複数プロセスで構成されるサービスに向けたスケジュール法

スカンヤ・スラナワラッ† 谷口秀夫†

既存の多くオペレーティングシステムが提供しているプロセスのスケジュールは、プロセスの動作内容に関係なく、実時間や時分割の処理といったプロセス実行の以前に得られる利用形態を基に、プロセスの実行を制御している。このため、プロセッサの有効利用を妨げ、プロセスの処理時間を無用に引き延ばしている場合がある。ここでは、プロセスの動作を予測し、その予測動作に合わせて、サービスを構成する複数のプロセスの実行を制御する新たなスケジュール法について述べ、Web サーバに適用した場合について評価結果を報告する。

Process Scheduling Policy for A Service Consisting of Multiple Processes

Sukanya SURANAUWARAT† and Hideo TANIGUCHI†

Process schedulers of many existing operating systems, control process execution based on predetermined policies such as real-time or time sharing system, not based on a process behaviour. Hence, this can hinder an effective use of a processor or, extend the processing time of a process unnecessarily. In this paper, we propose a new process scheduling policy that controls a service, which consists of multiple processes such as a server, by adjusting the execution of those processes according to their predicted behaviour. And we also report the effect when applied to a Web server.

1. はじめに

既存の多くオペレーティングシステムが提供しているプロセスのスケジュールは、プロセス実行の以前に得られる利用形態を基に、プロセスの実行を制御している。例えば、実時間の処理を行う利用形態では、実行優先度により、実時間性の高い処理を優先的に実行している。また、多くの利用者が利用する計算機では、処理の均等化を図るため、タイムスライスによりプロセスの連続実行時間を制限している。いずれの場合も、「プロセスの動作内容」に合わせて実行制御を行っているわけではない。このため、プロセッサの有効利用を妨げ、また、プロセスの処理時間を無用に引き延ばしている場合がある。具体的な例としては、残り僅かでプロセッサ処理を終えようとするプロセスでも、タイムスライス機能によって、強制的に再スケジュールが行われる場合が挙げられる。この例では、そのプロセスの処理が次のプロセッサ割当まで待たされることとなり、プロセスの処理時間が増加することをもたらす。また、強制的な再スケジュールは、プロセス切替処理の増加も招いている。もし、この時、強制的な再スケジュールを行わなければ、プロセスの継続実行によりプロセスの処理時間が短縮され、プロセス切替処理も減少される。すなわち、プロセスの動作を予測し、プロセスのプロセッサ利用に合わせたスケジュールを行えば、上述のような無駄は発生しない。そこで、プログラム動作内容に合わせてプロセスの実行制御法を変更するという考え方である、プログラム指向スケジュール[1] (POS : Program Oriented Schedule) を提案した。

POS の基本的な考え方を以下に示す。

(1) プロセスの動作を把握し、プログラム動作内容の知識として保存する。

(2) プログラム動作内容の知識を利用して、プロセスの実行制御法を変更する。

文献 [1] ではプロセスの動作を予測してプロセスの切替時期を変更する方式を提案した。また、文献 [2] ではプログラムの動作を予測して入力処理を効率化する方式を提案した。前者の方式は、POS の考え方を演算スケジュールへ適用したものであり、サービスが 1 プロセスで構成されている場合に有効である。しかし、

† 九州大学大学院 システム情報科学研究科
Graduate School of Information Science and Electrical Engineering, Kyushu University

複数のプロセスで構成されることが多い今日のサービスへの適用には不十分である。

このため、本論文では、複数のプロセスで構成されるサービスにも適用できる新たなスケジュール法を提案する。具体的には、サーバの処理の効率向上を重点において、現在、よく使われているWebサーバを対象サービスとし、サーバプロセスの動作内容を把握し、クライアント・サーバによるトランザクションの応答時間を短縮するスケジュール法を提案する。

2. Webサーバ

複数のプロセスを構成するサービスの例として、サーバが挙げられる。サーバでは、同じプログラムによるプロセスが同時に多数、生成と消滅を繰り返して動作を行なう。サーバの処理は同じ内容の繰り返し、つまり、トランザクション処理に近い処理である。また、分散システムでは、サーバの性能は重要である。したがって、本論文では、サーバの処理の効率向上を重視して、現在、よく使われているWebサーバを対象サービスとした。

