# 自由市場原理に基づくスケジューリング方式

松本　尚 †‡　　平木　敬 ‡

† 科学技術振興事業団さきがけ研究 21 「情報と知」領域
‡ 東京大学 大学院理学系研究科 情報科学専攻
Email: tm@is.s.u-tokyo.ac.jp

**あらまし**　タイムシェアリングシステムの従来のスケジューリングはプロセッサの占有時間による公平性の調節と入出力装置との同期によって主に決定されていた。しかし、近年ネットワークの発展とそれに伴う分散協調処理の実用化やマルチメディア処理の進歩により、プロセッサの計算負荷以上にネットワークの通信や外部入出力が重要となるアプリケーションが増加している。これらのアプリケーション間では、プロセッサの占有時間のみで公平性を判断しても良いスケジューリングは行えない。つまり、従来のスケジューリング方式は時代遅れとなっており、分散並列処理やマルチメディア処理に適した新しいスケジューリング（資源割り当て）方式を研究開発する必要がある。本稿では、ワークステーションクラスタ等の分散資源環境に適合する「自由市場原理に基づくスケジューリング方式」と呼ぶ新しい方式を提案する。この方式ではシステム提供のグローバルスケジューラは不要であり、動的最適化の決断はアプリケーションに任される。一方、マイグレーションによる負荷分散や資源使用要求の抑制による過度の資源競合の緩和といった最適化を自律的に行うことが奨励されるようにノード内のスケジューラが構成される。

**キーワード**　分散協調処理、スケジューリング、マイグレーション、公平性

# A Scheduling Scheme Based on Free Market Mechanism

Takashi MATSUMOTO†‡　　Kei HIRAKI‡

† PRESTO, Japan Science and Technology Corporation
‡ Department of Information Science, Faculty of Science, University of Tokyo
Email: tm@is.s.u-tokyo.ac.jp

**Abstract**　On existing systems, conventional scheduling methods use processors' utilization to keep fairness between users' application tasks (processes). As the systems have provided non-blocking I/O facilities for user-programs, new-types of applications that eagerly exploit I/O devices or network communications are coming out. For these applications the bottlenecks of systems are not processor resources but I/O or network ones. Therefore, conventional scheduling methods are old-fashioned for these applications . In this paper a brand-new scheduling scheme "FMM scheme (Free Market Mechanism scheme)" is proposed for workstation cluster systems. In the FMM scheme complicated global schedulers are unnecessary and dynamic optimizations are performed by user-programs. The FMM scheme provide the information disclosure mechanism which enable user-tasks to inexpensively access information on loads, configurations and usages of system resources. The FMM also presents fair node-level schedulers which take usages of I/Os or communications into account.

**key words**　parallel distributed processing, scheduling, migration, fairness

## 1 はじめに

ユーザアプリケーションの多用化につれて、マルチメディア処理のように大量のデータ入出力を必要としたり、数値シミュレーションの並列処理のように大量のデータ通信を必要とするアプリケーションが増えてきている。一方、これらのアプリケーションを効率良く実行するために、入出力や通信に対するユーザレベルの裁量権が拡大し、入出力や通信をユーザレベルで最適化して高効率で行えるようになってきている。具体的に述べると、ノンブロッキングの入出力手段やノンブロッキングの通信手段がユーザに提供されるようになってきている。通信の例を挙げると、MPI[1] にはノンブロッキング送受信関数が用意されており、MBCF[2, 3] のようなメモリを介した通信手段であれば任意のメモリをバッファとして実行をブロックすることなしに通信が行える。

ただし、これらのノンブロッキングの入出力操作手段を過剰に使用すると、周辺装置やネットワークの性能を飽和させて、システム全体の性能を大幅に低下させかねない。従来の周辺装置の入出力要求やネットワークへのデータ送受信要求はブロッキング操作であったため、一つのアプリケーション（タスク）が同時に複数の要求を出すことにより、システムの能力を飽和させてしまうことはなかった。最近になってノンブロッキング操作が提供されるようになってきているが、多くのオペレーティングシステムにおいて資源のスケジューリング自体は昔ながらのプロセッサ使用時間に基づくプロセッサの時分割による割り当てである。このため、ネットワークの性能が飽和している場合には、通信を必要としないタスクを優先的にスケジューリングする等の考慮がなされるべきであるにもかかわらず、なされていない。つまり、ノンブロッキングの入出力操作や通信操作が提供されている環境（オペレーティングシステム）では、従来型の時分割タスクスケジューリング方式では役不足である。

多くの資源が共用される分散システムや並列システムの方が単一プロセッサの独立した計算機と比べて、資源競合によってシステムの能力低下する可能性が高い。また、分散メモリ（分散資源）型のシステムでは負荷分散のためには、タスク（プロセス）を実行途中でマイグレーションする必要が出る。ワークステーションクラスタや PC クラスタシステムが徐々に実用化されつつあるが、資源競合の回避、公平性、マイグレーションによる負荷分散といった問題を解決するシステムは現れていない。

