情報処理学会論文誌 Vol. 49 No. 8 2862–2872 (Aug. 2008)

スケジューラとの協調による プロセス優先度に基づくパケット受信処理手法

一野 浩太朗[†1] 楢崎 修二[†2]

近年，ネットワークの通信帯域が増加し，複数のプロセスが同時にパケット受信を行う場合が増えてきた．多くの OS ではプロセスの重要度に従った資源割当てを行うためにプロセス優先度を導入し，優先度が高いプロセスほど多くの資源を与える．よってパケット受信においても宛先プロセスの優先度に基づいた処理を期待するが，一般的な OS では受信処理においてはプロセス優先度が考慮されておらず，すべてのパケットは平等に扱われてしまう．これによって複数のプロセスが同時に受信を行う場合に，高優先プロセスへのパケットよりも低優先プロセスへのパケット受信に多くの資源を割り当ててしまう可能性がある．これはシステムが高負荷の状態において特に問題となる．そこでこの問題を解決するため，本論文ではネットワークサブシステムとプロセススケジューラとが互いに情報を伝える受信処理手法 *PacketFlow* を提案する．*PacketFlow* は早期段階でのパケットの選択破棄とソケットバッファ使用率の高いプロセスの優先度調整とを行うことで高優先プロセスのパケット受信数を増加させる．本手法を Linux の上に実装し評価実験を行った結果，提案手法では受信処理において高優先プロセスのスループットおよび全体のスループットが向上していることが確認できた．

A Packet Receiving Method on Process Priority by Cooperating with Scheduler

Kotaro Ichino[†1] and Shuji Narazaki[†2]

As the communication bandwidth is increasing in recent years, more processes on a computer are excuted and receive packets simultaneously. Since priority of process is used in most operating systems in order to reflect the importance of process, a process with higher priority has more resources than processes with lower priorities. But in networking subsystem, process priority is ignored in packet receiving and packets are treated equally. Thus a process with low priority may use more resources then processes with higher priority in some situations. This becomes a critical problem when the system's load is high. To solve this problem, we propose a new receiving method *PacketFlow*, in which network subsystem and process scheduler communicate each other to reflect process priority in networking. In *PacketFlow*, network subsystem selects and drops the packets to be delivered to processes with low priorities in an early stage of networking, and CPU scheduler increases the priorities of processes which have many packets in their socket buffers. Experimental results with a testbed built on Linux show that both of processes with high priorities and the whole system achieve better throughputs compared with the original kernel.

1. はじめに

近年，ネットワークの通信速度が大幅に向上している．それにともなってインターネット上でのサービスが多様化され，IP 電話や動画ストリーミング配信サービス等のネットワーク通信を行う新しいアプリケーションやサービスも登場し，オペレーティングシステム（以下 OS）の処理内容のうちパケット受信のための処理の割合が増加している．動画像のストリーミング再生を行いつつファイル転送を行うといった複数のプロセスが同時に受信を行うことも少なくない．通信の内容や量によってプロセスの処理の重要度は変わってくるため，重要度の低いプロセスは重要度の高いプロセスの処理を妨げないようにしなくてはならない．

一般的な OS ではプロセスの重要度に従った処理を行うためにプロセス優先度を導入しており，優先度が高いプロセスほど多くの資源が割り当てられることを期待する．しかし，現在の OS でのパケット受信処理ではすべてのパケットは公平に処理されてしまい，受信処理において宛先プロセスの優先度が適切に反映されていない[1)]．したがって，先に述べたような多重受信状況において，高優先プロセスへのパケット受信よりも低優先プロセスへのパケット受信に多くの資源を割り当て，低優先プロセスの方がより多くのパケットを受信してしまう可能性がある．また，受信処理では状況に応じてパケットの破棄を行うことがあるが，その破棄においてもプロセス優先度は反映されていないため，低優先プロセス宛のパケットではなく高優先プロセス宛のパケットが破棄されてしまう可能性もある．以上の理由により，ネットワーク通信を行うプロセスが複数存在した場合，ユーザの意図どおりにプロセスの処理を進めることが困難であるという問題がある．この問題は CPU およびネットワークサブシステムの処理に余裕がある場合には問題とならないが，システムが高負荷の状

†1 九州大学大学院システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University
†2 長崎大学工学部
Faculty of Engineering, Nagasaki University

態において特に問題となる．

本研究の目的は OS のパケット受信処理における上記の問題点を解決し，システムが高負荷の状態においてプロセス優先度を反映した効率の良いパケット受信処理を実現することである．そこで本論文では，ネットワークサブシステムとプロセススケジューラが協調しプロセス優先度を反映した効率の良い受信処理を行う手法 *PacketFlow* を提案する．本手法ではプロセス優先度に基づいた早期段階でのパケットの選択破棄，ならびに受信処理を優先的に進めるべきであるプロセスを自動的に検出しスケジューリング時の優先処理を施すという 2 つの機能を実現する．我々は本手法を導入したシステムを設計し，Unix 系 OS の 1 つである Linux[2] の上に実装し，その評価実験を行った．実験の結果から，本手法を導入したシステムではオリジナルのシステムよりも高優先プロセスのスループットおよび全体のスループットの両方が向上していることが確認できた．

本論文の構成を以下に示す．まず 2 章ではパケット受信処理について説明する．次に 3 章では，2 章であげた問題点を解決した受信処理手法 *PacketFlow* の提案を行い，4 章で本手法の評価実験の方法とその結果について述べる．そして 5 章では関連研究の比較を行い，最後に 6 章でまとめを述べる．

2. パケット受信処理

まず，本章では**図 1** に沿って一般的な OS でのパケット受信時の処理を述べ，その問題点について説明する．

2.1 受信処理の流れ

あるプロセス（図 1 中のプロセス A とする）がソケットを用いて受信を行う場合，そのプロセス宛のパケットがソケットの持つパケット格納用キューであるソケットバッファ（図 1 中のソケット 1）に届いていなければ，プロセス管理部がそのプロセス A を待ち状態へと遷移させる．

次に，パケットがネットワークから送られてくるとそのパケットは Network Interface Card（以下 NIC）に到着し，NIC 内メモリに格納される．そして，NIC に対応する受信ハンドラを用いて[⋆1]，ネットワークサブシステムはパケットを図に示す処理待ちキューに格納する．

