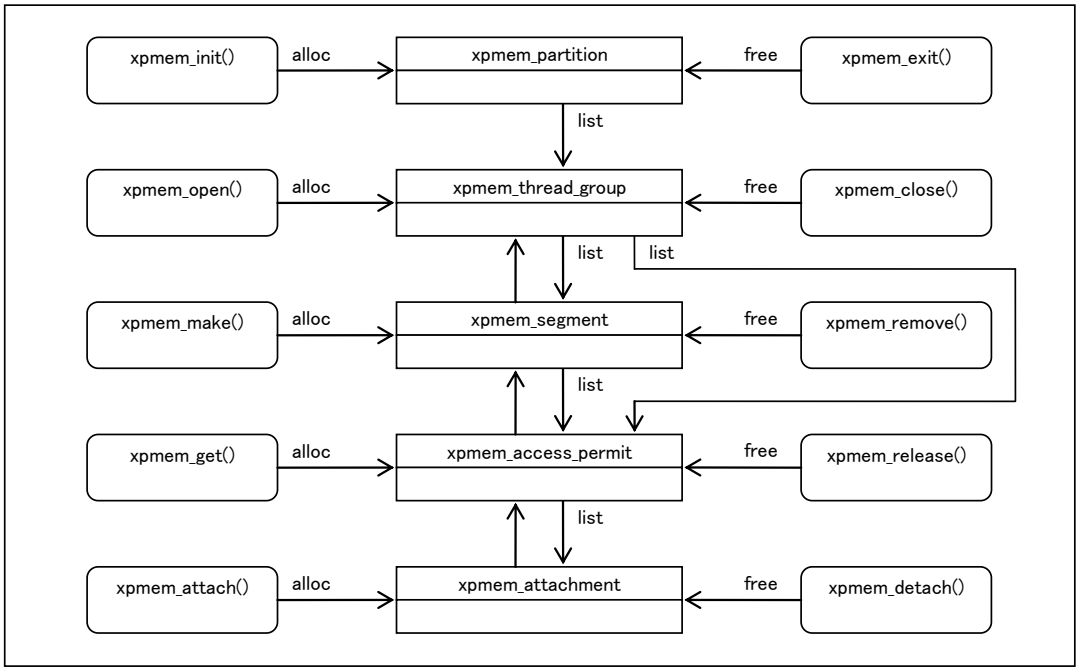


Figure 2.11: XPMEM のデータ構造を生成・破棄する関数

XPMEM 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

XPMEM のデータ構造を図 2.12 に示す。

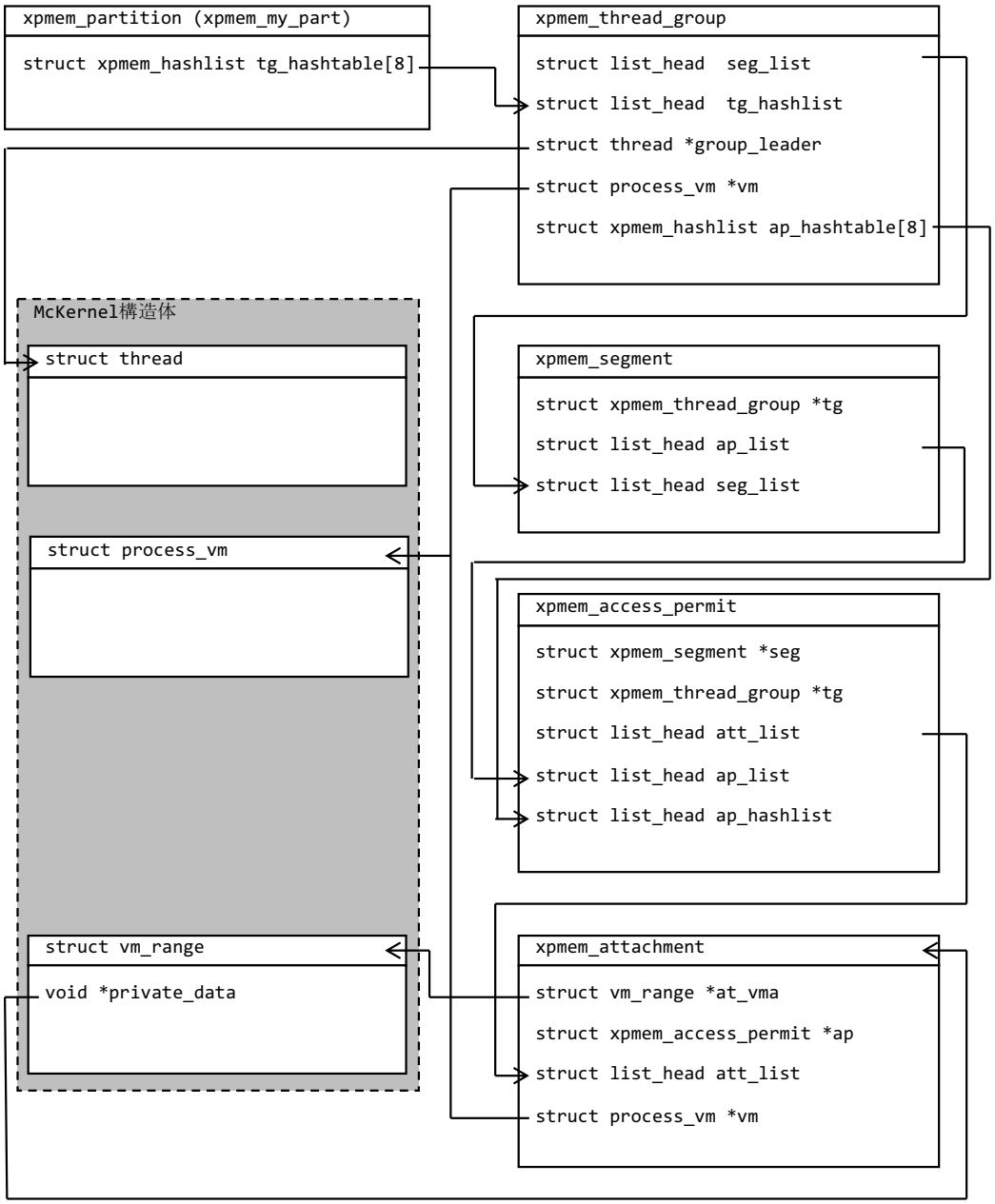


Figure 2.12: XPMEM のデータ構造

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() 呼び出しで、/dev/null デバイスファイルをオープンして、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戻す。
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 を戻す。
   (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。

(c) XPMEM_CMD_REMOVE
xpmem_cmd_remove データを取得する。
xpmem_remove() を呼び出す。

(d) XPMEM_CMD_GET
xpmem_cmd_get データを取得する。
xpmem_get() を呼び出す。xpmem_cmd_get データの apid を設定する。

(e) XPMEM_CMD_RELEASE
xpmem_cmd_release データを取得する。
xpmem_release() を呼び出す。

(f) XPMEM_CMD_ATTACH
xpmem_cmd_attach データを取得する。
xpmem_attach() を呼び出す。xpmem_cmd_attach データの vaddr を設定する。

(g) XPMEM_CMD_DETACH
xpmem_cmd_detach データを取得する。
xpmem_detach() を呼び出す。

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

2.8.3 XPMEM デバイスファイルのクローズ

書式

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

説明

xpmem_close は、以下の処理を行う。

1. pid から xpmem_thread_group(tg) を取得する。
2. xpmem_release_aps_of_tg() を呼び出して、xpmem_access_permit、xpmem_attachment を破棄する。
3. xpmem_remove_segs_of_tg() を呼び出して、xpmem_segment を破棄する。
4. xpmem_destroy_tg() を呼び出して、xpmem_thread_group を破棄する。
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() を呼び出して XPMEM を終了する。
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() でクローズする。

戻り値

0	正常終了

2.8.4 XPMEM の初期化

書式

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

説明

xpmem_init は、以下の処理を行う。

1. xpmem_partition を生成して、初期設定する。

戻り値

0	正常終了
-ENOMEM	十分な空きメモリ領域が無い

2.8.5 XPMEM の終了

書式

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

説明

xpmem_exit は、以下の処理を行う。

1. xpmem_partition を破棄する。

戻り値

なし。

2.8.6 xpmem_segment の生成

書式

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

説明

xpmem_make は、以下の処理を行う。

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

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

2.8.7 xpmem_segment の破棄

書式

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

説明

xpmem_remove は、以下の処理を行う。

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

戻り値

0	正常終了
-EACCES	許可がない
-EINVAL	引数が無効である

2.8.8 xpmem_access_permit の生成

書式

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

説明

xpmem_get は、以下の処理を行う。

1. segid から xpmem_thread_group(seg_tg) を取得する。
2. seg_tg、sigid から xpmem_segment を取得する。
3. 自プロセスの xpmem_thread_group(ap_tg) を取得する。
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。
5. xpmem_access_permit を生成して、初期設定する。
6. apid_p に apid を設定する。

戻り値

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

2.8.9 xpmem_access_permit の破棄

書式

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

説明

xpmem_release は、以下の処理を行う。

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。

戻り値

0	正常終了
-EACCES	許可がない
-EINVAL	引数が無効である

2.8.10 xpmem_attachment の生成

書式

```
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)
```

説明

xpmem_attach は、以下の処理を行う。

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。

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

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

2.8.11 xpmem_attachment の破棄

書式

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

説明

xpmem_detach は、以下の処理を行う。

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

戻り値

0	正常終了
-EACCES	許可がない
-EINVAL	引数が無効である

2.8.12 vm_range の fault 処理

書式

```
int xpmem_fault_process_memory_range(struct process_vm *vm, struct vm_range
*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) を戻す。
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) を戻す。
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) を戻す。
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

0	正常終了

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。
- 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 monitoror.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。
- 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。
- 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。

2.11 Non-Maskable Interrupt

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。

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

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。

NMI の利用ステップは以下の通り。

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。
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。
3. 各 CPU が以下を実行する。
   (a) NMI を受けて、NMI ハンドラ nmi() に制御を移す。
   (b) nmi() で nmi_mode の値に応じた処理を行う。

以下、関連関数の動作を説明する。

2.11.1 NMI 動作設定

書式

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

説明

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。

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。

戻り値

0	正常終了

2.11.2 NMI 送信

書式

```
static int smp_ihk_os_send_nmi(ihk_os_t ihk_os, void *priv, int mode)
```

説明

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

戻り値

0	正常終了
-EINVAL	エラー

2.11.3 NMI ハンドラ

書式

