仮想計算機システムにおける論理プロセッサをスケジュールする新方式の開発と評価

梅野英典[†] 久保隆重[††] 今田豊寿[†††]

仮想計算機システム（VMS）は，1台の実計算機システム下で，複数台のOSを同時に動かす．仮想計算機（VM）は，該実計算機と同等のアーキテクチャを有する論理的な計算機である．各VMは，1台または複数台の論理プロセッサを持つ．ハイパバイザは，VMS全体を制御するソフトウェアで，論理プロセッサをスケジュールする．従来ハイパバイザは，それへの割込み（ホスト割込み）を契機として，待ち状態の論理プロセッサに当該割込みを報告し，その状態を実行可能状態に変える．このような従来のスケジューリング方式においては，上記割込みシミュレーションのオーバヘッドが大きくなり，VMモードとネイティブモードとの間の移行オーバヘッドが大きくなるという問題点がある．本論文は，この問題点を解決するために，待ち状態の論理プロセッサに関して新しいスケジューリング方式を提案する．すなわち，ハイパバイザに割込みを発生させることなく，割込み要因を検出し，それを検出したときに該当待ち状態論理プロセッサをスケジュールする方式を与える．また，ハイパバイザは負荷がなくなったとき待ち状態の論理プロセッサを探索し，ホストI/O割込みを発生させることなく負荷を検出する．これによって上記問題点を解決しVMSの性能を負荷のCPU利用率が低いときも，高いときと同様に，ほとんどネイティブ性能に近くなるまで向上させることができた．

Development and Evaluation of New Methods for Scheduling Logical Processors in Virtual Machine Systems

Hidenori Umeno,[†] Takashige Kubo[††] and Toyohisa Imada[†††]

Virtual machine system (VMS) can run multiple operating systems (OSs) in a single real computer system. Virtual machines (VMs) are logical computers with almost the same architecture as the real computer system. Each VM has single or multiple logical processors. A hypervisor is software to control an overall VMS, and schedules those logical processors. Traditionally, the hypervisor receives an interrupt that has to be simulated to a waiting logical processor. After simulating the interrupt, the hypervisor changes the logical processor's state to a ready state. This traditional scheduling has two problems. One is to increase the simulation overhead of that interrupt to the hypervisor, the other is to increase the mode change overhead between VM mode and native mode. This paper presents new methods for scheduling the waiting logical processors to solve these problems. That is, one is for detecting an interrupt pending without interrupting to the hypervisor, and the other is for dispatching the waiting logical processors in response to the detection. Moreover, when no workload is found, the hypervisor scans the waiting logical processors, finding a workload by using no host I/O interrupt. Thus, these methods have solved those problems, and enhanced the performance of the VMS to near native performance, even when the CPU utilization of the workload is low, as well as when it is high.

1. はじめに

仮想計算機システム（Virtual Machine System：VMS）は，1台の実計算機システム（以下，ホスト実計算機システムという）下で複数台のオペレーティングシステム（OS）を同時に動かすことができる．このVMSは，メインフレームの分野では，広く使用されるに至っている[1]．パソコンの分野でもvmware[2]のように複数のUNIXやWindowsを同時に動かすものが出荷されてきている．仮想計算機（Virtual Machine：VM）は，ホスト実計算機と同等のアーキテクチャまたはその部分を有する論理的な計算機である．VMSにおいて，このVMが複数台定義され，それらは，同時に動作し，各VM上で実計算機上と同じOSが動

† 熊本大学工学部数理情報システム工学科
Department of Computer Science, Faculty of Engineering, Kumamoto University
†† 株式会社日立システムアンドサービス
Hitachi System & Service, Ltd.
††† 株式会社日立製作所
Hitachi, Ltd.

作する[3),4)]．したがって，複数台の OS がホスト実計算機上で同時に走行することになる．

ここでいうホスト実計算機は，ユニプロセッサ（UniProcessor：UP）または対称型マルチプロセッサ（Symmetric MultiProcessor：SMP）といわれるものである．この UP は，1 台の実プロセッサを持つ．一方，SMP は，複数台の実プロセッサを持つ．各実プロセッサは，同等機能を持ち，主メモリを共有する．同様に，VM も 1 台または複数台の論理プロセッサを持ち，それらの論理プロセッサは，その VM のメモリ領域を共有する．論理プロセッサは，実プロセッサに相当し，それとほぼ同等のアーキテクチャを有する．

本論文は，VMS の運用形態として複数台の VMs を日常業務用に運転している形態を前提とし，そのような運用形態でのシステム性能向上方式を提案する．

ハイパバイザは，仮想計算機システム全体を制御するソフトウェアである．ハイパバイザは，各 VM へ実リソース（実プロセッサ，実メモリ，I/O 系）を割り当てる．ハイパバイザは，論理プロセッサに実際に実プロセッサを割り当て，その論理プロセッサを走行させる．すなわち，ハイパバイザは論理プロセッサをスケジュールする．このスケジューリングは，ハイパバイザの構成要素であるスケジューラ（以下，単にスケジューラと呼ぶ）によって行われる．

この VMS における論理プロセッサのスケジューリングは，通常の OS におけるプロセスのスケジューリング[5)] に相当する．この従来のプロセス・スケジューリングから，容易に，以下のハイパバイザにおけるスケジューリング方式を導くことができる．すなわち，スケジューラは各論理プロセッサに 3 つの状態を与える．1 つは走行状態であり，これは論理プロセッサが実際に実プロセッサ上で走行している状態を表す．2 つ目は待ち（wait）状態であり，これは論理プロセッサが自らある事象の発生を待っている状態である．3 つ目は実行可能（ready）状態であり，これは論理プロセッサは走行可能であるが実プロセッサの割当てを待っている状態である．従来，I/O 等の割込みが走行状態の論理プロセッサに対して発生したときは，大部分は，ハードウェアにより直接実行される[4),9)]．したがってこの場合の割込みは，オーバヘッドとはならない．

待ち状態の論理プロセッサは，対象となる事象が発生するのを待っている．従来，この事象はハイパバイザへの割込みを契機として発生していた．一般に，ハイパバイザへ報告する割込みをホスト割込みという．このホスト割込みには，ハイパバイザ自身が設定したタイマの割込みや，ハイパバイザ自身が設定した I/O の割込み等がある．一方，論理プロセッサ自身の設定したタイマの割込みや，論理プロセッサ自身の I/O の割込みは，その論理プロセッサが走行状態であるときは，直接，すなわち，ハイパバイザを介することなく，その論理プロセッサに報告されることが多い[4)]．

しかし，論理プロセッサが待ち状態のときは，従来，これらの割込み（最終的には論理プロセッサに報告すべきもの）はホスト割込みとなっていた．それは，ハイパバイザが該割込み発生を認識し，該論理プロセッサの状態変更をしなければならないからであった．すなわち，この場合，ホスト割込みが発生すると，スケジューラ（ハイパバイザの一部）は該事象発生を認識し，該論理プロセッサに該割込みを報告（すなわち，割込みシミュレーション）し，その待ち状態を解除し，実行可能状態に変える．これは通常の OS において，プロセス・スケジューラが待ち状態のプロセスを事象発生を契機として実行可能状態に変えるのと同様である．このような従来のスケジューリング方式は以下の問題点を持っている．

問題点 1. 上記割込みシミュレーションのオーバヘッド

問題点 2. VM モードとネイティブ（native）モードとの間の移行オーバヘッド

問題点 1 について：上記の待ち状態論理プロセッサへの割込みは該当の VM に対するシミュレーションオーバヘッドとなる．VM 上の OS に I/O 命令を多発する負荷をかけて，その CPU 利用率が低いときは，その論理プロセッサが待ち状態のときに I/O 割込みが多発する可能性が高い．このような負荷のとき，その I/O 割込みシミュレーションオーバヘッドはネイティブに対して 20〜30%に及ぶことがある[10),11)]．これにより CPU 速度は実質的に低下し，I/O スループットが半分以下になることもある[12)]．さらに，このオーバヘッドは個々の VM に付随するので，同時に走行している他の論理プロセッサに与える CPU サービス量が減少する．したがって，システム全体の性能を低下させることとなる．

問題点 2 について：ハイパバイザは，ネイティブモード，すなわち，通常のように唯一の OS を走行させるモードで走行する．それに対して VM は VM モードで走行する．これは単なるソフトウェア上のモードだけではなく，ハードウェアとしてサポートされているモードである．両者間のモード移行には通常の命令実行時間の 100 倍程度の実行時間がかかる．それは，ハードウェア的に多くの状態設定（制御レジスタの切替えやハードウェア内部レジスタの切替え等）がなさ

