Vol.2015-ARC-215 No.2
Vol.2015-OS-133 No.2
2015/5/26

HPC 向けメニーコア OS を用いた バッチジョブ運用の課題検討

平井浩一[†1]　小田和友仁[†1]　岡本高幸[†1]　二宮温[†1]　住元真司[†1]
高木将通[†2]　Balazs Gerofi[†2]　山口訓央[†2]　小倉崇浩[†2]　亀山豊久[†2]
堀敦史[†2]　石川裕[†2]

将来の HPC 向けの OS としては，メニーコアへの最適化が必須となってきており，それを実現するための OS として McKernel を選択し，計算センターにおけるバッチジョブ運用への適応を進めている．本論文では，将来のスーパーコンピュータ上で，McKernel に適応したバッチジョブ運用を実現する場合の課題を述べ，現状の検討状況について述べる．

Issues of Batch Job Execution Environment using Many-core OS for HPC

KOUICHI HIRAI[†1]　TOMOHITO OTAWA[†1]　TAKAYUKI OKAMOTO[†1]
ATSUSHI NINOMIYA[†1] SHINJI SUMIMOTO[†1]
MASAMICHI TAKAGI[†2]　BALAZS GEROFI[†2]　NORIO YAMAGUCHI[†2]
TAKAHIRO OGURA[†2]　TOYOHISA KAMEYAMA[†2]　ATSUSHI HORI[†2]
YUTAKA ISHIKAWA[†2]

1. はじめに

近年，HPC 向け OS としては，Linux が主流となっている．2014 年 11 月の TOP500 では，ランクインしたシステムのうち 97%が Linux を採用している[1]．今後も Linux が多く利用されることが想定されるため，ユーザに対して Linux 環境での可搬性を保つことが非常に重要となる．

また，HPC 向けのプロセッサとしては，省電力化が求められており，周波数の向上による性能向上ではなく，コア数増加により性能を高めるメニーコアプロセッサが台頭し始めている．例えば，スーパーコンピュータ「京」[2]の後継である FUJITSU Supercomputer PRIMEHPC FX100 [3]に搭載された SPARC64 XIfx では 34 コア，2014 年 11 月の TOP500 で No.1 となった Tianhe-2 でも採用された Xeon Phi [4]では 60 コア以上もある．文部科学省が進めている FLAGSHIP2020 プロジェクトで検討されているスーパーコンピュータ「京」後継機であるポスト「京」においても，メニーコア型スーパーコンピュータの検討が進んでいる．メニーコア化に伴い，プロセッサコア間の結合方式もクラスタ接続（SPARC64 XIfx）やリング接続（Xeon Phi）が採用され，メモリアクセス方式もメモリ位置によりアクセス性能が異なる NUMA 型が採用されている．

このようなメニーコアプロセッサ上で OS を動作させる場合，以下のような課題がある．

- 共有資源の排他制御によるオーバーヘッドの低減
- アプリケーション資源の保障
- プロセッサ構成，NUMA を意識した資源割り付け
- システムコールや割り込み処理によるキャッシュ汚れの低減
- デーモンや割り込み処理によるアプリケーション性能影響の低減

これらの課題は， Linux においても必ずしもすべてが解決されていないことから，Linux の改造による解決は大きなコストを伴う． Linux の基盤機能からの大幅な変更が必要であり，汎用的な OS を目指すオープンソースコミュニティとしては，それらの修正をメインストリームへ取り込むことは難しい．さらに，Linux に独自の修正を入れた独自 Linux を作ることもバージョン追従やメンテナンスのためのコストがかかるため採用することが難しい．

そのため，HPC 向けに特化し，極力機能を削ぎ落とすような軽量カーネルにより，これら課題を解決する研究が進められてきた．ハード専用に特化した軽量カーネルの研究としては，Blue Gene/L の CNK[5]や，Red Storm（Cray XT-3）の Catamount[6]などがあげられる．しかし，このようなアプローチでは Linux と比べると機能制限が多く，実行できるアプリケーションが限定されるという問題があった[7]．

これに対し，Linux 互換によるアプリケーションの可搬性を保ちつつ，上記課題を解決するための軽量カーネルによるアプローチを両立するアプリケーション実行の枠組み

†1 富士通株式会社
FUJITSU LIMITED
†2 理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science

ⓒ2015 Information Processing Society of Japan

Vol.2015-ARC-215 No.2
Vol.2015-OS-133 No.2
2015/5/26

として，理化学研究所を中心として研究が進められているMcKernel [8][9]がある．McKernel は，プロセス・スレッド管理，メモリ管理，シグナル処理などカーネルが提供しなければならない機能及びアプリケーション特性ごとに特化した機能のみを実装し，それ以外の OS 処理は Linux に処理を委譲することにより，軽量カーネルを実現しながら Linux と同等機能を提供している．

