4.4 ブロードバンドネットワークサービスを実現するシステム技術

近年のネットワークの高速化により、ネットワークサーバの負荷が大きな問題となってきている。そのため、ネットワークサーバにおいては、数千から数万ものコネクションを効率よく管理する、性能を重視したプログラミングモデルが重要になってきている。そこで、従来より広く用いられているポーリングI/O のインタフェースを、より効率的なコネクションの並行処理が可能となるPOSIX 実時間シグナルを利用して実現することにより、サーバ処理の高速化を低い実装コストで実現する手法について研究を行った。

4.4.1 研究の狙い

従来、高い性能を目的としたネットワークサーバでは、select()や poll()によって実装される、ポーリングI/O モデルと呼ばれるI/O 機構がよく用いられる。これは、複数の記述子におけるI/O の即時完了性を1回の呼び出しでチェックするもので、単一の実行エンティティ（プロセスあるいはスレッド）で複数の記述子を並行して扱うことが可能となる。そのため、それぞれの記述子にエンティティを割り当てるマルチプロセスモデルやマルチスレッドモデルでは問題となるエンティティ管理コスト（生成、切替えのコスト）が発生せず、効率的な通信管理が可能となる。一方で、ポーリングI/O モデルは、扱う記述子の数の増加に対するスケーラビリティが低く、性能劣化の問題が広く知られている[1]。

そこで近年着目されているのが、明示的な通信イベント通知機構を用いた通信管理である[1-5]。これらの中で、POSIX 実時間シグナルを用いたイベント通知機構は、POSIX 実時間機能の拡張（IEEE 1003.1b-1993）の中で標準化され、現在では多くの Unix オペレーティングシステムに実装されている機能である。しかし、この機構をそのまま用いるには、プログラミングモデルが従来のポーリングI/O と大きく異なる点や、例外処理の煩雑さなど課題が多い。実際にこの機能を用いているネットワークサーバは非常に少ないというのが現状である。

本研究では、この POSIX 実時間シグナルを用いて擬似的に従来のポーリングI/O モデルを実現する通信管理ライブラリを提案する。具体的には POSIX 実時間シグナルから得られる情報およびノンブロッキングI/O の実行結果を用

いて、各ソケットの状態遷移を管理し、ポーリング I/O の呼び出しにおいては記録しておいた情報を返すというものである。本方式により、個々の記述子のポーリング処理におけるオーバヘッドが軽減され、より高速なポーリング I/O が実現される。

4.4.2 実時間シグナルを用いた通信管理機構の設計

本研究で提案する通信管理機構では、実時間シグナルおよびノンブロッキング I/O を用いて、擬似的にポーリング I/O の機能を実現する。本節では、その基本的な通信管理の仕組みと、ポーリング I/O 機能の実現について述べる。

実時間シグナルによる通知イベントの管理

ポーリング I/O は、管理する記述子集合に対して個々の状態を得るものである。一方で、実時間シグナルは、個々の記述子が I/O 可能に変化する際に発行されるものである。また、I/O 不可への変化は、ノンブロッキング I/O の実行結果から検出することができる。そのため、ポーリング I/O を実現するためには、実時間シグナルやノンブロッキング I/O によって検出される状態変化を静的な記憶領域に記録しておき、要求されたときに返せば良い。図 1 は、各ソケットの状態を保持する I/O 状態管理テーブルを示したものである。