れるからである．このモード移行は論理プロセッサのスケジューリングにおいて一般的に発生する．すなわち，論理プロセッサがタイムスライス（CPU 時間）を使い切るとタイムスライスエンド割込み（ホスト割込み）が発生し，スケジューラはこれを契機として論理プロセッサを切り替える．このときのモード移行オーバヘッドは，タイムスライスが 1〜10 ms であるためその頻度が低く，ほとんど無視できる．しかし，I/O 割込み等（VM に反映すべきもの）が頻繁にホスト割込みとして発生すると，モード移行頻度が高くなる．そうすると，そのオーバヘッドは 1 回あたりのコスト（CPU 時間）が大きいため無視できなくなってくる．

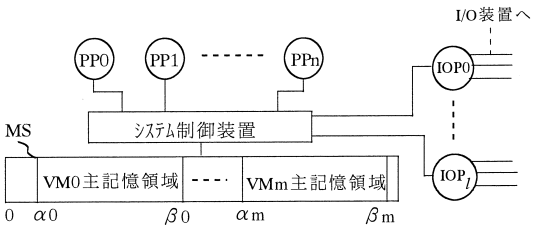
このような問題点は，スケジューラが上記のような方式（これをホスト割込み方式という）をとる限り避け難いものである．本論文の目的は，仮想計算機システムにおいて，待ち状態の論理プロセッサに関して新しいスケジューリング方式を提案し，上記の 2 つの問題点を解決することである．すなわち，(1) ホスト割込みを発生させることなく，待ち状態論理プロセッサに報告すべき割込み要因（主に I/O 割込み要因）を検出する方式を与え，(2) それを検出したときに待ち状態論理プロセッサをディスパッチ（dispatch）する方式を与えるものである．これによって，上記 2 つの問題点を解決し仮想計算機システムの性能を向上させることができる．

本論文では，2 章で従来のスケジューリング方式とその問題点を述べる．3 章でそれらの問題点を解決するための新しいスケジューリング方式を提案する．4 章で関係する従来技術と関係するアーキテクチャを述べ，5 章で実験的評価を与える．6 章で結論を述べる．

2. 従来のスケジューリング方式とその問題点

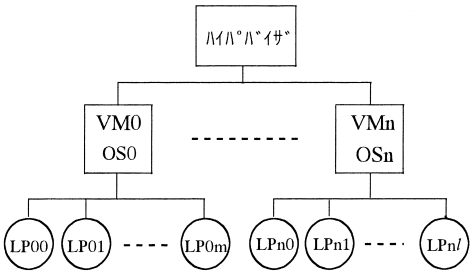
本論文で前提とする実システムの構成を**図 1** に示す[4]．この図においては，実プロセッサ PPi (i = 0, 1, ···) がシステム主記憶 MS を共有している．実プロセッサは 1 台だけでもよい．また，I/O プロセッサ IOPi (i = 0, 1, ···) も示されている．実プロセッサの台数や I/O プロセッサの台数は任意である．同図に示すように仮想計算機 VM の主記憶領域はシステム主記憶 MS の領域を分割して使用するものとする．この図においては，仮想計算機 VMk の主記憶の下限アドレスは α_k であり上限アドレスは β_k $(k = 0, 1, \cdots, m)$ である．仮想計算機 VM は，主記憶 MS のサイズの許す範囲で複数台定義することができる．このような分割領域を持つ VM は通常業務運用に使用する．

本論文で前提とする仮想計算機システム（VMS：Virtual Machine System）の構成を**図 2** に示す[6]．この図においては，仮想計算機 VM0, VM1, ···, VMn が定義されており，各 VMi は，論理プロセッサ LPi0, LPi1, ··· を含む．各仮想計算機 VM は，図 1 に示す主記憶領域を持つ．また，VM は 1 台または複数台の論理プロセッサを持ち，それらの論理プロセッサは，当該 VM の主記憶を共有する．VM あたりの論理プロセッサ台数は任意であり，また VM ごとに論理プロセッサの台数は異なってもよい．



PPi： 実プロセッサ- i i=0, 1, ···, n
IOPj： I/O プロセッサ- j j=0, 1, ···, *l*
MS： システム主記憶装置（Main Storage）
VM_k： Virtual Machine（仮想計算機）-k k= 0, 1, ···, m
α_k, β_k :VM_kの主記憶の下限、上限アドレス

図 1 ホスト実計算機システムの構成
Fig. 1 Configuration of host real computer system.



VMi: Virtual Machine - i, i=0, 1, ···, n
OSi: Operating System - i, i=0, 1, ···, n
LPij: Logical Processor - j of VMi

図 2 仮想計算機システムの概念図
Fig. 2 Concept of virtual machine system.

本論文は，VMS の運用形態として複数台の VMs を日常業務用に運転している形態を前提とし，そのような運用形態でのシステム性能向上方式を提案する．日常業務運用では特に実計算機に近い性能（あるいはそれ以上のシステム性能）が要求される．もちろん，性能的に許せばテスト用の VMs が何台か動いていてもよい．

ハイパバイザ（hypervisor）は，仮想計算機システム全体を制御するプログラムであり，各 VM に実プロセッサや実メモリ等の実資源を分け与え，複数台の VM をスケジュールする．ハイパバイザは，通常のネ

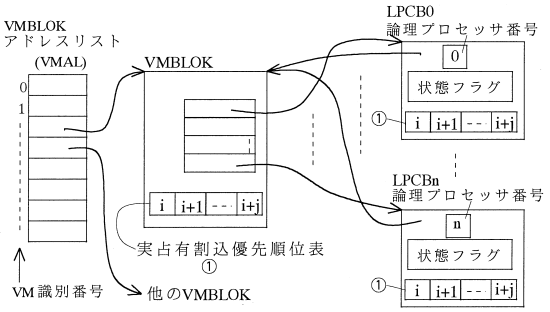


図 3 VM 関連制御ブロックの構成例
Fig. 3 Example of VM control blocks.

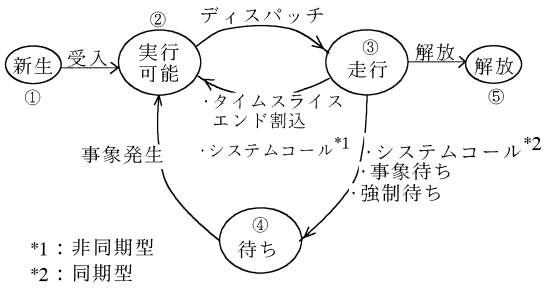


図 4 OS における典型的なプロセススケジューリング方式
Fig. 4 Traditional process scheduling method in OS.

イティブモードで走行するが，各 VM の論理プロセッサを動かすときは該実プロセッサは VM モードにおかれる．これは，以下の理由による．

たとえば，論理プロセッサが自分自身全体を待ち状態にするような命令を発行したとする．この命令はネイティブモードでは，実システム全体を待ち状態にさせるような命令である．1 つの論理プロセッサがホスト実計算機全体を待ち状態にすることは避けなければならない．このため，VM モードでかかる命令が発行されたときは，ハードウェアは当該命令の実行を抑止し，ハイパバイザへ制御を渡す．これを契機として，ハイパバイザは，当該論理プロセッサを待ち状態とし，他の論理プロセッサに制御を渡す．このモード移行のためには通常大きな実行時間が必要となる．その時間は通常の命令実行時間の 100 倍程度の実行時間がかかる．それは，ハードウェア的に多くの状態設定がなされるからである．

また，**図 3** に仮想計算機 VM の制御ブロック VMBLOK およびその論理プロセッサの制御ブロック LPCBi $(i = 0, 1, \cdots, n)$ の構成例を示す．これは制御ブロック構成例を示すものであり，従来技術より容易に推定できるものである．VMBLOK のアドレスリスト VMAL は VM 識別番号 0, 1, 2, $\cdots$ の順番に該当する VMBLOK のアドレスを含む．これらの制御ブロックはハイパバイザにより作成・管理される．また，各論理プロセッサ制御ブロック LPCB は状態フラグを持ち，該当論理プロセッサの状態を表す．ハイパバイザは，各論理プロセッサの状態を管理し，その状態に従って各論理プロセッサをスケジュールする[6]．図 3 の実割込み優先順位①については 3.3 節で説明する．

スケジューリング方式は，従来の OS における通常のプロセススケジューリング方式に準ずる方式をとる．そこで，従来の OS における代表的なプロセススケジューリング方式を**図 4** に示す[5]．すなわちプロセスは新しく生成されて新生（New）状態（①）となり，それをシステムが受け入れる（admit する）と実行可能状態（ready：②）となる．実行可能状態のプロセスをディスパッチ（dispatch）する（実際に実プロセッサを割り当てる）ことにより走行状態（running：③）となる．走行状態のプロセスが与えられた CPU タイムスライスを使い切るとタイムスライスエンド割込み（time slice end interrupt）によって実行可能状態にされる．また，走行状態のプロセスが同期型システムコール（System-call）を発行すると該当処理が完了するまで該プロセスは待ち状態（wait：④）にされる．非同期型のシステムコールのときは当該プロセスは実行可能状態（ready：②）におかれる．走行状態のプロセスが何かの事象発生を待つ（事象待ち）場合や何もすることがなくなると（強制待ち），そのプロセスは待ち状態に置かれる．待ち状態のプロセスに対して該当の事象が発生すれば（事象発生）該プロセスは実行可能状態となる．走行中のプロセスがその処理をすべて完了すれば解放（release）されて解放状態（⑤）となる．