本論文では，McKernel をスーパーコンピュータ向けの OS としてセンター運用することを目指し，バッチジョブ運用する上での課題と解決手段について議論する．2 章で McKernel の概要を，3 章でバッチジョブ運用の概要を紹介する．4 章で将来のスーパーコンピュータ上でのバッチジョブ運用の要件として，メニーコア OS をバッチジョブ運用に組み込む必要があることを述べる．5 章でその実現上の課題として，McKernel 実行に必要な資源割り当てと管理，ユーザ利便性の解決，アプリケーションの実行シナリオの検討が特に重要であることを述べ，課題に対する対応方法の検討結果を述べる．

2. McKernel の概要

McKernel は，同じ計算機内の一部のコアで Linux を動作させたまま，残りのコアで McKernel を起動し，OS が提供すべき機能の一部を Linux に委譲することによって，軽量カーネルを実現しながら Linux と同等の機能を提供している．Linux と McKernel とのカーネル間インターフェースとして IHK（Interface for Heterogeneous Kernels) [10]が実装されている．アプリケーションは McKernel 上で稼動し，各種デーモンやカーネルスレッド，デバイス割り込み，タイマー割り込みなど，OS ノイズ源となる可能性のある処理は， Linux 上で処理される．また，McKernel 上で実行されるプロセスから要求されたシステムコールの処理も大部分を Linux 側に委譲することで，McKernel 自体は軽量な作りのままで Linux と同等の機能を提供している．

本論文では，McKernel をバッチジョブスケジューラと連携させる場合に考慮が必要となる 3 つの処理について以下.2.1 節から 2.3 節で述べる．

2.1 McKernel に割り当てる CPU 及びメモリの分割

McKernel は，Linux の制御から独立した CPU とメモリを使用して動作する．これは Linux との排他制御などのオーバーヘッドを排除し，独立して CPU やメモリの割り当てを制御可能とするためである．初期の実装では Linux の初期化処理部分にパッチを当てることで，Linux と独立した CPU 及びメモリを確保していた．しかし，現在では起動した状態の Linux 上から，カーネルモジュールによって動的に CPU とメモリを確保することを可能としている[8].

また，Linux から一旦分割確保した CPU 及びメモリを，McKernel の終了後に Linux に返却することも可能である．

2.2 McKernel の起動・終了操作

前述の CPU とメモリの確保を含めた McKernel の起動及び終了の流れを説明する．全体としては，以下の(1)から(7)のような流れとなる．

(1) CPU 及びメモリの確保, McKernel 用資源領域の作成
(2) McKernel バイナリのロード
(3) McKernel の起動
(4) McKernel 上へのプロセス起動
(5) McKernel 上でのプロセス終了
(6) McKernel の終了
(7) McKernel 用資源領域の解放，Linux への資源の返却

(1)及び(7)は 2.1 節で述べた資源の確保及び返却の処理である．メモリの確保については McKernel に割り当てるメモリの領域サイズを指定する．CPU の確保については数量ではなく確保する CPU の論理 ID を指定する．また，複数の McKernel を並列に動作させるため，Linux から確保した資源をさらに複数の領域に分割して，それぞれ異なった McKernel に割り当てることも可能である．

(2)では使用する McKernel バイナリファイルのパスを指定する．指定されたファイルが読み出され，(1)で確保したメモリ領域に McKernel の実行命令列及びデータが書き込まれる．

(3)で McKernel の実行が開始される．命令実行は資源領域の先頭の CPU で開始され，それ以外の CPU は McKernel 自体で命令実行を開始させる．McKernel の起動処理が終了すると各 CPU でアイドル状態を示すプロセスが動作する．

(4)及び(5)は McKernel 上で動作させるプロセスを管理する操作である．プロセスの起動には専用の起動プログラム（mcexec）を使用する．以降，McKernel 上のプロセスに対する制御は，この起動プログラムのプロセスに対する制御を通して実施される．このプロセスについては 2.3 節で詳細を述べる．プロセスの終了処理は，特に外部から指示せずとも，プログラム自体の終了によって自動的に処理が行われる．

(6)では Linux 上から管理者権限のコマンドによって McKernel の終了処理を開始する．

(1),(2),(3),(6),(7)の処理はすべて Linux の管理者権限（root 権限）でのみ実行可能である．

2.3 システムコール委譲処理と Ghost Process

McKernel はアプリケーションに対し，Linux API 互換のシステムコール機能を提供する[11]．　これは 300 以上存在する Linux システムコールの大部分を，Linux に委譲することで実現している．McKernel 上でプロセスを起動すると同時に，Linux 上でもそのプロセスのシステムコールを代行するプロセス（Ghost Process）を起動する．2.2 節で述べた起動プログラム（mcexec）がこのプロセスとなる．

McKernel 上のプロセスからシステムコールが発行され

ⓒ2015 Information Processing Society of Japan

ると，その処理は McKernel 内部で実行されるか，または Linux の Ghost Process に委譲される．McKernel の管理する資源に対する処理（brk, mmap など）や高速な応答が必要とされる処理（futex など）は McKernel 内部で直接処理される．それらを除いた大部分のシステムコールは IHK を通して，Linux 上の Ghost Process に委譲される．

