平成29年度 特別研究報告書

カーネル空間からユーザ空間へ大容量データを通信するための機構の提案

龍谷大学 理工学部 情報メディア学科

学籍番号 ： T140504
氏名 ： 松浦 拓巳

指導教員 ： 三好 力 教授，芝 公仁 助教

内容梗概

Linuxでは，一般的にカーネルとプロセスの間でデータをやり取りする手法として，writeやreadシステムコールを用いたsysfsやdebugfsといった仮想ファイルシステム，あるいはソケット通信をベースとするNetlinkなどを用いることが多い．

しかし，これらの既存の手法では，通信データの容量に制限が設けられている．さらに，ユーザ空間リスナの遅延によるバッファオーバーランが発生する可能性がある．これらの問題により大容量なデータの通信や，高頻度なデータを通信する状況での使用には適さない．そこで本研究では，カーネル内機構とプロセスがメモリを共有して通信する手法に着目し，既存手法の問題を解決する通信機構の開発を目指した．

本論文では，カーネル内機構が持つ大容量なデータをプロセスへ提供する通信機構として，共有メモリを用いてコピーを伴うことなく効率的に通信する手法を提案する．また，その有用性を確認するための基礎的な評価実験を行う．実験の結果，カーネル内機構からプロセスへ大容量なデータの通信が実現でき，さらにデータ通信における速度の向上が確認された．

目 次

1 はじめに **1**

2 カーネルとプロセスの通信 **2**

2.1 カーネルが提供する通信機構 2
2.2 仮想ファイルシステム 2
2.3 BSD ソケット 3
2.4 既存手法の問題点 4

3 提案手法 **6**

3.1 メモリを共有した通信機構 6
3.2 Memlink の概要 6
3.3 Memlink の動作 9
3.4 共有メモリの管理を行う Memlink の提案 11

4 評価 **13**

4.1 評価方法 13
4.2 実験結果 14
4.3 考察 16

5 おわりに **17**

謝辞 **18**

参考文献 **19**

A 付録 **20**

1　はじめに

オペレーティングシステムは，一般的にユーザ空間とカーネル空間の二つの層に分けられ構成されている．カーネル空間とユーザ空間との間でデータを交換するために，Linux カーネルでは RAM ベースの仮想ファイルシステムやソケットに基づく通信機構などがしばしば用いられている．

仮想ファイルシステムとして proc ファイルシステムや sys ファイルシステムなどが代表的で，これらの通信機構はファイルに基いてカーネルからプロセスへ情報を提供する．プロセスは標準の read や write システムコールを用いて情報にアクセスを行い，読み書きにはユーザコマンドの cat や echo といった多くのツールが利用できる．しかし，これらの仮想ファイルシステムには，書込みバッファに対する制限を設けられている．そのため，大容量なデータの通信ができない．また，物理メモリページにバッファリングを行うため，データの通信時に遅延が発生する可能性がある．

ソケットベースのメカニズムでは，カーネル内機構とプロセスの間でデータを通信するために，特殊な IPC である Netlink ソケットが用いられている．Netlink ソケットはデータの通信にメッセージキューイング機能を用いるため，非同期なプロセス間通信を提供する．しかし，カーネル内機構からプロセスへ大容量データを高頻度に通信する状況ではプロセスのリスナの遅延によりバッファオーバーランが発生することがある．さらに，バッファ空間の使用に上限が設けられているため，カーネル内機構が持つ大容量データをプロセスに通信することができない．

本論文では，カーネル空間からユーザ空間に大容量データを通信するための機構を提案する．この機構では，カーネル内機構の持つデータをメモリに書き込み，プロセスではそのメモリを共有することでコピーを伴わない効率的な通信をすることができる．さらに，メモリを共有することで大容量なデータを通信することが可能になる．

第 2 章ではカーネル内機構とプロセスがデータの通信をするために用いられる既存の機構の概要を紹介し，その問題点について分析する．第 3 章は提案手法について述べ，第 4 章に提案手法を用いて，カーネル内機構の持つ大容量なデータをプロセスへ通信する実験を行い，結果と考察について記述する．第 5 章では本論文のまとめを行う．