その後，ネットワークサブシステムはパケットを処理待ちキューから取り出し，エラー処理や TCP におけるシーケンス番号の確認等のプロトコル処理を行った後，パケットの宛先プロセスと対応するソケットバッファ（ソケット 1）に格納し，プロセス管理部の起床関数を呼び出す．起床関数ではパケットの到着を待っていたプロセス A を実行可能状態へと遷移（起床）させ，プロセスキューに格納する．

そして，ディスパッチャによりプロセス A に実行権が与えられると，プロセス A はパケットをソケットバッファから取り出し受信する．

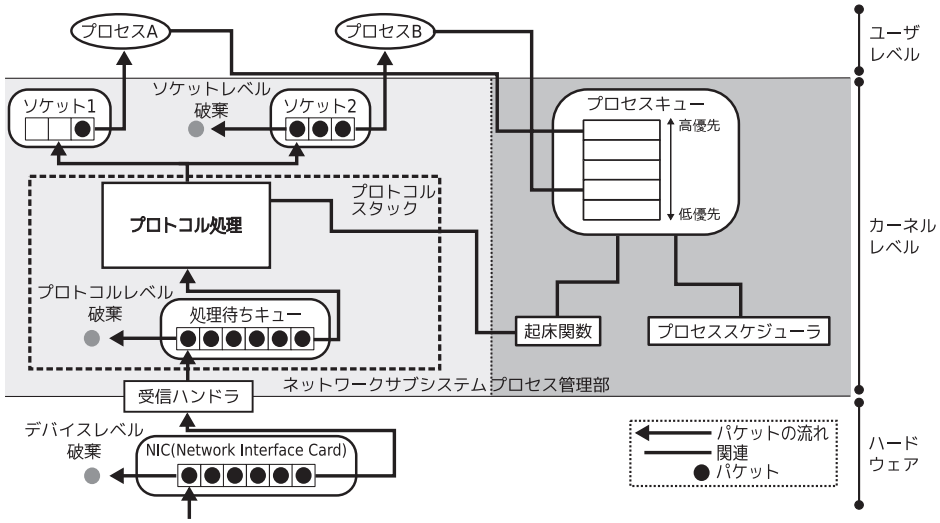

図 1　パケット受信処理
Fig. 1　The process of receiving packets.

2.2 パケットの破棄の発生

受信処理では資源の利用状況に応じて処理の途中でパケットの破棄を行うことがある．

まず，NIC 内メモリに空きがない状態で NIC に新しいパケットが届くと，NIC 内メモリに格納する際にそのパケットは破棄される（以下，この破棄を**デバイスレベル破棄**と呼ぶことにする）．

次に，プロトコルスタックの処理待ちキューはパケットのプロトコル処理を行うため，送られてきたパケットを一時的に格納しておく固定長のキューである．このキューでのパケット格納数は決められており，その数を超えて届くパケットは格納されることなく無条件で破

⋆1 この呼び出しは通常ハードウェア割込みによる．

棄される（以下，この破棄を**プロトコルレベル破棄**と呼ぶことにする）．

そして，プロトコルスタックでの処理が正常に終了した場合は，パケットを宛先プロセスと対応するソケットバッファに格納するが，ソケットバッファでもパケット格納数は決められておりその数を超えて届けられるパケットは破棄される（以下この破棄を**ソケットレベル破棄**と呼ぶことにする）．

2.3 問　題　点

本論文で考える OS でのパケット受信処理の問題点を以下にあげる．

問題点 1　優先度を無視したパケット破棄の発生

デバイスレベル破棄やプロトコルレベル破棄では，パケットの宛先プロセス優先度は考慮せず，最後に到着したパケットをそのまま破棄している．そのためこれらの破棄においては，低優先プロセス宛のパケットではなく高優先プロセス宛のパケットが破棄される可能性がある．

問題点 2　ソケットレベル破棄および通信帯域の抑制の発生

ソケットバッファに空きがない場合に新たなパケットが届けられると，ソケットレベル破棄が発生する．ソケットレベル破棄が発生した場合にはそれまでそのパケットに対してプロトコル処理層で使用した CPU 資源が無駄となる．このソケットレベル破棄は UDP [3] を使用した通信時に発生する．

TCP [4] を使用した通信ではフロー制御[⋆1]が働くため通常ソケットレベル破棄は発生しない．しかし一方で，ソケットバッファの利用可能な領域が少ない場合，フロー制御での通知するウインドウサイズが小さな値に自動的に設定される．すると単位時間あたりに送られてくるパケットの量が制限されてしまうため，対応するプロセスの通信帯域すなわちスループットが減少してしまうことになる．

問題点 2 はプロセスのソケットバッファからのパケット取り出しが追いついていないことにより発生する問題であるので，対応するプロセスの実行時間を長くすることにより解決できるはずである．

3. 提 案 手 法

本論文では前章の問題点が存在する，以下に示す環境を対象とする．

⋆1 ウインドウサイズ（ソケットバッファの利用可能な領域の大きさ）を通信を行うホストが互いに通知しあうことで，受信処理の途中でパケットの破棄が発生しないよう送信パケット量の調整を行う制御．

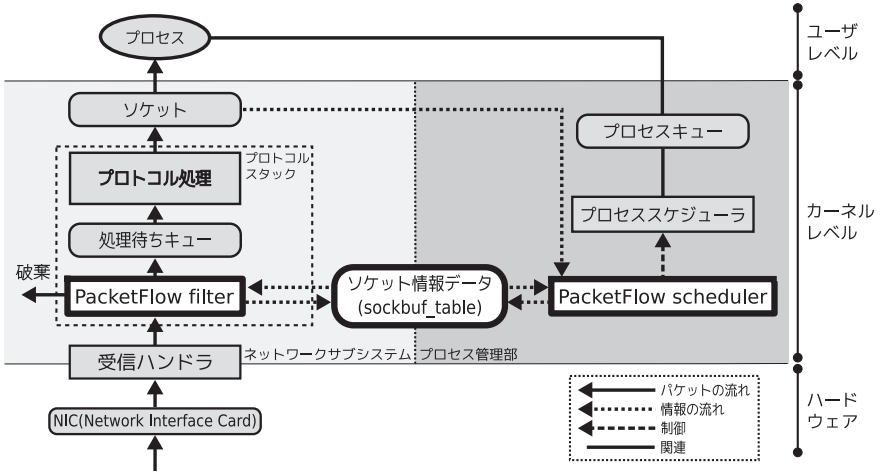

図 2　*PacketFlow* におけるパケット受信処理
Fig. 2　The process of receiving packets on *PacketFlow*.