本論文では、無料で入手し利用することが可能であり、また、インターネット上で約5割のシェアを占めている[3]Apacheサーバプログラムを使用することにした。したがって、以降にApacheサーバの動作について説明し[3),4]、クライアント・サーバによるHTTPトランザクション構造の例を取り上げる。

2.1 Apacheサーバの動作

Apacheはスタンドアロンまたはinetdモードのどちらかのモードで稼働する。

(1) スタンアロンモード
このモードは、サーバがスタンアロンでApacheを起動している場合である。Apacheはインタフェースへ割り当てられた、1つ以上のIPアドレスの1つ以上のポート番号(標準設定80番)で接続を待ち受ける。このモードでは、同時に生じる数の接続を扱うため、実際には複数個のApacheプロセスが稼働することになる。

(2) inetdモード
このモードは、サーバがUNIXのコマンドであるinetdを利用する設定になっている場合である。inetdは監視するように設定された全ポート上で接続を待ち受ける。接続要求が到着すると、inetdは設定ファイルである/etc/inetd.confから、どのサービスが該当ポート番号に対応付けされているかを判別し、設定されたプログラムを起動する。したがって、WWWサービスと該当ポートを/etc/inetd.confに設定すれば、Apacheはinetdによって起動される。

どちらのモードで稼働していたとしても、到着した接続は結局Apacheに渡されることになる。スタンドアロンモードにおいては、サーバが起動されたままとされ、クライアントのリクエストに対して返答を直ちに返す。inetdモードにおいては、クライアントからのリクエストごとにサーバが起動され、その間に変更された設定内容が次回の起動時に反映される。

Apacheの機能のいくつかはinetdモードで動作しないので、このモードでのApacheの使用は非常に単純な場合に限られる。したがって、一般に、スタンドアロンモードは通常運用、inetdモードは、実験運用で各々利用される。以降、特に断らない限り、Apacheはスタンドアロンモードで稼働していると仮定する。

Apacheを稼働させると、1つのプロセスが生成される。このプロセスは、初期化の処理やプログラムの起動時に指定されたオプションに対応する処理を行なう。その後、このプロセスは新たなプロセス(以降、親サーバプロセスと呼ぶことにする)を生成して終了する。

親サーバプロセスはソケット処理などを行ない、Configファイルに指定された起動時のサーバプロセスの数(以降、StartServersと略す)で新たなプロセス(以降、子サーバプロセスと呼ぶことにする)を起動する。そして、起動したこれらの子サーバプロセスの状態を監視する。具体的には、接続持ち状態の子サーバプロセスの数が設定ファイルConfigに指定された接続待ち状態の子サーバプロセスの最小数(以降、MinSpareServersと略す)よりも少なくなった場合は、親サーバプロセスが新しい子サーバプロセスを1秒に1つ子サーバプロセスを起動する。

実際に、クライアントからの接続要求を待機して処理を行うのは、子サーバプロセスである。子サーバプロセスは、Configファイルに指定された接続待ち状態の子サーバプロセスの最大数(以降、MaxSpareServersと略す)や各子サーバプロセスに対する処理できる要求の最大数(以降、MaxRequestsPerChildと略す)に達すると、終了する。

2.2 HTTPトランザクションの構造

クライアントの一種であるブラウザがWebサーバにアクセスしHTMLドキュメントを要求する例を取り上げ、HTTPトランザクションの構造について図1に示し、説明する。