本稿では、ワークステーションクラスタまたは PC クラスタを対象として[1]、ノンブロッキング入出力操作が提供されている環境で必要とされるスケジューリング方式の要件を議論し、ユーザレベルの自律的最適化に適した「自由市場原理に基づくスケジューリング」を提案する。

[1]分散メモリ型並列計算機にも本稿の議論は適用できる。

## 2 資源割り当ての公平性をいかに算出するか

すでに述べたように、従来の汎用オペレーティングシステムは、外部入出力がない場合には単位時間当たりのプロセッサ資源の使用時間が公平になるようにタスクのスケジューリングを行っていた。以前は外部入出力（通信を含む）がブロッキング方式で行われていたため、外部入出力の性能が飽和してシステム全体の性能が低下するという事態はほとんど起こらなかった[2]。ノンブロッキング方式の外部入出力が可能になると、一つのタスクが外部入出力装置を性能限界まで使用することが容易になった。ノンブロッキング方式では操作内容や処理結果をバッファリングしているため、外部入出力を要求したタスクをプロセッサのスケジューリングから外しても、外部入出力装置における処理は続けられる。空いたプロセッサに割り当てられた別のタスクが同じ外部入出力装置に対して要求を出すと、外部入出力装置は処理しきれずに性能が飽和してしまう。この外部入出力装置がネットワークや共有ディスク装置のような共有資源である場合には、事態はもっと深刻であり、同時刻に同一の資源を頻繁に使用するタスクが複数のノード（マシン）において割り当てられれば、外部入出力装置の性能が飽和してしまう。

具体例を挙げると、ワークステーションクラスタ上の並列処理における高速ユーザレベル通信はその典型例である。また、連続メディア処理のための外部記憶装置アクセスや通信もこの例に含まれる。これらの処理ではプロセッサの計算能力ではなく、外部入出力装置の能力がボトルネックになる可能性があるわけである。性能のボトルネックとなっているものが外部入出力装置であるならば、スケジューリングにおける公平性もプロセッサではなくて、そのボトルネックとなっている資源の使用比率を公平に分けるべきなのではないのだろうか。また、汎用スケーラブルオペレーティングシステム *SSS–CORE* の提案当初に述べたように [4]、メモリ資源も無視できない。メモリを多くのタスクが浪費して、スラッシングのような事態が発生してはシステム全体の性能が大幅に低下してしまう。メモリ浪費を抑えてつつましやかに処理を行うタスクは、メモリを盛大に浪費するタスクよりも優先されるべきであろう。少なくとも、浪費タスクによってスラッシングが発生しても、倹約タスクはその被害があまり及ばないようにする責任が汎用オペレーティングシステムにはあるのではないだろうか。

上記議論から、プロセッサ資源の使用時間だけを特別扱いする理由はないことが判る。資源割り当て時にはすべての資源の使用状況が考慮されるべきである。ただし、性能が飽和していない資源に関しては、公平性の観点では（課金や節電の観点では別であろう）、考慮する必要がない。なぜなら、余っている資源は有効活用してもらった方がシステム（というよりシステム設計者）としてはありがたいからである。従来のプロセッサ資源の使用時間に基づく時

[2]バス型のネットワークに多数のマシンを接続するような非常識な場合は除く

分割というのも、資源要求が使用可能な資源量を上回っているために行われる方策である。この方策を自然に他の資源に拡張すると、資源（または性能）が不足する場合のみスケジューリングの公平性の算出に算入すればよいことになる。

## 3　スケジューリングすべき対象は何か

前節の議論において性能が飽和している資源の使用時間（使用総量）を公平性の算出に使用すべきであるという結論を得た。それでは、何をスケジューリングするのであろうか。答を先に述べると、これは従来通りプロセッサ資源である。ノンブロッキング入出力操作はオーバヘッド削減のため、システムコール不要でユーザレベルにおいて行えるようになりつつある。このため、入出力操作自体をスケジューリングの対象にすることは難しい。プロセッサ資源への割り当てが行われていなければ、当然入出力操作を行うことは不可能である。プロセッサ資源への割り当てを調節することによって、たとえユーザレベルの処理要求であっても、間接的に他の資源への処理要求の頻度を調節することが可能である。ただし、従来のプロセッサ資源スケジューリングと異なるのは、他の資源の能力が大幅に飽和している場合は、飽和状態を緩和するために、しばらくの間プロセッサに何もアプリケーションタスクを割り当てないという状況もあり得る。つまり、そのマシンに存在するすべてのタスクが飽和している資源を高頻度で使う場合にこの状況が起こり得る。

なお、メモリ資源の不足時にはリプレース対象（犠牲）をどのタスクのどの種類のメモリから選ぶかという選択の余地が存在し、そのための判断つまりスケジューリングが必要となる。分散共有メモリを使用する場合のリプレース対象の選択戦略に関する提案を文献 [4, 5] において行っている。メモリリプレース戦略については本稿の議論の範囲外とする。

## 4　中央集権か自由主義か

本稿では、分散並列処理環境におけるスケジューリングを議論している。負荷分散をタスクの実行単位よりも細かく実現するために、タスクのマイグレーションが可能であると仮定する。