- ユーザが適切にプロセス優先度を設定している．
- 優先度が異なる受信プロセスが複数存在している．
- 多量のパケットを受信している，またはプロセスレベルでのパケット受信量の時間変化が存在する．

このような環境に対して，問題を解決するためにネットワークサブシステムとプロセススケジューラとが協調し，プロセス優先度を反映した効率の良い受信処理を行う手法 *PacketFlow* を提案する．

PacketFlow は優先度に基づく早期段階でのパケットの選択破棄を行うモジュールである *PacketFlow filter*，ならびに優先すべきプロセスの自動検出とそのプロセスへの優先処理を行うモジュールである *PacketFlow scheduler* の 2 つのモジュールからなる．*PacketFlow* を導入した受信処理システムの概要を**図 2** に示す．

同様の環境に対する改善手法として QoS（Quality of Service）に基づく手法があげられる[5),6)] が，QoS はアプリケーションに対して帯域の保証を目指すものであるのに対して，本手法は高負荷時においてアプリケーションに対して割り当てることのできる帯域の拡大を目指すものである．

以下では各モジュールについて説明を行う．

3.1 *PacketFlow filter*

PacketFlow filter は NIC とプロトコルスタックの処理待ちキューとの間に位置し，パケットの選択破棄を行うモジュールである．このモジュールは受信ハンドラの処理の先頭で呼び出される．

通常パケットをプロトコルレベル破棄やソケットレベル破棄で破棄する場合にはすでに何らかの処理を行っているため，CPU 資源の無駄が生じる．そこで *PacketFlow filter* ではパケットを受信した早期の段階でパケットのトランスポートヘッダ情報を読み出し，受信プロセスに対応したソケットバッファおよびプロトコルスタックの処理待ちキューの状況を調べる．そしてそのパケットが後述する破棄条件に該当した場合，この時点でパケットを破棄する．*PacketFlow filter* で早期に破棄を行うことで，既存方法でパケットを破棄する場合よりも使用する CPU 資源を抑制することができる．その結果，使用しなかった CPU 資源を他のパケット受信処理やプロセスの実行に使用することができるようになるため，パケット受信処理の効率が向上する．また，デバイスレベル破棄やプロトコルレベル破棄が発生している場合には低優先プロセスへのパケットを選択的に破棄することによって高優先プロセスへのパケットの破棄の可能性を減らし，高優先プロセスのスループットを向上させることができる．

3.1.1 パケット破棄条件

PacketFlow filter でのパケット破棄条件は以下のとおりである．

条件 A　パケットの使用プロトコルが UDP である．

条件 B1　デバイスレベル破棄またはプロトコルレベル破棄が発生しておりパケットの宛先プロセスが高優先度でない．

条件 B2　パケットの宛先ソケットバッファでソケットレベル破棄が発生している．

条件 A では，使用プロトコルが UDP であるパケットは既存の受信処理で破棄が起こる可能性があるため，破棄対象として選出する．TCP パケットの場合は受信処理で破棄を行うとパケットの再送が起こってしまい，処理を増やしてしまうことになるため *PacketFlow filter* で破棄は行わない．**条件 B1** では，デバイスレベル破棄やプロトコルレベル破棄が発生していた場合は高優先度プロセス宛のパケットを優先的に処理できるようにするため，低優先プロセス宛のパケットを破棄対象として選出する．**条件 B2** では，宛先ソケットバッファでソケットレベル破棄が発生しているパケットは受信処理を進めたとしてもソケットバッファに格納する際に破棄されるため，破棄対象として選出する．

```
 1: /* ソケット情報ハッシュ表 */
 2: extern struct sock_data sockbuf_table[];
 3:
 4: /* 破棄確率変数 */
 5: unsigned int drop_rate;
 6:
 7: /* パケット選択破棄関数
 8:    この関数はパケット受信関数の先頭で呼ばれる */
 9: PF_filter(p){
10:   if(前回の drop_rate の変動から一定時間が経過)
11:     if(一定時間の間にパケット受信時の破棄が発生)
12:       drop_rate を増加;
13:     else
14:       drop_rate を減少;
15:   if(!条件 A)
16:     return;
17:   if(条件 B1){
18:     drop_rate に基づく p の破棄;
19:     return;
20:   }
21:   if(p の宛先ソケット情報が sockbuf_table に存在)
22:     sockbuf_table からソケット情報を取り出す;
23:   else{
24:     ソケット検索を行いソケット情報を得る;
25:     ソケット使用率を算出しソケットレベル破棄発生の有無を調べる;
26:     ソケット情報を sockbuf_table へ登録する;
27:   }
28:   if(条件 B2)
29:     p を破棄;
30: }
```

図 3　*PacketFlow filter* のアルゴリズム
Fig. 3　The algorithm of *PacketFlow filter*.