仮想計算機システムにおけるスケジューリング方式も論理プロセッサを通常 OS でのプロセスと見なすことにより同様に実現できることは明らかである[6]．しかし，以下の主要な相違点がある．

(1) 割込み直接実行時の論理プロセッサの走行状態の継続：論理プロセッサが走行中に割込み要因が発生したとする．それが該走行中の論理プロセッサに属するものである場合は，該論理プロセッサに反映しなければならない．その反映はハードウェアで直接実行される場合と，ハイパバイザでシミュレーションされる場合がある．前者の場合は当該論理プロセッサは走行状態のままである．後者の場合もハイパバイザは割込みシミュレーション後，直前に走行中であった論理プロセッサを優先してディスパッチするので，当該論理プ

ロセッサは走行状態となる．他の論理プロセッサに属する割込みが発生したときはホスト割込みとなることがあり，そのときは，従来どおり優先度による切替えが発生しうる．

一方，一般OSのプロセス・スケジューリングの場合は，走行中のプロセスに割込みが発生した場合は，そのプロセスは，中断されて実行可能状態にされる．これは，優先度に基づくプロセス切替えの可能性があるからである．割込みが発生すると，まずコントロールがOSのカーネル部分の割込みハンドラに渡され，さらにスケジューラに渡される．そこで，当該割込みに関する事象を待っている待ち状態のプロセスが実行可能状態に更新される．その後，優先度に基づきプロセスがスケジュールされる．

(2) 論理プロセッサへの割込み直接反映：ハードウェアは割込み要因発生時アーキテクチャとして以下の割込み動作を行う．すなわち，CPUの現在のPSWレジスタの内容（Processor Status Word：プログラムカウンタや状態制御フラグを含む[7]）をスタックに退避し，該当のPSWを割込みベクトルテーブルから読み出してPSWレジスタに設定する．この後，制御が当該OSの割込みハンドラへと渡される[7]．論理プロセッサはネイティブマシンと同一アーキテクチャを持つので論理プロセッサに割込み可能な割込み要因が発生したときは上記のハードウェアとしての割込み動作をシミュレーションしなければならない．このシミュレーションは，従来より，ハイパバイザにより実施されるか，または走行中の論理プロセッサの割込み要因については，VM専用の割込み制御論理によって直接実施されるかしている[4]．このように論理プロセッサへの割込みはハードウェアにより該当論理プロセッサへ直接反映されうる．

一方，一般のOSの場合，割込みハンドラでの処理が完了後，プロセス（実行可能状態の）へ制御が渡る．このようにプロセスに割込みが直接反映されることはなく必ずOSの割込みハンドラ経由である．

(3) 論理プロセッサの待ち状態の要因：業務運用でハイパバイザが論理プロセッサを待ち状態にすることはほとんどない．また，論理プロセッサ上のOS配下のどれかのプロセスが走行中であれば論理プロセッサ全体としても走行状態である．さらに，同一VM下の論理プロセッサ間の同期はシステム立ち上げのときやロック（lock）命令等での排他制御だけである．さらに，異なるVM間は通常独立であり，同期待ちはない．したがって，論理プロセッサは，大部分は自らの原因で待ち状態となる．すなわち，多くは，論理プロセッサ上のOSが何もすることがなくなってシステム待ち状態（system wait）になったり，システムアイドル（system idle）状態となったりする場合，当該論理プロセッサは待ち状態となる．この待ち状態を解除する事象はほとんど当該論理プロセッサに対する割込み発生である．従来は，スケジューリング方式として，図4に示すスケジューリング方式を取っているので，まず割込み（待ち状態の論理プロセッサに属する割込み）をハイパバイザで受けて，そこで割込みを該当の論理プロセッサにシミュレーションしなければならなかった．これにより，当該論理プロセッサの待ち状態を解除し，実行可能状態としていた．

一方，通常のOSにおけるプロセスの待ち状態は割込み発生のほかにプロセス間の同期事象，排他事象の発生に対する待ち状態である．OSのプロセススケジューラは，プロセスに対して，きめ細かな待ち要因管理を行う[5]．

さて，以上によって分かるとおり，従来の図4におけるスケジューリング方式においては以下の2つの問題点が発生する．

問題点1：待ち状態の論理プロセッサを実行可能状態に移行させるにはハイパバイザへの割込みを契機とし，それによる該論理プロセッサへの割込みシミュレーションを必要とする．したがって，I/O負荷の高いジョブを走らせると，ホスト割込みが頻発し，この割込みシミュレーションのオーバヘッドが大きくなること．

問題点2：VMモードとネイティブモードとの間の移行オーバヘッドが大きくなること．上記のホスト割込みが，VMモード中（ある論理プロセッサ走行中）に発生すると，ネイティブモードへの移行が発生する．さらに，再び該論理プロセッサ（または別の論理プロセッサ）をディスパッチすることにより，再びネイティブモードからVMモードへ移行することになるからである．このモード移行には多くの実行時間（通常命令の100倍程度）が必要である．このため，ホスト割込みの頻度が高いとモード移行の頻度が高くなりオーバヘッドが大きくなる．次の章でこれらの問題点を解決するための新しいスケジューリング方式を提案する．

3. 新しいスケジューリング方式の提案

前章で指摘した2つの問題点を解決するために，本論文では以下の4つからなる新しいスケジューリング方式を提案する．

(1) 自己待ち（self-wait）状態の導入

(2) 自己待ち状態の論理プロセッサに対する割込み

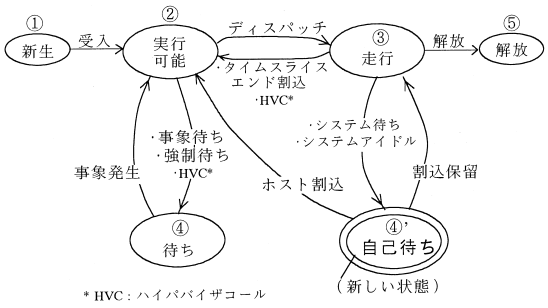


図 5 論理プロセッサ Self-wait 状態の導入
Fig. 5 Self-wait state of logical processors.

保留の検出方式

(3) 自己待ち状態の論理プロセッサをディスパッチする方式

(4) 2 階層活性待ち（active wait）方式

これらについて以下に述べる．

3.1 自己待ち状態の導入

走行中の論理プロセッサが待ち状態になるのは 2 つの場合がある．1 つはハイパバイザのコマンドにより停止させられた場合（前者）であり，他の 1 つは自ら何もすることがなくなって該論理プロセッサ上の OS がシステム待ち状態となるか，またはシステムアイドル（system idle）となり結果的に待ち状態となる場合（後者）である．仮想計算機 VM を業務運用システムとして運用しているときは，ハイパバイザのコマンドは投入しないので大部分は後者の場合である．しかし，開発用，テスト/デバッグ用の VM が複数台同時運用されている場合は前者による待ち状態も発生し，その状態の論理プロセッサも多くなる可能性がある．

これら 2 つの待ち状態を区別するために，**図 5** に示すように，後者の待ち状態を自己待ち状態（④'）として分離する．その他の状態（①〜⑤）は従来どおりである．自己待ち状態の論理プロセッサは，独立な「自己待ちキュー」（self-wait queue）を構成する．本論文では自己待ち状態の論理プロセッサを走査し，その割込み要因を検出する方式をとる．この検出を速くし，割込み遅延を防止する（3.4 節に後述するように自己待ちキューを実行可能キューよりも優先することによりゲスト OS の割込み認識遅延を防止する）ために独立な自己待ちキューを構成する．

割込み保留要因としては，I/O 割込み要因とタイマ等の外部割込み要因がある．これらの割込み要因は，通常の OS のプロセススケジューリングにおいては，ハードウェアの割込みによって検出される．しかし，仮想計算機システムにおいては，非走行論理プロセッサに反映すべき割込みが，ハイパバイザへの割込みとなりオーバヘッドが増加する．そのため，本論文では，ハードウェア命令によって自己待ち状態の論理プロセッサの割込み保留要因を検出する機構と，それを用いたスケジューリング方式を提案する．この方式は仮想計算機システムでのみ意味（性能向上という意味）があり，ネイティブモード（唯一の OS で動かす方法）においては何ら意味を持たない．

3.2 走行中 VM に対する I/O 割込み直接実行方式（従来方式）

従来より，VM の性能を実計算機に近づけるために，走行中の VM に属する I/O 命令や I/O 割込みはハードウェアによって直接実行されてきた[4]．本論文は，この I/O 直接実行方式を前提とし，さらに，自己待ち状態にある論理プロセッサに対しても，スケジューリング待ちはあるが，この I/O 割込み直接実行を適用させようとする方式を提案する．この提案方式をより明らかにするために，以下に，この従来の I/O 割込み直接実行方式（走行中論理プロセッサに属する I/O 割込みに対する方式）について簡単に述べる[4]．