どのノードでタスクを実行するかの判断を中央集権的に行う方式とユーザもしくはアプリケーションタスク自体が走行するノードを選択する方式が考えられる。さらに、ノードの中のスケジューリングを他のノードと同期して（連携して）行う方式と独立して行う方式が考えられる。言い替えると、実行場所の制御を中央制御するか、自由放任で行うかの区別と、実行時刻の制御を中央制御するか、ノード毎に任せる（地方分権）かの区別である。この類別によると 4 種類のスケジューリング方式が考えられる。ここでは、両極端にある二つのケースについて議論する。より中央制御の徹底した実行場所も実行時刻も中央制御する方式を**中央集権方式**と呼び、よりユーザの自由度が高い実行場所はユーザが選択し、実行時刻はノードのスケジューリングに任せる方式を**自由主義方式**と呼ぶ。なお、実行時刻をノード内の独立したスケジューリングに任せたとしても、自分の都合に合わない時刻にプロセッサを割り当てられた場合に、プロセッサの使用権を放棄する自由／手段があるものと仮定する。プロセッサ割り当て後すぐに使用権を放棄した場合に、実行優先度が下げられないで ready キューの先頭（もしくは先頭付近）に戻されるような仕組みがあれば、実行時刻をタスク自体が制御することが可能である。中央集権方式はグローバルなスケジューラを持ち、時分割のギャングスケジューリングが可能である。自由主義方式はノード毎のスケジューラのみを持ち、ノードの選択はユーザまたはアプリケーション任せである。したがって、中央集権方式ではグローバルスケジューラがマイグレーションの実行の決断およびマイグレーション先の選定を行い、自由主義方式ではマイグレーションの実行の決断およびマイグレーション先の選定をアプリケーションタスク自体が行う。後者の場合、タスクの移送行為自体はシステムコール等でオペレーティングシステムに委託される可能性がある。

この二つの方式の特徴をまとめると以下のようになる。

- 中央集権方式の特徴
  1. グローバルかつ詳細な負荷分散が可能
  2. グローバルな最適化スケジューラが必要
  3. ギャングスケジューリングの実現が容易
  4. グローバルスケジューラが公平性を考慮
  5. プロセッサプール方式に適合
- 自由主義方式の特徴
  1. 自己責任による自律的負荷分散
  2. 自律最適化、ノードスケジューラの構成が単純
  3. ギャングスケジューリングは疑似的な実現
  4. ノードスケジューラが公平性を考慮する必要
  5. 個別ワークステーション的使用と両立

分散並列処理環境が均質かつ平坦な構成でない場合には、協調処理もしくは並列処理ではタスクの環境内の配置やタスクのスケジューリング順序が実行効率に影響する。中央集権方式の場合は、グローバルスケジューラが細かい資源割り当てまで決定するため、ユーザアプリケーションの都合を反映するためには何らかのインタフェースが必要である。つまり、グローバルスケジューラにユーザ側から制約やヒントを提示して、それらを満たすようにスケジューリングしてもらう必要がある。制約やヒントが複雑になると、複数の協調並列タスクの制約やヒントを準最適に満足するスケジューリングを求めることさえ困難になる。簡単なスケジューリング制約しか指定できないと協調並列タスクの実行最適化の機会を減らしてしまう。これに対して、自由主義方式ではアプリケーションタスク自

体が資源割り当ての方向性（どのノードで実行するか、割り当てられたタイムスロットを受け入れるか）を決断するため、プログラム次第で自由な実行時最適化が可能である。しかし、逆に言えば「最適化」のつもりで行った行動が実行時間を大きくする方向に働いたとしても他人（スケジューラ）の責任にはできない。システムを作る立場からは、複雑な資源制約を解決するグローバルな最適化スケジューラを実装する必要のない自由主義方式の方が圧倒的に簡単である。

次に、中央集権方式のメリットと考えられるギャングスケジューリング実現の容易さについて議論する。ノンブロッキング通信操作によって複数のタスクが通信している場合は、タスク間の厳密な同期は不要である。もちろん、タイムスライス（tick）オーダの実行のずれが許容できるかどうかは協調タスクの性質による。しかし、前述のように自由主義方式と言えども、実行時刻を調整することが不可能であるわけではない。ギャングスケジューリングが必要なタスクがノードに複数割り当てられた場合は、自然にタスクがノードに割り当てられるタイミングが揃うようにノードスケジューラを構成することは可能である[3]。自由主義方式の方がギャングスケジューリングと同等の動作を行うためには少し余計にコストがかかる可能性があるが、ギャングスケジューリング自体の有効性はまだ明らかではなく、ギャングスケジューリング実現のためのコスト差による効果は今後の検討課題である。