さらに，Ghost Process に外部から指示をする場合も特別なインターフェースはなく，通常の Linux のプロセスと同様にシグナルによって制御を行う．

システムコール委譲処理やシグナルの取り扱いについての詳細は論文[11]を参照のこと．

3. バッチジョブ運用について

HPC の大規模クラスタの運用においては，多数のユーザの要求を効率よく実行するため，一般にバッチジョブ方式によるシステム運用が採用されている．本章ではバッチジョブ運用の目的と，その典型的な動作について述べる．

3.1 バッチジョブ運用の目的

バッチジョブ運用とは，ジョブスケジューラがクラスタを一元管理し，ユーザのジョブ実行要求を保持して適切な実行ポリシーにもとづき，順次実行する方式である．ユーザが投入するバッチジョブを管理し，運用するシステムをバッチジョブシステムと呼ぶ．バッチジョブシステムは一般的に，以下の 3 つの要求を満たす機能を提供する．

(1) 使いやすいジョブ実行インターフェース

ジョブ実行ユーザが複雑なクラスタ構成を意識することなくジョブを実行し，ジョブの種別や並列度をオプションなどで指定可能とすることで，統一的なインターフェースでユーザが必要な実行環境を提供する．また，対話的に実行するインターフェースや，ジョブの性能情報を取得するインターフェースを提供することにより，大規模ジョブの性能チューニングなどを容易にする．さらに，ユーザは登録したジョブの実行状態や処理完了予定時刻などのジョブに関する管理情報を把握するインターフェースを提供する．

(2) ジョブ実行要求に対する適切なスケジューリング

多数のユーザのジョブ実行要求を即座に受け付け，適切なタイミングでクラスタ構成内の計算機の CPU やメモリなどの計算機資源を割り当て，ジョブを実行させるようなスケジューリングを行う．

(3) 計算機資源の効率的利用

クラスタ全体の計算機資源の利用状況を管理し，ジョブの要求資源や優先度などの属性を加味して適切に計算機資源を割り当てることで，クラスタ全体の資源の利用効率を最適化する．

3.2 バッチジョブ実行の流れ

図 1 に具体的なバッチジョブ運用におけるジョブ実行処理の流れの例を示す．

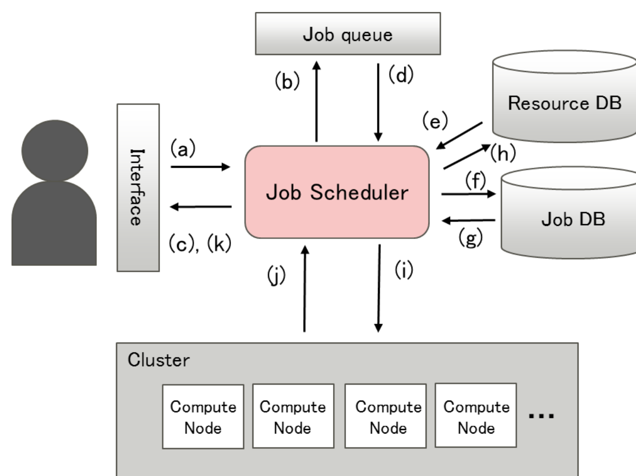

図 1 バッチジョブ実行の流れの例

(1) ジョブ実行要求受け付け処理

(a) ユーザによるバッチジョブ実行要求を受け付け
(b) ジョブ実行要求を Job queue に登録し，保持
(c) ユーザへジョブ実行要求の受け付け完了を通知
　ジョブ実行要求を Job queue に登録した時点で，ユーザインターフェースを即座に復帰させることにより，大量のジョブ実行要求を短時間に受け付けることを可能とする．

(2) ジョブスケジューリング処理

(d) Job queue からジョブ実行要求を取り出し
(e) 計算機資源を管理するための Resource DB (Database) から計算機資源の利用状況を取得
(f) (e)の情報をもとに，ジョブ実行予定時刻と割り当て予定資源を決定し，スケジューリング情報を Job DB (Database)に登録
　Resource DB に保持した計算機資源の利用状況を参照し，計算機資源の利用効率を最適化するようにスケジューリングを実施する．

(3) ジョブ実行処理

(g) 実行されているバッチジョブの状態を管理するための Job DB からジョブスケジューリング情報を取得
(h) Resource DB を更新し，ジョブ実行に必要な計算機資源を割り当て
(i) ジョブ実行を開始
(j) ジョブの実行結果を回収し，計算機資源を解放
(k) ジョブの実行結果をユーザに通知

ジョブスケジューラは新たなジョブが投入された時，あるいは実行していたジョブが終了した時に，スケジュール情報を参照して必要な計算機資源が割り当て可能ならば，計算機資源を割り当てた計算ノード上でジョブを実行する．計算ノード上では，ノード内ジョブマネージャがアプリケーション用の CPU やメモリなどの資源をジョブ単位で割り当て，その資源上で動作するジョブプロセスと結びつけて占有利用可能としている．

ⓒ2015 Information Processing Society of Japan

図 2にすべての計算機資源を利用して Linux を動作させている場合のジョブ管理方式の概要を示す．ジョブは，ジョブスクリプトと呼ばれるシェルスクリプトで記述された手順に従って実行される．ジョブスクリプトでは通常の Linux コマンドが呼び出されるとともに並列アプリケーションを動かすためのコマンドが呼び出される．

