2008－OS－107 (9)
2008／1／31

プロセス優先度を考慮したパケット受信処理手法 PacketFlowのLinuxへの実装

一野 浩太朗　　楢崎 修二

長崎大学工学部情報システム工学科
852-8521 長崎市文教町 1-14
Email: {ichino, narazaki}@cs.cis.nagasaki-u.ac.jp

概要

現在のOSにはパケット受信処理においてプロセス優先度が適切に反映されないという問題がある．そこでネットワーク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ことで，プロセス優先度を反映した受信処理を行う手法 **PacketFlow** を提案する．PacketFlowは早期段階でのパケット選択破棄を行う **PacketFlow filter**，ソケットバッファ使用率の高いプロセスの優先処理を行う **PacketFlow scheduler**，およびプロセス優先度に基づくソケットバッファ領域の動的割り当てを行う **PacketFlow memory-manager** の3つのモジュールから構成される．本論文ではPacketFlowをLinux上に実装し，評価を行った結果について述べる．

Implementation of PacketFlow, a Process Priority based Packet Receiving Method on Linux

Kotaro Ichino　　Shuji Narazaki

Department of Computer and Infomation Science, Faculty of Engineering,
Nagasaki University
1-14 Bunkyo Machi, Nagasaki, 852-8521 Japan
Email: {ichino, narazaki}@cs.cis.nagasaki-u.ac.jp

Abstract

In most operating systems, process priority isn't reflected on packet receiving process in network subsystem properly. In this paper, we propose a new packet process receiving method called *PacketFlow* which is reflecting process priorities of destination processes by communicating network subsystem and process scheduler. PacketFlow consists of three modules: *PacketFlow filter* that drops packets for processes with low priorities, *PacketFlow scheduler* that accelerates a process that has many packets in the socket buffer, and *PacketFlow memory-manager* that controls the length of socket buffer dynamically based on process priority. In this paper, we describe some results of evaluations based on an implementation of PacketFlow system in Linux 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，ネットワーク通信を利用するアプリケーションが増加しており，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．複数のプロセスが同時に受信を行うことも少なくなく，この場合宛先プロセスのプロセス優先度に応じたパケット受信処理を行う必要があるが，現在のOSでのパケット受信処理では全てのパケットが公平に処理されてしまい，受信処理においてプロセス優先度が反映されない[3]．一方で一般的なOSではプロセス優先度を用いてスケジューリングや資源の割り当てを行っており，プロセス優先度の高いプロセスはより多くの資源を使用することができるという方針と齟齬が生じている．

この問題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，ユーザの意図通りにプロセスの処理を進めることが困難となる．そこで我々は，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協調しプロセス優先度を反映した受信処理を行う手法 **PacketFlow** を提案する．PacketFlowは早期段階でのプロセス優先度に基づくパケットの選択破棄を行うモジュールである **PacketFlow filter**，優先すべきプロセスの自動検出と優先処理を行うモジュールである **PacketFlow scheduler**，ならびにプロセス優先度に基づいてソケットバッファ領域の動的割り当てを行うモジュールである **PacketFlow memory-manager**（以下 **PacketFlow mm**）の3つのモジュールからなる．PacketFlow filterとPacketFlow schedulerは我々の先行研究であるPBPF & STPB[7]を元に判定条件やデータ構造を見直し，再設計を行ったものである．PacketFlow mmは今回追加するモジュールである．

本論文の構成を以下に示す．2章で一般的な受信処理とその問題点を述べ，3章で受信処理手法PacketFlowを提案する．そして4章で本手法の評価実験を行った結果について述べ，5章で関連研究，6章でまとめを述べる．

2 パケット受信処理

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対応する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ではソケット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，プロセスキューに格納する．

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，ネットワークサブシステムではプロトコルスタックのパケット受信関数を呼び，パケットを図1に示す処理待ちキューに格納する．このキューでのパケット格納数は決められており，格納時にキューに空きがなかった場合はこの時点でパケットは破棄される．（以下この破棄を**パケット受信時の破棄**と呼ぶ）

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とり出し，プロトコル処理を行った後，ソケット1に格納する．このソケットバッファも処理待ちキューと同様に固定長であり，空きがなかった場合にはパケットが破棄される．（以下この破棄を**ソケットレベル破棄**と呼ぶ）

パケットを格納するとプロセス管理部の起床関数を呼び出し，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）させる．そして，ディスパッチャがプロセスAを選ぶと，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．