```
void nmi()
```

説明

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 に示す。

2.12 全 CPU 一時停止

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。

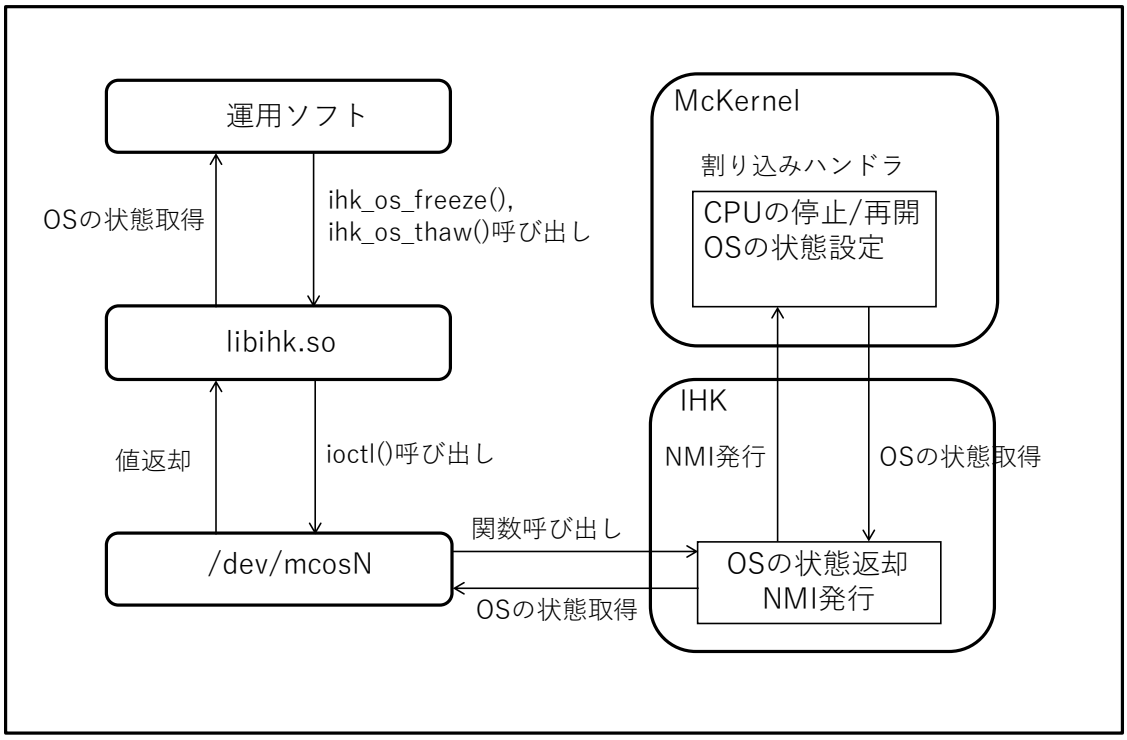


Figure 2.13: 構成要素関連図

全 CPU 一時停止機能の構成を図 2.13 に示す。 1

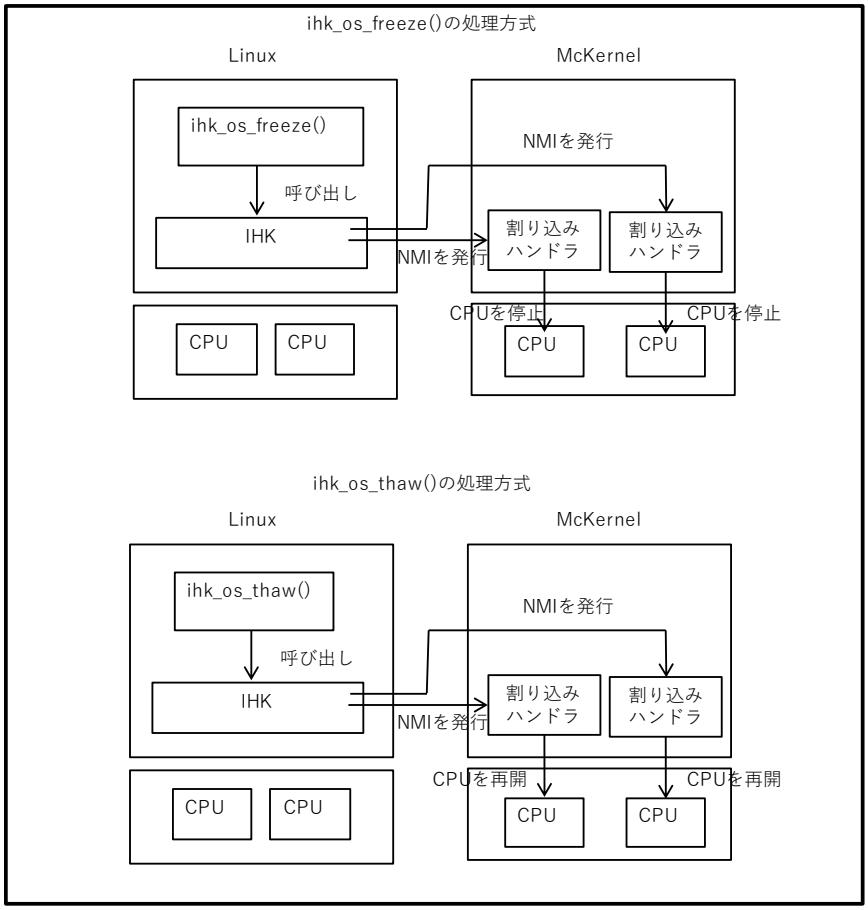


Figure 2.14: 全 CPU 一時停止および一時停止からの復帰のフロー

全 CPU 一時停止の動作を図 2.14 を用いて説明する。

1. バッチジョブスケジューラが `ihk_os_freeze()` 経由で `IHK_OS_FREEZE` コマンドを指定して `ioctl()` を呼ぶ。
2. IHK-master core が `__ihk_os_freeze()` 経由で IHK-master driver の `smp_ihk_os_freeze()` を呼ぶ。
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_os_send_nmi()` を呼び、各 CPU に NMI を送る。
4. 各 CPU が以下を実行する。
   (a) NMI を受けて、NMI ハンドラ `nmi()` に制御を移す。
   (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への遷移であるため、`freeze_thaw()` を呼ぶ。
   (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` 命令後のジャンプ先を `__freeze()` にする。

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

i. CPU の状態をバックアップ用変数に保持する。

ii. CPU の状態を一時停止状態に設定する。

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

一時停止からの復帰の動作を図 2.14 を用いて説明する。

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。

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

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。

4. 各 CPU が以下を実行する。

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

   (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。

   (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。

以下、関連関数のインターフェイスと動作を説明する。

2.12.1 一時停止指示（IHK-master core）

書式

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

説明

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。

戻り値

0	正常終了

2.12.2 一時停止からの復帰指示（IHK-master core）

書式

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

説明

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() を呼び出す。

戻り値

0	正常終了

2.12.3 一時停止指示（IHK-master driver）

書式

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

説明

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態へ遷移させ、また CPU を NMI を受けるまで停止させる。

戻り値

0	正常終了

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

書式

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

説明

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

戻り値

0	正常終了

2.12.5 一時停止および一時停止からの復帰指示

書式

```
long freeze_thaw(void *nmi_ctx)
```

説明

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。

戻り値

0	一時停止を行った
1	一時停止からの復帰を行った

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

書式

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

説明

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 待ちの状態で停止させることができる。

2.12.7 一時停止指示（ラッパー）

書式

```
void __freeze()
```

説明

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。

2.12.8 一時停止指示

書式

```
void freeze()
```

説明

ステップは以下の通り。

1. CPU 状態を保存する。
2. CPU 状態を IHK_OS_MONITOR_KERNEL_FROZEN に遷移させる。
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。
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。

2.13 カーネルダンプ

カーネルダンプの採取と解析のステップは以下の通り。

1. 以下のいずれかの方法でダンプファイルを作成する。
   (a) 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（以降、McKernel 主導ダンプと呼ぶ)。
   (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、コマンド vmcore2mckdump を用いて McKernel 形式に変換する（以降、Linux 主導ダンプと呼ぶ)。
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。

以下、詳細を説明する。

2.13.1 全体の処理の流れ

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 を用いて説明する。

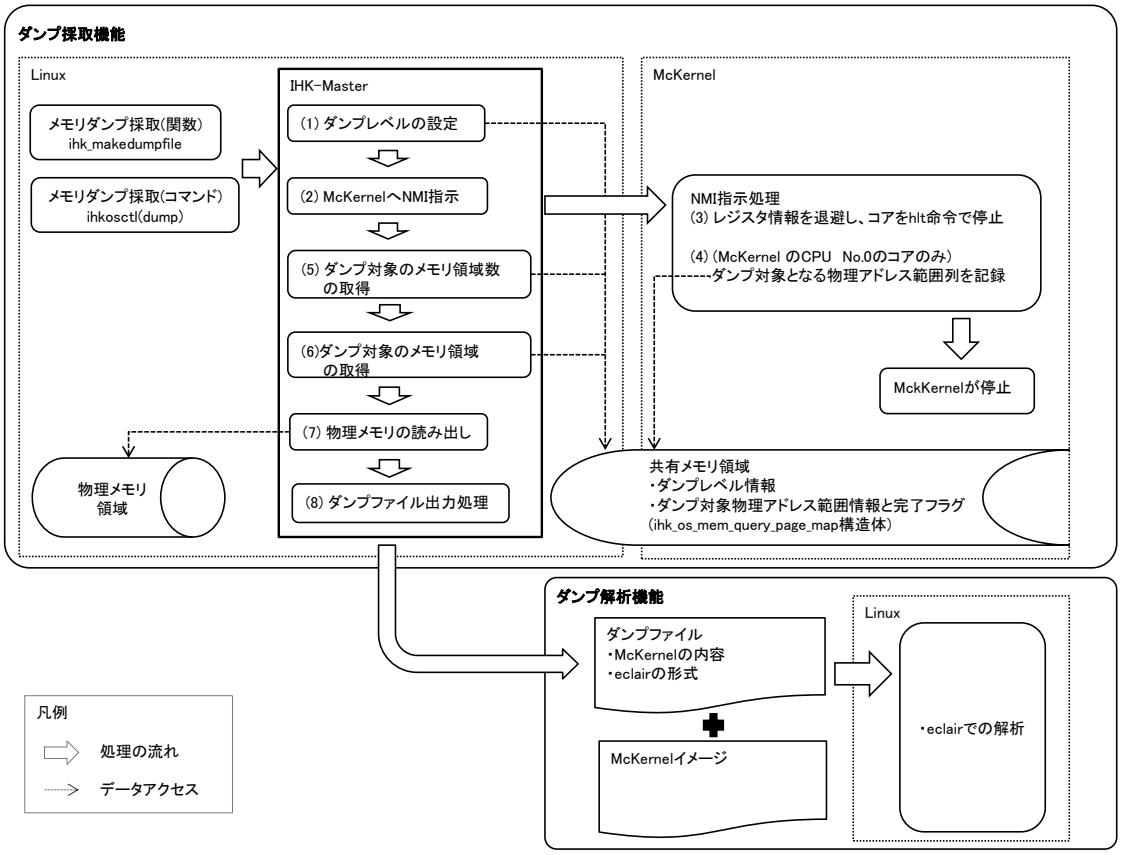


Figure 2.15: McKernel 主導ダンプの場合のダンプ採取機能とダンプ形式変換機能の処理の流れ

1. IHK が OS ブート時に、各物理ページがダンプ対象であるかを示す情報（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。

2. 管理者が `ihk_os_makedumpfile()` でダンプを指示する。

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)）

4. IHK が McKernel の各コアへ NMI を送る。（図の (2)）

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)）

   (a) レジスタ情報を退避する。

   (b) 共有メモリを参照してダンプレベルを取得する。

   (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。

(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。

(e) 完了フラグに 1 をセットする。

(f) `hlt` 命令で停止する。

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を前記メモリ領域に記録する。(図の (5)、(6))

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 (7)、(8))

8. 管理者は eclair を用いてダンプファイルの解析を行う。

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