3.1.2 *PacketFlow filter* のアルゴリズム

PacketFlow filter のアルゴリズムを**図 3** に示す．両モジュールの共有変数である `sockbuf_table` については後に説明する．

PacketFlow filter を導入した受信処理では，受信ハンドラの処理の先頭で送られてきたパケット `p` を引数にして関数 `PF_filter` を呼び出し，送られてきたパケットの選択破棄を行う．そして，関数 `PF_filter` で破棄されなかったパケットに対して通常の受信処理を行う．

関数 PF_filter では，まずパケット破棄確率変数 drop_rate を変動させる．デバイスレベル破棄またはプロトコルレベル破棄が発生している間は新たに到着したパケットも破棄する可能性が高いと考え，一定時間内にパケット受信時の破棄が発生している場合は drop_rate を増加させ，発生していない場合は drop_rate を減少させる．パケット受信状況に応じて動的に破棄確率の変更を行うため，パケットの不必要な破棄を防ぐことができる．

続いて，パケットのトランスポートヘッダから通信プロトコルを割り出し，**条件 A** を満たしているかどうかの判定を行う．そしてさらに**条件 B1** を満たしているならばそのパケットを確率的に破棄し，**条件 B2** を満たしているならばそのパケットを破棄する．

条件 B1 ではパケットの宛先ポート番号を用いて宛先プロセスが低優先プロセスであるかどうかを判断するが，これは計算量を減らすために後述するソケットに関する情報を記憶する両モジュールの共有変数である sockbuf_table の情報を用いることで行う．破棄は破棄確率変数 drop_rate とパケットのデータ部分の値とを使った乱数との比較により確率的に行う．

また**条件 B2** では宛先ソケット情報の読み出しを行うが，同じプロセスに対するパケットが連続することが多いので，計算コストを抑えるため，sockbuf_table を用いる．パケットのヘッダ情報から検索したソケット情報は sockbuf_table に保存しておき，その有効期限を設定する．パケットに対応するソケット情報がすでに sockbuf_table に存在する場合にはそのデータを再利用することにした．

3.2 *PacketFlow scheduler*

PacketFlow scheduler はパケット受信の観点から優先すべきプロセスを自動的に選出し，プロセス管理部においてそのプロセスが優先的に処理されるようにプロセスの優先度を制御するモジュールである．

すなわち *PacketFlow scheduler* では，プロセス起床時にそのプロセスと対応するソケットバッファの使用率を調べ，使用率が高かったものは優先プロセスとして選出する．優先プロセスとして選出したプロセスは，そのプロセスの動的優先度とタイムスライスとを増加させる．

この処理を行うことでプロセススケジューラはそのプロセスの実行順序を早め，実行時間を増加させるので，優先プロセスの単位時間あたりのパケット受信量が増加する．ソケットバッファからデータを取得する受信プロセスの実行順序を決定するプロセススケジューラがパケット受信処理のボトルネックとなっているという報告[7),8)] もあるため，このことからもプロセススケジューラの制御は有用であると考えられる．また，ネットワークサブシステムの観点からはパケットのソケットバッファ滞在時間を減少させ，利用可能な領域を増加させることになるため，ソケットレベル破棄および通信帯域の抑制の発生を防ぐことができる．

3.2.1 優先プロセス選出条件と優先処理

プロセスを優先プロセスとして選出するかどうかは，対応するソケットバッファ使用率が閾値 T を超えている，という**選出条件**によって判定する．プロセスに対応するソケットバッファ使用率が高ければ，UDP 通信の場合はそのソケットバッファでソケットレベル破棄が発生する可能性が高いと考えられ，TCP 通信の場合はフロー制御によって通信帯域が減少している可能性があるからである．ソケットバッファ使用率が高いと判断するための閾値 T はあらかじめユーザが設定を行うものとする．条件を満たすプロセスが複数存在する場合はそのプロセス優先度が高い方から優先プロセス選出数 N 個まで選出する．

また，優先プロセスとして選出されたプロセスはその動的優先度[⋆1]とタイムスライスを増加して設定を行うことで優先処理を行う．

ここで，伝統的な UNIX の優先度の設定との比較を述べる．OS での最終的な優先度は静的優先度と動的優先度の合計値となり，対話型プロセスの応答性を上げるために，プロセスの待ち時間を計測しバッチプロセス[⋆2]の動的優先度を低く設定することを意図的に行っている[9)]．通常，ネットワークからパケットの受信を行うプロセスはバッチプロセスに分類され，優先度は低く設定される．しかし対話型プロセスにおいて入力がいつ発生するか知ることができないように，ネットワークからパケットが届くタイミングを受信側では知ることができない．そこで我々は，ネットワークからパケット受信を行うプロセスも対話型プロセスと同様に，その受信状況に応じて優先度を上げて処理を行う必要があると考える．*PacketFlow scheduler* を用いたプロセススケジューラでは動的優先度の算出において，対話型かどうかの判定に加えソケットバッファ使用率の判定も行う．

3.2.2 *PacketFlow scheduler* のアルゴリズム

PacketFlow scheduler のアルゴリズムを**図 4** に示す．

PacketFlow scheduler を導入したシステムでは，プロセスが起床する際に起床するプロセス P を引数にして関数 PF_scheduler を呼ぶ．P と対応するソケットのソケットバッファ使用率を調べ，**選出条件**を満たしていた場合，優先プロセスに選出される可能性のあるプロセスとして，今選出している優先プロセスとの比較を行う．

⋆1 一般的な OS のプロセス優先度には，ユーザがプロセス実行時に設定を行う静的優先度と OS がプロセスの状態を考慮して算出する動的優先度がある．
⋆2 非対話的に CPU 資源を消費するようなプロセスのことを指す．

```
 1:/* ソケット情報ハッシュ表 */
 2: extern struct sock_data socubuf_table[];
 3:
 4: /* 優先プロセスデータ構造 */
 5: struct prior_process{
 6:   struct task *tk;/* プロセスへのポインタ */
 7:   int    priority;/* プロセス優先度 */
 8:   int    time;    /* プロセス選出時間 */
 9: };
10:
11: /* 優先プロセスリスト */
12: struct prior_process prior_process_list[N];
13:
14: /* 優先プロセス選出関数
15:    この関数はプロセスが起床する際に呼ばれる */
16: PF_scheduler(P){
17:   P と対応するソケットバッファ使用率を調べる;
18:   if(選出条件 && 優先プロセスの中で P のプロセス優先度が上位 N 個内){
19:       prior_process_list に P を追加し更新;
20:       sockbuf_table 中のフラグを更新する;
21:     }
22: }
```

図 4　*PacketFlow scheduler* のアルゴリズム
Fig. 4　The algorithm of *PacketFlow scheduler*.