まず、ブラウザはユーザから与えられたURLを読み込んで解釈する。次に、ブラウザはHTTPプロト

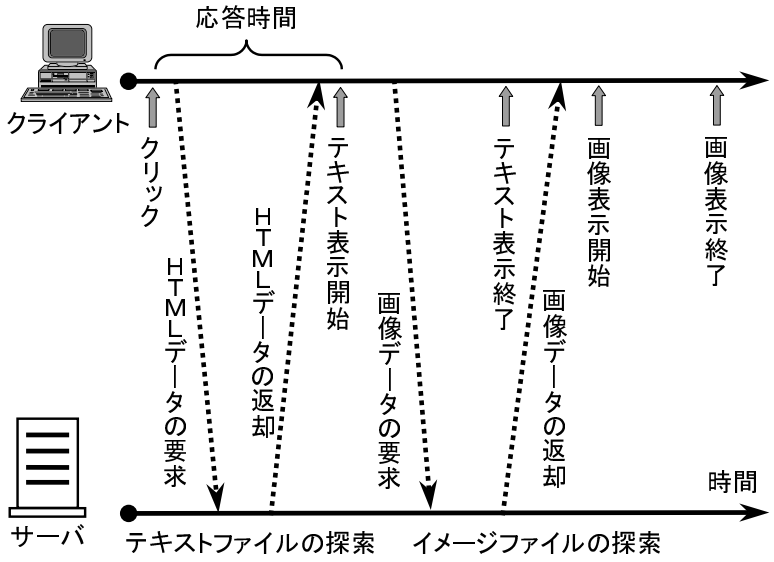

図 1 HTTP トランザクションの構造
Fig. 1 A Structure of HTTP transaction

コルを使って URL に指定されたサーバマシンへ接続し、そのサーバマシンに保管している HTML データを要求する。

そして、サーバマシン上でクライアントからの接続を待機している複数の子サーバプロセスの中の 1 つはその要求を受け付け、URL に指定されたドキュメントパスに対応する HTML データのテキストファイルを探して要求元のブラウザに返す。

そして、最後に、ブラウザはサーバから取得した HTML データを解釈しながら、必要な整形を行ない、画面に表示する。例えば、
と
に囲まれた部分にある<li>アイテムは字下げして行頭に点を打つというような整形を行なう。また、ブラウザがこの HTML のテキストファイルを解釈していき、このドキュメントの表示には画像データが必要なことが判明したら、そこで、ブラウザは Web サーバとの間に接続している接続を通して、2 つめの要求を送信する。

3. スケジュール法

本章では、POS の基本制御機構[1] を用いて Web サーバの動作内容を把握し、分析した結果を報告する。さらに、サーバマシン上で稼働している子サーバプロセスのプロセスの状態は、どのように変化するのかについて説明する。そして、最後にクライアント・サーバによるトランザクションの応答時間をどのように短縮することができるのかを示し、新しいスケジュール法を提案する。

3.1 サーバの動作内容

本節では、サーバの動作内容を把握するために、POS の基本制御機構[1] を用いて、各子サーバプロセスのプロセスの状態とその時刻を記録した。そして、それぞれの子サーバプロセスの識別子とその状態の列からなる情報である PFS[1] を、記録した情報に基づいて作成した。以降、PFS を基にして分析したサーバの動作内容について説明する。

図 2には、ユーザが同じブラウザに同じ URL を 3 回入力したときの分析したサーバの動作内容を示す。PFS の作成の際に使用したサーバプログラムは、Apache Ver.1.2.5 であり、クライアントプログラムは、Netscape Navigator Ver3.04 である。この時、サーバの Config ファイルに指定された StartServers や MinSpareServers や MaxSpareServers や MaxRequestsPerChild の値は、それぞれ 1、1、3、3 である。また、サーバが提供している URL の中には、1 つの画像が含まれている。

図 2において、親サーバプロセスは、Config ファイルに指定された StartServers の値と同じ数で新たな子サーバプロセスであるプロセス A を生成してから WAIT 状態となる。