```
struct ihk_dump_page_set {
    unsigned int completion_flag;  /* 書き込み完了フラグ */
    unsigned int count;            /* ダンプ対象のページ情報数 */
    unsigned long page_size;       /* ダンプ対象のページ情報の全体サイズ */
    unsigned long phy_page;        /* ダンプ対象のページ情報の物理アドレス
                                      (struct ihk_dump_page の配列) */
}

struct ihk_dump_page {
    unsigned long start;       /* マップ情報の開始物理アドレス */
    unsigned long map_count;   /* マップ情報の領域数 (map[] の配列数) */
    unsigned long map[];       /* マップ情報 (ビットマップ形式) */
};
```

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 を用いて説明する。

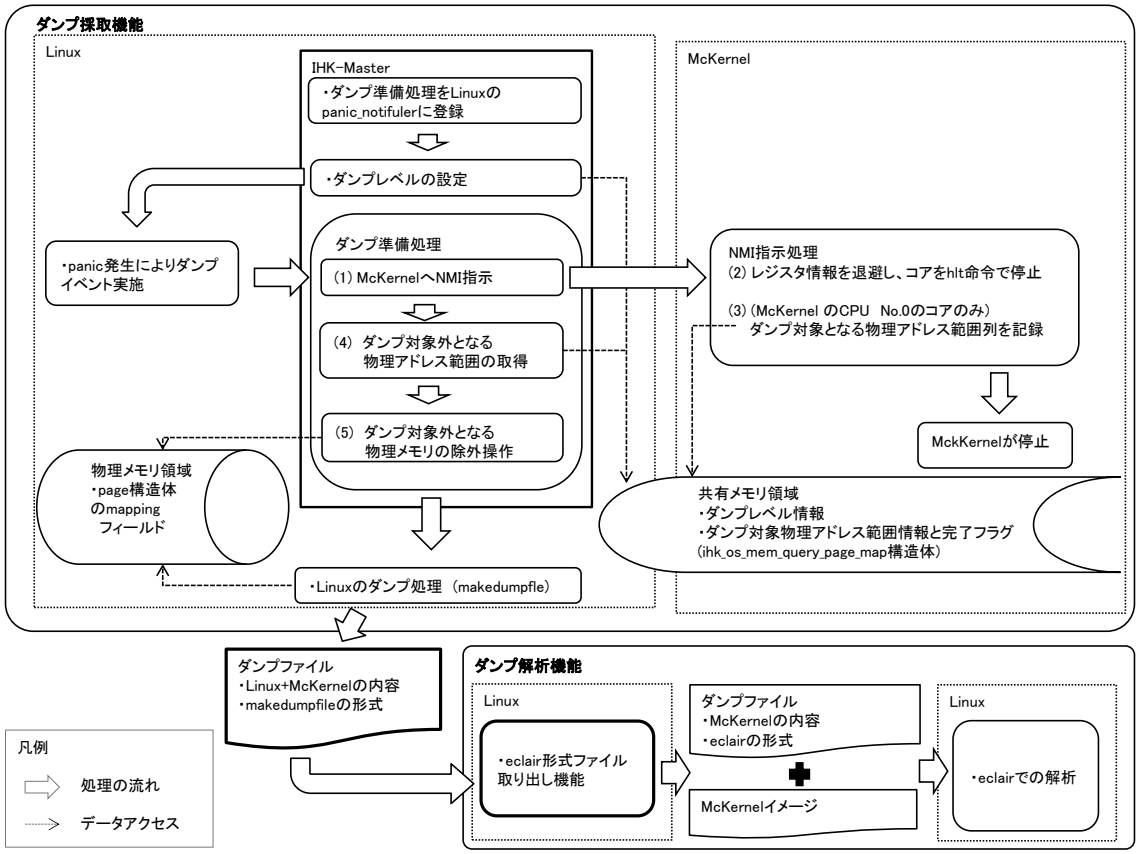


Figure 2.16: Linux 主導ダンプの場合のダンプ採取機能とダンプ形式変換機能の処理の流れ

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 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. 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 に設定する。
3. IHK がダンプ準備処理関数を Linux の `panic_notifier` に登録する。
4. 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。
5. IHK が McKernel の各コアへ NMI を送る。（図の (1)）
6. 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（図の (3)）。McKernel の第 0 CPU は以下を実行する。（図の (2)、(3)）
   (a) レジスタ情報を退避する。
   (b) 共有メモリを参照してダンプレベルを取得する。
   (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にダンプからの除外を記録する。
   (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。

(e) 完了フラグに 1 をセットする。

(f) `hlt` 命令で停止する。

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 に設定する。（図の (4)、(5))）

8. Linux が `makedumpfile` コマンドを実行する。

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。

10. 管理者が `ldump2mcdump` コマンドで、`makedumpfile` 形式のダンプファイルを eclair 形式に変換する。

11. 管理者は eclair を用いてダンプファイルの解析を行う。

2.13.2 ユーザメモリ領域情報取得

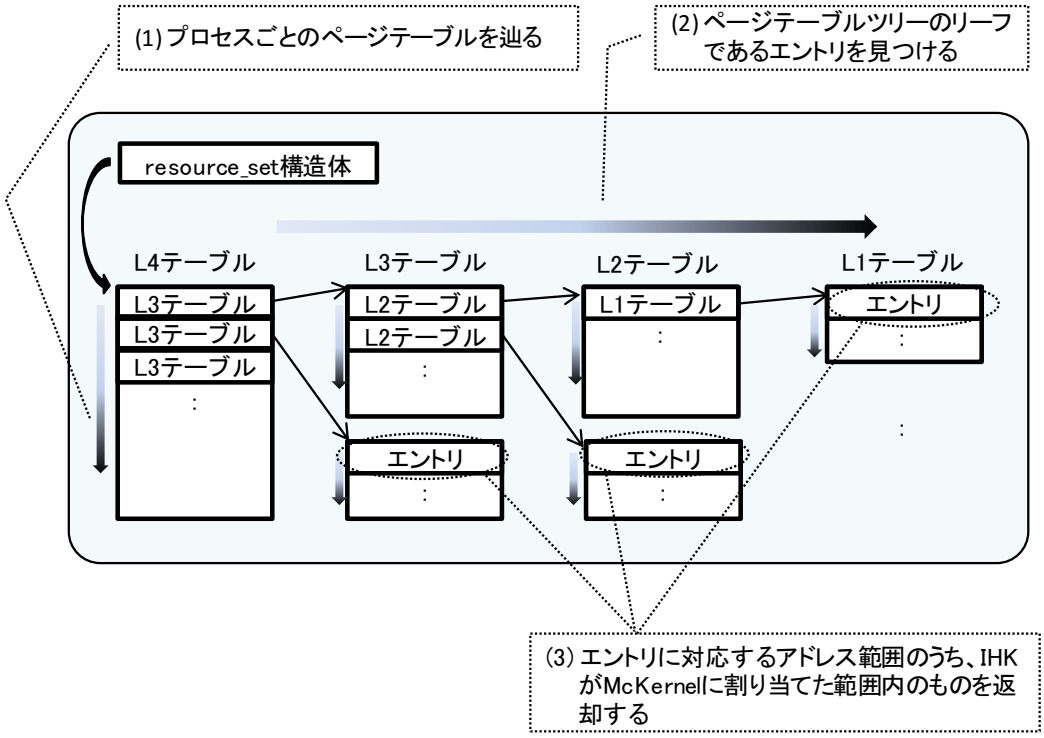


Figure 2.17: ユーザメモリ領域情報取得処理の流れ

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。

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以下を行う。（図の（ 1 ））

   (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 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

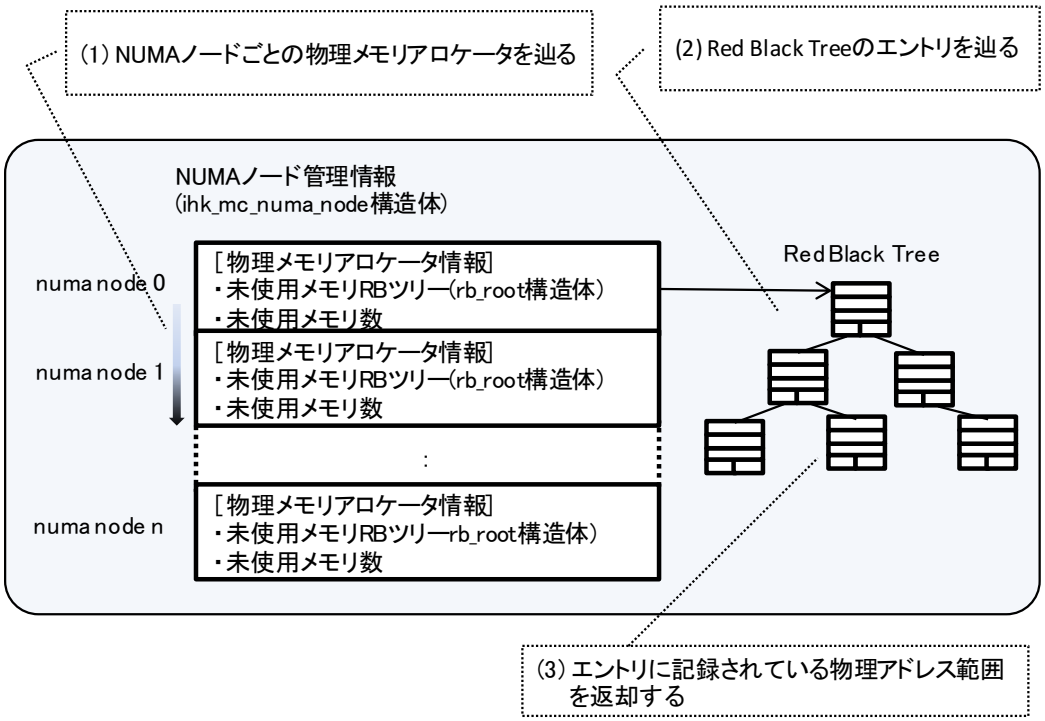


Figure 2.18: 未使用メモリ領域情報取得処理の流れ

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。

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

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

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

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	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。 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。 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。
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。

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

2.13.4.1 ダンプファイルの形式

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、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、 ALLOC、 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:
 771800000 00000000 00f0ffff 00000000 00400000  .............@..
 771800010 21000000 0c000000 48004800 00000000  !.......H.H.....
 771800020 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うが、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う。ダンプ解析コマンドが gdb コマ

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すため、以下の手順で動作する。

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 gdb エージェントとしての初期化をする。
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。
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。

   `-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ル名＞

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。
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 エージェントとしての動作に専念する。

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。

2.13.6 ダンプ形式変換（crash プラグイン）

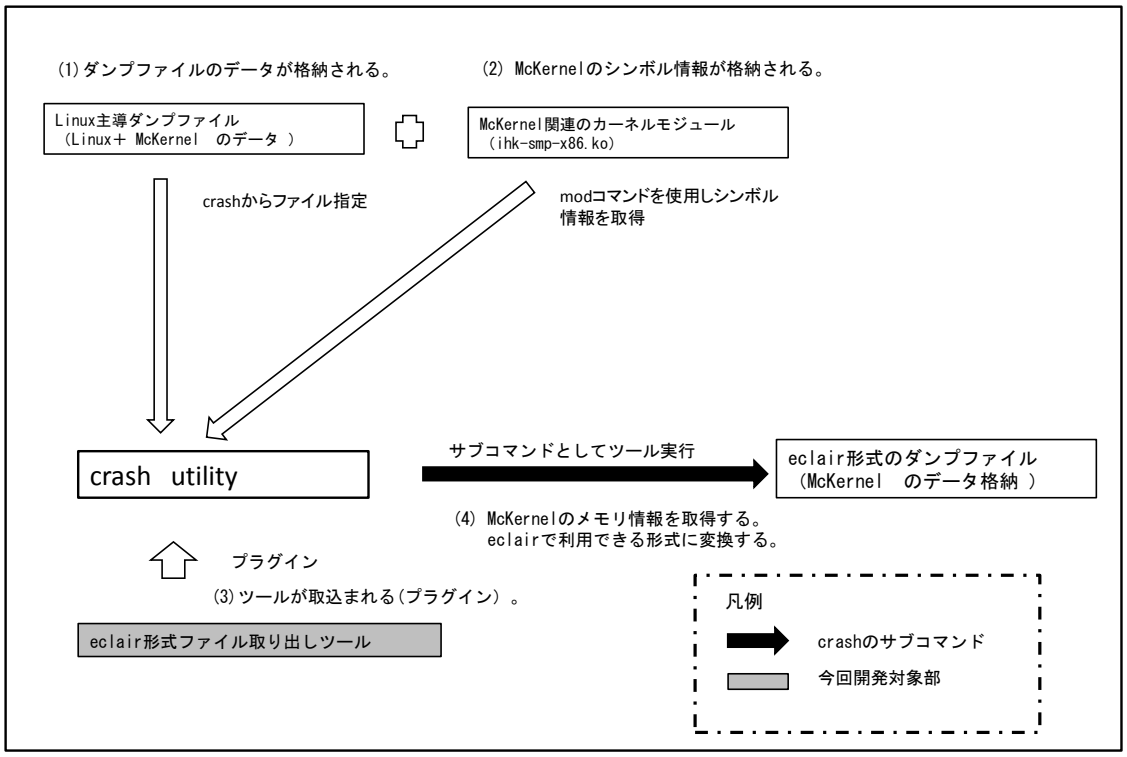


Figure 2.20: ダンプ形式変換処理の流れ

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 主導ダンプは 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。

2.14 プロセスダンプ

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。

2.14.1 実装の制限

出力される情報についての制限は以下の通り。

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。

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、以下のフィールドに対応する情報は格納しない。