プロセッサプールは無名のプロセッサ資源の集合を指し、個々のプロセッサ資源を特定してタスクの実行が行われることはない。グローバルスケジューラがプロセッサプールから負荷や各種制約を考慮して適したプロセッサ資源を選んでタスクに割り当てる。プロセッサプールというのはバックエンドの計算資源という位置付けである。これに対して、分散並列計算機環境がワークステーションクラスタやPCクラスタである場合は、個々のマシンをワークベンチとして使用したいと言うケースも多い。この場合、ディスプレイカードや音声合成デバイスといったマシン依存のデバイスが大きな役割を果たす。このため、特定のマシンで動く必要のあるタスクや特定のマシンで動いた方が圧倒的に有利なタスクが存在する。また、従来オペレーティングシステムにおける使い方では、単一実行アプリケーションはユーザがマシンを指定して起動することが一般的である。このことから、ワークステーションクラスタやPCクラスタではプロセッサプールによるタスクスケジューリングのみを行うことは望ましくない。各ノードに対して個別ワークステーションとしての使用を認める場合には、そのノードを特定して投入したタスクによって、ノードの負荷が大幅に変動してしまう。グローバルスケジューラがこの変動を事前に予期することは不可能であるため、事後的に負荷調整を行うと調整のためのオーバヘッドが発生する。また、この負荷変動のために、それ以前に行ったマイグレーション等の負荷均衡のための動作が裏目になるかもしれない。もちろん、自由主義方式でも同じことが言えるが、この場合は元々自己責任に基づいているので、予期しない負荷変動による不利益も自己責任の範囲内である。

中央集権方式は全知全能のグローバルスケジューラがスケジューリングの公平性に関する考慮を行う。これに対して、自由主義方式ではノードレベルのスケジューラしか存在しないため、ノードレベルのスケジューラにおいて公平性を実現する必要がある。プロセッサの割り当て頻度を調整することにより公平性を実現するため、ノードレベルでも十分に実現可能である。自由主義方式に関する公平性の議論は次節で行う。

結局、中央集権方式を採ろうが自由主義方式を採ろうが実現できる機能には差がない。しかし、自由主義方式はスケジューラの実装が容易であり、アプリケーションから見てインフラストラクチャであるスケジューラを変更することなしにユーザレベルの最適化を強化できる可能性がある。もちろん、逆にユーザレベルの実行時最適化の戦略が幼稚であれば、タスクの実行効率を低下させる危険もある。しかし、ランタイムライブラリやコンパイラ生成コードとして実行時最適化ルーチンが提供されるのであれば、この危険は幼稚な戦略しか採れないグローバルスケジューラを実装してしまう危険と変わらない。このため、自由度が高く、システムの実装が楽な自由主義方式を我々は採用する。

## 5 自由市場原理に基づくスケジューリング方式

前節において望ましいスケジューリング方式に対して「自由主義」方式という言葉を使用したが、理想の市場主義経済体制がそうであるように、決して無政府状態の自由放任方式を意味しているわけではない。ある種の公平性は保証しなくてはならず、自己責任でスケジューリングの判断を下すためには判断を下すのに十分な情報が低コストで入手可能でなくてはならない。

必要性が明白な低コストの情報入手について先に補足する。実行時に動的に負荷分散を自らの判断で行うためには、自分のノードの負荷情報のみではなく、他のノードの負荷情報も必要である。また、システムが非均質であれば、マイグレーション可能なノードを知るために各ノードのプロセッサの種類といった情報も必要になる。これらの情報を獲得するために、各タスクが毎回他ノードに通信を行うようではコストが大き過ぎる。他ノードも含めた資源の使用状況や割り当て情報がユーザアプリケーションから低コストでアクセスできる情報開示機構が必要である。この情報開示機構には *SSS–CORE* で提案した情報開示機構 [4] がそのまま使用可能である。

公平性を算出する方式を議論する節でもすでに述べたように、全体の利益を無視してひたすら利己的に振舞うタスク（結果的にはシステム全体の性能を低下させ自分にも不利益をもたらしている）がシステム全体の利益を考慮して自律的に外部入出力やメモリの使用量を抑制しているタ

[3] すべてのノードで同じ大きさの tick を採用し、自律的なタイムシフトが動的に可能な機構（システムコール等）を用意する。

スクに迷惑をかけるようでは良いシステムとは言えない。具体例を挙げると、 ethernet 等の通信網では競合がある程度以上になると通信の成功率が大幅に低下して、意識的に通信を控えた方が通信の全体性能が向上する。ユーザレベルの高性能通信手段を利用するような性能重視のアプリケーションでは、常に最高性能が出せるようにプログラムを構成したい。そこで、ネットワークの使用状況を調べて、ネットワークがある程度以上の競合状態にある場合には、通信頻度を自分で抑える（TCP/IP 実装などで見られる輻輳回避をユーザレベルで行う）という最適化が考えられる。この自律的最適化を施したタスクとそうでないタスク（こちらもネットワークを高頻度で使用する）が同時に同じノードにスケジューリングされた場合に、ネットワークの競合状況を見て自律的最適化を行うタスクのみが通信（実行）を遠慮して、最適化を行わない「厚かましい」タスクが優遇されて実行される結果になってしまうようでは「公平な」システムとは呼べない。つまり、システム全体の性能低下の防止ならびにアプリケーション間の公平性堅持のための基本ルールを設ける必要がある。

「低コストでアクセスできる情報開示機構」を持ち、「公平性を守るための基本ルールを確立」した自由主義スケジューリング方式を**自由市場原理に基づくスケジューリング方式**（FMM 方式: Free Market Mechanism 方式）と呼ぶ。