2.2 問題点

以下ではPBPF & STPBで挙げていたOSの受信処理の問題点について述べる．

問題1 プロセス優先度を無視したパケット破棄の発生 プロトコルスタックの処理待ちキューに空きがない場合に新しいパケットが送られてくると，そのパケットを破棄しなければならない．

パケット受信時の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する．そのため低優先度プロセス宛ではなく高優先度プロセス宛のパケットが破棄される可能性がある．

問題2 タイムスライスの不足によるソケットレベル破棄の発生 プロセスに割り当てられたタイムスライスが短く，プロセスのパケット受信が追いつかずにソケットバッファに空きがない場合（図1のソケット2）に新しいパケットが送られてくると，ソケットレベル破棄が発生する．破棄を行った場合にはそれまでの処理に使用したCPU資源が無駄となる．

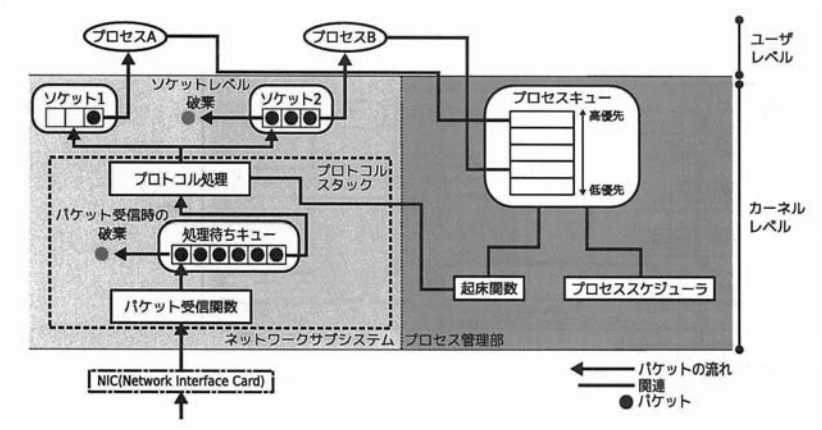

図 1: パケット受信処理

問題 3 静的なソケットバッファ割り当ての問題　TCP を使用した通信では，フロー制御[1]やパケットの応答確認取得などの機能を用いて通信における信頼性を高めているため先に説明したパケット受信時の破棄とソケットレベル破棄は発生しない．しかし一方で，与えられたソケットバッファが小さく利用可能領域が少ない場合，フロー制御によりウインドウサイズを減少させて通信相手に通知する．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が減少してしまうことになる．

また UDP を使用した通信の場合でもソケットバッファが小さければソケットレベル破棄がより多く起こってしまう可能性があるので，どちらの場合においてもできるだけ大きなソケットバッファを与えるべきである．

しかし割り当てることのできるメモリは有限であるため，プロセス優先度に基づいてソケットバッファの領域を割り当てるのがよいと思われる．

3　提案手法:PacketFlow

我々は，前章で挙げた 3 つの問題点の解決を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，プロセス優先度を適切に反映した受信処理を行う手法 PacketFlow を提案する．PacketFlow は先行研究の PBPF & STPB で解決できていなかった問題 3 を解決し，問題 1 と 2 のよりよい解決を図るために全体の再設計を行ったものである．PacketFlow を導入した受信処理システムの概要を図 2 に示す．以下，各モジュールの詳細を述べる．

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り，問題 1 を解決する．このモジュールはパケット受信関数の先頭で呼び出される．

通常パケットをパケット受信時の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，パケットの破棄を行う際に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，高優先プロセスのスループットを向上させることができる．ここで高優先プロセスとは，もともとのプロセスが持つプロセス優先度が高い，もしくは後述する優先プロセスに選出されているプロセスのことを指す．

[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合うことで，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

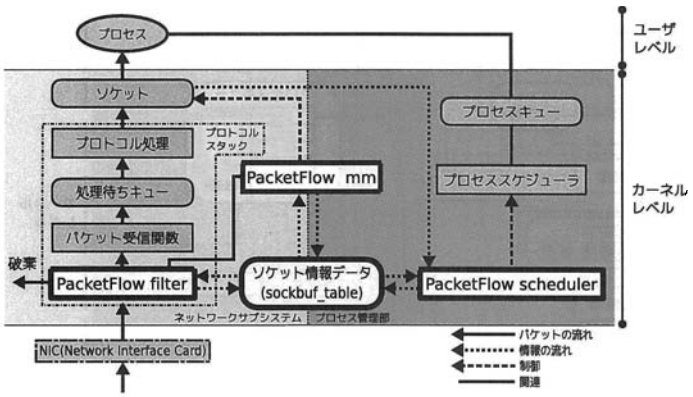

図 2: PacketFlow におけるパケット受信処理

3.1.1 PacketFlow filter での処理

PacketFlow filter のパケット破棄条件を示す.