従来より行われている走行中の論理プロセッサに対する I/O 割込み直接実行方式の特徴は，以下のとおりである．

(1) VM の割込み保留要因をハードウェア論理がソフトウェアを介さずに認識すること．

(2) VM の割込み保留要因に対する該当 VM 自身の割込み可能性をハードウェア論理がソフトウェアを介さずに判断すること．これは，走行中の論理プロセッサに対する判断だけである．

(3) 割込み可能であれば該論理プロセッサ（走行状態）にハードウェア論理がソフトウェアを介さずに直接当該割込みを反映すること．

これを実現するための代表的な法式を以下に説明する．まず，上記 (1) について説明する．ハードウェアは，通常，I/O 割込み要求をその優先順位別に保留する．そこで，I/O 割込み優先順位を各 VM に排他的に割り当てることにする[4]．たとえば，I/O 割込み優先順位を $0, 1, 2, \cdots, n$ とするとき，優先順位 1 と 2 を VM0 に割り当て，優先順位 3 と 4 と 5 を VM1 に割り当てるという具合にする．この割当ては，ユーザ指定に基づき，各 VM 起動時までにハイパバイザが行う．このような排他割当てを行うことにより，たとえば，優先順位 i に割込み要因が存在するとき，それは，その占有元の VM の I/O 割込み要因であることが分かる．そこで VM に占有させている優先順位を示すレジスタを用意すれば，VM の I/O 割込み保留要因をハードウェアとして認識できるようになる．

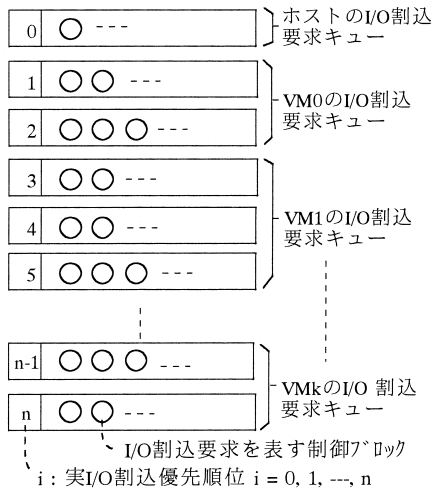


図 6 VMS における I/O 割込み要求キュー（従来例）
Fig. 6 I/O interrupt request queues in VMS (conventional).

この実装例を，**図 6** に示す．すなわち，I/O 割込み要求は，優先順位 i $(i = 0, 1, 2, \cdots, n)$ 別にキューイングし，割込み優先順位 0（最高位）は，ハイパバイザに割り当て，残りを，各 VM 間で排他的に割り当てる．優先順位 1 と 2 を VM0 に割り当て，優先順位 3 と 4 と 5 を VM1 に割り当て優先順位 $n-1$ と n を VMk に割り当てている．このキューは主記憶に構成してもよく，別の I/O プロセッサ側に構成してもよい．

次に上記 (2)，(3) を実現するためには VM 用割込み制御論理が必要である．このためには，少なくとも，走行中の VM の I/O 割込み制御マスクレジスタと当該 VM 自身の I/O 割込み優先順位に関する割込み制御マスクを反映するレジスタが必要である．この制御論理はいろいろなものがある．たとえば，各 VM にただ 1 つの実割込み優先順位しか与えない方式や，各 VM に複数個の実割込み優先順位を与える方式や，あるいは，各 VM にすべての実割込み優先順位を与える方式がある．詳細は文献 4) を参照されたい．

注意すべきは，この従来方式は，ある実プロセッサ上で走行中の論理プロセッサに反映すべき I/O 割込みに対してだけ適用されるということである．本論文はこの I/O 割込み直接実行を前提とし，さらに，それを自己待ち状態の論理プロセッサにも適用するためのスケジューリング方式を提案する．

3.3 自己待ち状態の論理プロセッサの I/O 割込み保留要因の検出方式

ゲスト OS が走行中でないときにも，当該 OS に反映すべき I/O 割込み要求は発生しうる．その頻度は該 OS 上に I/O ヘビー（heavy）な負荷をかけた場合は高くなる可能性がある．そのような I/O 割込みは，従来はホスト I/O 割込みとしてハイパバイザに入り，ハイパバイザがソフトウェア的に該 I/O 割込みを該当論理プロセッサに反映してその状態を実行可能状態にしていた．

本論文で提案する方法は，ホスト I/O 割込みを発生させない．このために，ホスト用の割込み優先順位マスクレジスタの中で，どれかの VM に占有させている実割込み優先順位（図 6 では VM0 は 1，2；VM1 は 3，4，5 等）のホストマスク値はつねに 0 とする．これは，VM が走行中も，そうでないときも，あるいはハイパバイザが走行中もつねにそのようにするということである．これにより，実占有割込み優先順位のホスト I/O 割込みの発生を防止できる．その代わり，実占有割込み優先順位の I/O 割込み要因は，占有元の VM の論理プロセッサをディスパッチしたときにハードウェアにより直接実行させるようにする．

以上のマスク設定により実占有割込み優先順位に対するホスト I/O 割込みを防止できる．しかし，そのために，論理プロセッサが I/O 割込み待ち等により自己待ち状態に陥ると，そのホスト割込みが発生しないため，そこから抜け出すことができなくなる．そのため，自己待ち状態から抜け出すための新たな方式が必要となる．従来，スケジューラは，待ち状態または自己待ち状態の論理プロセッサはディスパッチしない．それは，従来の OS のプロセススケジューラの考え方からすれば当然である．また，無理矢理それら待ち状態の論理プロセッサをディスパッチしても何もすることはないので，すぐにまた待ち状態となってしまい，スケジュールオーバヘッドを増やすことになるからである．

仮想計算機システムにおいて，スケジュール対象は論理プロセッサであり，これは実計算機と同等のアーキテクチャを持つ．論理プロセッサは，自己待ち状態に遷移後，システム活動の結果，I/O や外部割込み要因を内部に保留する（ハードウェアアーキテクチャとして）ようになる可能性がある．そのような論理プロセッサは，自己待ち状態であってもディスパッチするとただちに，ハードウェア動作として，その保留していた割込みの処理に入り，VM モードとしての動作・走行が可能となる．このことから，ハードウェア内部に保留されているゲスト I/O 割込み要因を検出する機能があれば，ハイパバイザは，その機能を使用して自己待ち状態の論理プロセッサに割込み要因があるかどうかを判定することができる．それがある場合は，

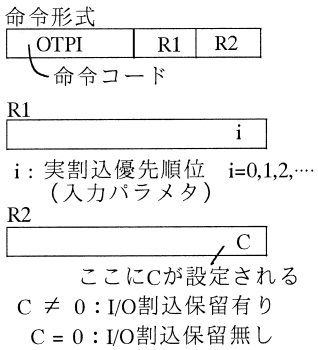


図 7 Only Test Pending I/O Interrupt (OTPI) 命令
Fig. 7 Only Test Pending I/O Interrupt (OTPI) instruction.

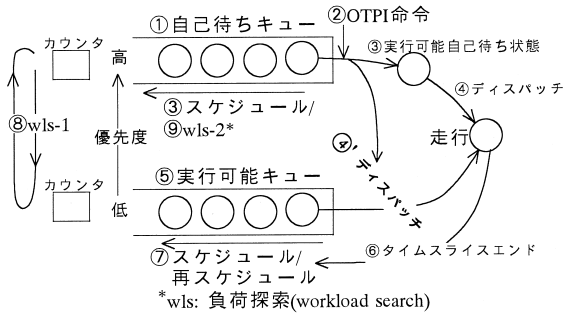


図 8 自己待ちキューを用いたスケジュール方式
Fig. 8 Scheduling with self-wait queue.