## 6 SS-Wait 同期方式と情報開示機構

FMM 方式の情報開示機構として *SSS–CORE* の情報開示機構が使用可能である。本節では、その情報開示機構が用意された理由である同期方式と情報開示機構の構成について簡単に説明する。

筆者らが開発している汎用オペレーティングシステム *SSS–CORE* では、 Time Sharing System を維持しつつ、効率の良いスピン同期を実現するために、松本が 1989 年に考案した Snoopy Spin Wait（SS-Wait）[6] と呼ばれる同期方式を採用している。これはスピン（ポーリング）方式の変形で、フラグをチェックすることにより同期の確認を行う点ではスピンウェイト方式とまったく同一であり、同期が成立していれば、そのまま処理を続行する。同期が成立していない場合に、フラグのチェックを繰り返す方式が単純なスピンウェイト方式であるが、 SS-Wait では資源割り当て情報や同期相手の進捗情報を利用してフラグチェックを繰り返した方がよいかどうかの判断が追加される。もし、同期相手がプロセッサの割り当てを受けていなかったり、大幅に進捗状況が遅れている場合には、フラグチェックの繰り返しを放棄して、タスク内の別の処理を行うかプロセッサの実行権を他のタスクに委譲する。 FMM 方式では単に実行権を放棄すると言う選択肢の他にマイグレーション（つまり異なるノードにコンテクストを移送して実行を継続）する選択肢が増えることになる。ただし、マイグレーションは実行が大幅に遅れた場合に行う必要がある選択肢であるため、 SS-Wait の結果直接自らのマイグレーションが選択されるようにプログラムを構成するのは難しい。定期的に自ら判断するか実行が先行するタスクから非同期メッセージを送ってもらってマイグレーションを実行する。 SS-Wait はこの非同期メッセージを送る機会を提供するのに使用可能である。

SS-Wait の実現でキーとなるのは自分のノードのみではなく他ノードの資源割り当て情報や資源使用状況といった情報が低コストで参照可能な機構である。このために、*SSS–CORE* は情報開示機構を持っている。カーネルが管理している資源量、資源の種類、資源の機能、資源割り当て情報や資源使用状況といった情報を格納しているメモリ領域がユーザ空間に read-only でマップされており、低コストで参照できる。しかも、情報開示機構には自分のノードの情報のみではなく、他ノードの情報も適宜交換され開示されている。負荷分散に関する動的な最適化は、最適化のためのアクション（例えばマイグレーション）のコストが大きいため、比較的大きな粒度で判断すべきものである。このため、情報開示機構の他ノードの情報は細粒度で更新する必要はなく、最も細かい粒度でもたかだかタイムスライスのオーダである。ユーザアプリケーションは情報開示機構の情報を参照して、 SS-Wait を行い、また今後の負荷分散戦略や実行タイミングの検討を行う。なお、同期相手の進捗状況に関しては適当な間隔で、協調処理を行うタスク間においてお互いに情報交換すればよいため、特に情報開示機構の開示対象とはなっていない。

FMM 方式ではマイグレーション先を情報開示機構を使ってユーザタスク自らが決定する。情報開示機構の情報にはプロセッサのアーキテクチャの種別等を含む（資源の種類に該当）ため、システム全体は非均質な構成であっても構わない。同一実行コードでなくてはマイグレーションできない場合には、タスク自らが条件に合ったマイグレーション先を選択する。

## 7 優先度再計算方式の拡張による公平性の導入

ノードにおけるスケジューラにおいて公平性を実現するために、従来はプロセッサ占有時間しか考慮されていなかった優先度再計算（エイジング）を拡張することを提案する。なお、ノード単位でスケジューリング時にこの優先度の高い（若い）タスクからプロセッサ資源の割り当てがなされる。

優先度再計算方式を以下の要因が反映できるように構成する。

1. 性能が飽和している（資源が不足している）場合に限り、該当資源の使用量と使用時間の積に応じたエイジング（加齢）
2. 競合度の大きさに応じたエイジング
   （入出力実現コストに応じたエイジング）
3. 明示的かつ即時の実行権委譲に対する若返り
4. 明示的な実行待機の待機時間に応じた若返り

5. ノードオーナーのタスクもしくはノード固有のデバイスを使用するタスクの優遇（エイジングの度合を小さくする）

第一項目は公平性を実現するためのエイジング要素であり、対象はプロセッサのみではなくメモリ、通信、ディスク I/O 等のすべての不足している資源に関して積算する。第二項目は自律的動的最適化を奨励するための要素であり、 ethernet 等では使用率がある程度以上大きくなるとエイジング（一種のペナルティ）の度合を加速的に大きくして、ネットワーク混雑時の使用を制限する。優先度がある程度以上低くなったタスクは、タスクのプロセッサへの割り当てを中断する[4]。第三項目はギャングスケジューリングを実現するための優先度再計算の補正項目である。意に沿わないタイミングで実行権を得た場合に、すぐに実行権を放棄すると次のスケジューリング時に実行権が割り当てられる可能性を高める。第四項目はマルチメディア系リアルタイム処理を実現するための優先度再計算の補正項目である。その時点の CPU 負荷ならびに自分の実行優先度に応じた待ち時間以上に明示的に sleep するタスクに対して、sleep 時間が終了した時の優先度を高めてスケジューリングされる可能性を向上させる。第五項目は Wisconsin 大の Condor システム [7] に触発されて考案した項目であり、各ノードをワークベンチとして使用する際に数値計算等のアプリケーションの実行によって使用者にストレスをかけないための補正項目である。 Condor との比較は関連研究の節に記述する。