条件 A パケットの使用プロトコルが UDP である

条件 B1 パケット受信時の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，パケット受信時の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．

PacketFlow ではパケットを受信するたびに上記の条件判定を行い，**条件 A** かつ**条件 B1** が成立したパケットは確率によって破棄し，**条件 A** かつ**条件 B2** が成立したパケットは破棄する．破棄を行う確率はパケット受信時の破棄の発生数によって変動するため，状況に応じたパケットの破棄を行うことができる．また，条件判定に必要なソケット情報は後述する共有変数から読み出すことで PacketFlow filter で行う処理を最小限に抑え，処理のオーバヘッドを削減している．

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検出し，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り，問題 2 を解決する．

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[2]とタイムスライスを増加させる．

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ため，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．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 PacketFlow scheduler での処理

優先プロセス選出のための条件を示す.

条件 a 全プロセスの中で一番高いプロセス優先度ではない

[2]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．

条件 b ソケットバッファの使用率が閾値 T より高い

条件 a では，プロセスが最も高優先ではないならば，そのプロセスはソケットバッファからパケットを取得するために他のプロセスの終了を待つ可能性があるため選出する．**条件 b** ではプロセスと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ため選出する．ここでの閾値 T はあらかじめユーザが設定を行うものとする．

提案システムでは，プロセスが起床する際に上記の条件判定を行い，**条件 a** および **b** を同時に満たすものを優先プロセスとして選出する．条件を満たすプロセスが複数存在する場合はプロセス優先度の上位 N 個を選出する．

優先プロセスとして選出されたプロセスの場合，プロセススケジューラがその動的優先度とタイムスライスを設定する際に通常より増加させて設定する．また優先プロセスはそのプロセスと対応するソケットが閉じられるか，選出されて一定時間が経過した場合には除外される．

3.3 PacketFlow mm

PacketFlow mm は対応するプロセスのプロセス優先度とソケットバッファの状況に基づいて，各ソケットバッファのサイズを動的に変更するモジュールであり，問題 3 を解決する．

通常ネットワークサブシステムではソケットバッファのサイズは静的に設定されており，ソケットに対応するプロセスのプロセス優先度に関係なく全て同じサイズが割り当てられる．ソケットオプションを用いることでソケット毎にサイズを設定できるが，使用状況に応じてサイズが動的に変わることはない．

PacketFlow mm ではソケットバッファ使用率が高いものはプロセス優先度に応じてその領域を増加させることでソケットレベル破棄の発生を防ぎ，プロセス優先度を適切に反映した資源割り当てを行うことができる．

3.3.1 PacketFlow mm での処理

PacketFlow mm は新たなソケットが作られると，そのソケットの通信開始時に対応するプロセス優先度を調査し，そのソケットに割り当てることのできるソケットバッファ領域の最大の大きさ SB_{max} を計算する．計算した値は後述する共有変数に格納しておく．そして使用率が高くなっているソケットバッファではその領域を，SB_{max} まで増加させる．また，全体で確保できるソケットバッファ領域にも制限があるので，ソケットバッファ領域を増加させる時にはその値を越えないようにする．PacketFlow filter を有効にしている場合は，ソケットバッファを増加させることのできる間は PacketFlow filter での破棄を行わないようにする．

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[6] に実装し，その評価実験を行った結果を示す．なお，PacketFlow mm については現在実装段階にあるため，他の 2 つのモジュールとは別に実験を行っている．

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+5 の範囲で値が変動するため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，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．

4.2 実験環境

実験環境として，計測用サーバと計測クライアント，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全て同様で，CPU は AMD Athlon 64 X2 3800+（2.0GHz），メモリは 2048MByte，OS は Debian GNU/Linux 4.0（Linux 2.6.22）となっている．

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プロセス優先度の異なるプロセスが複数存在した場合に，それぞれのプロセスのパケット受信率がプロセス優先度を適切に反映しているかどうかを調べた．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．このプロセスは全て同じプロセス優先度（nice 値[3]では 0），同じ送信率（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 256Byte である．一方，計測クライアント上ではそれぞれ送られてきたパケット数を数えるプ