生成されたプロセス A は、初期化などの処理を行なった後、ブラウザからの接続を待機して WAIT 状態となる。

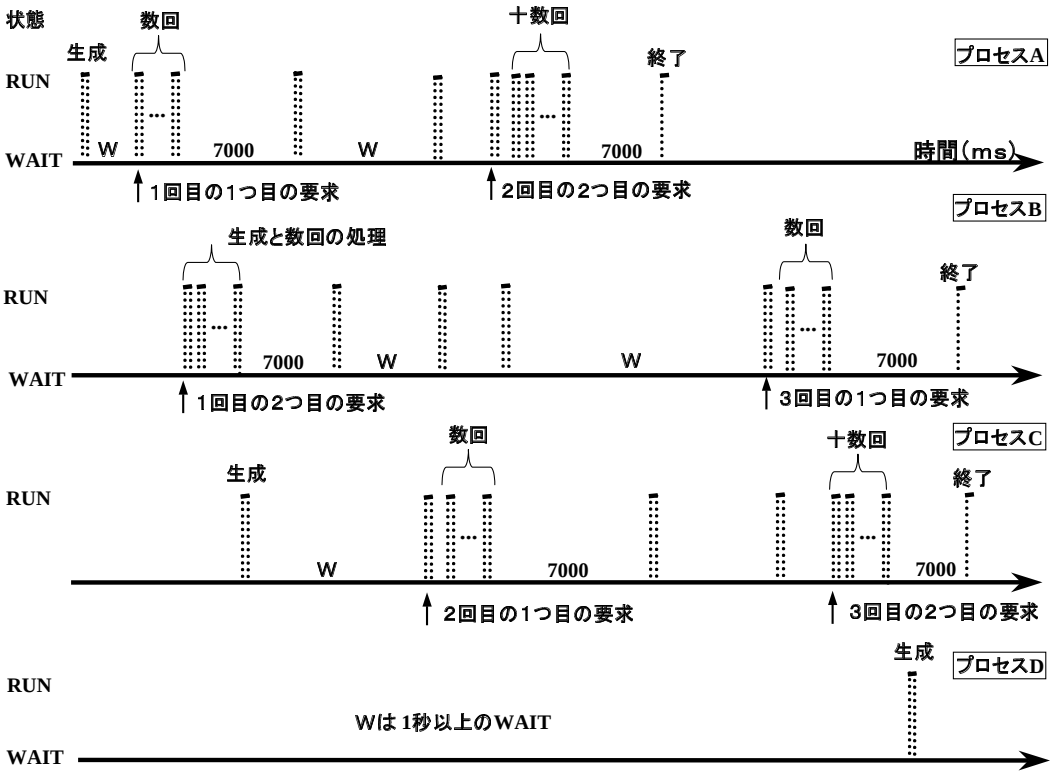

図 2 PFS によるサーバの動作内容
Fig. 2 Contents of Processes according to PFS

一方、ブラウザはユーザに入力された一回目の URL を読み込んで解釈し、該当のマシンへ接続して要求を送信する。

すると、接続を待機していたプロセス A は、RUN 状態となり、URL に指定されたドキュメントパスに対応する HTML データのテキストファイルを探して要求元のブラウザに返す。そして、ブラウザからの再接続時の遅延を最小限とするための最大要求待ち時間 (以降、KeepAliveTimeout と略す) で接続を開設したままの状態で維持し WAIT 状態となる。但し、KeepAliveTimeout は 7 秒と設定した。ちょうどこの時、親サーバプロセスは、再び子サーバプロセスの状態を調べ、1 と設定した MinSpareServers の値より子サーバプロセスの数が少なくなったことが分かり、新たな子サーバプロセスであるプロセス B を生成し WAIT 状態となる。

一方、ブラウザがサーバからの HTML データのテキストファイルを解釈し、このドキュメントの表示には画像データが必要であると判明し、2 つめの要求を送信する。

そして、生成されたプロセス B は、プロセス A と同様な処理を行なってから接続を待機して WAIT 状態になる直前に、2 つめの要求が来たので、RUN 状態のままでイメージファイルを探し、要求元のブラウザに返す。そして、プロセス A と同様に KeepAliveTimeout により WAIT 状態となる。ちょうどこの時、親サーバプロセスは、再び子サーバプロセスの状態を調べ、1 と設定した MinSpareServers の値より子サーバプロセスの数が少なくなったことが分かり、新たな子サーバプロセスであるプロセス C を生成し WAIT 状態となる。

その後、プロセス A も B も、順に KeepAliveTimeout による WAIT 状態から RUN 状態へ移行し 2.1節で述べた終了条件である MaxSpareServers と MaxRequestsPerChild を調べ満足しないので、再び接続を待機して WAIT 状態となる。

そして、2 回目の URL が入力されるとブラウザが同様な処理を行った後に、先に接続を待機して WAIT 状態となったプロセス C は、RUN 状態となり、HTML テキストファイルを探して要求元のブラウザに返す。そ