```
    struct elf_siginfo pr_info;   /* シグナル情報 */
    short int pr_cursig;          /* 現在のシグナル */
    a8_uint64_t pr_sigpend;       /* ペンディングされているシグナル */
    a8_uint64_t pr_sighold;       /* hold されているシグナル */
    pid_t pr_pid;
    pid_t pr_ppid;
    pid_t pr_pgrp;
    pid_t pr_sid;
    struct prstatus64_timeval pr_utime;   /* ユーザ時間 */
    struct prstatus64_timeval pr_stime;   /* システム時間 */
    struct prstatus64_timeval pr_cutime;  /* 累積ユーザ時間 */
    struct prstatus64_timeval pr_cstime;  /* 累積システム時間 */
```

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

```
struct elf_siginfo {
    int si_signo; /* signal number */
    int si_code;  /* extra code */
    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。

2.15.1 スレッドマイグレート処理

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。

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システムコールを用いて指示する。

2. McKernel はシステムコールを受けて以下の処理を行う。

   (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。

   (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。

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コール完了を待ってスリープする。

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。

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、スレッドを終了する。

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。

1. McKernel からスレッドマイグレート指示を受ける。

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成されることはない。

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（次節で説明する）。

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。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に mcexec のコンテキストに復帰する必要があるためである。

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
`mmap,mprotect,munmap,brk,futex`	IKC を用いて McKernel に処理を依頼する
`getpid, gettid`	`mcexec` に記録しておいた id を返す
`open, read, write` 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
`exit_group, _exit`	`mcexec` 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。

1. 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。

2. 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 領域を破壊する可能性があるためである。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 (例えば、x86_64 では syscall 命令の発行)。

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 `mcctrl` において mcexec の TLS と McKernel スレッドの TLS を保持しておく。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 に切り替え、出口では McKernel スレッドの TLS に切り替える。

2.15.4 スレッド終了処理

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。

1. スレッド終了の捕捉
   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、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 (`trace_sched_process_exit`) を用いて捕捉する。

2. McKernel 側でのスレッド終了
   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 側のスレッドを終了させる。

3. Linux 側でのスレッド終了
   以下のステップで Linux 側のスレッドを終了させる。

   (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。

   (b) tracer プロセスが終了する。

   (c) `mcexec` のスレッドが `_exit()` を呼び出すことで終了する。

2.15.5 実装詳細

2.15.5.1 Linux CPU へのスレッド生成の構成

Linux CPU へのスレッド生成の構成を図 2.21 に示す。

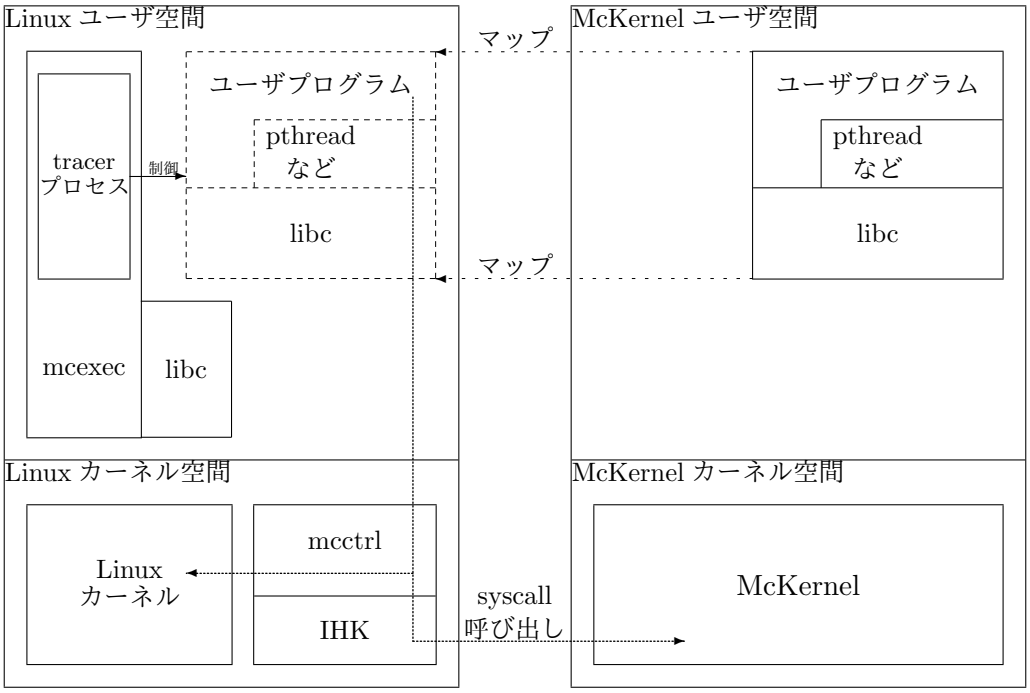


Figure 2.21: Linux CPU へのスレッド生成の構成

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 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 とユーザプログラムの libc が異なる実体として配置されている。

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、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 経由で McKernel 上で処理するように制御する。

2.15.5.2 util_indicate_clone システムコール

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 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。

thread 構造体の関連フィールドは以下のように定義される。

```
struct thread {
    ... 略 ...
    int mod_clone;                  // 生成対象 OS
    void *mod_clone_arg;            // CPU 位置の指示、スレッドの振る舞いの記述
};
```

2.15.5.3 util_migrate_inter_kernel システムコール

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

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。コピーに失敗した場合、EFAULT を返却する。
2. コンテキスト退避用ページを確保する。
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。
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、sched_setaffinity をオフロードする[2]。
5. コンテキスト退避用ページを解放する。
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。
   (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() でプロセスを終了する。
   (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] 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

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 が処理するときに 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 SPAWINING_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。

2.15.5.9 do_syscall の処理

do_syscall に対して、以下の変更を行う。

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 McKernel へのシステムコール委譲)、以下の処理を行う。
   (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返却する。
   (b) システムコール番号が rt_sigreturn 以外の場合、以下を行う。
      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、登録されていない場合は ENOSYS を返却する。
      ii. システムコールコンテキストを作成し、syscall_table に登録されているシステムコール処理を呼び出す。結果を返却する。
   (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 Linux に通知する。この処理は remote page fault と同様である。