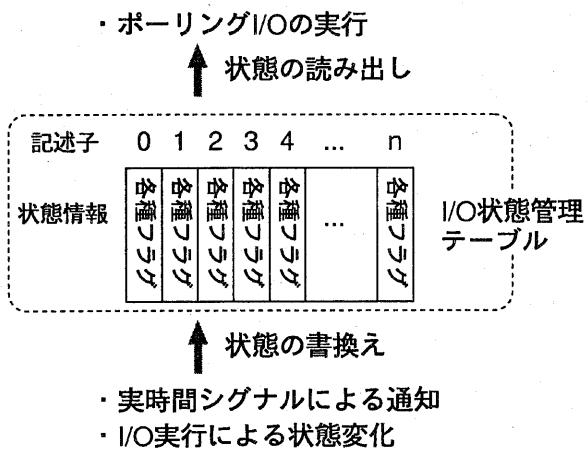

図 1 ソケット状態管理の基本構造

一方、実時間シグナルによるイベントとソケットの状態遷移の様子をまとめ

たものが図 2 である。このように、I/O 実行による状態変化の記録では、読み込み、書き込み、エラーの状態について管理することができる。

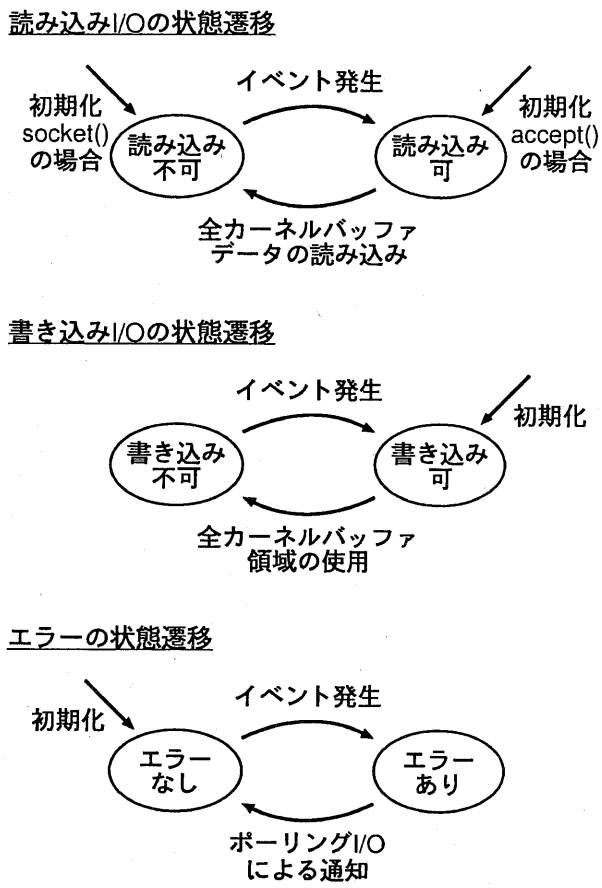

図 2 イベント通知とソケット状態の遷移

読み込み I/O

読み込み I/O では、読み込み可と判明している記述子に対してのみ実際のシステムコールを伴う I/O を実行する。そこで、用意したユーザバッファのサイズ以下のデータしか読み込めなかった場合に、そのソケットは読み込み不可に変化したことが分かる。また、状態の初期値については、socket()呼び出しで得られたソケットの場合、読み込み不可としておけばよい。サーバにおいて accept()を呼び出して得られた新しいソケットについては、accept()終了後、実時間シグナルをセットするまでの間にデータが到着してしまう可能性がある。そのため、accept()によって得られたソケットについては、読み込み可として

初期化することでこの問題を回避する。一方で、データが届いているか未確認のまま読み込み可に設定してしまうため、1 回余分に読み込み I/O が実行されるというオーバヘッドがある。しかし、この I/O によりバッファの状態が正しく検出されるため、次からは無駄に I/O が実行されることはない。

書き込み I/O

書き込み I/O については、書き込もうとしたデータサイズよりも小さいサイズしか実際には書き込めなかった場合、そのソケットは書き込み不可に変化したことが分かる。ここで注意しなければならないのは、実時間シグナルは I/O 不可状態から I/O 可能状態への変化をイベントとして通知する点である。そのため、書き込みバッファの状態については、状態の初期化において書き込み可に設定する必要がある。

エラー

エラーについては、実時間シグナルはその発生時に一度通知してくれるだけである。そのため、対処によってエラー状態が解除されても、知ることができない。一方で、エラー情報が通知されれば、サーバプログラムによって何らかの対処がなされると考えられる。本通信管理機構では、エラーはポーリング I/O によりサーバプログラムに通知されたときに解除している。

ノンブロッキング I/O の併用

提案する通信管理機構では、実際の I/O においてノンブロッキング I/O を併用している。ノンブロッキング I/O を用いる理由は、以下の 3 つの理由による。

1 つは、I/O における例外的なブロックを排除することである。ブロッキング I/O では、それがポーリング I/O などにより I/O 可能であると判断される場合でも、バッファの空き容量を超えるデータの書き込みを試みた場合など、ブロックされる場合がある。

2 つめの理由は、I/O によりソケットの状態を管理することである。特に書き込み I/O においては、ブロッキング I/O を用いる場合、カーネルバッファの空き容量が足りない場合でも、ブロックすることで最終的には全 I/O が実行さ

れるため、状態を知ることができない。ノンブロッキング I/O では、即時可能な分のみ実行されるため、その結果によりバッファの空き具合いを知ることができる。

3 つめの理由は、記述子の状態変化の曖昧性の問題を回避することである。記述子の状態変化の曖昧性とは、以下のように説明される。記述子からデータを読み込む際に、アプリケーションバッファのサイズと同じサイズのデータが得られた場合、本通信管理機構ではまだデータがカーネルバッファに残っていると判断し、読み込み可の状態を保持している。しかし、読み込み I/O 前にカーネルバッファに格納されていたデータのサイズが、アプリケーションバッファのサイズと同じであった可能性もある。実時間シグナルは、読み込み可の状態から読み込み不可の状態へ遷移した場合には発行されない。そのため、さらなるデータの読み込みを実行しなければこの判断はできない。ここで、ブロッキング I/O を用いると、カーネルバッファにデータが残っていない場合にブロックされるため、非効率的である。同様の問題は、データの書き込みの際にも発生する。データ書き込みにおいて、用意したアプリケーションバッファのデータを全て書き込むことができた場合でも、丁度カーネルバッファが一杯になり、書き込み不可の状態に遷移している場合がある。本通信管理機構では、こうした問題を回避するために、ノンブロッキング I/O を用いている。