ハイパバイザは該論理プロセッサをディスパッチ対象とする．

さて，3.2 節で述べたように，従来，ハードウェアが I/O 割込み直接実行機能を持つときは，各 VM の I/O 割込み保留要因は，ハードウェアによって認識される．そのために I/O 割込み優先順位を VM 間で排他的に割り当てることは前述のとおりである．そこで，**図 7** に示す Only Test Pending I/O Interrupt（OTPI）命令を提案する．この命令は，実割込み優先順位 i $(=0,1,\cdots,n)$ をオペランド（レジスタ R1 に含まれる）とし，その優先順位 i の割込み保留要因が存在するかどうかを調べる．存在するときは非 0 の値を R2 に設定し，存在しないときは 0 を R2 に設定する．この命令自体は，ハードウェアが I/O 割込み要因を $0,1,\cdots,n$ の優先順位ごとに保留する機能を持つ場合は，どのアーキテクチャでも定義可能である．

ハイパバイザは自己待ち状態の論理プロセッサに対して，その実占有割込み優先順位を求め，この命令を使用して該論理プロセッサが I/O 割込み保留要因（その実占有割込み優先順位に対するもの）を持つかどうかを判定することができる．図 3 に示すように，各論理プロセッサ制御ブロック LPCB は，内部に実占有割込み優先順位表（図 3 の①）を有しているので，ハイパバイザはその順位表にアクセスして，この命令の入力オペランドを得ることができる．複数個占有しているときは各々について調べ，どれかについて，I/O 割込み保留要因が存在すれば，該論理プロセッサは I/O 割込み保留を持つと判断することができる．

この命令はテストするだけなので，簡単な機能であり，その実行時間はホスト割込み処理に比べれば半分以下である．なぜなら，ハードウェアの割込み処理は一般には割込みキューの操作や割込み用スタックの操作を含むからである．このようにして，ハイパバイザは，ホスト I/O 割込みによらず，自己待ち状態の論理プロセッサが I/O 割込み要因を持つかどうかを判断することができる．自己待ち状態の論理プロセッサが I/O 割込み保留要因を持つとき，該論理プロセッサは「実行可能自己待ち」（self-wait ready）状態であると呼ぶ．

ハードウェアは，通常，図 6 に示すような I/O 割込み要求キューを持つ．このような場合は，この命令ついて，以下のように実装することができる．すなわち，R1 によって指定された優先順位 i のキューを調べ，そこにキューエントリ（すなわち割込み保留要因）あればレジスタ R2 に非 0 の値を設定し，存在しないときは 0 を設定すればよい．

この OTPI 命令は，従来の Test Pending Interrupt（TPI）命令[7] に類似している．この TPI 命令は，各割込み優先順位の割込み要因をチェックする点は同じである．しかし，もし，割込み要因が存在するときは，その割込み要因を刈り取ってしまう点が最も大きな違いである．すなわち，図 6 の I/O 割込み要求キューの I/O 割込み要求を示す制御ブロックをデキュー（dequeue）してしまうことである．これは TPI 命令は元々ネイティブ OS において割込みを刈り取るために使用される命令だからである．これに対してここで提案する OTPI 命令は指定割込み優先順位の I/O 割込み保留をテストするだけが目的のハイパバイザ専用命令であり，性能的にもデキューしない分 OTPI 命令の方が速い．

3.4 自己待ちキューを用いたスケジューリング方式

図 8 に自己待ちキューを用いた新しいスケジューリング方式を示す．スケジューラは，自己待ちキュー（①）の先頭の論理プロセッサから状態を調べ，「実行可能自己待ち」状態であればそれをディスパッチする．そうでないときは OTPI 命令（②）を用いて当該論理プロセッサの属する VM に占有されている実割込

み優先順位の割込み保留要因があるかどうかをチェックする．それがあれば当該論理プロセッサを「実行可能自己待ち」状態とし（③），それをディスパッチする（④）．なければ，自己待ちキュー内の次の論理プロセッサの状態を調べる．自己待ちキューをすべて調べてディスパッチ可能な論理プロセッサが見つからないときは，実行可能キュー（⑤）を検索に行く．そこにエントリがあればそれをディスパッチする（④'）．このように，スケジューラは，自己待ちキューを通常の実行可能キューより優先する．ここでも見つからないときは負荷探索（workload search）部分（⑧，⑨：3.5 節参照）へ行く．

スケジューラは，各走行中の論理プロセッサが与えられたタイムスライスを使い切るか，待ち状態になるか，または，自己待ち状態になるまで走行させる．走行中の論理プロセッサへの I/O 割込みは直接実行され該論理プロセッサは走行状態を継続する．非走行論理プロセッサへの I/O 割込みは前節で述べたように，その間ハードウェア的に抑止される．このように，スケジューラは，各論理プロセッサを走行させるとき，CPU タイムスライスの間，サービスを保証しようとする．これは論理プロセッサ上でネイティブモードのときと同等の大きさの負荷を持つ OS とその配下のプロセス群を走行させるからである．この意味で各論理プロセッサは同等の優先度を持つ．しかし，論理プロセッサがタイムスライスエンド（⑥）になったとき，ハイパバイザは，これまでのサービス量に基づき，該論理プロセッサのディスパッチ順序を更新し（⑦），サービス量の低い論理プロセッサを優先する．

また，もし，走行中の論理プロセッサがハイパバイザコール（hypervisor-call）を発行した場合は，スケジューラは，そのサービス後，発行元の論理プロセッサが走行可能であれば，ただちに，それをディスパッチする．もし，ハイパバイザコールの処理が完了するまで走行不可能であれば該論理プロセッサを一般の待ち状態とする．一般の待ち状態の論理プロセッサは，通常のプロセススケジューリングと同じようにハイパバイザへのイベント発生により実行可能状態に遷移させる（図 5 参照）．

自己待ち状態の論理プロセッサは，与えられたタイムスライスを使い切る前に自己待ち状態となっている．この理由により，スケジューラは，実行可能キューよりも先に，まず自己待ちキューを走査（scan）し，「実行可能自己待ち」状態の論理プロセッサがあれば，それをディスパッチする．自己待ちキュー内の論理プロセッサは，FIFO でサービスされる．これにより当該論理プロセッサは走行状態となり，その直後，当該割込み保留要因が当該論理プロセッサに対してハードウェア的に直接実行され，該論理プロセッサ上の OS の割込みハンドラに制御が渡る．

各 VM の優先順位は CPU サービス量制御を除き同等である．CPU サービス量制御とは論理プロセッサがタイムスライスを使い切ったときに，スケジューラは，これまでのサービス量に従って，そのサービス量の少ない論理プロセッサを優先するようにスケジュール順序を更新することである．したがって，各ゲスト OS の I/O 割込みの認識は自己待ちキューおよび実行可能キュー（ready queue）内の論理プロセッサの順番に従い遅れることとなる．しかし，この I/O 割込み認識の遅れは通常の OS 内でのプロセススケジューリングによる遅れと同等である．さらに，スケジューラは，自己待ち状態の論理プロセッサを実行可能状態の論理プロセッサより優先させて，このスケジューリングによる遅れを軽減する．これ以外の I/O 割込み認識遅れは，前節で述べたようにホスト I/O 割込み（本来ゲスト OS に報告すべきもの）の抑止により発生する．すなわち，各論理プロセッサに対して最低 1 タイムスライスの連続使用を保証する（該論理プロセッサが待ち状態にならない限り）ため，最大 1 タイムスライスの I/O 割込み認識遅れが追加される．しかし，これは，タイムスライスが 1〜10 ms 程度であり問題とはならない．

さらに厳しいリアルタイム性を要求するアプリケーションを走行させる VM は従来どおり実プロセッサを占有させる方式をとる．このときは該実プロセッサはつねに VM モードで実行させ，当該 VM はつねに実行中となり，その I/O 割込みはすべて直接実行されるので問題ない[6]．

3.5 2 段階活性待ち方式

マルチプロセッサシステムでの仮想計算機システムにおいて，ある実プロセッサ上で動かすべき論理プロセッサがなくなり，その負荷がなくなったときは，1 つの方法として，その実プロセッサは，すべての割込みマスクを開けてシステム待ち状態となり，ホスト割込みを待つという方法がある．これは，通常の OS のスケジューラでも行われていることである．しかし，この方法はシステム待ち状態中の他のプロセッサを起動するためのシグナルプロセッサ（signal processor）割込みが多くなり，そのオーバヘッドが大きくなるという問題点がある．このことから，従来から行われている別の手法として，負荷がなくなったときにすべての割込みマスクを開けて自プロセッサのキャッシュライ

ン（cache line）の中の事象発生表示ビット（bits）をチェックしながらループ（loop）する活性待ち方式[8]がとられている．この方法だと負荷を検出したプロセッサが該当（その負荷を処理すべき）プロセッサの事象発生表示ビットを 1（on）にするだけでよく，シグナルプロセッサの割込みは不要となる．

また，従来，1 つの実プロセッサへのホスト割込みにより負荷が発生したとき，すべての実プロセッサの事象発生表示ビット（bits）を 1 にしていた．これは，その負荷をどの実プロセッサで処理してもよいということからは妥当な方式である．しかし，一方で，負荷が 1 つしか存在しないときは，他の（該負荷処理以外の）実プロセッサはスケジューラを走行後，再び活性待ち状態となる．すなわち，スケジューラの空振りが発生しスケジュールオーバヘッドを増すこととなる．

これを避けるために別の従来方式もとられている．それは，負荷の個数をチェックする方法である．さらに，自プロセッサに負荷が存在しないときは他プロセッサの負荷の個数をチェックし，もし余分にあれば，そこからスチール（steal）して全体の負荷バランスを図るという方法もとられている[8]．