2.15.5.10 mcexec の処理

mcexec は以下の処理を行う。

1. sched_set_affinity に対するオフロード処理として、以下を行う。
   (a) create_worker_create を呼び、新しいワーカースレッドを作成する。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ある。
   (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。
   (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スト退避領域へコピーさせる (MCEXEC_UP_UTIL_THREAD1)。
   (d) tracer プロセスを生成する。tracer プロセスの詳細は 4 に示す。
   (e) uti_attr が指定されている場合、mcctrl に uti_attr 処理を依頼する (MCEXEC_UP_UTI_ATTR)。
   (f) 以下に示す switch_ctx を行う。
      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。
      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 (MCEXEC_UP_UTIL_THREAD2)。
      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。
   (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。
   (h) mcexec スレッドのコンテキスト退避領域を解放する。
   (i) スレッドを終了する。
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。
   (a) mcctrl に MCEXEC_UP_SIG_THREAD を要求し、mcexec の TLS に切り替える。
   (b) mcexec のスレッドの場合、McKernel に受信したシグナルを通知する。
   (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。
      i. mcctrl に rt_sigaction で MCEXEC_UP_SYSCALL_THREAD 要求を行い、シグナルハンドラの設定を取得する。
      ii. シグナルハンドラが SIG_IGN の場合、何もしない。
      iii. シグナルハンドラが SIG_DFL の場合、シグナルが SIGCHLD、SIGURG、SIGCONT 以外の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。これによって、当該シグナルを受信して自プロセスが終了する。
      iv. シグナルハンドラがアドレスの場合、一時的に TLS を元に戻してアドレスの関数 (シグナルハンドラ) を呼び出す。
   (d) mcctrl に MCEXEC_UP_SIG_THREAD を要求し、元の TLS に切り替える。
3. tracer プロセスは以下の処理を行う。
   (a) tracee にて、tracer と待ち合わせに使用するパイプを作成する。
   (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 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。
   (c) tracee は以下の処理を行う。
      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。

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 になる)。
iii. パイプの入力イベントの発生を (最大)1 秒待つ (select)。
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。
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。
vi. パイプから 1 バイト読み込み、パイプを閉じる。
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。
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。)

(d) パイプの入力側を閉じる。
(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。
(f) /dev/mcos 以外のファイルディスクリプタを全て閉じる。
(g) 標準入出力を/dev/null に割り当てる。
(h) tracee スレッドに PTRACE_ATTACH する。
(i) tracee の停止を待つ (wait)。
(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ために、PTRACE_SETOPTIONS で PTRACE_O_TRACESYSGOOD を指定する。
(k) パイプに 1 バイト書き出し、パイプを閉じる。
(l) 以下、無限ループ。
i. PTRACE_SYSCALL により tracee を再開する。
ii. tracee の停止を待つ (wait)。
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。
iv. 停止以外の場合、continue[3]。
v. システムコールで停止した場合、以下を行う。
A. PTRACE_GETREGS を行い、tracee のレジスタを得る。
B. システムコール番号が ioctl で引数に MCEXEC_UP_SYSCALL_THREAD が指定されている場合、戻り値を逆オフロード結果に書き換える。
C. システムコール番号が逆オフロード対象で、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)、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数を設定する。
D. PTRACE_SETREGS を行い、tracee のレジスタを更新する。
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 に指定するシグナルとして、停止シグナルを設定する。

2.15.5.11 `mcctrl` の処理

`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。

`MCEXEC_UP_UTIL_THREAD2` コマンドに対しては、`host_thread` 構造体を作成する。

`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、以下のように定義される。

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

```
struct host_thread {
    struct host_thread *next;                // 同一 PID 内のリスト
    struct mcos_handler_info *handler;       // LWK 情報へのハンドラ
    int pid;                                 // プロセス ID
    int tid;                                 // スレッド ID
    unsigned long usp;                       // mcexec コンテキストの SP
    unsigned long lfs;                       // mcexec の TLS ベース
    unsigned long rfs;                       // ユーザプログラムの TLS ベース
};
```

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

```
strruct mcos_handler_info {
    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。

i. 全ての NUMA ドメインについて、親スレッドが属しているかどうかを調べ、UTI_FLAG_SAME_NUMA_DOMAIN が指定されている場合は当該ドメインに属す CPU 集合の、また、UTI_FLAG_DIFFERENT_NUMA_DOMAIN が指定されている場合には親スレッドが属していないドメインの CPU 集合の和集合を求める。

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。

(c) フラグに UTI_FLAG_SAME_L3 か UTI_FLAG_DIFFERENT_L3 が設定されており、且つ、親スレッドの CPU が L3 キャッシュと持つ場合、以下を行う。

i. UTI_FLAG_SAME_L3 が指定されている場合、キャッシュを共有する CPU の集合を求める。

ii. UTI_FLAG_DIFFERENT_L3 が指定されている場合、キャッシュを共有する CPU の集合の補集合を求める。

iii. 求めた CPU 集合と作業用 cpumask の積集合を求め、作業用 cpumask に設定する。

(d) フラグに UTI_FLAG_SAME_L2 か UTI_FLAG_DIFFERENT_L2 が設定されており、且つ、親スレッドの CPU が L2 キャッシュと持つ場合、以下を行う。

i. UTI_FLAG_SAME_L2 が指定されている場合、キャッシュを共有する CPU の集合を求める。

ii. UTI_FLAG_DIFFERENT_L2 が指定されている場合、キャッシュを共有する CPU の集合の補集合を求める。

iii. 求めた CPU 集合と作業用 cpumask の積集合を求め、作業用 cpumask に設定する。

(e) フラグに UTI_FLAG_SAME_L1 か UTI_FLAG_DIFFERENT_L1 が設定されており、且つ、親スレッドの CPU が L1 キャッシュと持つ場合、以下を行う。

i. UTI_FLAG_SAME_L1 が指定されている場合、キャッシュを共有する CPU の集合を求める。

ii. UTI_FLAG_DIFFERENT_L1 が指定されている場合、キャッシュを共有する CPU の集合の補集合を求める。

iii. 求めた CPU 集合と作業用 cpumask の積集合を求め、作業用 cpumask に設定する。

6. 以下の処理によって、CPU アフィニティの設定、及び、スケジューラの設定を行う。

(a) 作業用 cpumask が空集合の場合は何もしない。

(b) フラグに UTI_FLAG_EXCLUSIVE_CPU が設定されている場合、以下の処理を行う。

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)

ii. スケジューラに SCHED_FIFO を設定する。

(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。

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)

(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。

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)

ii. スケジューラに SCHED_FIFO を設定する。

(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。

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)

(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。

7. 作業用 cpumask を開放する。

作業用 cpumask から CPU を 1 つ選択する手順を以下に示す。

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。

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。

3. 更新した CPU ID 以外の作業用 cpumask のビットをクリアする。

2.15.5.11.2 MCEXEC_UP_SIG_THREAD の処理

MCEXEC_UP_SIG_THREAD の要求に対して、以下の処理を行う。

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。

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。

2.15.5.11.3 MCEXEC_UP_SYSCALL_THREAD の処理

MCEXEC_UP_SYSCALL_THREAD の要求に対して、以下の処理を行う。

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。

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。

3. wait_queue_head_list_node を作成し、システムコールの結果待ちに備える。

4. util_migrate_inter_kernel のレスポンスに syscall_request の物理アドレス設定する。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. __notify_syscall_requester を呼び出して、McKernel 側にレスポンスの変更を通知する。

6. wait_event_interruptible によって、システムコール完了を待つ。

7. 結果を返却する。

2.15.6 実装の制限

制限は以下の通り。

- 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スは行えない。
- 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説明したもののみである。
- 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。

2.16 高速プロセス起動

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 プログラム起動時間を短縮する機能を提供する。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ションが挙げられる。

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帰で置き換えることで実現する。高速プロセス起動は以下の機能から構成される。

- プロセス実行停止および停止からの再開機能
  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うにする。ライブラリ関数として実装する。
- MPI プログラムの繰り返し起動指示機能
  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
ql_mpiexec_start	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 ql_talker と ql_server 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。
ql_mpiexec_finalize	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 ql_talker と ql_server 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	mpiexec 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 ql_mpiexec_start コマンドより起動され常駐する。また、mpiexec プロセス終了と共に終了する。
mpiexec	MPI プロセスを生成する。本プロセスは mpiexec 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。
mcexec	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。
ql_server	高速プロセス起動対象の MPI プログラムを記録し、ql_mpiexec_{start,finalize} コマンドの指示を MPI プロセスに送る。ql_server は ql_mpiexec_start コマンドから ssh 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。
ql_talker	ql_mpiexec_{start,finalize} と ql_server との間の通信を仲介する。ql_mpiexec_{start,finalize} コマンドから ssh で ql_server 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。

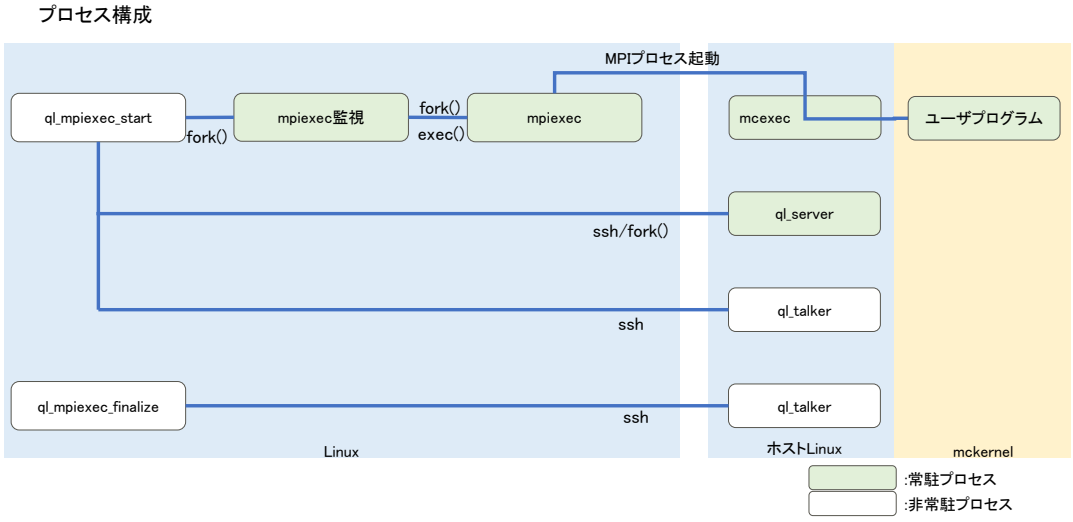


Figure 2.22: プロセス構成

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	mcexec 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 mcexec 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られなかった場合に返答として用いる

プロセス間通信の通信電文フォーマットは以下の通り。

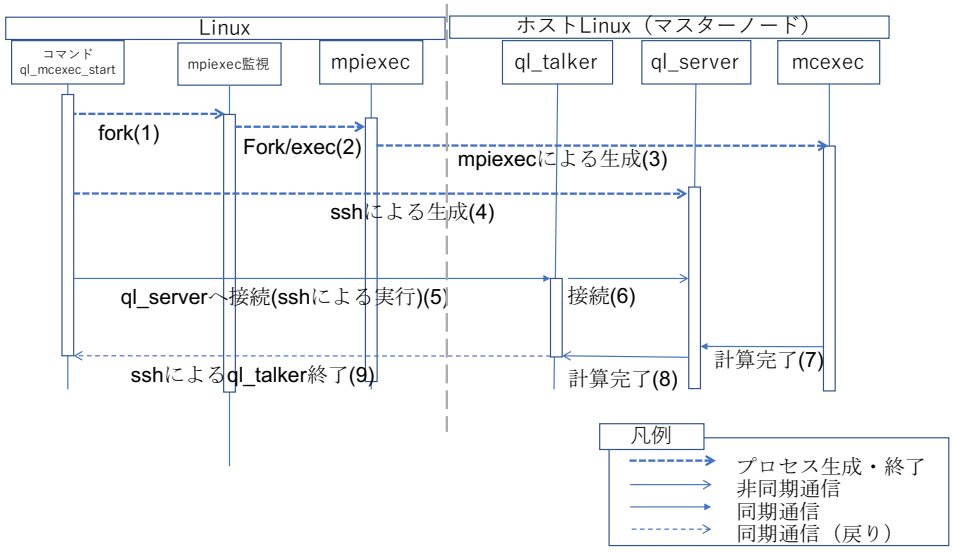
<コマンド> <データサイズ> <データ>

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

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

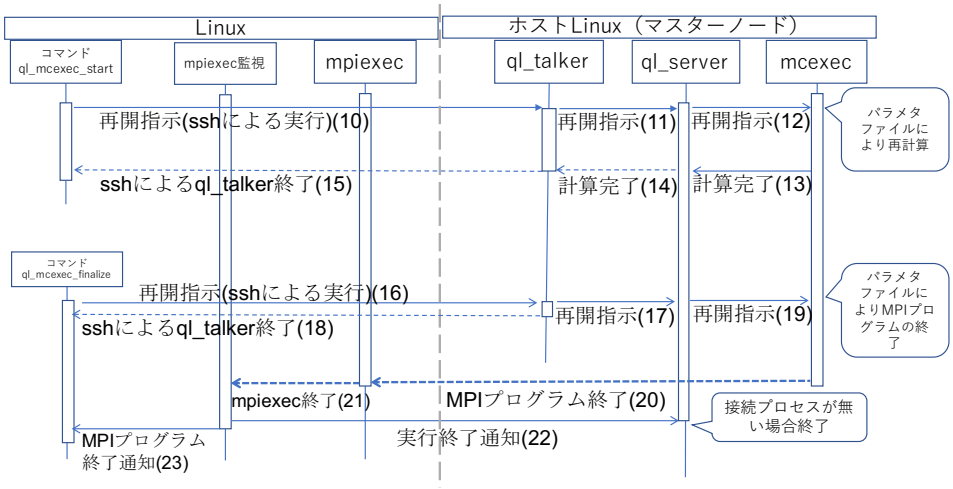


Figure 2.23: 関連コマンドと関連プロセスの動作フロー

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））

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））

03 `mpiexec` が `mcexec` を用いて MPI プロセスを `McKernel` 上に起動する。（図の（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））

05 `ql_mpiexec_start` が `ssh` でホスト Linux 上に `ql_talker` を起動する。（図の（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））

07 mcexec は MPI プログラムの一回の計算完了後 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))

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))

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 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る。(図の (9))

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 でホスト Linux 上に ql_talker を起動する。(図の (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 からの返信を待つ。(図の (11))

12 ql_server は mcexec へ R コマンドを送信する。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 (12))

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

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

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))

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 を起動する。(図の (16))

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 (17) (18))

18 ql_server は mcexec へ R コマンドを送信する。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確認し終了処理を行う。(図の (19))

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

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 プロセス終了を検知し、戻り値を取得する。(図の (21))

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、管理表が空になった場合は終了する。(図の (22))

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、戻り値を渡し、終了する。(図の (23))

23 ql_mpiexec_finalize 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、mpiexec の戻り値を自身の戻り値として終了する。

2.16.2 MPI プロセス起動指示コマンド

書式