して、KeepAliveTimeoutによりWAIT状態となる。

そして、プロセスCの次に接続を待機していたプロセスAは、2回目の2つめの要求に対してイメージファイルを探し要求元のブラウザに返す。そして、KeepAliveTimeoutによりWAIT状態となる。

その後、プロセスCもAも、順にKeepAliveTimeoutによるWAIT状態からRUN状態へ移行して終了条件ををを調べる。プロセスCは、終了条件を満足しないので、再び接続を待機してWAIT状態となる。そこで、プロセスAは、終了条件であるMaxRequestPerChildに達したので、終了する。

そして、ユーザから3回目のURLが入力されると、上述の処理と同様に、プロセスBは、テキストファイルを、プロセスCは、イメージファイルを探して、要求元のブラウザに返す。そして、KeepAliveTimeoutによりWAIT状態となる。この時、親サーバプロセスは、再び子サーバプロセスの状態を調べ、1と設定したMinSpareServersの値より子サーバプロセスの数が少なくなったので、新たな子サーバプロセスであるプロセスDを生成しWAIT状態となる。

生成されたプロセスDは、初期化などの処理を行なった後、ブラウザからの接続を待機してWAIT状態となる。

3.2 サーバのプロセスの状態変化

子サーバプロセスのプロセスの状態がどのように変化するのかについて図3に示し、説明する。

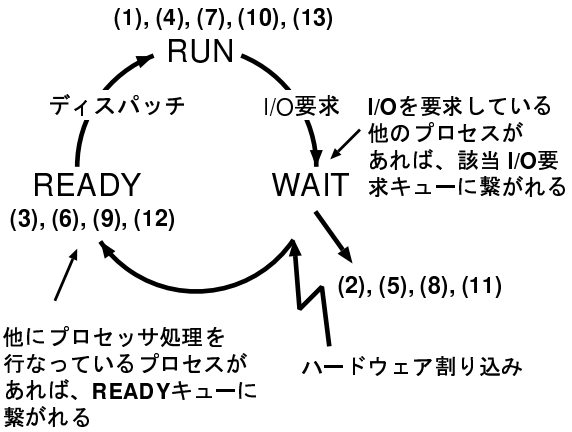

図3 子サーバプロセスのプロセス状態の変化
Fig. 3 The State Transition Of a Server's Process

(1) 子サーバプロセスが生成されてRUN状態になると、初期化の処理やクライアントからの接続の準備を行なう。

(2) その後、クライアントからの接続を待機してWAIT状態となる。接続要求が来ると、WAIT状態となったその子サーバプロセスはREADY状態となる。

(3) READY状態となった子サーバプロセスは、他にプロセッサ処理を行なっているプロセスが存在すれば、READYキューに繋がれ、再びスケジューラによって、ディスパッチされるまで、待ち続ける。

(4) 再びRUN状態となった子サーバプロセスがクライアントからの要求を処理するために、受信を要求する。

(5) この時、受信要求が既に他のプロセスからあり受信処理中の場合、その子サーバプロセスが受信要求キューに繋がれ、WAIT状態となる。その後、受信処理を行い、受信終了後、ハードウェア割り込みにより、READY状態となる。受信処理中でない場合は、受信要求キューでの待ちは発生せずに、受信処理によるWAIT状態になる。

(6) (3)と同様。

(7) 再びRUN状態となった子サーバプロセスが受信した要求を、「ドキュメントパスに対応するファイルを送り返せ」というような具合に解釈し、サーバマシンの磁気ディスクにそのファイルをデータの入力として要求する。

(8) この時、入出力処理が既に他のプロセスからあり入出力処理中の場合、その子サーバプロセスが実I/O待ちキューに繋がれ、WAIT状態となる。その後、入力処理を行い、入力終了後、ハードウェア割り込みにより、READY状態となる。入出力処理中でない場合は、実I/O待ちキューでの待ちは発生せずに、入力処理によるWAIT状態になる。