優先プロセスデータのリストである prior_process_list では優先プロセスのデータが優先度順に並んでおり，新しい優先プロセスが選出された場合にはその順番を考慮して prior_process_list の適切な場所にプロセスのデータを挿入する．

プロセススケジューラでは prior_process_list に入っているプロセスである場合，タイムスライスと動的優先度の設定を行う際に増加した値を設定することで優先プロセスの優先処理を行う．

また，同じプロセスがいつまでも優先プロセスとして選出されていることを防ぐため，優先プロセスとして選出されて一定時間が経過したプロセスも prior_process_list から外す．プロセス終了時の処理も含めアルゴリズムの詳細は省略する．

3.3　両モジュール間の共有変数

PacketFlow filter と *PacketFlow scheduler* の計算コストを削減するために両モジュールから参照できる共有変数である sockbuf_table を持つことにした．この共有変数を用いることで両モジュールでの計算量を減らし本手法でのオーバヘッドを減らすことができる．sockbuf_table の定義を**図 5** に示す．

```
 1: /* ソケット情報保存用構造体 */
 2: struct sock_data{
 3:   struct sock *sk;   /* ソケットへのポインタ */
 4:   long     time;     /* 情報を取得した時刻 */
 5:   double   rcv_rate; /* ソケットバッファ使用率 */
 6:   unsigned char ds_f;/* ソケットレベル破棄発生フラグ */
 7:   unsigned char pp_f;/* 優先ソケットフラグ */
 8: };
 9:
10: /* ポート番号をキー値とするハッシュ表 */
11: struct sock_data sockbuf_table[];
```

図 5　共有変数 sockbuf_table の定義
Fig. 5　The definition of shared variable sockbuf_table.

構造体 sock_data はソケット情報を保存しておくデータ構造である．この構造体のデータをポート番号をキー値とするハッシュ表 sockbuf_table に格納しておくことで，両モジュールから素早く参照を行うことができる．

PacketFlow filter では，パケットの宛先ソケット検索を行った際にパケットの宛先ソケットのアドレスを sk に，時刻を time に記憶する（図 3 の 24 行目）．ソケットバッファ使用率はソケットレベル破棄の検出時（図 3 の 25 行目）に計算して rcv_rate に書き込み，*PacketFlow scheduler* ではこの値を参照することにする．ソケットレベル破棄が発生していた場合はフラグ ds_f を立てておく．

PacketFlow scheduler では，プロセスが優先プロセスとして選出される際にそのソケットデータには優先プロセスのソケットであることを示すフラグである pp_f を立てておく（図 4 の 20 行目）．これにより *PacketFlow filter* ではこのフラグを参照するだけで優先プロセス宛のパケットかどうかの判定が可能となる．

4.　実装と評価実験

本章では提案手法である *PacketFlow* を Linux 2.6.22 [2] に実装し，その評価実験を行った結果を示す．

4.1　実　　装

PacketFlow を Linux に実装する際の実装方針を述べる．Linux の動的優先度は −5 から

表 1　ハードウェアの仕様
Table 1　Hardware configuration of experimental environments.

CPU	AMD Athlon 64 X2 3800+ 2.0 GHz
メモリ	2048 MB
OS	Debian GNU/Linux 4.0 (Linux 2.6.22)
NIC	Realtek RTL-8169

+5 の範囲の値をとる[10]．システムへの影響を考慮して *PacketFlow scheduler* で優先プロセスとして選出したプロセスには動的優先度として −5 を与えることにした．また，破棄確率変数 `drop_rate` の変動の幅は 5%，優先プロセス選出数 `N` は 2，優先プロセス選出のための条件であるソケットバッファ使用率の閾値 `T` は 50%に設定した．以上の実装も含め，両モジュールとも追加したコードはそれぞれ 700 行程度となった．

4.2 実験環境

実験環境として，計測用サーバと計測用クライアント，および負荷生成用サーバ各 1 台からなるネットワークを構築した．3 台のマシンはスイッチングハブを介して接続されている．3 台のマシンの仕様はすべて同様で，**表 1** に示すとおりである．

NIC は Gigabit Ethernet に対応するもので，高負荷時には受信割込み負荷が多大になると考えられるが，今回 *PacketFlow* の実装を行った Linux 2.6 では NAPI[10] が導入されており，高負荷時には割込み処理をポーリング処理に切り替えてパケット受信処理を行う機能が備わっているため，この状況には対応できていると考えられる．

4.3 UDP パケット受信による評価

この実験では，UDP パケットの受信を行う優先度の異なるプロセスが複数存在する状況での，高優先プロセスのパケット受信数の増加を確認する．計測用サーバ上では計測クライアントに向けて UDP パケットを送信するプロセスを 3 つ生成する．このプロセスはすべて同じ優先度（nice 値[⋆1]では 0），同じスループット（この実験では 37,000 packets/sec）でパケットの送信を行う．パケットのペイロード長は 256 Byte である．一方，計測クライアント上では対応するプロセスを 3 つ生成する．このプロセスは単純に到着したパケット数を数えるだけのものである．これら 3 つのプロセスはそれぞれ優先度が異なり（nice 値では 0，5，10），それぞれ P_{U0}，P_{U5}，P_{U10} とし，各プロセスの受信スループットとパケット破棄数の計測を行った．なお，今回は NAPI を用いた受信処理を行っており処理待ちキュー（バックログ）は使用しないため，プロトコルレベル破棄は発生しない．

⋆1 Linux の静的優先度（nice 値）は-20 から 19 の間の値をとり，値が小さい方が優先度が高い．一般的に root 権限を持たないユーザが設定できる nice 値の値は 0 から 19 である．

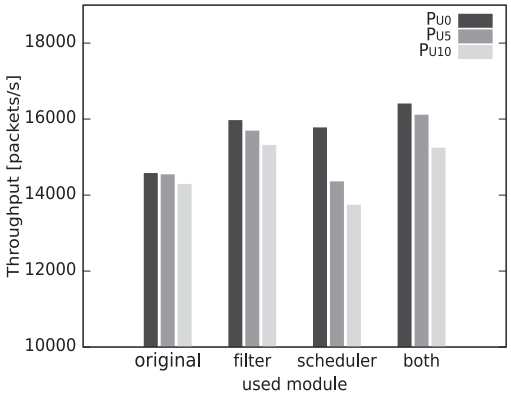

図 6　UDP パケットにおける受信スループット
Fig. 6　Throughput of UDP packets.