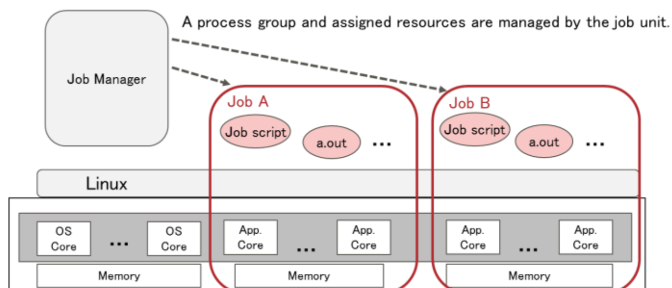

図 2　計算ノード上でのジョブ管理方式（Linux）

ジョブの実行が終了した後，ジョブスケジューラは実行結果を回収して計算機資源を解放する．

4. 将来のスーパーコンピュータ上でのジョブ運用の要件

本章では，将来のスーパーコンピュータ上でセンター運用した場合おける実行環境のジョブ運用の要件について述べる．

まず，将来のスーパーコンピュータのシステムと運用の要件について述べる．将来のスーパーコンピュータは，第 1 章で述べたように，NUMA 構成を取るメニーコアプロセッサを利用したシステムを想定している．また，アプリケーションの実行環境としては，Linux とメニーコア OS である McKernel が混在した環境となり，これらを意識しないシームレスな実行環境を提供すべきである．ユーザは McKernel を特別意識することなく Linux で動作するアプリケーションを開発し，実行の際にアプリケーションの特性に応じて，Linux か複数の McKernel の中から実行環境を選択する．また，このスーパーコンピュータは多数のユーザが利用することから，バッチジョブ運用が主体となり，ユーザの利用形態に合わせた実行環境が提供される．

このような実行環境を実現する要件を以下に述べる．

① 実行バイナリの互換性確保
Linux 上で実行するアプリケーションバイナリは，McKernel 上でも実行可能とする．

② アプリケーション特性に合わせた McKernel を実行可能
実行環境の選択において，アプリケーション特性に合わせた McKernel バイナリの配置，差し替えを可能とする．

③McKernel を意識しないジョブ実行・運用が可能
アプリケーションの性能評価をより容易に比較評価できるよう，実行環境をジョブ実行時に選択可能とする．実行環境として，Linux 環境と複数のメニーコア OS 環境の中から選択可能とする．また，Linux と McKernel は意識されることなく同等のジョブ実行と運用が可能となる機能を提供する．

5. McKernel のバッチジョブ運用に向けた課題と検討

本章では，4 章に述べた要件を整理した後，メニーコア OS として McKernel を採用した場合のバッチジョブ運用に向けた課題と検討結果について述べる．

5.1 要件整理

(1) 実行バイナリの互換性確保

McKernel は Linux と ABI の互換性があるため，アプリケーションバイナリの互換性が保たれており，本要件を満たしている．

(2) アプリケーション特性に合わせた McKernel を実行可能

この要件をバッチジョブ運用に適用する場合，アプリケーション特性に合わせた McKernel バイナリをどのように実行環境に展開するのかが検討課題となる．加えて，シームレスな実行環境の実現のためには，McKernel の起動を Linux 上でのジョブ実行の流れの中で，自動的に起動するなど，ユーザに McKernel の起動自体を意識させないことが求められる．

(3) McKernel を意識しないジョブ実行・運用が可能

この要件は，次に述べる観点から検討する必要がある．

① McKernel 向けの計算機の資源割り当てと管理
McKernel の実行に必要な計算機資源は McKernel と Ghost Process 向けの CPU とメモリであり，Linux の資源から割り当てられる．NUMA 構成などシステムのアーキテクチャにあわせた資源の割り当てと管理が必要となる．

② Linux 上で実行されるジョブとの統一性
Linux 上のジョブと McKernel 上のジョブを意識しないシームレスな実行環境を実現するためには，McKernel 並びに McKernel 上で実行されるジョブは次の条件を満たす必要がある．

a) Linux ジョブと同等のジョブタイプを実行可能であること（バッチジョブ，会話型ジョブなど）
b) 複数の McKernel バイナリから選択できること
c) Linux のアプリケーション実行と同一な方法で実行が可能であること
d) McKernel 並びに McKernel 上のジョブプロセスについて，状態管理，異常状態の検出，ジョブの統計情報の収集ができること

McKernel のバッチジョブ運用に向けては以上の述べた課題があるが，本論文では，次の 3 つの課題についてさらに詳しく述べる．

1. McKernel 実行に必要な資源割り当てと管理
2. McKernel バイナリの選択
3. McKernel におけるアプリケーション実行シナリオの検討

5.2 McKernel 実行に必要な資源割り当てと管理

本節では, McKernel を実行するために必要な資源は CPU 資源，メモリ資源，ネットワークを含む I/O 資源，そして，システムコール委譲処理が動作するための資源が必要である．本稿では I/O 資源以外のそれぞれの資源について述べる．