```
ql_mpiexec_start -machinefile <hostfile_path> [<mpiopts>...]  <exe> [<args>...]
```

説明

処理ステップは以下の通り。

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 を作成する。ID を環境変数 QL_NAME に記録する。

2 ql_server との通信のためのソケットファイルのパスを環境変数 QL_SOCKET_FILE に記録する。また、ssh 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 ql_server 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。

3 mpiexec 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、mpiexec 監視プロセスを fork する。mpiexec 監視プロセスは、mpiexec を fork/exec で生成する。また、mpiexec 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。

4 再開指示のためパラメタファイルを作成する。

5 mpiexec 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。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。

6 第 1 回の計算開始時は ssh で ql_talker をマスターノード上に起動する。ql_talker は、ql_server へ接続を意味する N コマンドを送信し、各回の計算完了を意味する E コマンドを受信するまで待機する。

7 第 2 回目以降の計算開始時は、ssh で ql_talker をマスターノード上に起動する。ql_talker は、ql_server へ再開指示を意味する R コマンドを送信し、各回の計算完了を意味する E コマンド受信まで待機する。

8 ql_talker コマンドがその終了をもって ql_mpiexec_start へ計算完了を通知する。ql_mpiexec_start は mpiexec 監視プロセスと通信を行って mpiexec が終了していないことを確認する。mpiexec が終了している場合は、mpiexec の終了コードを取得する。

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。mpiexec 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。

ql_mpiexec_start が使用する環境変数は以下の通り。

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

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

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。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。

フォーマットは以下の通り。

<ヘッダ部>
<データ部>
[<データ部>...]

<ヘッダ部>のフォーマットは以下の通り。

0 COM=<mcexec への指示> <引数の数> <環境変数定義の数>

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

<データ部>のフォーマットは以下の通り。

<種別> <データ長> <データ値>

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

書式

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

説明

処理ステップは以下の通り。

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。

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。

3 終了指示のためのパラメタファイルを作成する。

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。

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。

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。

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。

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

名前	説明	作成・参照
QL_NAME	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。

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	参照
${QL_PARAM_PATH}/<MPI プログラム ID>.param	ql_mpiexec_finalize から mcexec 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)

書式

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

説明

argc, argv を用いて高速プロセス起動の初期化を行う。本関数は PMPI インタフェースにより MPI_Init() を置き換える。

処理のステップは以下の通り。

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。

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期化する。

3 PMPI_init() の戻り値自身の戻り値に設定して戻る。

戻り値

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

2.16.5 MPI 実行環境初期化関数 (fortran)

書式

```
subroutine MPI_INIT(INT ierr)
```

説明

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 インタフェースにより MPI_INIT を置き換えることで実装される。処理のステップは以下の通り。

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。

2 pmpi_init_() を呼び出し、MPI 環境を初期化する。

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初期化する。

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

戻り値

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。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 を返す。

3 MPI プログラム ID から、パラメタファイルのパスを作成する。

4 QL_SUCCESS を返す。

戻り値

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

2.16.9 計算ノードの管理サーバ

書式

```
ql_server
```

説明

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、以下の処理を行う。

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。

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。

3 select() で当該ソケットを監視する。

4 電文を読み込み、コマンドとデータを取得する。

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スを返すマップ（接続マップと呼ぶ）に ql_talker を登録する。
接続マップは struct client_fd で実装される。struct client_fd は以下のように定義される。

```
struct client_fd {
        int fd;         // 接続元プロセスのファイルディスクリプタ
        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>	ql_server へ送信するコマンド（1 文字）を指定する。
<receive_command>	ql_server 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 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 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 <send_command>と<MPI_Program_ID>より電文を作成し、ql_server へ電文を送信する。失敗した場合は終了コード-1 で終了する。

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。

5 <receive_command>を受信した場合、終了コード 0 で終了する。<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。

戻り値

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

2.16.11 swapout システムコール

書式

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

引数

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

説明

A. スワップアウト処理

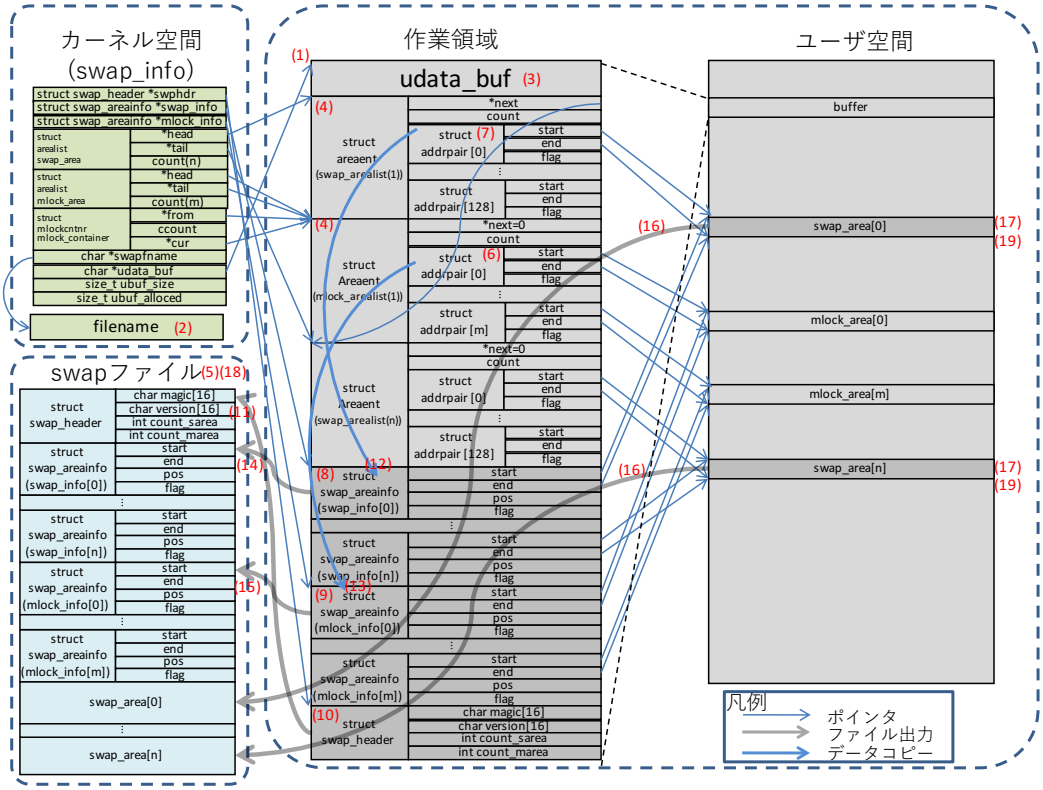


Figure 2.24: スワップアウトの処理フロー

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))