クローズされた記述子の扱い

実時間シグナルを用いたイベント通知では、クローズされた記述子の扱いには注意が必要である。シグナルキューにイベント情報が残っている状態で、その記述子がクローズされた場合、そのイベント情報は誤ったものになる。文献[2]では、FreeBSD の kqueue における同様の問題が議論されている。こうしたイベント情報は、記述子がクローズされている間に取り出されれば対応が可能である。しかし、記述子がクローズされた後に直ちに再利用される場合、問題である。

この問題は、記述子をクローズする際に、シグナルキューを走査し、該当するイベント情報を取り出すか無効にすることで解決することができる。しかし、そのためにはオペレーティングシステムへの変更が必要であり、現実的には移植性の問題などが発生する。そこで本通信管理機構では、以下の手順によりこの問題を回避する。

1. 記述子をクローズする場合には、shutdown()を用いてコネクションを閉じ、その後別のリストでその記述子を（クローズしないで）管理する。
2. イベントキューから情報を取り出す際に、イベントがキューに残っていないことを検出した場合に、リストに保管されている記述子集合をクローズする。

本方式が正しく動作するためには、シグナルキューにイベントを残さないように、可能な限り高速に取り出すようにしなければならない。そのため、このイベントの取り出しにタスクを限定した別スレッドを用いて実装し、その実行優先度を高く設定している。これは次節で述べるシグナルキューの溢れの問題を回避するためにも重要である。

シグナルキュー溢れへの対処

実時間シグナルを用いる場合、シグナルキューの溢れの問題がある。しかし、この問題への対処は一般的に困難とされている。

本提案手法では、この問題を回避するために、シグナルキューからのイベント取り出しを最優先で行うように設計している。しかし、それでも不十分なほどシステムが過負荷状態になることも考えられる。そこで、シグナルキューの溢れに対して、全ソケットをI/O可能状態（読み込み、書き込み共に可）に設定することで対処している。これにより、読み込みI/Oおよび書き込みI/Oがアプリケーションの必要に応じて実行され、その結果によりI/O状態管理テーブルの情報が正しく再設定されることになる。本方式は次の利点を有している。

- 処理をI/Oに集中することにより、スループットを向上させ、一時的な過負荷状態をすばやく解決することができる。
- 他のI/Oモデルへの移行が不要であり、継続して実時間シグナル駆動モデルを利用することができる。

ブロッキングI/Oの実現

提案する通信管理機構では、全てのソケットをノンブロッキング化している。一方で、ブロッキングI/Oが必要になる場合もある。そこで、各ソケットに条

件変数を用意し、読み込みおよび書き込みが可能になるイベントを待つことができるようにした。実時間シグナルにより、ソケットがI/O可能状態に変化したことを検出すると、そのイベントを待っているスレッドがあるかどうかチェックし、あればそのスレッドのブロックを解除する処理を行っている。

4.4.3 結論と今後の課題

本研究では、実時間シグナルを用いたネットワークI/Oの多重処理を行う通信管理機構を提案した。本機構の利点は、以下のようにまとめることができる。

- ポーリングI/Oのプログラミングモデルをそのまま利用することができる。
- サーバにおける並行ソケット数に対するスケーラビリティが高い。
- 実時間シグナルは現在多くのUnixオペレーティングシステムで実装されており、移植性に問題が少ない。

今後は、本研究で用いた実時間シグナル以外の明示的なイベント通知機構（pollデバイス、kqueue、epollなど）もサポートしていきたいと考えている。こうした機構は、現在いくつかのオペレーティングシステムにおいて先進的な機能として実装されており、今後も改良が期待される。最終的には、システムにおける実装の詳細を隠蔽する高レベルかつ高性能な通信管理ミドルウエアとしての実装を目標としている。

参考文献

[1] G. Banga, et al. “A scalable and explicit event delivery mechanism for UNIX”, in Proceedings of USENIX Annual Technical Conference, 1999.

[2] J. Lemon. “Kqueue: A generic and scalable event notification facility”, in Proceedings of USENIX Annual Technical Conference, 2001.

[3] N. Provos, et al. “Scalable Network I/O in Linux, in Proceedings of USENIX Annual Technical Conference, 2000.

[4] N. Provos, et al. Analyzing the Overhead Behavior of a Simple Web Server”, in Proceedings of the 4th Annual Linux Showcase & Conference, 2000.

[5] A. Chandra, et al. Scalability of Linux Event-Dispatch Mechanisms", in Proceedings of USENIX Annual Technical Conference, 2001.