表 2　UDP パケット破棄数
Table 2　Dropping count of UDP packets.

		original	filter	scheduler	both
デバイスレベル破棄 [packets/s]		804.3	333.6	210.0	296.0
条件 B1 による破棄 [packets/s]	P_{U0}	-	453.7	-	0.1
	P_{U5}	-	513.4	-	0.1
	P_{U10}	-	461.6	-	304.7
	合計	-	1,428.5	-	305.0
ソケットレベル破棄 [kpackets/s]	P_{U0}	22.4	0	21.2	0
	P_{U5}	22.5	0	22.6	0
	P_{U10}	22.7	0	23.2	0
	合計	67.6	0	67.0	0
条件 B2 による破棄 [kpackets/s]	P_{U0}	-	20.4	-	20.4
	P_{U5}	-	20.7	-	20.7
	P_{U10}	-	20.7	-	21.0
	合計	-	61.5	-	62.1
合計 [kpackets/s]		68.4	63.2	67.2	62.7

実験結果を**図 6** および**表 2** に示す．図 6 の縦軸は受信スループット，横軸は **original** が *PacketFlow* を導入していないオリジナルのシステム（Linux 2.6.22），**filter** が *PacketFlow*

filter のみを用いた場合，**scheduler** が *PacketFlow scheduler* のみを用いた場合，**both** が両モジュールを用いた場合である．表 2 はパケット破棄の発生回数である．

PacketFlow filter を用いた場合ではオリジナルのシステムと比較してすべてのプロセスにおいてスループットが向上している．これはパケットを早期段階で破棄することにより破棄を行う際に使用する CPU 資源が抑制され，パケット受信処理の効率が向上したことを示している．

PacketFlow scheduler のみを用いた場合，P_{U0} のスループットが 1,200 packets/s 程度向上している．これは P_{U0} が優先プロセスとして選出され，優先処理を施された結果であると考えられる．P_{U10} でのスループット減少は優先度の高いプロセスの優先処理を行うことで，P_{U10} への CPU 資源の割当て量が減少したためであると考えられる．また表 2 中段に示すように，ソケットレベル破棄の発生数の合計値はオリジナルのシステムを用いた場合とあまり違いは見られなかったが，各プロセスの発生数を見ると破棄の発生回数に違いが見られた．

両モジュールともに用いた場合は *PacketFlow filter* のみを用いた場合よりも，P_{U0} と P_{U5} のスループットが向上した．また，表 2 の**条件 B1** による破棄結果を見ると，デバイスレベル破棄の発生時にプロセス優先度に応じて破棄が行われていることも分かる．

比較のためにネットワーク通信を行わないプロセスを同時に実行した場合の資源割当て量の変化を調べてみると，nice 値が 0 のプロセスに対し 5 のプロセスでは割り当てられる資源の量が 24%減少し，nice 値が 10 のプロセスでは 36%減少していた．ネットワーク通信を行うプロセスの場合でもこれに準じたスループットの違いがあるべきであるが，今回の実験では優先度によるスループットの差が一番大きな *PacketFlow scheduler* のみを用いたときにおいても，そのスループットの差は P_{U0} に対し P_{U5} では 9%，P_{U10} では 13%の減少であり，その変化はネットワーク通信を行わないプロセスに比べて小さいものとなった．これは本来受信可能であるパケットを破棄することで nice 値によるスループットの違いを実現することは受信処理の改善とはいい難いため，*PacketFlow* では本来受信できるパケットはなるべく破棄しないような方針を選んだためである．

4.4　TCP パケット受信による評価

次に TCP についても同様の実験を行った．プロセス数，優先度の設定は先の実験と同様である（nice 値が 0 のプロセスを P_{T0}，5 のものを P_{T5}，10 のものを P_{T10} とする）．ここで用いる計測用クライアント上に生成するプロセスも単純に届けられたパケットを数えるだけのものとした．TCP の場合，スループットはウインドウサイズによる影響が大きく通信の途中でスループットが変化してしまうため，プロセスがパケットの受信を開始してから 100,000 パケットを受信完了するまでの時間を計測した．

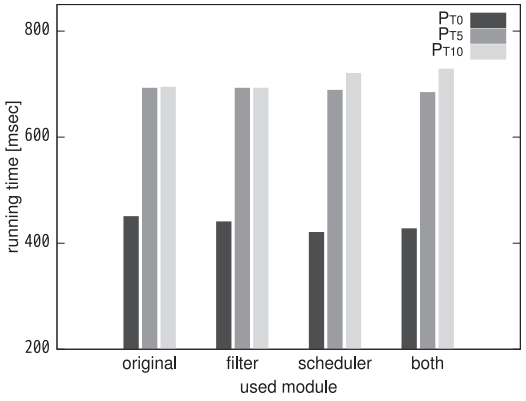

図 7　TCP パケットを受信するプロセスの実行時間
Fig. 7　Running time to receive TCP packets.

実験結果を**図 7** に示す．縦軸はプロセスが通信を開始してから 100,000 パケット受信するまでに経過した時間，横軸は先の実験と同様である．

オリジナルのシステムの場合，P_{T0} は実行時間が短くなっているが，他の 2 つのプロセスの実行時間はほぼ等しくなっている．

PacketFlow filter のみを用いる場合は，オリジナルのシステムとほぼ同じ挙動を示した．*PacketFlow filter* では UDP パケットを処理の対象としているため，TCP パケットを受信した場合は処理は行わずオリジナルのシステムと変わらない．しかしこの結果から *PacketFlow filter* のオーバヘッドが小さいことが分かる．

PacketFlow scheduler のみを用いる場合と両モジュールともに用いる場合ではオリジナルのシステムよりも P_{T0} の実行時間は約 30 msec 短く，P_{T10} の実行時間は約 25 msec 長くなっている．TCP についても *PacketFlow* を導入したシステムではオリジナルのシステムと比較して優先度を反映したパケット受信処理を行うことができているといえる．

また，各プロセスへのパケット送信量が変化する状況でのスループットの変動を調べた．

先の実験で用いた P_{T0}，P_{T10} を用いて，P_{T0} に対してパケット送信を行い，その 10 秒後に P_{T10} に対してパケット送信を行うことで P_{T0} のスループットの変化を調べた．なおこの実験では優先プロセス選出数 `N` を 1 に設定している．

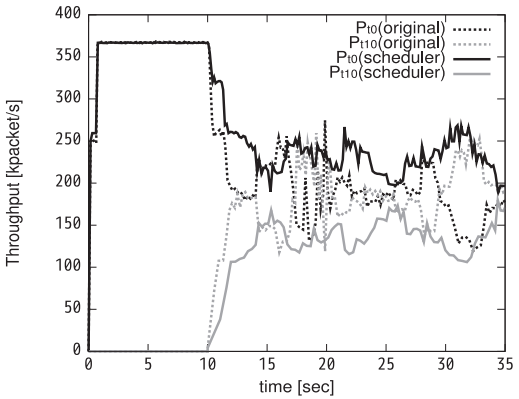

図 8　パケット量に対するスループットの変動
Fig. 8　Throughput against the number of packets.