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 メンバに作業領域の swap_info 配列の先頭アドレスを記録する。 (図の (8))

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 メンバに作業領域の mlock_info 配列の先頭アドレスを記録する。 (図の (9))

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 メンバに先頭アドレスを記録する。 (図の (10))

10. 作業領域の swap_header の magic メンバに”McKernel swap”、version メンバに”0.9.0”、count_sarea メンバに swap_arealist のエントリ数、count_marea メンバに mlock_arealist のエントリ数を記録する。

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))

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 配列へコピーする。 (図の (12))

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 配列へコピーする。 (図の (13))

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 (14))

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 (15))

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 (16))

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ものを ihk_mc_pt_free_range() でアンマップする。 (図の (17))

18. スワップファイルを close() を用いてクローズする。 (図の (18))

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を mcexec に依頼することでアンマップする。 (図の (19))

B. スワップイン処理

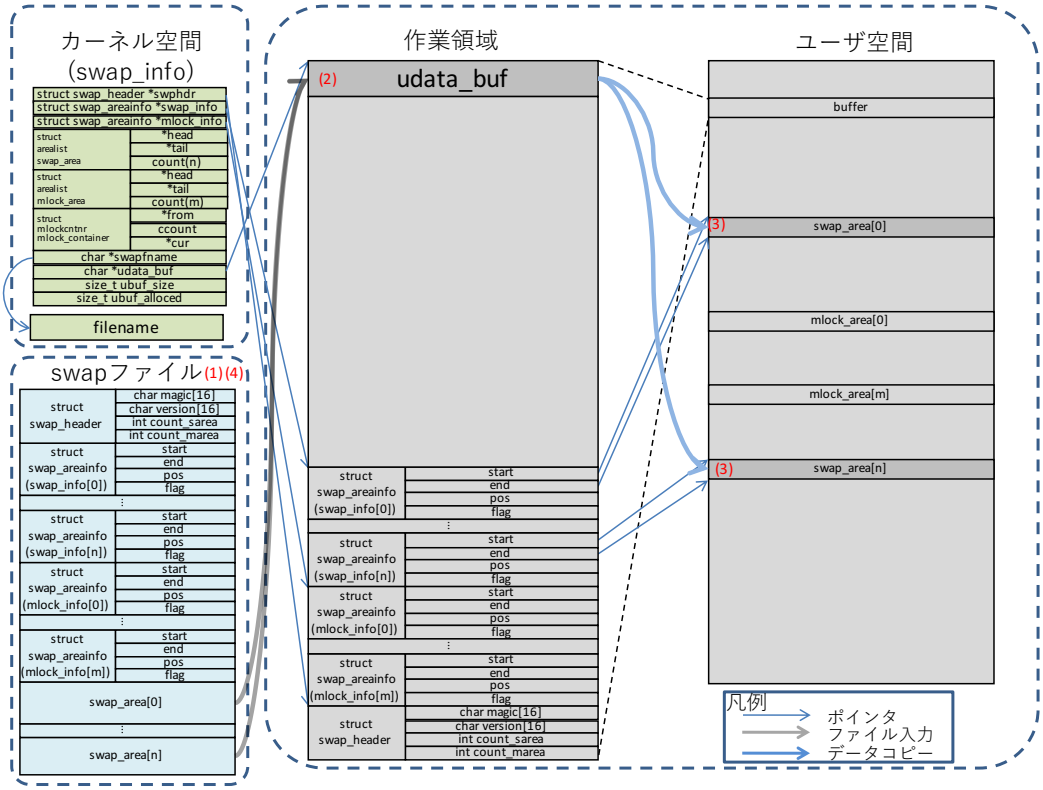


Figure 2.25: スワップインの処理フロー

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 indepent source codes are created and codes are separately stored into those two directories. That is, source codes, including header files, for some specific architecture are located in its architecture dependent directory.

The source codes, accessing some hardware registers, are hardware specific, and thus those are machine dependent. Low-level interrupt handlers, some memory management codes, context switch codes, and signaling codes are the examples. Those source codes are located in an architecture dependent directory.

Any program code and header files must not include any machine dependent codes including conditional compile macros, such as `#ifdef ARCH`. As much as possible, we define machine independent interfaces so that those interfaces are implemented for each architecture.

2.18 Formal Verification

Some of the behaviors of McKernel is verified in a formal way by embedding behaviors in code and running a verification engine. We employ an extented version of the ANSI/ISO C Specification Language, whose extensions[1] were developed at the project "Dependable Operating Systems for Embedded Systems Aiming at Practical Applications" in the research area named Core Research for Evolutional Science and Technology (CREST), sponsored by Japan Science and Technology Agency (JST).

2.18.1 Specification Language

The following are expressions defined in the formal specification language. The behavior of each function is formally specified by using those expressions that are written as C comments.

`\result` specifys return vaule.

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

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

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

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

`is_locked(`⟨`pointer variable`⟩`)` returns true if a memory block pointed by the pointer variable is the lock status, otherwise returns false.

`requires` ⟨**condition expression**⟩ The condition expression must be satisfied at the begining of the function execution.

`ensures` ⟨**condition expression**⟩ The condition expression must be satisfied at the end of the function execution.

`invariant` ⟨**condition expression**⟩ The condition expression must be satisfied during the function execution.

Here is a sample code.

```
/*@
  @ behavior valid_vector:
  @   assumes 32 <= vector <= 255;
  @   requires \valid(h);
  @   assigns handlers[vector-32];
  @   ensures \result == 0;
  @ behavior invalid_vector:
  @   assumes (vector < 32) || (255 < vector);
  @   assigns \nothing;
  @   ensures \result == -EINVAL;
  @*/
int ihk_mc_register_interrupt_handler(int vector,
                                      struct ihk_mc_interrupt_handler *h)
{
    if (vector < 32 || vector > 255) {
      return -EINVAL;
    }
    list_add_tail(&h->list, &handlers[vector - 32]);
    return 0;
}
```

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
arch_prctl	It returns the EOPNOTSUPP error when ARCH_SET_GS is passed.
brk	It extends the heap more than requestd when -h (--extend-heap-by=)<step> option of mcexec is used with the value larger than 4 KiB.
clone	It supports only the following flags. All other flags cause clone() to return error or are simply ignored. • CLONE_CHILD_CLEARTID • CLONE_CHILD_SETTID • CLONE_PARENT_SETTID • CLONE_SETTLS • CLONE_SIGHAND • CLONE_VM
getrusage	The time spent is measured in a different way than Linux for RUSAGE_THREAD. 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).
mbind	Per-memory-range policy can be set but it is not used when allocating physical pages.
set_mempolicy	• MPOL_F_RELATIVE_NODES and MPOL_INTERLEAVE flags are not supported. • MPOL_BIND works in the same way as MPOL_PREFERRED. That is, MPOL_BIND 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.
migrate_pages	It returns the ENOSYS error.
msync	Only the modified pages mapped by the calling process are written back.
setpriority, getpriority	They could set/get the priority of a random mcexec thread. This is because there's no fixed correspondence between a McKernel thread which issues the system call and a mcexec thread which handles the offload request.
set_rlimit	It sets the limit values but they are not enforced.
set_robust_list	It returns the ENOSYS error.
signalfd	It returns the EOPNOTSUPP error.
signalfd4	It returns a fd, but signal is not notified through the fd.
setfsuid, setfsgid	It cannot change the id of the calling thread. Instead, it changes that of the mcexec worker thread which takes the system-call offload request.
mmap (hugeTLBfs)	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.
Anonymous shared mapping	Mixing page sizes is not allowed. mmap creates vm_range with one page size. And munmap or mremap that needs the reduced page size changes the sizes of all the pages of the vm_range.
ihk_os_getperfevent	It could time-out when invoked from Fujitsu TCS (job-scheduler).
madvise, mbind	The behaviors of madvise and mbind are changed to do nothing and report success as a workaround for Fugaku.
mmap	It allows unlimited overcommit. Note that it corresponds to setting sysctl vm.overcommit_memory to 1.
mlockall	It is not supported and returns -EPERM.
munlockall	It is not supported and returns zero.

Chapter 3

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

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 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
hidos	IKC を有効にする。
dump_level= <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。 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。 指定がなかった場合は 24 が用いられる。
allow_oversubscribe	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 clone(), fork(), vfork() などで生成しようとすると、当該システムコールが EINVAL エラーを返す。

3.1.2 ブートスクリプト

書式

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

オプション

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。 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。 指定がなかった場合は 24 が用いられる。
-O	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。

説明

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、前記パーティションに McKernel をブートする。

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 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_PGSIZES];
    /* 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_PGSIZES];
    /* 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） */
    unsigned long memory_kmem_max_usage;
    /* カーネルのメモリ使用量最大値（バイト単位） */
    unsigned long memory_numa_stat[IHK_MAX_NUM_NUMA_NODES];
    /* NUMA ごとのユーザのメモリ使用量現在値（バイト単位） */
    unsigned long cpuacct_stat_system;
    /* システム時間（USER_HZ 単位） */
    unsigned long cpuacct_stat_user;}
    /* ユーザ時間（USER_HZ 単位） */
    unsigned long cpuacct_usage;}
    /* ユーザの CPU 時間（ナノ秒単位） */
    unsigned long cpuacct_usage_percpu[IHK_MAX_NUM_CPUS];
    /* コアごとのユーザの CPU 時間（ナノ秒単位）*/
    int num_threads;
    /* スレッド数現在値 */
    int max_num_threads;
    /* スレッド数最大値 */
};
```

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。

```
#define IHK_OS_PGSIZE_4KB 0
#define IHK_OS_PGSIZE_2MB 1
#define IHK_OS_PGSIZE_1GB 2
```

3.1.6 ダンプ解析コマンド

インターフェイスは第 1.2.1 節に記載する。

3.1.7 ダンプ形式変換コマンド

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

mcreboot.sh を用いてブート手順を説明する。
スクリプトは以下の通り。

```
#!/bin/bash

# IHK SMP-x86 example boot script.
# author: Balazs Gerofi <bgerofi@riken.jp>
#       Copyright (C) 2014  RIKEN AICS
#
# This is an example script for loading IHK, configuring a partition and
# booting McKernel on it.  Unless specific CPUs and memory are requested,
# the script reserves half of the CPU cores and 512MB of RAM from
# NUMA node 0 when IHK is loaded for the first time.
# Otherwise, it destroys the current McKernel instance and reboots it using
# the same set of resources as it used previously.
# Note that the script does not output anything unless an error occurs.