5.2.1 McKernel への CPU 資源割り当て

(1) McKernel 実行向け CPU 割り当て

McKernel には Linux が動作する CPU とは異なる独立した CPU を割り当てる．この CPU の割り当てにおいて，システム運用を止めずに，動的に McKernel 運用に切り替える実現手段が求められる. 2.1 節で記載したように，現状の McKernel で，動的に CPU を割り当てる手段が実現されており，この点での課題はないと考えている．

(2) McKernel 実行向け CPU の状態管理

Linux からは CPU が切り離されてしまってしまうため，Linux から McKernel の CPU で動作しているプロセスの状態が直接見えない状態となることから，McKernel 上の動作をシステムとしてどのように扱い，管理するのかが課題となる．

この課題に対し McKernel は，専用の CPU とメモリを割り当てるため，それらの資源をすべてユーザが利用していると見なし，McKernel 上の動作は Ghost Process である mcexec プロセスを通して確認や管理をすることができる．そのため，動作中は mcexec プロセスを通して，終了時は McKernel と新たなインターフェースを作成して，動作状況や結果を引き渡す必要がある．

例えば，mcexec プロセスのプロセス情報（/proc インターフェースから得られる情報)などから McKernel 上のプロセスの状態が取得できたり，McKernel の異常終了なども mcexec プロセスの異常終了として感知できたりするような仕組みが必要となってくる．また，McKernel が動作することによって利用した資源の統計情報を，McKernel の終了時に Linux 側のメモリに書き込んで情報を引き渡すようなインターフェースも必要となるが，さらに具体的な検討が必要である．

5.2.2 McKernel へのメモリ資源割り当て

(1) McKernel ごとの効果的な割り当て手法

McKernel が利用するメモリは物理連続でなくても割り当てることが可能であるが，物理連続な領域を確保した方が性能及び，割り当てたメモリを活用する上で有効であり，いかにメモリ資源を効果的に管理し，割り当てるのかが課題となる．

そのため，できるだけ物理連続な領域を確保するためには，Linux の起動処理の中でバッチジョブシステムが物理連続領域を先に獲得してしまうことで，実現することも可能であると考えている．これにより，割り当て時には必ず，物理連続な領域を割り当てることができる．

(2) NUMA 構成に適した割り当てポリシー

McKernel に対してメモリを割り当てる際，すべての NUMA ノードに対して均等に割り当てる方法と McKernel が利用する CPU に近い NUMA ノードのメモリからできるだけ多く取る方法がある．複数の McKernel を起動する際には，McKernel ごとに別々なメモリ割り当てポリシーを選択することも可能となるが，図 3 のように，異なる割り当てポリシー（Job A は近くに割り当て，Job B は均等に割り当て）を持つ複数のジョブの割り当てができない場合があるため，すべての組み合わせが運用上，成立できるのかが課題となる．

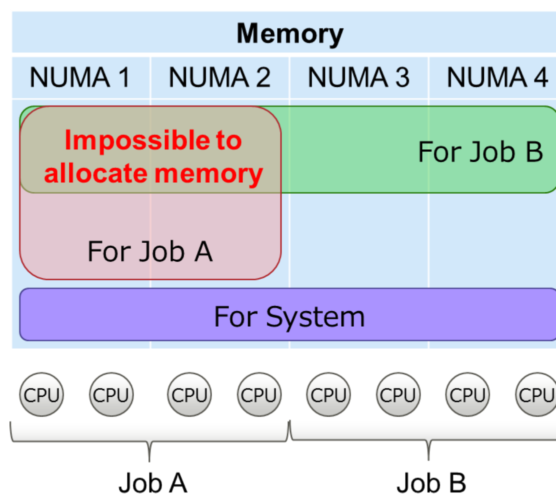

図 3 メモリ割り当て不可なケース

この課題に対し，(1)の施策による起動時のメモリ獲得の際に，NUMA 構成を意識した領域管理も行っておくことで，NUMA を意識したメモリ割り当てが可能となる．ただし，複数の McKernel を実行する場合には，各 McKernel が要求する NUMA ノードを意識したメモリ割り当てポリシーを混在させないようにする必要がある．

(3) メモリ資源の断片化

McKernel の起動と停止を繰り返したり，終了時刻の異なる複数のバッチジョブが 1 台の計算機で実行されたりすると，メモリは断片化されていくために，断片化を防ぎながら，効果的な物理連続領域の確保と NUMA 構成を意識した割り当てが課題である．

McKernel の起動・停止によるメモリの断片化は(1)の対策で防ぐことができる．終了時刻の異なる複数のバッチジョブ投入によるメモリは断片化に対する解決手段としては，以下のような対応の検討が必要である．

① 1 台の計算ノードに複数のバッチジョブを実行させない

この方法では，必ずしも 1 つのジョブが CPU やメモリを使わない可能性があるため，計算ノードの利用効率が悪化してしまう．しかし，バッチジョブ終了時には，確実にすべてのメモリが返却されるため，メモリ断片化は確実に防ぐことができ，同一ジョブ内の同質なプログラムが動作することから，プログラムの性能向上につながる可能性がある．