(9) (3)と同様な処理。

(10) 再びRUN状態となった子サーバプロセスがそのHTMLのテキストファイルをクエスト元のブラウザに返すために、データの送信を要求する。

(11) この時、送信要求が既に他のプロセスからあり送信処理中の場合、その子サーバプロセスが送信要求キューに繋がれ、WAIT状態となる。その後、送信処理を行い、送信終了後、ハードウェア割り込みにより、READY状態となる。送信処理中でない場合は、送信要求キューでの待ちは発生せずに、送信処理によるWAIT状態になる。

(12) (3)と同様な処理。

(13) 再びRUN状態となった子サーバプロセスが終了条件を調べ、満足したらその子サーバプロセスが終了する。そうでなければ、再びクライアント

からの接続の準備を行ない、(2) へ戻る。

3.3 スケジュール法

図 1において、ユーザにとっては、クリックしてから HTML のテキストファイルを表示するまでの応答時間が短い方が望ましいと考えられる。以下のことに着目されたい。

(1) プロセッサがボトルネックになった場合には、子サーバプロセスが READY キューに繋がれる。(図 3の (3),(6),(9),(12))

(2) ディスクがボトルネックになった場合には、子サーバプロセスが実 I/O 待ちキューに繋がれる。(図 3の (8))

(3) 通信がボトルネックになった場合には、子サーバプロセスが送受信キューに繋がれる。(図 3の (5),(11))

したがって、テキストファイルに関する処理を行っている子サーバプロセスが上述のそれぞれのキューに置かれている時、キューに繋がれている時間を短くすることにより、クリックしてから HTML のテキストファイルを表示するまでの応答時間を短くすることができる。以下のスケジュールを行う。

(1) READY キューでの優先制御
プロセッサがボトルネックになった場合には、READY キューに繋がっているプロセスの中で、HTML データのテキストファイルを要求するプロセスを優先してキューを繋ぎ換える。そうすることにより、テキストの表示および終了が早くなる。

(2) WAIT 時の実 I/O キューでの優先制御
ディスクがボトルネックになった場合には、実 I/O 待ちキューに繋がれているプロセスの中で、HTML データのテキストファイルを要求するプロセスを優先してキューを繋ぎ換える。そうすることにより、テキストの表示および終了が早くなる。

(3) 送受信待ちキューでの優先制御
送受信がボトルネックになった場合には、送受信待ちキューに繋がれているプロセスの中で、HTML データのテキストファイルを要求するプロセスを優先してキューを繋ぎ換える。そうすることにより、テキストの表示および終了が早くなる。

4. 実　　装

READY キューでの優先制御のスケジュール法について実装方式を述べる。

4.1 課題と対処

実装に当たって、その課題と対処について述べる。

＜課題 1 ＞ HTML データのテキストファイルの要求を処理するプロセスであることをどのように知ることができるのか

＜対処 1 ＞ ブラウザからの接続を待機して WAIT 状態となっている子サーバプロセスは、HTML データのテキストファイルの要求を処理するプロセスであることが 3.2節の (1),(10) から容易に分かる。他の事象で WAIT 状態にいる子サーバプロセスと区別するためには、ブラウザと Web サーバが 1 対 1 の場合、図 1に示すように、ユーザが URL を入力しクリックをしてから次のクリックをするまでの時間が比較的に長いので、長く WAIT 状態にいる子サーバプロセスはブラウザからの接続を待機してトランザクションの開始を処理するプロセスであるとみなすことができる。しかし、この対処は、ブラウザとサーバが複数対 1 の場合、ブラウザ間の相関が生じるため、有効ではないことがある。この場合の対処は今後の課題としている。

＜課題 2 ＞ 時間が長いと判断する基準 (SLP) をどのように設定するのか。

＜対処 2 ＞ 3.1節で述べた PFS を基に、クライアントからの接続を待機している時間を予測して SLP の値を設定する。

＜課題 3 ＞ ＜対処 2 ＞で述べた SLP の値の予測が外れて、HTML データのテキストファイルの要求を処理するプロセスでないプロセスの優先度を上げてしまったときには、どのように対処するのか。

＜対処 3 ＞ HTML データのテキストファイルの要求を処理するプロセスの処理内容を予測し、RUN と WAIT 状態繰り返し回数 (RW) を記録する。そして、以前に優先度が上げられたプロセスが RW の値を越えると、その優先度を元に戻す。