しかし，これらの従来方式は，ホスト割込みを入れることにより負荷を検出するという点では同じであり，シミュレーションオーバヘッドの増加をもたらす．このような従来方式に対して，本論文ではホスト割込みを発生させず，ハードウェア的にゲスト割込み保留要因を検出する方式を提案した．このホスト割込み禁止方式では，ハイパバイザはシステム待ち状態となることはできない．それは，ハイパバイザは割込み保留要因（VM に属するもの）の発生を検知しなければならないからである．このため，たとえすべての論理プロセッサの負荷が存在しなくなったときにもハイパバイザはシステム待ち状態になることはできない．したがって，本方式では，ある実プロセッサ上で動かすべき論理プロセッサがなくなり，その負荷がなくなったときは，その実プロセッサは，システム待ち状態とならず，負荷を探し続ける．この部分を負荷探索部分という．

従来の活性待ち方式では負荷を検出するためにすべての割込みを受け付ける．しかし，本方式では，実占有割込み優先順位に対する I/O 割込み要因は，ホスト割込みとするわけにはいかない．なぜなら I/O シミュレーションオーバヘッドを招くからである．そこで，本論文では，2 段階からなる負荷探索方式を提案する（図 8 参照）．第 1 ステップ（図 8 の⑧）では部分的に（すなわち，実占有割込み優先順位以外の割込み要因の）ホスト割込みマスクを開けて各キューカウンタ（queue counter）をチェックしながらループして，負荷の存在可能性を検知し，第 2 ステップ（図 8 の⑨）で実際の自己待ちキューを走査して負荷を検出し，スケジューラへ制御を渡す．これにより，一般の負荷は，第 1 ステップで検出し，実占有割込み優先順位に対する I/O 割込み要因は，ホスト割込みとすることなく，第 2 ステップで検出することができる．

負荷探索ステップ 1：負荷探索部分は，動かすべき論理プロセッサを探すために，実行可能キューのカウンタ（counter）と自己待ちキューのカウンタを調べ続ける．その間は，ホスト側の I/O 割込みマスクは以下のように設定する．すなわち，各 VM の実占有割込み優先順位についてはホスト割込みを発生させないようにホスト側のマスクを 0 に設定する．これにより，実占有割込み優先順位の I/O 割込み要因は，ホスト I/O 割込みとはならずハードウェア側へ保留される．これらはすべて該当の VM の I/O 割込み要因である．その他の割込み優先順位，すなわち，共有割込み優先順位については 1 に設定し，ホスト I/O 割込みを入れる．これは，テスト用 VM およびハイパバイザに与える割込み優先順位である．業務用 VM には通常 I/O 割込み優先順位を占有化させる．この負荷探索の走査ループ（scan loop）において，ホスト外部割込み等が発生した場合は，スケジューラは，図 5 に示すように，待ち状態，または，自己待ち状態の論理プロセッサを実行可能状態に遷移させる場合もある．また，他の実プロセッサで走行していた論理プロセッサが割込みを待って自己待ち状態に陥ることもある．このようにして，実行可能キューが非空となったときはスケジューラへ制御を渡す．実行可能キューが空（empty）で，自己待ちキューが非空となったとき，負荷探索は第 2 段階へと進む．また，自己待ちキューも空であるときは，負荷探索の先頭へ戻りループする．

負荷探索ステップ 2：負荷探索は，自己待ちキューを走査し，各論理プロセッサに対して OTPI 命令（図 7 参照）を発行し I/O 割込み保留要因を持つかどうかを調べる．持っている場合は，該論理プロセッサはディスパッチ対象となるので，そのフラグ（「実行可能自己待ち」状態フラグ）を該当の LPCB に設定する．このフラグを持つ論理プロセッサはスケジューラでディスパッチ対象となる．負荷探索は自己待ちキューのすべての論理プロセッサについてこのことを調べる．負荷探索が，ある実占有割込み優先順位について，I/O 割込み保留要因を検出したとする．その占有元の VM を VM0 とする．そのとき，その I/O 割込み保留要

因は複数存在しうるので，負荷探索は，自己待ち状態の論理プロセッサで当該 VM0 に属するものすべてに「実行可能自己待ち」状態フラグを設定する．

具体的には，負荷探索は，該当の論理プロセッサの制御ブロック LPCB（図 3 参照）より，その実占有割込み優先順位 $i, i+1, \cdots, i+j$ $(j \geq 0)$ を求め，これに対して，図 7 に示す OTPI 命令を発行する．これによって各実占有割込み優先順位について割込み保留要因が存在するかどうかを判断する．このうち，少なくとも 1 つの実占有割込み優先順位について割込み保留が存在するとき当該 VM の自己待ち状態の論理プロセッサすべてを「実行可能自己待ち」状態とし，ディスパッチの対象とする．図 3 に示す VMBLOK（VM を表す）と LPCB（論理プロセッサを表す）の関係を参照して，ある自己待ち状態の論理プロセッサから，その属する VM と，その VM に属するすべての論理プロセッサを容易に見つけ出すことができる．

I/O 割込みはどの論理プロセッサに反映してもよい．しかし，実行可能状態や走行状態の論理プロセッサよりも自己待ち状態の論理プロセッサに反映した方が全体の性能が良くなる．それは，VM 下の走行中の論理プロセッサの台数が増加するからである．

以上の自己待ちキューの走査においては，1 回の走査が完了するまでの間，ホスト割込みはすべて禁止し，そのキューをロックする．これは，走査の間，キューの構造を一定にしておかなければならないことと，走査完了後確実にロックをはずすためである．以上の負荷探索において負荷が検出されたとき，すなわち，自己待ちキューの中に「実行可能自己待ち」状態フラグを少なくとも 1 つ設定したときは，スケジューラへ制御を渡す．もし，負荷が検出できないときは負荷探索の第 1 ステップへ戻る．

4. 関係する従来技術および関係するアーキテクチャ

3 章で述べたスケジューリング方式のうち，自己待ち状態方式の導入とそのディスパッチ方式は，従来の通常の OS のスケジューリング方式にはない新しい方式である．それは，従来の OS の考え方では，プロセスは待ち状態から割込み等の事象発生により，割込みハンドラを経由して，実行可能状態へ移行して初めてスケジュールの対象となるのであり，待ち状態のプロセスをディスパッチするという考え方は従来ではなかった．また，活性待ち方式は従来より行われているが，その多くは，全割込みマスクを開けて事象（event）発生をチェックする割込み可能状態ビジーループ（enabled busy loop）である．それに対して，本論文では，実占有割込み優先順位については割込みマスクを閉じて割込み不可能にしている．すなわち本論文の活性待ち方式は部分的に割込み不可能にしてカウンタ（自己待ちキューと実行可能状態キューのカウンタ）をチェックするビジーループと，カウンタの値が正であることを検出したときのキュー走査からなる．このキュー走査は割込み禁止で該当キューヘッドをロックして行う．これにより，従来と同じく，シグナルプロセッサの割込みを発生させる必要はなく，実行可能キューにプロセス（論理プロセッサ）がキューイングされたときは，ただちに，どれかの活性待ち状態プロセッサが，その論理プロセッサをスケジュールすることができる．さらに，これに加えて，自己待ち状態の論理プロセッサに I/O 割込み保留要因が発生すれば，ホスト I/O 割込みを発生させることなく，それをただちに検出し，スケジュールすることができることが新規点である．

VM の割込み保留要因を調べる命令は IBM 社より特許[9] が出願されいる．しかし，その特許で開示されている命令は，割込み保留要因を調べるだけでなく，それを刈り取ってしまう機能も持っていることが示されている．すなわち，この特許はそのような命令の仕様を開示しただけであり，これを，ソフトウェアとして VM の制御用にどのように使用するかは何も開示していない．

本論文は 3.2 節で述べた I/O 割込み直接実行方式を前提としているが，その方式は S/390 をはじめとするメインフレームアーキテクチャで広く実現されている[4]．この I/O 直接実行方式は，S/390 と同様の VM 用のアーキテクチャのサポート，たとえば，VM 専用の割込みベクトル論理や割込み制御論理のサポートが必要であるが，他のアーキテクチャでも同様にサポート可能と考えられる．

提案方式の中でハイパバイザに I/O 割込み保留要因を検出する機能を与えているアーキテクチャは，公表されている範囲では，IBM 社の S/390[7] 型のアーキテクチャだけである．しかし，この機能は，I/O 割込み保留要因をその優先順位ごとに保留しているハードウェアであれば，同様に実装可能であると考える．これらのハードウェア機能があれば，その他の提案するスケジューリング方式は PC 上の vmware や virtual PC に対して適用可能である．

5. 実装と性能評価

5.1 実　　装

ハードウェア実装においては I/O 割込み保留要因

表 1 ホスト割込方式（従来）と割込保留検出方式（本提案）の性能測定値比較
Table 1 Performance comparison of host interrupt method with interrupt pending detection method.