②メモリ割り当てポリシーが同一のバッチジョブだけを実行させる

この方法であれば，余った CPU に別のバッチジョブを実行可能であるため，計算ノードの利用効率は①よりは高い．さらに，メモリポリシーの違いによる一部の NUMA ノードでのメモリの奪い合いは発生しない．しかし，複数のバッチジョブが必ずしも同時に終わらず，メモリの断片化が発生する可能性があるため，メモリの断片化が必ずしも防げない問題が残る．具体的には，図 4 のように 2 CPU を利用する 4 つのジョブが均等割り当てのポリシーで実行されている際を例にする．ここで，Job A と Job C が終了し，4 CPU を利用する Job E が均等割り当てのポリシーで実行されると図 5 のように断片化されたメモリが割り当てられることになる．

実際のバッチジョブ運用としては，上記①と②のどちらかを選択できるような実装が有効である．そのようにすることで，運用管理者は運用ポリシーに従って，ユーザ視点の運用である①，もしくは，運用管理者視点の運用である②を選択することになる．②のポリシーにおけるメモリの断片化を解消する方法としては，メモリの断片化解消機能（デフラグ機能）などを利用することも考えられるが，アプリケーションの停止と物理的なメモリ内容の移動などが伴い，かつ，時間もかかることから，メモリの断片化が顕著に進行しない限り，現実的な解ではないと考えている．

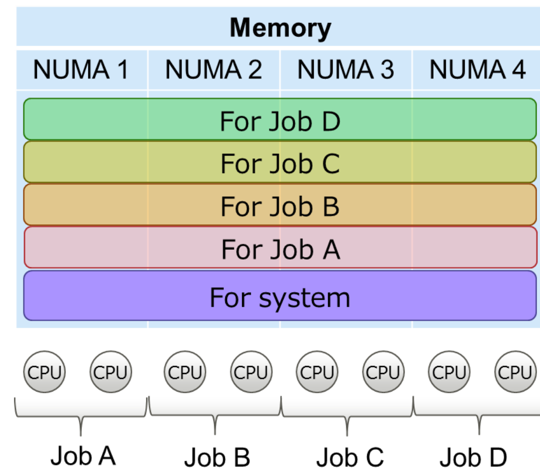

図 4　4 ジョブを均等割り当て

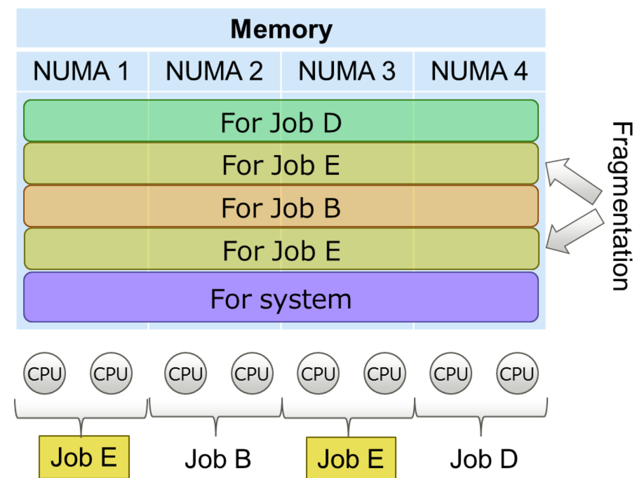

図 5　メモリ断片化状態での割り当て

5.2.3 システムコール委譲処理のための資源確保

McKernel では Ghost Process と呼ばれる McKernel から委譲されたシステムコール処理を実行するプログラム（mcexec)が Linux 上で動作している．この mcexec プロセスは，McKernel 上のプロセス 1 つに対して 1 つ実行されるため，McKernel 上で実行されるプロセスが増加すると，それに比例したプロセス資源が必要になる．

スーパーコンピュータでのジョブ運用においては，mcexec プロセスとジョブ運用プロセス向けの処理を限られた最小限の CPU 資源で実行する必要があり，これをいかに効率よく資源配分するかが課題である．なぜなら，ノード内のほとんどの CPU やメモリはアプリケーション向けに割り当てられるため，残りの限られた CPU 資源とメモリ資源で実行する必要があるからである．このような環境では，メモリ不足や特定のプロセスに CPU 資源が占有されるなどにより，システムコールの委譲処理が滞る，あるいは，ジョブ運用ソフトへの影響によりシステム処理が滞るなど影響があるため，双方への処理の影響を最小限に抑える仕

組みが必要である．

システムコール処理は Linux 上で動作するため，CPU やメモリは Linux で動作する他の処理と共有して利用する必要がある. CPU については, Linux 側には最小限の数の CPU しか残さないと考えられるため，委譲処理のために専用 CPU を割り当てることは不可能である．メモリについては, メモリ資源をいかに確保するかという点と，確保したメモリをどのように管理し，制限するのかという 2 つの論点がある．前者については，I/O を含めた様々な処理が動作することを考えると，ある程度，メモリ量を確保しないと，I/O 処理の遅延によるバッチジョブの処理遅延が発生する可能性がある．後者については，個々のバッチジョブ単位でメモリを確保しないと，バッチジョブ同士でメモリを奪い合い，他のバッチジョブ処理の委譲処理の影響による遅延が発生する可能性がある．