prefix="/home/takagi/project/os/install"
BINDIR="${prefix}/bin"
SBINDIR="${prefix}/sbin"
ETCDIR=/home/takagi/project/os/install/etc
KMODDIR="${prefix}/kmod"
KERNDIR="${prefix}/smp-x86/kernel"
ENABLE_MCOVERLAYFS="yes"

mem="512M@0"
cpus=""
ikc_map=""

if [ "${BASH_VERSINFO[0]}" -lt 4 ]; then
        echo "You need at least bash-4.0 to run this script." >&2
        exit 1
fi

redirect_kmsg=0
mon_interval="-1"
DUMP_LEVEL=24
facility="LOG_LOCAL6"
chown_option=`logname 2> /dev/null`

if [ "`systemctl status irqbalance_mck.service 2> /dev/null |grep -E 'Active: active'`"\
 != "" -o "`systemctl status irqbalance.service 2> /dev/null |grep -E 'Active: active'`"\
 != "" ]; then
        irqbalance_used="yes"
else
        irqbalance_used="no"
fi

```

```
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 occured.
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 [ "$enable_mcoverlay" == "yes" ]; then
99                         umount /tmp/mcos/mcos0_proc
100                fi
101                ;&
102        mcoverlayfs_loaded)
103                if [ "$enable_mcoverlay" == "yes" ]; then
104                        rmmod 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 $ind" >&2
129                                 fi
130                         done
131                 fi
132                 ;&
133         mcctrl_loaded)
134                 rmmod mcctrl 2>/dev/null || echo "warning: failed to remove mcctrl" >&2
135                 ;&
136         cpus_reserved)
137                 cpus=`${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 $mem > /dev/null; then
148                                 echo "warning: failed to release memory" >&2
149                         fi
150                 fi
151                 ;&
152         ihk_smp_loaded)
153                 rmmod ihk_smp_x86 2>/dev/null || echo "warning: failed to remove ihk_smp_x86" >&2
154                 ;&
155         ihk_loaded)
156                 rmmod ihk 2>/dev/null || echo "warning: failed to remove ihk" >&2
157                 ;&
158         irqbalance_stopped)
159                 if [ "`systemctl status irqbalance_mck.service 2> /dev/null |'\
160 'grep -E 'Active: active'`" != "" ]; then
161                         if ! systemctl stop irqbalance_mck.service 2>/dev/null; then
162                                 echo "warning: failed to stop irqbalance_mck" >&2
163                         fi
164                         if ! systemctl disable irqbalance_mck.service >/dev/null 2>/dev/null; then
165                                 echo "warning: failed to disable irqbalance_mck" >&2
166                         fi
167                         if ! etcdir=/home/takagi/project/os/install/etc perl -e \
168 '$etcdir=$ENV{'etcdir'}; @files = grep { -f } glob "$etcdir/proc/irq/*/smp_affinity";'\
169 ' foreach $file (@files) { $dest = substr($file, length($etcdir));'\
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 [ "${ENABLE_MCOVERLAYFS}" == "yes" ]; then
201         if [ "${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
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                 exit 1
230         fi
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 ! rmmod 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 contigous 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=$ihk_ikc_irq_core 2>/dev/null; then
342                 echo "error: loading ihk-smp-x86" >&2
343                 error_exit "ihk_loaded"
344         fi
345
346         # Offline-reonline RAM (special case for OFP SNC-4 mode)
347         if [ "`hostname | grep "c[0-9][0-9][0-9][0-9].ofp"`" != "" ] && [ "`cat /sys/devices/system/nod
348                 for i in  0 1 2 3; do
349                         find /sys/devices/system/node/node$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/$cpuid
554                                 fi
555                         done
556
557                         rm -f /tmp/mcos/mcos0_sys/devices/system/node/$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 $1;}}'`
573
574     sed "s/%mask%/$smp_affinity_mask/g" $ETCDIR/irqbalance_mck.in | \
575 sed "s/%banirq%/$banirq/g" > /tmp/irqbalance_mck
576         systemctl disable irqbalance_mck.service >/dev/null 2>/dev/null
577         if ! systemctl link $ETCDIR/irqbalance_mck.service >/dev/null 2>/dev/null; then
578                 echo "error: linking irqbalance_mck" >&2
579                 error_exit "mcos_sys_mounted"
580         fi
581
582     if ! systemctl start irqbalance_mck.service 2>/dev/null ; then
583                 echo "error: starting irqbalance_mck" >&2
584                 error_exit "mcos_sys_mounted"
585         fi
586 #    echo cpus=$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)

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)

3. irqbalance を停止する。(251–257行目)

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)

5. ihk.ko を insmod する。(307行目)

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行目)

   (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

   (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 オブジェクトのうち可能なものを破棄する

   (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. ihk-smp-x86.ko を insmod する。(340行目) ihk-smp-x86.ko は関数を ihk.ko に登録する。このため、ihk-smp-x86.ko は ihk.ko を insmod した後に insmod する必要がある。

8. メモリを予約する。(370行目)

9. CPU を予約する。(374行目)

10. McKernel のカーネルモジュール mcctrl.ko を insmod する。(382行目) mcctrl.ko 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 insmod は ihk.ko の insmod の後に、またブートの前に行う必要がある。

11. OS インスタンスを作成する。(406行目)

12. OS インスタンスに CPU を割り当てる。(412行目)

13. McKernel CPU の IKC メッセージ送信先の Linux CPU を設定する。(419行目)

14. OS インスタンスにメモリを割り当てる。(426行目)

15. カーネルイメージをロードする。(432行目)

16. カーネル引数をカーネルに渡す。(438行目)

17. カーネルをブートする。(444行目)

18. /proc, /sys ファイルの準備をする。また、その中で mcoverlayfs.ko を insmod する。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。

19. irqbalance を、Linux CPU のみを対象とする設定で開始する。(569–587行目)

3.3 シャットダウン手順

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

```
#!/bin/bash

# IHK SMP-x86 example McKernel unload script.
# author: Balazs Gerofi <bgerofi@riken.jp>
#       Copyright (C) 2015  RIKEN AICS
#
# This is an example script for destroying McKernel and releasing IHK resources
# Note that the script does no output anything unless an error occurs.

prefix="/home/takagi/project/os/install"
BINDIR="/home/takagi/project/os/install/bin"
SBINDIR="/home/takagi/project/os/install/sbin"
ETCDIR=/home/takagi/project/os/install/etc
KMODDIR="/home/takagi/project/os/install/kmod"
KERNDIR="/home/takagi/project/os/install/smp-x86/kernel"

mem=""
cpus=""
irqbalance_used=""

# No SMP module? Exit.
if ! grep ihk_smp_x86 /proc/modules &>/dev/null; then exit 0; fi

if [ "`systemctl status irqbalance_mck.service 2> /dev/null |grep -E 'Active: active'`" \
!= "" ]; then
        irqbalance_used="yes"
        if ! systemctl stop irqbalance_mck.service 2>/dev/null; then
                echo "warning: failed to stop irqbalance_mck" >&2
        fi
        if ! systemctl disable irqbalance_mck.service >/dev/null 2>/dev/null; then
                echo "warning: failed to disable irqbalance_mck" >&2
        fi
fi

# Destroy all LWK instances
if ls /dev/mcos* 1>/dev/null 2>&1; then
        for i in /dev/mcos*; do
                ind=`echo $i|cut -c10-`;
                # Retry when conflicting with ihkmond
                nretry=0
                until ${SBINDIR}/ihkconfig 0 destroy $ind || [ $nretry -lt 4 ]; do
                    sleep 0.25
                    nretry=$[ $nretry + 1 ]
                done
                if [ $nretry -eq 4 ]; then
                    echo "error: destroying LWK instance $ind failed" >&2
                    exit 1
                fi
        done
fi

# Query IHK-SMP resources and release them
if ! ${SBINDIR}/ihkconfig 0 query cpu > /dev/null; then
        echo "error: querying cpus" >&2
        exit 1
fi

cpus=`${SBINDIR}/ihkconfig 0 query cpu`
if [ "${cpus}" != "" ]; then
        if ! ${SBINDIR}/ihkconfig 0 release cpu $cpus > /dev/null; then
                echo "error: releasing CPUs" >&2
                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 $mem > /dev/null; then
74                  echo "error: releasing memory" >&2
75                  exit 1
76          fi
77  fi
78
79  # Remove delegator if loaded
80  if grep mcctrl /proc/modules &>/dev/null; then
81          if ! rmmod mcctrl 2>/dev/null; then
82                  echo "error: removing mcctrl" >&2
83                  exit 1
84          fi
85  fi
86
87  # Remove mcoverlay if loaded
88  if grep mcoverlay /proc/modules &>/dev/null; then
89          if [ "`cat /proc/mounts | grep /tmp/mcos/mcos0_sys`" != "" ]; \
90  then umount -l /tmp/mcos/mcos0_sys; fi
91          if [ "`cat /proc/mounts | grep /tmp/mcos/mcos0_proc`" != "" ]; \
92  then umount -l /tmp/mcos/mcos0_proc; fi
93          if [ "`cat /proc/mounts | grep /tmp/mcos/linux_proc`" != "" ]; \
94  then umount -l /tmp/mcos/linux_proc; fi
95          if [ "`cat /proc/mounts | grep /tmp/mcos`" != "" ]; then umount -l /tmp/mcos; fi
96          if [ -e /tmp/mcos ]; then rm -rf /tmp/mcos; fi
97          if ! rmmod mcoverlay 2>/dev/null; then
98                  echo "warning: failed to remove mcoverlay" >&2
99          fi
100 fi
101
102 # Remove SMP module
103 if grep ihk_smp_x86 /proc/modules &>/dev/null; then
104         if ! rmmod ihk_smp_x86 2>/dev/null; then
105                 echo "error: removing ihk_smp_x86" >&2
106                 exit 1
107         fi
108 fi
109
110 # Remove core module
111 if grep -E 'ihk\s' /proc/modules &>/dev/null; then
112         if ! rmmod ihk 2>/dev/null; then
113                 echo "error: removing ihk" >&2
114                 exit 1
115         fi
116 fi
117
118 # Stop ihkmond
119 pid=`pidof ihkmond`
120 if [ "${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 '$etcdir=$ENV{'etcdir'}; @files = grep { -f } glob "$etcdir/proc/irq/*/smp_affinity";'\
128 ' foreach $file (@files) { $dest = substr($file, length($etcdir));'\
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
```

手順は以下の通り。

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。（24–33 行目）
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）
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）
4. `mcctrl.ko` を `rmmod` する。（81 行目）
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。
6. `ihk-smp-x86.ko` を `rmmod` する。（104 行目）
7. `ihk.ko` を `rmmod` する。（112 行目）
8. `ihkmond` を停止する。（121 行目）
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. 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）

Bibliography

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

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