＜課題 4 ＞ どの契機で対象プロセスの優先度を操作するのか。

＜対処 4 ＞ 対象プロセスは WAIT 状態から READY 状態へ移行する契機で、優先度を操作する。

4.2 実現アルゴリズム

READY キューでの優先制御のスケジュール法を実現するアルゴリズム方式を以降に示す。

(1) プロセスが生成されたときに、READY キューでの優先制御のスケジュール法の対象プロセス

表 1 対象プロセスの登録表
Table 1 A Register Table of Processes with regard to Our Schedule Policy

pid	status	flag	rw_cnt	wait_cnt
⋮				
⋮				

であるために、そのプロセスを表1に示す登録表に登録する。表1における1要素は、プロセスの識別子(pid)、プロセスの状態(status)、プロセスの優先度への操作(上げる、下げる)が有無かを示すフラグ(flag)、RUNとWAIT状態の繰り返しの回数を記録するカウンタ(rw_cnt)、およびWAIT状態の連続時間を記録するカウンタ(wait_cnt)の5つの項目から成る。なお、プロセスの状態とは、そのプロセスが動作中か消滅したかの情報である。

(2) 対象プロセスがWAIT状態になると、wait_cntを1秒毎にカウントアップする。

(3) 対象プロセスがWAIT状態からREADY状態へ移行する際、

(A) その対象プロセスがWAIT状態にいる時間が長い場合(wait_cnt >= SLP)、そのプロセスがHTMLデータのテキストファイルの要求を処理するプロセスとみなし、優先度を上げ、flagをONにする。そして、wait_cntをクリアする。

(B) その対象プロセスがWAIT状態にいる時間が短い場合(wait_cnt < SLP)、

(a) 以前に優先度を上げたことがない、すなわち、flagがOFFであれば、wait_cntをクリアし、その対象プロセスの優先度を何にもせずに通常の制御を返す。

(b) 以前に優先度を上げたことがある、すなわち、flagがONであれば、wait_cntをクリアする。そして

(i) RUNとWAIT状態の状態繰り返し回数が小さい場合(rw_cnt <= RW)、優先度を上げたままにし、rw_cntをカウントアップする。

(ii) rw_cntがRUNとWAIT状態繰り返し回数が大きい場合(rw_cnt > RW)、優先度を下げて、flagをOFFにし、rw_cntをクリアする。

なお、3.1節で述べたPFSを基に、クライアントからの接続を待機している時間を予測してSLPの値を、HTMLデータのテキストの要求を処理するプロセスの処理内容を予測してRWの値を設定する。

5. 評 価

実装したREADYキューでの優先制御のスケジュール法はどれくらい効果があるかについて評価する。

5.1 処理モデル

実測に用いたサーバプログラムは、Apache Ver.1.2.5であり、3.1節と同じConfigファイルで設定した。また、実測に用いたクライアントプログラムは、Netscape Navigator(以降、Netscapeと略す)Ver.3.04である。

処理の内容は、SLPを5秒に固定し、1から10までの各RSの値に対して、クライアントマシンでNetscapeを走らせ、サーバマシンへ接続しドキュメントを取得する処理を、5秒以上の間隔で6回繰り返して行った。なお、サーバが提供しているURLの中には、1つの画像データが含まれている。また、サーバマシンのバッファ・キャッシュやNetscapeのキャッシュを無効にした。

5.2 実 測 環 境

実測に用いた計算機は、無限ループのプロセッサ処理と共存させてApacheを走らせる486DX2 33MHz機およびNetscapeを走らせるPentium Pro 180MHz機であり、それぞれBSD/OS 2.1を使用し、10Mbpsのイーサネット型通信路で結ばれている。

5.3 結果と考察

実測した結果は図4に示す。図4は、READYキューでの優先制御のスケジュール法を利用した応答時間と利用しない応答時間で割算した値を示す。ここでいう応答時間とは、以下の2つの場合示す。

(1) NetscapeにURLを入力してからHTMLデータのテキストファイルを取得するまでの時間(T1)

(2) NetscapeにURLを入力してから画像データのイメージファイルを取得までの時間(T2)