以下に 2 つの論点に分けて検討結果を示す．

(1) メモリ資源の確保

McKernel 上でアプリケーションを実行するため，McKernel 用に多くのメモリを割り当てていることから, 通常は Linux が利用可能なメモリ量は限られる. そのため，Linux 上でシステムコール委譲処理を行う mcexec プロセスのために，それほど多くのメモリを割くことはできない．mcexec プロセスは委譲されたシステムコール処理をしているのみで，多くのメモリ量を必要としない．しかし，委譲されたシステムコール処理によって，Linux カーネル内で確保されるバッファ等のメモリ量が問題になる可能性がある．また，5.4 節で述べるように, ジョブスクリプトを Linux 上のプロセスで実行させる場合には，ジョブスクリプトから実行されるコマンド用のメモリ量も問題になる．今後，McKernel を実際にバッチジョブ運用に適応しながら，メモリ割り当てに対する最適解を検討していく．

(2) 確保したメモリの管理と制限

(1)の手法で確保したメモリを管理し，制限する方法として, すでに Linux が提供している資源管理機能である cgroups[12]機能を利用することで実現可能であると考えている．

cgroups 機能は，Docker[13]などのコンテナ技術の基盤に使われており，主にメモリ資源の割り当てや制限に利用される機能である．具体的には，バッチジョブシステムが，システムコール委譲処理のために必要なメモリ量を割り当て，割り当てられたメモリ量で cgroups の枠組みを作成し，その枠組みの中で Ghost Process である mcexec プロセスを実行する．これにより，システムコール委譲処理に必要なメモリの確保と制限を実現することが可能である．

5.3 McKernel バイナリの選択

5.1 節でも必要性を述べたが，ユーザの使い勝手の向上とシステムの運用の安定化を両立させる McKernel バイナリの選択機能の検討が課題である

ユーザが改変した McKernel バイナリを利用するためには, McKernel バイナリをアプリケーションバイナリと同様に，ユーザが自由に選択する仕組みがユーザ利便性の面では最善である．しかし，実行時に自由に選択するとセキュリティ上の問題が発生する可能性がある．このためにも，すべてのユーザに McKernel バイナリの登録を許可させることは避けなければならない．これを回避するためには，システム管理者が McKernel バイナリを管理し，安全性を確認したものだけを登録し，ユーザが望む McKernel バイナリを選択する手法が望ましい．

以上に述べた機能を実現する際，バッチジョブシステムとしては，McKernel に特化した実装をするのではなく，汎用的な実行フレームワークとして，以下のような機能をシステムソフトウェアで準備することで, McKernel に依存しない様々な運用形態に対応できると考えている．

①ジョブ実行前の任意コマンド実行機能（各ノード上）
②ジョブ実行後の任意コマンド実行機能（各ノード上）
③クラスタシステムへのファイル配布機能
④プロセス生成方法の切り替え（MPI 実行向け）

5.4 アプリケーション実行シナリオの検討

Linux における McKernel の実行環境については，原理的に 3 つのアプリケーション実行シナリオを考えることができる．図 6 では，Job A,B,C の 3 つの実行シナリオパターンを示す. Job A では，ジョブスクリプトを Linux カーネル上で動作させ，ジョブスクリプトから並列プロセスを起動するためのコマンドである mpiexec コマンドにより，McKernel 上で並列アプリケーションを動作させる．この際, Linux が使用している資源の一部をジョブスクリプトの実行に割り当てている．Job B では，ジョブスクリプトも並列アプリケーションもすべて, McKernel 上で動作させている．Job C では，並列アプリケーションも Linux 上で動作させ，すべての資源を Linux に割り当てている．Job A，B，C の実行モードをそれぞれ，Linux+McKernel モード，McKernel モード，Linux モードと呼ぶことにする．

Linux+McKernel モードでは，ジョブスクリプトを Linux 上で実行することにより，複数のアプリケーションを同時に実行させるなどのプロセス制御もすることが可能である. また, ジョブスクリプトは Linux と共通な少ない CPU の範囲（OS コア）で実行されているため，mpiexec コマンドを使わずに実行されるプログラムは, 少ない CPU 上での実行となる．

McKernel モードでは，mpiexec コマンドを使わずに実行されるプログラムもジョブに割り当てられたすべての CPU を利用できる．ただし，複数アプリケーションの同時実行などのプロセス制御を行うためには，タイマー割り込みおよび，プロセスケジューラを持つ McKernel を選択す

る必要がある．これにより，少なからず OS ノイズが発生するため，OS ノイズによりアプリケーション実行に影響が出てくる．

Linux モードでは，すべての資源を利用して Linux を動作させた時と，同じ実行環境となる．この場合，McKernel が提供するメニーコア向け最適化が利用できない．しかし，現状の McKernel 実装では，fork/exec など一部のシステムコールの実行時間が Linux 上での実行時間よりも大幅に大きくなる．そのため，頻繁にシステムコールを発行するアプリケーションでは，McKernel が提供する最適化機能の恩恵よりも，Linux モードによる恩恵の方が大きくなることが想定される．今後，McKernel における fork/exec などのシステムコール実行時間の最適化を行うとともに，McKernel モードと Linux モードの得失を評価していく必要がある．