[3]Linux での静的優先度．nice 値は-20 から 19 の間の値を取り，値が小さい方が優先度が高い．

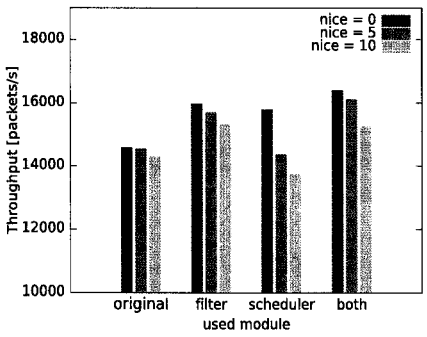

図 3: UDP パケットの受信率

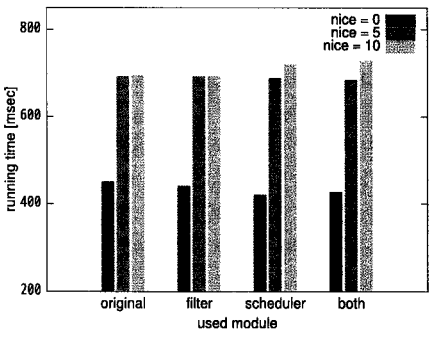

図 4: TCP パケットを受信するプロセスの実行時間

ロセスを生成する．この 3 つのプロセスはそれぞれプロセス優先度が異なり（nice 値では 0，5，10），各プロセスの受信率の計測を行った．

実験結果を図 3 に示す．縦軸は受信パケットの受信率，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．

まず，オリジナルのシステムではプロセス優先度によるプロセスの受信率の差はほとんど見られず，プロセス優先度が反映されていない．

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最も高優先であるプロセスでは 1,500 packets/s 程度，その他の 2 つのプロセスでも 1,000 packets/s 程度受信率が向上している．これはパケットの早期段階での破棄により使用する 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．またオリジナルのシステムと比較して，プロセス優先度によるパケット受信率の差もみられる．

PacketFlow scheduler のみを用いた場合，高優先プロセスの受信率が 1,200 packets/s 程度向上している．このプロセス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．一方，最も低優先であるプロセスでは受信率が 600 packets/s ほど減少している．これはプロセス優先度の高いプロセスを優先し，低優先プロセスへの CPU 資源の割り当て量が減少したためであると考えられる．

両モジュール共に用いた場合は PacketFlow filter のみを用いた場合よりも，最も高優先のプロセスと中程度の優先度のプロセスの受信率が向上した．最も高優先であるプロセスはオリジナルのシステムと比べて 2,000 packets/s 程度（約 14%）増加している．PacketFlow filter のみを用いた時の処理に加えて，PacketFlow scheduler によって優先プロセスとして選出される高優先プロセスと中程度の優先度のプロセスへの優先処理が行われることによる結果であると考えられる．

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．TCP の場合は通常全てのパケットを受信することができるため，プロセスがパケットの受信を開始してから 100,000 パケットを受信するまでの時間を計測した．

実験結果を図 4 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，横軸は使用したモジュールである．

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おり，TCP パケットを受信した場合は処理は行わないためである．

PacketFlow scheduler のみを用いる場合と両モジュール共に用いる場合ではオリジナルのシステムよりも最も高優先であるプロセスの実行時間は約 30 msec 短く（約 7%向上），最も低優先であるプロセスの実行時間は約 25 msec 長くなった．プロセスの優先処理を行うためオリジナルのシステムとは挙動が異なり，高優先プロセス以外でもプロセス優先度による実行時間の変化が見られた．

4.5 両プロトコルが混在した状態での実験

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，プロセス優先度を高

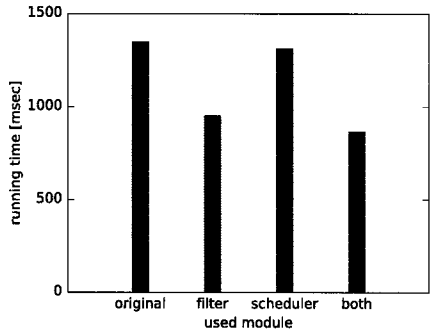

図 5: 高優先プロセスの実行時間

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）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，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 パケットを受信するまでの時間を計測した．

実験結果を図 5 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みられる．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，一番よい結果となった．

また逆に，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 いずれに対してもプロセス優先度を反映した受信処理を行うことができていると言える．