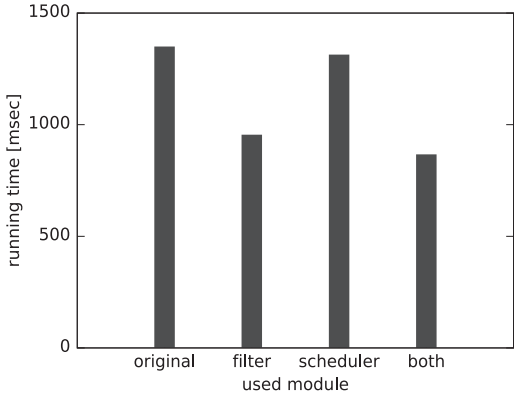

図 9　高優先プロセスの実行時間
Fig. 9　Running time of the highest priority process.

実験結果を**図 8** に示す．縦軸はスループット，横軸は経過時間である．

結果を見てみるとどちらも 10 秒経過後には P_{T0} のスループットが減少しているが，*PacketFlow scheduler* を用いた場合ではスループットすなわち通信帯域の減少量が小さくなっていることが確認できた．

4.5　通信プロトコルが混在した状態での評価

実際のネットワーク通信では TCP を使用するプロセスと UDP を使用するプロセスの両方が混在し，それぞれ通信を行っている．そこでこの実験では，クライアントマシンで TCP パケットと UDP パケットを同時に受信し，優先度を高く設定したプロセスのパケット受信処理が優先的に行われているかどうかを調べた．

計測用サーバ上で計測クライアントに向けて TCP パケットを送信プロセスを 1 つ生成，同時に負荷サーバ上で計測クライアントに向けて UDP パケットを送信するプログラムを 3 つ生成する．いずれも nice 値は 0 で実行する．計測クライアント上ではそれぞれに対応するプロセスを生成する．TCP パケットを受信するプロセスは nice 値 0 で実行し，残りの UDP パケットを受信するプロセスは nice 値 10 で実行する．負荷サーバからの UDP パケット受信によりネットワーク受信処理が高負荷な状態において，TCP パケットを受信するプロセスがパケットの受信を開始してから 100,000 パケット受信完了までにかかる時間を計測した．

実験結果を**図 9** に示す．*PacketFlow filter* のみを用いた場合ではオリジナルのシステムと比べて約 30%プロセスの実行時間が短くなっている．これは *PacketFlow filter* によって負荷サーバからの UDP パケットをネットワークサブシステムの早期の段階で破棄することができ，不要になった CPU 資源を TCP パケットの受信処理に回すことができたためであると考えられる．*PacketFlow scheduler* のみを用いた場合ではオリジナルのシステムと比べて若干改善が見られる．これは TCP パケットを受信するプロセスを優先プロセスとして選出し優先処理を行ったためであると考えられるが，今回は優先プロセス選出数は 2 に設定してあるため，ネットワーク負荷用 UDP パケット受信プロセスのうちの 1 つも優先プロセスとして選出し優先処理を行ってしまうため，改善量が少なくなったと考えられる．これは，状況に応じて選出する優先プロセス数を変更可能にすることで改善すべきである．両モジュールともに用いた場合はオリジナルのシステムと比べて約 36%プロセスの実行時間が短くなっており，一番良い結果となった．

またこの実験とは逆に，計測用サーバから送信するパケットを UDP パケット，負荷生成サーバから送信するパケットを TCP パケットとして先の実験とは逆の状況での実験を行ったところ，*PacketFlow* を導入し両モジュールを用いたシステムではオリジナルのシステムと比較して UDP パケットを受信するプロセスの受信率が約 10%増加した．

以上の実験から，提案する *PacketFlow* では高優先プロセスのパケット受信処理に多くの CPU 資源を割り当て，TCP，UDP いずれに対しても高優先プロセスのスループットを向上させることができた．さらに他のプロセスにおいてもスループットに優先度を反映させる

表 3　フレーム破棄数
Table 3　Number of frame drops.

	original	both
パケット処理高負荷時	1,499.8 (10.75%)	1,069.5 (7.66%)
CPU 処理高負荷時	1,747.3 (12.52%)	1,313.7 (9.41%)

ことができている．

4.6　実際のアプリケーションを用いての実験

この実験では実際のアプリケーションを用いて，ネットワークシステムでの状況を考慮した処理が行われているかどうかを調べた．実験は計測用マシンで動画像のストリーミング再生（30 frame/s，1280 × 720，13,956 frame）を行うプログラムとその他のプログラムを同時に実行し，動画のフレーム破棄数を調べることで行う．また，パケット受信処理の負荷が高い場合と，CPU 処理が高負荷である場合との両方の状況を設定した．CPU 処理の負荷を高める場合には動画像のエンコードプログラム（ffmpeg）を実行し，パケット受信処理の負荷を高める場合には cp でネットワークファイル（NFSv4）サーバからファイルのコピーを行うことにした．動画像の再生は nice 値 0 に設定し，その他のアプリケーションは nice 値 10 に設定する．

実験結果を**表 3** に示す．*PacketFlow* を用いることによってどちらの場合でもフレーム破棄数の減少が見られた．どちらの場合においてもオリジナルのシステムと比べてフレーム破棄数の 3 割程度の減少が確認できた．