#	負荷	項目	従来方式 (a)	本提案方式(b)	ネイティブモード(c)	b−a	(b−a)/a (%)	a/c (%)	b/c (%)	注釈
1	Batch_1 4800jobs Compile,Link,Go	ETR*1(Ended/sec)	4.67	4.73	4.31	0.05	1.18	108.47	109.75	Host:4way
		Simul. ovh.*2(%)	4.32	2.45	−	−1.87	−43.29			4 VMs
		Sched. ovh.*3(%)	27.70	2.48	−	−25.22	−91.05			各VM:4way
		経過時間*4(sec)	1,029.42	1,017.81	1,116.53	−11.61	−1.13			ρ^*=40%（VMモードから推定）
2	Batch_2 4000jobs Compile,Link,Go	ETR*1(Ended/sec)	8.08	8.11	8.61	0.04	0.47	93.76	94.21	Host:2way
		Simul. ovh.*2(%)	0.48	0.38	−	−0.10	−20.83			2 VMs
		Sched. ovh.*3(%)	0.62	0.38	−	−0.24	−38.71			各VM:2way
		経過時間*4(sec)	495.37	493.05	464.44	−2.32	−0.47			ρ^*=99%
3	TSS 100端末	ETR*1(Ended/sec)	51.60	53.57	53.66	1.97	3.82	96.16	99.83	Host:4way
		Simul. ovh.*2(%)	3.21	1.85	−	−1.36	−42.37			2 VMs
		Sched. ovh.*3(%)	15.38	1.77	−	−13.61	−88.49			各VM：4way
		測定区間*5(sec)	120.00	120.00	120.00	0.00				ρ^*=26%
4	I/O_heavy	ETR*1(EXCPs/sec)	12.65	16.34	16.90	3.69	29.15	74.87	96.69	Host:2way
		Simul. ovh.*2(%)	12.95	7.43	−	−5.52	−42.63			2 VMs
		Sched. ovh.*3(%)	20.17	5.79	−	−14.38	−71.29			各VM:2way
		測定区間*5(sec)	300.00	300.00	300.00	0.00				ρ^*=49%（VMモードから推定）
5	I/O_heavy	ETR*1(EXCPs/sec)	13.30	16.31	16.70	3.01	22.63	79.64	97.66	Host:4way
		Simul. ovh.*2(%)	13.86	8.95	−	−4.91	−35.43			2 VMs
		Sched. ovh.*3(%)	42.96	8.47	−	−34.49	−80.28			各VM:4way
		測定区間*5(sec)	300.00	300.00	300.00	0.00				ρ^* = 25%

Host Machine Cycle : 6.2ns
*: ρ : Native CPU utilization
*1 : ETR : External Throughput Ratio : 処理件数/実時間1秒
*2 : システム待ち命令(Load PSW wait)やホストタイマ（ゲストへ反映すべき）割込等のシミュレーション（CPU利用率）
*3 : 論理プロセッサをスケジュールするためのCPU利用率
*4 : 全ての投入されたジョブ(jobs)が終了するまでの時間
*5 : 測定データを取得した時間

を検出する命令（図 7 の OTPI 命令）を実現した．ソフトウェア実装においては，自己待ち状態や，ホスト I/O 割込みを抑制する割込みマスク設定方式や，2 段階構造を持つ活性待ち方式をサポートする新しいスケジューラを実装した．従来はホスト I/O 割込み（本来ゲスト OS に反映すべき割込み）によって待ち状態から実行可能状態に遷移していた．しかし，本方式では，ホスト I/O 割込みは発生させず，自己待ち状態の論理プロセッサに対して上記 I/O 割込み保留を直接検出し，それを直接ディスパッチすることができるようになった．この方式の評価を実測に基づき以下に述べる．**表 1** に従来のホスト割込み方式と本論文で提案した割込み保留検出方式の性能測定結果を載せる．以下に測定方法と測定データを論述する．

5.2 測定方法

本論文は，1 台の大型実計算機（Symmetric Multiprocessor）の下で日常業務運用として複数台の VMs を同時運用しているユーザを主たる対象としている．ユーザサイトでは非常に大きな規模のところもあるが，2-way〜4-way マルチプロセッサ（n-way マルチプロセッサというと n 個のプロセッサからなる SMP のこととする）の下で 2 台〜4 台の業務運用 VMs を同時に動かしているところも多い．

このようなユーザ使用状況を考慮して，ホストマルチプロセッサは 4-way マルチプロセッサシステムと 2-way マルチプロッセッサシステムの 2 種類を使用した．VM 下の論理プロセッサの台数は，実質的に同時に動く論理プロセッサの台数をできるだけ増やして性能を上げるために，実プロセッサの台数と同じにした．すなわち，ホストが 4-way の場合は 2〜4 台の VMs を同時走行させ，かつ，そのときの各 VM も 4-way（すなわち 4 台の論理プロセッサ）の構成とした．ホストが 2-way の場合は 2 台の VMs を同時走行させ，かつ，そのときの各 VM も 2-way（すなわち 2 台の論理プロセッサ）の構成とした．

1 つのケースを測定するのに準備時間を含めて 1〜3 時間かかる．測定した負荷の種類は少ないが実際の負荷と比較しうるだけの規模と属性を持つと考えられる．測定ツールはゲスト OS 内蔵のパフォーマンスモニタを用いた．表 1 に測定結果を載せる．以下に表 1

の各測定ケースを説明する．

(1) **ケース 1（表 1 での#1）**：このケースでは 4-way マルチプロセッサ下で 4 台の VMs を同時に走行させる．各 VM もまた 4-way すなわち 4 台の論理プロセッサを持つ．負荷はバッチジョブ群 Batch_1（Compile, Link, Go の 4,800 個のジョブ群）である．ネイティブの場合（表 1 の (c) の欄）は，これらのジョブ群をネイティブ OS のジョブキューに登録完了後，測定を開始した．VMS の場合（表 1 の (a) 欄と (b) 欄）は，各 VM（ゲスト OS はネイティブのものと同じ）に 1,200 個ずつのジョブ群をジョブキューに登録完了後，同時にスタートさせて測定した．このように，各 VM のシステムボリュームは分離してあること以外は，ネイティブと VMS の測定において全体のハードウェア量と負荷量は同じである．

(2) **ケース 2（表 1 での#2）**：このケースでは 2-way マルチプロセッサ下で 2 台の VMs を同時に走行させた．各 VM もまた 2-way すなわち 2 台の論理プロセッサを持つ．負荷はバッチジョブ群 Batch_2（Compile, Link, Go の 4,000 個のジョブ群）である．VMS の場合（表 1 の (a) 欄と (b) 欄）は，各 VM に 2000 個ずつのジョブ群を投入して測定した．その他はケース 1 と同様．

(3) **ケース 3（表 1 の#3）**：ケース 3 では 4-way マルチプロセッサ下で 2 台の VMs を同時に走行させた．各 VM は 4-way すなわち 4 台の論理プロセッサを持つ．負荷は会話型システム（TSS）で端末数は全体で（表 1 の (c) の欄の場合）100 端末である．VMS の場合（表 1 の (a) 欄と (b) 欄）は，各 VM に 50 端末ずつである．負荷として端末シミュレータを用いてトランザクション（edit コマンドおよびそのサブコマンド群）を生成し，それを入力して 10 分間〜14 分間測定し，その中の定常状態の 2 分間（測定区間）を取り出した値である．

(4) **ケース 4（表 1 の#4）**：2-way マルチプロセッサ下で 2 台の VMs を同時に走行させた．各 VM は 2-way すなわち 2 台の論理プロセッサを持つ．負荷は I/O_heavy と呼んでいるが，I/O マクロ命令である EXCP（Execute Channel Program）命令を高頻度に発行するジョブ群の連続実行である．その中の定常状態の 5 分間（測定区間）のデータを取得した．

(5) **ケース 5（表 1 の#5）**：4-way マルチプロセッサ下で 2 台の VMs を同時に走行させた．各 VM は 4-way すなわち 4 台の論理プロセッサを持つ．その他はケース 4 と同様である．

5.3 測定データ

(1) **シミュレーションオーバヘッド（CPU 利用率）**

シミュレーションオーバヘッドは各 VM に付随し，そのゲスト OS の発行した特権命令や，それに属する割込みに対するシミュレーションである．従来から主要なものはシステム待ち命令や，非走行論理プロセッサに属する I/O 割込みやタイマ系割込みシミュレーションである．シミュレーションオーバヘッドが高いと他の VM に与えるべき CPU 時間が減少し，全体の性能が低下する．

I/O ヘビーなジョブでシミュレーションオーバヘッドの減少は顕著である（#4：従来 対 提案 = 12.95% 対 7.43%，#5：13.86% 対 8.95%）．これは，自己待ち状態の論理プロセッサをディスパッチする方式によりホスト I/O 割込みが減少し直接実行となったためであると考えられる．走行中の論理プロセッサの I/O 割込みは従来方式でも直接実行されるため，全体の CPU 利用率が高い場合は従来方式でもシミュレーションオーバヘッドは低い（#2 の Batch_2：従来 対 提案 = 0.48% 対 0.38%）．今回はタイマ系の割込み（VM に属するもの）は走行中以外はホスト割込みとして発生するためそのシミュレーションオーバヘッドが提案方式においても発生していると考えられる．