ここで述べた方針の主たる目的は不足もしくは飽和している資源への要求をなるべく控えることを奨励する基本ルール（第一項目および第二項目）を定めることにある。このことはユーザレベルのノンブロッキング入出力操作に一定の条件を加える。つまり、ユーザレベル操作であってもタスクが操作した回数、データ量、状況、操作タスク名といった情報のプロファイリングが可能である必要がある。詳細な操作ログは不要であるが、課金情報程度の情報を収集する必要がある。ユーザレベル入出力操作をハードウェア的に実現する場合には、この情報収集機能をハードウェアで持たせることが望ましい。逆に、この情報が得られなければ、たとえ FMM 方式以外のスケジューリング方式であっても、ユーザレベル入出力操作に関して使い過ぎのペナルティを課すことはできない。

また、ノードスケジューラにコストの大きな処理を行わせないために、スケジューラの実装は以下の方針で行う。

- タスクの状態遷移時点でのみの優先度計算

この方針は計算コストが大きくしないために、スケジューリング時に優先度再計算を行うタスクをタスクの状態（active,ready,wait,sleep）が変化したものに限定することを意味している。この方針の実現でキーとなるのは優先度計算の発散を防止する方法である。発散防止法に関してはスケジューリング時刻をベースにした優先度を使用することで解決を図る。

[4]優先度が発散しないようにタイムスライスごとに若返らせることにより、将来的にはスケジューリングされる

## 8 優先度再計算方式案

FMM 方式で重要な役割を担うノードスケジューラの優先度再計算方式について、具体的なイメージを提示するために一つの計算方式案を示す。なお、この計算方式案は研究の進展に従って変更される可能性がある。

以下、優先度再計算方式と再計算式の一案について説明する。一定間隔（tick）のタイムスライスでタスクがプリエンプトされるタイムシェアリングシステムを対象に想定する。優先度の再計算はタスクがプロセッサの実行権を失った時に行われる。タイムスライスでプリエンプトされた場合は、再計算された優先度でソートされた ready キューに登録され、一番優先度が高い（本稿では一番値が小さい）タスクが次の実行タスクとしてキューから取り出される。タイムスライス時に再計算した優先度が、 ready キュー内の他のタスクよりも高ければ、次の tick も同じタスクの実行が継続される。

明示的に実行権の放棄（*SSS–CORE* では時間ゼロの sleep）を行った場合は、再計算した優先度が他の ready キューのタスクよりも高くても、 ready キューに他にタスクがある場合はタスクを切替える。明示的に sleep システムコールを行った場合は、 sleep キューにタスクをつなぐ前に優先度再計算を行う。タイムアウトで sleep していたタスクが起き上がってくる場合には、 sleep システムコール発行時に計算した優先度で、 ready キューにつながれる。

優先度再計算式の表記法は以下の通りである。

- 実行権を失って優先度再計算対象のタスク: $\tau$
- 単調増加するシステム内時間: $T$
- OS が管理する資源: $k$
- 時刻 T 直前のプロセッサ割り当て期間内にタスク $\tau$ の資源 $k$ に対する消費量: $n_k(\tau, T)$
  （実メモリなら占有実ページ数、ディスク読み出し要求ならセクタ数）
- 時刻 T 直前のプロセッサ割り当て期間内にタスク $\tau$ が資源 $k$ を占有した時間: $t_k(\tau, T)$
  （通信等では占有時間ではなく要求回数）
- 時刻 T における資源 $k$ の競合度: $c_k(T)$
  資源量もしくは能力に余裕がある場合は 0（もしくは十分に小さい値）をとり、そうでない場合の値は競合が性能におよぼす大きさを考慮して決められる。ただし、 $c_k(T)$ は資源使用量と資源占有時間の積に対する比例係数を兼ねる
- タスク $\tau$ のノード使用優先度: $np(\tau)$
  $np(\tau)$ は正の値を取り、値が小さいほどエイジングの度合が小さくなって優先される。ノードのオーナー（コンソール使用者）のタスクやノード固有のデバイスを使うタスクのみを小さな値にして優遇する

- 時刻 T における即時実行権放棄に対する若返り係数: $C1(T)$
  システムの負荷状況によって係数が変化するため T の関数
- タスク $\tau$ が時刻 T において即時の実行権放棄を行ったかどうかを示す関数: $f(\tau, T)$
  放棄時に 1 非放棄時に 0 の値を取る
- 時刻 T における明示的実行待機（sleep）に対する若返り係数 $C2(T)$
  システムの負荷状況によって係数が変化するため T の関数
- タスク $\tau$ が時刻 T において sleep を行った場合の sleep の設定時間: $s(\tau, T)$
  非 sleep 時には 0 の値を取る
- ユーザが（増大方向にのみ）設定可能なタスク $\tau$ に対する優先度のオフセット: $O_\tau$