5. 関 連 研 究

以下では本研究と関連の深い研究の概要と本研究との比較について述べる．

5.1　PBQs

尾崎らによる PBQs [1] では通常のバックログとは別に，受信プロセスの優先度に従った優先度別のバックログを持つ．このバックログを使用して，高負荷時には低優先度の受信プロセスを宛先とするパケットの処理を遅延させる．しかし，本手法のようにソケットバッファの状況を考慮していないため，ソケットレベル破棄や通信帯域の抑制の改善には対応できないと考えられる．また，NAPI を導入した Linux では受信処理を行う際にバックログを使用しないため，PBQs を用いることはできない．

5.2　LRP，SRP

BSD [11] に関する Druchel らによる LRP [12] や，Brustoloni らによる SRP [13] では，NIC から受け取ったパケットを受信プロセス別のキューに格納し，プロセスに対応するソケットバッファが受信可能であった場合にパケットのプロトコル処理を許可している．パケットの破棄はプロセス別のキューで発生し，そのキューはプロトコルスタックの低い層にあるため，パケット破棄による無駄な CPU 資源の利用は抑えられている．しかし，本手法のように対応するプロセスの処理を優先的に進めるなどの処理は行っていない．

5.3　E-ATBT

寺井らによる E-ATBT [14] はソケットバッファがボトルネックとなるスループットの低下を防ぐため，受信側のソケットバッファが送信側のウインドウサイズ以上になるように動的に割り当てる手法である．また割り当てた領域を使いきれていないと判断した場合には，領域を減少させる．しかし TCP パケットの受信を問題解決の対象としているため，本手法のように UDP パケットの受信時に発生するパケット受信時の破棄，ソケットレベル破棄に関しては対応できない．

6. お わ り に

本論文ではプロセススケジューラとネットワークサブシステムが協調し，優先度に基づいた受信処理を行う手法，*PacketFlow* の提案を行った．*PacketFlow* では早期段階でのパケット選択破棄とプロセスの優先処理という手法を導入し，受信処理の問題点の解決を図った．また *PacketFlow* を導入したシステムを Linux 2.6.22 の上に実装し，その評価実験を行った．

実験結果から，*PacketFlow* を用いたシステムでは UDP，TCP どちらのパケット受信処理においてオリジナルのシステムよりも優先度を反映したパケット受信処理を行っていることが確認できた．また，全体のスループットが増加しており，効率的なパケット受信処理を行うことができていると考えられる．そして，アプリケーションを用いて動画像の再生を行った結果，*PacketFlow* を導入したシステムではフレーム破棄数が 27%程度減少した．

今後の課題としては，*PacketFlow filter* での TCP パケットの処理を追加，優先プロセス選出の順位による優先処理の度合いの制御，状況に応じた優先プロセス選出数の動的変更等があげられる．これらを導入することにより，より優先度を反映し状況に応じた受信処理を行うことができるようになると考えられる．また本研究では，優先度が異なるプロセスが複数存在する状況を前提としているが，同じ優先度のプロセスのみが複数存在する状況でもパケットが送られてくるタイミングに違いがあれば改善効果があると考えられるので，今後評価実験を行う予定である．

参 考 文 献

1) 尾崎亮太，中山泰一：Linux におけるプロセス優先度に基づく受信処理の実現，情報処理学会論文誌，Vol.45, No.3 (2004).
2) The Linux Kernel Achives. http://www.kernel.org/
3) Postel, J.: User Datagram Protocol (IETF RFC 768) (1980).
4) Postel, J.: Transmission Control Protocol (IETF RFC 793) (1981).
5) 西尾信彦，徳田英幸：QOS の 3 階層指定とその翻訳を用いたセッションの単純化調停方式，情報処理学会論文誌，Vol.39, No.2 (1998).
6) Nichols, K., Blake, S., Baker, F. and Black, D.L.: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers (IETF RFC 2474) (1998).
7) Hibino, H., Kourai, K. and Chiba, S.: Difference of Degradation Schemes among Operating Systems – Experimental analysis for web application servers, *The International Conference on Dependable Systems and Networks* (*DSN-2005*), pp.172–179 (2005).
8) 日比野秀章，松沼正浩，光来健一，千葉　滋：アクセス集中時の Web サーバの性能に対する OS の影響，日本ソフトウェア科学会第 21 回大会（2004 年度）論文集，日本ソフトウェア科学，pp.25–29 (2004).
9) McKusick, M.K., Bostic, K., Karels, M.J. and Quarterman, J.S.: 4.4BSD の設計と実装，アスキー (2003).
10) 高橋浩和，小田逸郎，山幡為佐久：Linux カーネル読解室，SoftBank Creative (2006).
11) The FreeBSD Project. http://www.jp.freebsd.org/www.FreeBSD.org/
12) Druschel, P. and Banga, G.: Lazy Receivevr Processing (LRP): A Network Subsystem Architecture for Server System, *USENIX Symposium on Operating System Design and Implementation*, pp.261–276 (1996).
13) Brustoloni, J., Gabber, E., Silberschatz, A. and Singh, A.: Signaled Receiver Processing, *USENIX 2000 Annual Technical Conference*, pp.211–223 (2000).
14) 寺井達彦，岡本卓也，長谷川剛，村田正幸：Web サーバの高速・高機能化のためのソケットバッファ管理方式の実装と評価，電子情報通信学会技術研究報告，SEE，交換システム，Vol.100, No.670, pp.47–54, 電子情報通信学会 (2001).

(平成 19 年 12 月 10 日受付)
(平成 20 年 5 月 8 日採録)

一野浩太朗（学生会員）

1985 年生．2008 年長崎大学工学部情報システム工学科卒業．現在，九州大学大学院システム情報科学府情報工学専攻修士課程在籍．ユビキタスコンピューティング，オペレーティングシステムの分野に興味を持つ．

楢崎　修二（正会員）

1965 年生．1988 年九州大学工学部情報工学科卒業．1990 年同大学大学院工学研究科情報工学専攻修士課程修了．博士（工学）．九州大学工学部助手，九州工業大学講師を経て，現在，長崎大学工学部情報システム工学科准教授．分散並列協調処理，プログラミング言語等に関する研究に従事．