また、図4において、t1とt2は、READYキューでの優先制御のスケジュール法を使用するとしないときのT1とT2のそれぞれの比である。

図4から見ると、各RSの値に対してt1が1以下である。これは、NetscapeをSLPの値(5秒)以上の間隔で走らせ、NetscapeにURLを入力してクリックする度に、図2に示した1回目の1つ目や2回目の1つ目のなどの要求を処理するようなプロセスが、HTMLデータのテキストファイルに関して処理を行なったか

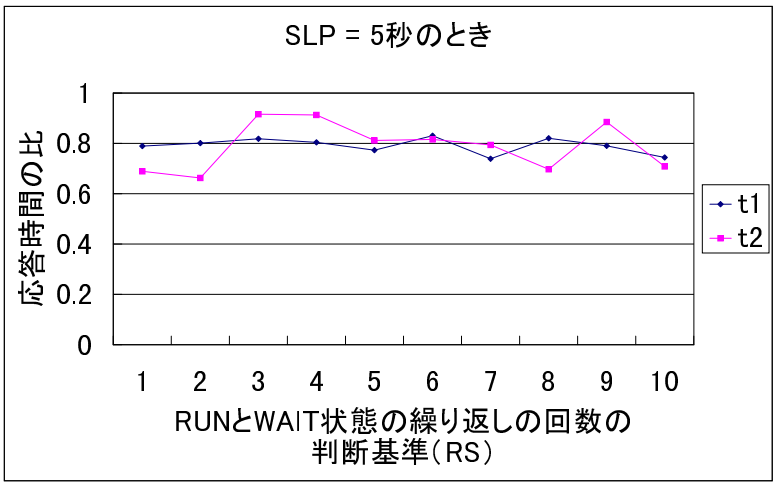

図 4 READY キューでの優先制御法の効果
Fig. 4 The Effect of Our Schedule Policy

らである。したがって、T1 は短くなり、わずかであるが期待している結果が得られた。

また、各 RS の値に対して、t2 も 1 以下である。これは、以前に長い wait をしており、かつ今回画像データのイメージファイルの要求を処理しようとするプロセス、すなわち、図 2に示した 2 回目の 2 つ目や 3 回目の 2 つ目などの要求を処理するようなプロセスにも、優先度が上げられたからである。したがって、T2 の平均値は短くなり、t2 が 1 以下であることをもたらした。

6. おわりに

READY キューでの優先制御のスケジュール法を提案し、実現時の課題と対処を示した。スケジュール法の基本的な考え方は、プロセッサがボトルネックになった場合、READY キューに繋がっているプロセスの中で、HTML データのテキストファイルの要求を処理するプロセスを優先してキューを繋ぎ換えることである。提案したスケジュール法の効果については、実際に Netscape を用いて、Netscape に URL を入力してから HTML データのテキストファイルを取得するまでの時間が短くなるという期待している結果が得られた。

今後は、SLP と RS の値を作成した PFS の基に、自動的に設定できるようにする。また、PFS を更新する契機を検討する。さらに、ブラウザとサーバが複数対 1 の場合に生じる、ブラウザ間の相関への対処やこの時の SLP と RS の値の設定について検討し、実現と評価を行う。最後に、残りの 2 つのスケジュール法の実現と評価を行う。

参考文献

1) 谷口秀夫: POS:プログラム指向スケジュールの提案, 情報処理学会シンポジウム論文集, Vol. 96, No. 7,pp. 123–130 (1996)
2) 谷口秀夫: 入出力要求流れの学習による入出力処理の効率化, 情報処理学会シンポジウム論文集, Vol. 97, No. 8, pp. 141–148 (1997)
3) 著者:ルートボックス, 訳者:三輪 幸男 : Apache 活用カイド (UNIX 編), プレンティスホール発行所
4) 著者:Ben Laurie, Peter Laurie, 監訳者:田辺 茂也, 訳者:三代川 信義: Apache ハンドブック (Apache The Definite Guide), オライリー・ジャパン発行所
5) 著者:Clinton Wong, 監訳者:法林 浩之, 訳者:須田 隆久: Web クライアントプログラミング (Web Client Programming with Perl), オライリー・ジャパン発行所