優先度再計算式は以下のような形式になる。なお、値が小さいほど優先度が高い。

$$P(\tau,T) = T + \sum_k n_k(\tau,T)t_k(\tau,T)c_k(T)np(\tau) \\ -C1(T)f(\tau,T) - C2(T)s(\tau,T) + O_\tau(1-f(\tau,T))$$

本優先度再計算式において単調増加するシステム内時間 $T$ を優先度計算の基準に使っており、タスクがプロセッサの割り当てを解かれた時に上式に基づいて優先度を計算する。その計算結果に持って ready キュー（もしくは wait キューまたは sleep キュー）に登録する。基準時刻 $T$ は自然に増大するため、キュー内のタスクの優先度を若返らせる必要はない。$T$ が単調増加すると言っても、現実には上限が存在する。しかし、値の巡回を考慮して符号つき整数を $T$ に使用することにより、この問題は解決できる。同時に扱われる優先度の振幅が常に表現できる値の半分未満に抑えられていればよい。32bit もしくは 64bit 長の時刻を用いればこの制約は実用レベルで満足可能である。

# 9 協調動作するタスクのスケジューリング

ユーザアプリケーションが自由にスケジューリング戦略を決定できる場合に、複数のタスクが協調して一部のタスクの実行を妨害するようなことができないように公平性を守るための基本ルールを決める必要がある。FMM 方式の場合、システム提供のスケジューラはノードレベルのスケジューラのみであり、このノードスケジューラが協調動作するタスクによって一部のタスクに不利益になるような決定を下さなければ良い。ノードスケジューラにとって協調動作するタスクというのはノード内に割り当てられている同一ユーザのタスクと考えられる。よって、優先度計算をユーザ単位に行って、同一ユーザのタスク間ではラウンドロビンでスケジューリングするというような手法を採ることによって公平性を保つことができる。複数のユーザが協調（結託）して一部のタスクのみが優遇もしくは冷遇されるようにすることは可能かもしれないが、この結託の事実を見破る方法がなければ対処の方法は存在しない。

# 10 関連研究

## 10.1 Mach のスケジューリング

マイクロカーネル構造にしてユーザレベルでスケジューリングポリシが記述できるようになった初期のオペレーティングシステムとして Mach[8] がある。特に、ユーザレベルのページャプロセスが有名である。ただし、これらのスケジューラはミドルウェアであり、ユーザアプリケーションが自由にスケジューリング戦略を変更できるわけではない。スケジューリング関連のサーバプロセスの実装がカーネルレベルではなくユーザレベルで行われているだけである。Mach は分散環境を指向したオペレーティングシステムとして開発されていたが、特徴的な機構および機能はノード内のマルチプロセッシングをサポートするものが多かった。Mach は標準的なノード間の負荷分散機構をもっていないため、本稿の分類では自由主義方式の一種と見なせる。ただし、入出力操作を考慮した公平性を守るための基本ルールも、ユーザレベルにおけるスケジューリング戦略策定を助ける情報開示機構も存在していない。

## 10.2 SSS–CORE の二階層スケジューリング

*SSS–CORE* の研究開始当初、グローバルなカーネルレベルスケジューラと並列タスク（並列に動作するタスクグループ）毎のユーザレベルスケジューラを組み合わせた二階層スケジューリング方式を提案していた [4]。各並列タスクはカーネルレベルスケジューラに希望する資源量（プロセッサ数、メモリ量）、条件（地理的条件、時間的条件）、希望の強さを伝えて資源を割り当ててもらう。並列タスクに割り当てられた資源はユーザレベルスケジューラによって並列タスク内のスレッドやタスクに配分する。本稿の分類ではユーザレベルスケジューラはユーザプログラムの一部と考えられるため、中央集権方式の一種である。二階層スケジューリング方式に基づくスケジューラを実装する前に、ノンブロッキング入出力による資源競合がシステム性能の大幅な低下を引き起こすことが明らかになり、この事態を回避する必要が生じた。*SSS–CORE* には情報開示機構があるため、タスクは自律的に資源競合を回避することが可能である。しかし、すべてのタスクにこの自律的な回避動作を期待することは不可能である。そうだからと言って、プロセッサとメモリに関する資源制約を満たすカーネルレベルスケジューラを作ることすら容易ではないのに、他の資源に関する資源競合を配慮するカーネルレベルスケジューラを作ることは困難である。この *SSS-CORE* のスケジューラ開発困窮状況を打破するために考案したのが本稿で提案した FMM 方式である。プロセッサ資源以外の資源に関する総使用量をエイジングに反映するというアイデアは二階層スケジューリング方式のアイデアを継承している。