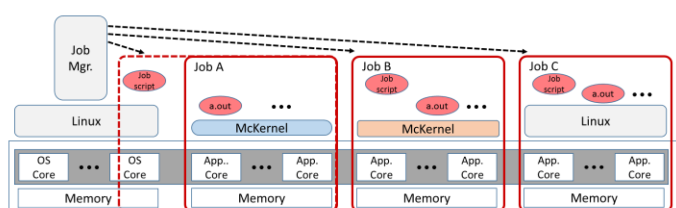

図 6　アプリケーション実行シナリオ

6. まとめ

以上のように McKernel をバッチジョブ運用に組み込むためには，いくつかの課題があり，それに対する施策に取り組む必要があることが分かった．しかし，これらの施策に取り組むことで，McKernel を利用したバッチジョブ運用は，Linux だけを用いた運用と同等な安定性を実現でき，両者を共存して運用することが可能であることを示している．

1 章で記述したように，今後，さらにメニーコア化が進むことが想定されており，メニーコアを意識した OS の需要は高まってくると考えている．今後，このようなメニーコア OS で実現した機能や考え方がさらに進んでいけば，実現した機能がメニーコア CPU で動作するための一般的な OS 実装となる可能性もあり，それらの機能が Linux などの汎用 OS へも徐々に取り込まれていくのではないかと考えている．

同時に，バッチジョブシステムについても，本論文のように，McKernel のようなメニーコア OS をサポートしていくことを検討していく必要がある．さらに，将来的には，メニーコア OS を VM（Virtual Machine）や Docker のような仮想化やコンテナ技術と同様な仮想的な OS の実行形態の 1 つと捉え，汎用的な作りで仮想環境のサポートの幅を広げていくべきであると考えている．それにより，様々な実行環境に対応することにつながり，多くのユーザ要望に応えることにつながっていくと考えている．

スーパーコンピュータ「京」の後継機であるポスト「京」に向けては，上記のような考えを取り入れ，メニーコア OS を仮想的な OS の 1 つと捉え，汎用的な仮想環境の実行をサポートする予定である．今後，必要となってくる様々なアプリケーションに合わせて，最適な実行環境を選択できるような使い勝手の良いシステムを目指している．

謝辞　本論文の一部は，文部科学省「特定先端大型研究施設運営費等補助金（次世代超高速電子計算機システムの開発・整備等 ）」で実施された内容に基づくものである．

参考文献

1) Super Computer TOP500, http://www.top500.org/
2) Riken AICS K computer, http://www.aics.riken.jp/en/k-computer/
3) FUJITSU PRIMEHPC FX100, http://img.jp.fujitsu.com/downloads/jp/jhpc/primehpc/primehpc-fx100-datasheet-ja.pdf
4) インテル® Xeon Phi™ 製品ファミリー, http://www.intel.co.jp/content/www/jp/ja/processors/xeon/xeon-phi-detail.html
5) Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from blue gene's cnk. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–10, Washington, DC, USA, 2010.
6) Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, catamount. In *In Cray User Group*, pp. 16–19, 2005.
7) 住元真司,小田和友仁,宇野俊司,石川裕, 次世代高性能計算機システムのためのシステムソフトウェア実現にむけて, 情報処理学会研究報告. [ハイパフォーマンスコンピューティング] 2013-HPC-141(8), 1-8, 2013-09-23
8) Masamichi Takagi, Balazs Gerofi, Tomoki Shirasawa, Gou Nakamura, Yutaka Ishikawa: McKernel Specification Version 1.0 (2015), http://www-aics-sys.riken.jp/project/mckernel/doc/mckernel-spec-1.0.pdf
9) Masamichi Takagi, Balazs, Gerofi, Norio Yamaguchi, Takahiro Ogura, Toyohisa Kameyama, Atsushi Hori, Yutaka Ishikawa, "Operating System Design for Next Generation Many-core based Supercomputer," IPSJ SIG Notes 2015-OS-133, 2015.
10) Taku Shimosawa, Balazs Gerofi, Masamichi Takagi, Gou Nakamura, Tomoki Shirasawa, Yuji Saeki, Masaaki Shimizu, Atsushi Hori, Yutaka Ishikawa, Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures, In IEEE International Conference on High Performance Computing (HiPC), IEEE, 2014.
11) 佐伯 裕治,清水 正明,白沢 智輝,中村 豪,高木 将通,Balazs Gerofi,思 敏,石川 裕,堀 敦史, ヘテロジニアス計算機上の OS 機能委譲機構, 情報処理学会研究報告. 計算機アーキテクチャ研究会報告 2013-ARC-205(15), 1-7, 2013-04-18 .
12) Documentation of cgroups on kernel.org, https://www.kernel.org/doc/Documentation/cgroups/
13) Docker - Build, Ship, and Run Any App, Anywhere, https://www.docker.com/