4.6 ソケットバッファ増加による受信率の改善

この実験では PacketFlow mm でソケットバッファのサイズを増加させることによるパケット受信率の改善効果を調べた．計測用サーバから計測クライアントへ UDP パケットを送信するプロセス優先度の異なるプロセスを 3 つ生成し（nice 値は 0，5，10，スループットは 37,000packet/s），計測クライアントで各プロセスの受信率を計測した．

PacketFlow mm でのプロセス優先度によるソケットバッファ領域の最大の大きさ $SOCKBUF_{max}$ を決定する式を以下のようにした．

$$SOCKBUF_{max} = \frac{default\_size \times (140 - prio)}{20}$$

この式は Linux でのプロセスのタイムスライス設定に用いられる式と同様のものとなっている．$default\_size$ は nice 値が 0 の時のソケットバッファの最大値で今回は 500KByte に設定した．$prio$ は nice 値を Linux 内での優先度に変換した値[4]である．

実験結果を図 6 に示す．オリジナルのシステムでは全てのソケットバッファが同じ大きさに設定され，パケットは公平に扱われるためプロセス優先度によって受信率に大きな違いはない．PacketFlow mm を用いた場合ではプロセス優先度によって確保するソケットバッファの大きさに違いがあるため，その受信率にも違いがみられた．通信終了時には全てのソケットバッファが $SOCKBUF_{max}$ までその領域を増加させていることが確認できた．また，オリジナルのシステムのソケットバッファサイズのデフォルト値はとても小さなものになっているため，全てのプロセスでオリジナルのシステムよりもよい結果が得られた．

以上より PacketFlow mm でソケットバッファサイズを制御することは有用であると考えられる．今後はこの

[4]nice 値で-20 から 19 は，Linux 内での優先度では 100 から 139 に変換される．

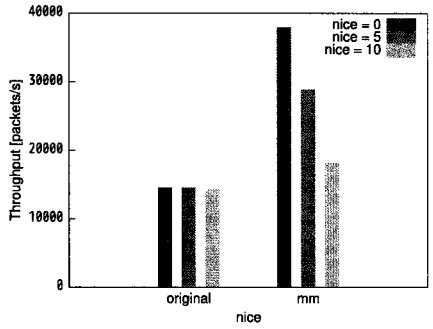

図 6: PacketFlow mm での改善効果

結果を考慮してソケットバッファサイズの増加方法に関するポリシーを決定し，また TCP ソケットバッファにも対応できるように実装を進めていく．

5 関連研究

尾崎らによる PBQs[3] ではプロトコルスタックの下位層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使用して，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ことで受信処理において優先度を反映させている．しかし，ソケットバッファで起こるソケットレベル破棄や通信帯域の減少は考慮していない．

BSD[5] に関する Druchel らによる LRP[2] や，Brustoloni らによる SRP[1] 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．パケット破棄を行う場合はこのキューで行うことで無駄な CPU 資源の利用は抑えられている．しかし，ソケットバッファから取り出すパケットの量に関しては考慮していない．

寺井らによる E^2-ATBT[4] 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，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考慮していない．

6 まとめ

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，プロセス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．また PacketFlow 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プロセス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．

今後の課題としては，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挙げられる．また PacketFlow mm のポリシーを決定し，実装を進めていく．

参考文献

[1] Jose Brustoloni, Eran Gabber, Abraham Silberschatz, and Amit Singh. Signaled Receiver Processing. In *USENIX 2000 Annual Technical Conference*, pp. 211–223, 2000.

[2] Peter Druschel and Gaurav Banga. Lazy Receivevr Processing(LRP):A Network Subsystem Architecture for Server System. In *USENIX Symposium on Operating System Design and Implementation*, pp. 261–276, 1996.

[3] 尾崎亮太, 中山奏一. 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, 第 45 巻, pp. 785–793, March 2004.

[4] 寺井達彦, 岡本卓也, 長谷川剛, 村田正幸. Web プロキシサーバにおける動的資源管理方式の提案と実装. 電子情報通信学会技術研究報告. IN, 情報ネットワーク, 第 101 巻, pp. 25–32. 電子情報通信学会, 2001.

[5] The FreeBSD Project. `http://www.jp.freebsd.org/www.FreeBSD.org/`.

[6] The Linux Kernel Achives. `http://www.kernel.org/`.

[7] 山本紫乃, 楢崎修二. ネットワークシステムとプロセススケジューラとの協調による udp パケット破棄の削減. 第 5 回情報科学技術フォーラム, No. 4M-8 L-044. 電子情報通信学会，情報処理学会, September 2006.