### 10.3 Condor のスケジューリング

Condor は UNIX カーネルに手を加えることなく、ミドルウェアとライブラリによってプロセスのマイグレーションを可能にした UNIX マシンクラスタシステムである [7]。ただし、現在のところプロセス間通信を行うプロセスはマイグレーションできない[5]ため、並行プロセスや並列プロセスのマイグレーションをサポートできない。Condor のスケジューリング方式 [9] は本稿で言うところの実行場所の制御を中央制御し、実行時刻の制御をノード（UNIX）に任せるタイプである。多くのノード（UNIX マシン）をプロセッサプールに登録してもらうために、マシンのオーナーに迷惑を掛けないというポリシを強く掲げている。このため、オーナーがマシンを使用し始めたら、そのマシンで動かされていたタスクは他のマシンにマイグレーションされる。負荷分散やノードの割り当てをグローバルスケジューラで行っているために、複雑な制御や新しい最適化を可能にするためには、スケジューラをどんどん複雑にせざるを得ない問題点を抱えている。優先度再計算方式の拡張の節において述べたように、FMM 方式でもノードスケジューラにおいてオーナーのタスクを優遇することで、オーナーの実行権の保護が可能である。このため、私達はノードオーナー保護の観点でシステムレベルの中央制御が必要になるとは考えていない。

## 11 おわりに

本稿では、まずワークステーションクラスタ等の分散協調環境における資源割り当て方式として、必要な要件について検討した。プロセッサ資源だけを考慮した公平なスケジューリングでは、最近の高性能入出力方式に対応できないことを明らかにし、容量もしくは性能が飽和した資源に関しては、すべて公平性の考慮に入れることを提案した。分散協調システムが提供するスケジューラとして、中央集権方式と自由主義方式を比較して実現される機能に差がないことを示した。しかし、資源割り当てに対してユーザアプリケーションの最適化戦略を反映しようとすると、中央集権方式ではユーザの要求を聞いて複数の資源の使用状況を考慮しつつ最適なスケジューリングを求める非常に複雑なスケジューラを実装する必要がある。これに対して、自由主義方式では資源割り当てに関する戦略がユーザの自己責任であるため、システムが提供するスケジューラ自体は単純になる可能性がある。自由主義方式ではユーザ自らが判断を行うために、資源の使用状況が把握できる「低コストでアクセスできる情報開示機構」が必要不可欠であり、システム全体に迷惑をかけないようにする「公平性を守るための基本ルールを確立」が重要である。そして、これらの条件を満たす自由市場原理に基づくスケジューリング方式（FMM 方式）を提案した。FMM 方式では、ノードごとのスケジューラの優先度再計算（エイジング）方式に資源競合等を反映することによりタスク間の公平性を守る。この FMM 方式に使用する優先度再計算方式として単調増加する内部時間を利用した、計算量が少ない計算方式を示した。

FMM 方式を汎用スケーラブルオペレーティングシステム *SSS–CORE* のスケジューリング方式として採用し、現在実装中である。並列もしくは協調実行中のタスクグループの一部もしくは全体が自由にマイグレーションできるシステムを構築する予定である。

[5]将来サポートすると表明しているが、UNIX カーネルの変更なしに実現が可能である保証はない。


## 謝辞

Condor のスケジューリングとの差異に関して議論していただいた東京工業大学 / さきがけ研究 21 の松岡聡先生に感謝いたします。


## 参考文献


[1] Gropp, W., Lusk, E., Doss, N., and Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard. *Parallel Computing*, Vol.22, No.6, pp.789–828 (September 1996).

[2] 松本 尚, 平木 敬: 汎用超並列オペレーティングシステム SSS–CORE のメモリベース通信機能. 第 53 回情報処理学会全国大会講演論文集 (1), pp.37–38 (September 1996).

[3] Matsumoto, T. and Hiraki, K.: MBCF:A Protected and Virtualized High-Speed User-Level Memory-Based Communication Facility. In *Proc. of the 1998 ACM Int. Conf. on Supercomputing*, pp.259–266 (July 1998).

[4] 松本 尚, 平木 敬: 汎用並列オペレーティングシステム SSS–CORE の資源管理方式. 日本ソフトウェア科学会第 11 回大会論文集, pp.13–16 (October 1994). *http://www-hiraki.is.s.u-tokyo.ac.jp/ssscore/index-j.html*

[5] 信国 陽二郎, 松本 尚, 平木 敬: 汎用超並列 OS *SSS–CORE* におけるスケジューリング方式. 情報処理学会論文誌, Vol.39, No.6, pp.1738–1745 (June 1998).

[6] 松本 尚: マルチプロセッサ上の同期機構とプロセッサスケジューリングに関する考察. 計算機アーキテクチャ研究会報告 No.79-1, 情報処理学会, pp.1-8 (November 1989).

[7] Condor Team: The Condor High Throughput Computing Environment. *http://www.cs.wisc.edu/condor/*.

[8] M. Accetta, et al.: Mach: A New Kernel Foundation for UNIX Development. *Proc. Summer 1986 UNIX Conf.*, USENIX, pp.93–112(1986).

[9] M. Livny, J. Basney, R. Raman, and T. Tanenbaum: Mechanisms for High Throughput Computing. *SPEEDUP Journal*, Vol.11, No.1 (June 1997). *http://www.cs.wisc.edu/condor/doc/hpdc98.ps*