(2) **スケジュールオーバヘッド（CPU 利用率）**

スケジュールオーバヘッドは論理プロセッサ間の切替えに要する CPU 利用率である．表 1 に示すようにスケジュールオーバヘッドは，I/O 割込みシミュレーションの減少に付随して，減少していると考えられる．

負荷の CPU 利用率（ネイティブモードでの値，以下同じ）が高い場合は従来方式でもオーバヘッドは低い（#2 Batch_2：従来 対 提案 = 0.62% 対 0.38%）．負荷の CPU 利用率が高い場合は従来方式でも各論理プロセッサがほとんどつねに実行可能状態であるため，負荷探索ルーチンはほとんど走行しない．さらに，I/O 割込みもほとんど直接実行される．このため，負荷の CPU 利用率が高い場合は，スケジュールオーバヘッドは低いと考えられる．

しかし，負荷の CPU 利用率が低い場合は従来方式ではオーバヘッドは高い（例：#5 I/O heavy：従来 対 提案 = 42.96 対 8.47%）．これは，負荷の CPU 利用率が低いときは負荷探索ルーチンが頻度高く走行することになるためである．提案方式では負荷の個数をチェックするため，プロセッサが無駄にスケジューラを動かすことは少ない．しかし，従来の活性待ち方式では負荷が 1 つでも発生すると全プロセッサに通知するため，スケジューラでの空振りが多くなることが 1

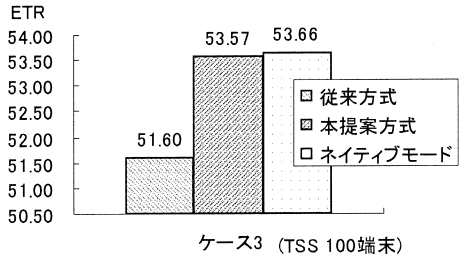


図 9 External Throughput Ratio（ETR）の比較（ケース 3）
Fig. 9 ETR comparison (case 3).

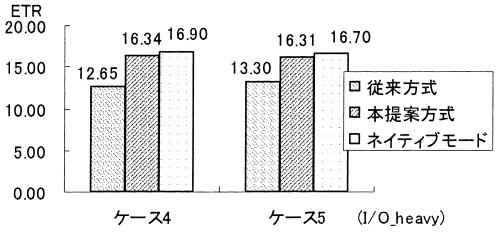


図 10 External Throughput Ratio（ETR）の比較（ケース 4 と 5）
Fig. 10 ETR comparison (case 4, 5).

つの原因と推定される．

負荷の CPU 利用率が低いときにスケジュールオーバヘッドが高いことは，以下の点で問題となる．たとえば，負荷の CPU 利用率が低いとき空きの CPU 時間を利用してハードウェアログ（hardware log）の解析・診断等を行うことがある．そのときに，スケジュールオーバヘッドのために，それができなくなるという問題がある．

(3) **ETR（External Throughput Ratio）**

この ETR は実時間 1 秒あたりに処理したトランザクション件数（ケース 1-2 ではジョブ処理件数，ケース 3 ではトランザクション件数，ケース 4-5 では EXCP 発行回数）であり，ユーザから見た性能となる．

ケース 1–2（表 1 の負荷：Batch_1，Batch_2）では従来方式の性能も高く本方式との差はわずかである．しかし，この場合は，経過時間の減少にその効果を見ることができる（Batch_1：11.6 sec 減少；Batch_2：2.3 sec 減少）．

ケース 3（表 1 の負荷：TSS 100 端末）では**図 9** に示すように明らかな性能向上を確認できる（従来 対 提案 = 51.60 対 53.57）．

ケース 4–5（表 1 の負荷 I/O_heavy）においても，**図 10** に示すように明らかな性能向上を確認できる（#4：従来 対 提案 = 12.65 対 16.34；#5：13.30 対 16.31）．これは，従来，ネイティブの 75〜80%であったが，本提案方式ではネイティブの 96〜97%にまで上昇しており，ほとんどネイティブに近い性能が得られていることを意味する．

この ETR の向上は，I/O 割込みシミュレーションオーバヘッドの減少とスケジュールオーバヘッドの減少によるところが大きい．

6. 結　　論

仮想計算機システムにおける論理プロセッサスケジューリング方式として自己待ち状態の論理プロセッサをスケジュールする方式を提案した．また，自己待ちキューの探索を含めた 2 段階負荷探索方式も提案した．これらの方式を実装し，その効果を測定した．自己待ち状態の論理プロセッサをスケジュールする方式により，ホスト I/O 割込み（実はゲスト OS に反映すべきもの）にともなう I/O 割込みシミュレーションオーバヘッドとスケジュールオーバヘッドを削減することができた．また，2 段階負荷探索方式によりスケジュールオーバヘッドを削減することができた．これらにより，ETR（External Throughput Ratio）を負荷の CPU 利用率が低いときも，高いときと同様に，ほとんどネイティブ性能に近くなるまで向上させることができた．

謝辞 本研究の推進にあたり株式会社日立制作所エンタプライズサーバ事業部ネットワークアーキテクチャ部チーフマネージャ下城孝様，日立電子サービス株式会社 CS 技術本部担当部長桑原敏憲様，EDMS 統括センタ推進部長檜山徹様には実装や性能測定や本論文の投稿にあたり多大のご協力をいただきました．ここに厚く御礼申し上げます．

参 考 文 献

1) IBM: Logical Partitioning. http://www-1.ibm.com/servers/eserver/iseries/lpar/documentation.htm
2) Sugerman, J., et al.: Virtulizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, VMware, Inc., *Proc. 2001 USENIX Annual Tech. Conf.*, Boston, Ma., USA (June 25–30 2001).
3) Goldberg, R.P.: Architectural Principles for Virtual Computer Systems, Ph.D. dissertation *Div. Eng. Appl. Phys.*, Harvard Univ., Cambridge, MA (1972).
4) 梅野，久保ほか：仮想計算機システムの高性能化方式，情報処理学会会誌，Vol.31，No.12，pp.1665–1680 (1990).
5) Stallings, W.: *Operating Systems: Internal and Design Principles*, 3rd Edition, Prentice

Hall (1998).
6) Umeno, H., et al.: Development of Methods for Reducing the Spins of Guest Multiprocessors, *Trans. IPS Japan*, Vol.36, No.3, pp.681–696 (1995).
7) IBM: z/Architecture Principles of Operation, SA22-7832-00.
8) Blandy, G.O., et al.: Active wait, US4631674, IBM, Filed (Feb. 5 1985).
9) Bean, G.H., et al.: Logical Resource Partitioning of a Data Processing System, IBM, USP4843541 (July 29 1987) Filed.
10) 池ケ谷，金子，梅野ほか：M シリーズ仮想計算機機構（VM/EX 機構）の開発・評価，情報処理学会第 36 回全国大会，pp.231–232 (1988).
11) 井上，梅野ほか：仮想計算機システムにおける I/O 直接実行方式の評価，情報処理学会第 38 回 (1989).
12) 江口，武永，小玉，梅野：仮想計算機の性能評価に関する研究，電気関係学会九州支部連合大会（第 54 回），p.678 (2001).

(平成 13 年 11 月 7 日受付)
(平成 15 年 1 月 7 日採録)

梅野　英典（正会員）

昭和 22 年生．昭和 45 年 3 月九州大学理学部数学科卒業．同年 4 月株式会社日立製作所中央研究所入所．昭和 51 年 8 月同システム開発研究所．平成 5 年 2 月同汎用コンピュータ事業部．平成 8 年 9 月同オフィスシステム事業部．平成 8 年 9 月博士（理学：東京大学）．平成 10 年 10 月熊本大学工学部数理情報システム工学科教授．研究分野：以下の性能向上と信頼性向上方式，オペレーティングシステム，仮想計算機，データベース管理システム，SMP クラスタ，最適化コンパイラ，Web サーバ，システムソフトウェアから見た計算機アーキテクチャ．ACM，IEEE Computer Society 会員．

久保　隆重（正会員）

昭和 17 年生．昭和 40 年 3 月京都大学工学部数理工学科卒業，昭和 42 年 3 月同工学研究科（数理工学専攻）修了．昭和 42 年 4 月株式会社日立製作所中央研究所入所．昭和 55 年 8 月同システム開発研究所．平成 2 年 8 月同副所長．平成 5 年株式会社日立西部ソフトウエア．平成 5 年 5 月〜7 年 4 月情報処理学会理事．平成 12 年 4 月株式会社日立システムアンドサービス．平成 14 年から日本 PKI フォーラム部会長．平成 14 年 12 月から日立システムアシスト株式会社．研究テーマ：計算機基本ソフトウェア，計算機システムの高性能・高信頼化，計算機用設計自動化，情報セキュリティと国際標準化，情報システムアーキテクチャー，情報システム開発技術等．

今田　豊寿

1960 年 7 月 9 日生．1983 年 3 月慶應義塾大学工学部数理工学科卒業．株式会社日立製作所エンタープライズサーバ事業部．サーバ製品の論理分割機能開発に従事．