2　カーネルとプロセスの通信

本章では，カーネルとプロセスの間の通信機構として既存手法の概念について説明する．また，その問題点について記述する．

2.1　カーネルが提供する通信機構

Linux では，カーネル空間とユーザ空間でデータを通信するための機構としてシステムコール，ioctl，proc ファイルシステム，ソケットなどを提供し，これらは様々な目的のために用いられ，各々異なった性質を有している．特にシステムコールに基づいた通信機構は，他の多くのオペレーティングシステムと同様に，Linux の主要なカーネルインタフェースである．これらの通信機構は，BSD ソケットにファイルのような仕組みを提供する仮想ファイルシステム及び，プロセスが BSD ソケット API を用いてカーネル内機構との間でデータを送受信する機能に大きく分類することができる．

2.2　仮想ファイルシステム

Linux は，カーネル空間とユーザ空間が通信するために使用できる，いくつかの仮想ファイルシステムを提供している．ファイルシステムは，コンピュータのリソースを操作するためのオペレーティングシステムが持つ機能の一つで，プロセスがデータの読み込みや書き込みをストレージに対して行う際に仲介するカーネルのサービスである．デバイスやプロセス，またカーネル内の情報といったものもファイルとして提供するシステムが仮想ファイルシステムである．図 1 に，仮想ファイルシステムの概要を示す．

　仮想ファイルシステムの上にあるのは，カーネルのシステムコールインターフェースで，このシステムコールによってユーザ空間からカーネル空間へと呼び出しを移行することができる．例えば，プロセスが POSIX の open を呼び出すと，その呼び出しは GUN C ライブラリを介してカーネル空間へ渡され，最終的にシステムコールの open によって仮想ファイルシステムが呼び出される．カーネルは様々な仮想ファイルシステムを持つ一方で，これらはすべて別々の目的のために設計されている．

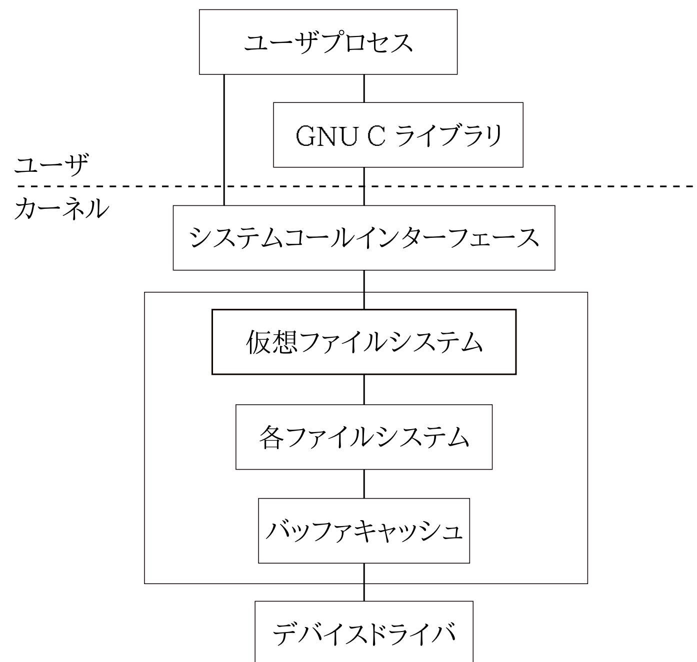

図 1: 仮想ファイルシステムの概要

2.3 BSD ソケット

Linuxではカーネル空間とユーザ空間がソケットを用いて通信する機構として，Netlinkがある．Netlink プロトコルは，カーネル内機構とプロセス間でデータを通信するために用いられるソケットベースの IPC 機構であり，BSD ソケットに基づいて，データグラム指向のメッセージング機能を提供するソケットファミリである．Netlink ソケットは，プロセス用の標準ソケット API とカーネルモジュール用の特別なカーネル API を使用し，単一の接続先だけにソケットからメッセージを送る他，マルチキャストグループにも送ることができる．図 2 に Netlink を用いたユニキャスト及びマルチキャストな通信の概要を示す．

Netlink ではソケットキュー領域が提供されているため，送信者はデータをキューイングするだけで受信者と非同期なデータの通信をすることが可能である．また，最大 32 のマルチキャストグループを各 Netlink プロトコルタイプごとに定義することができる．

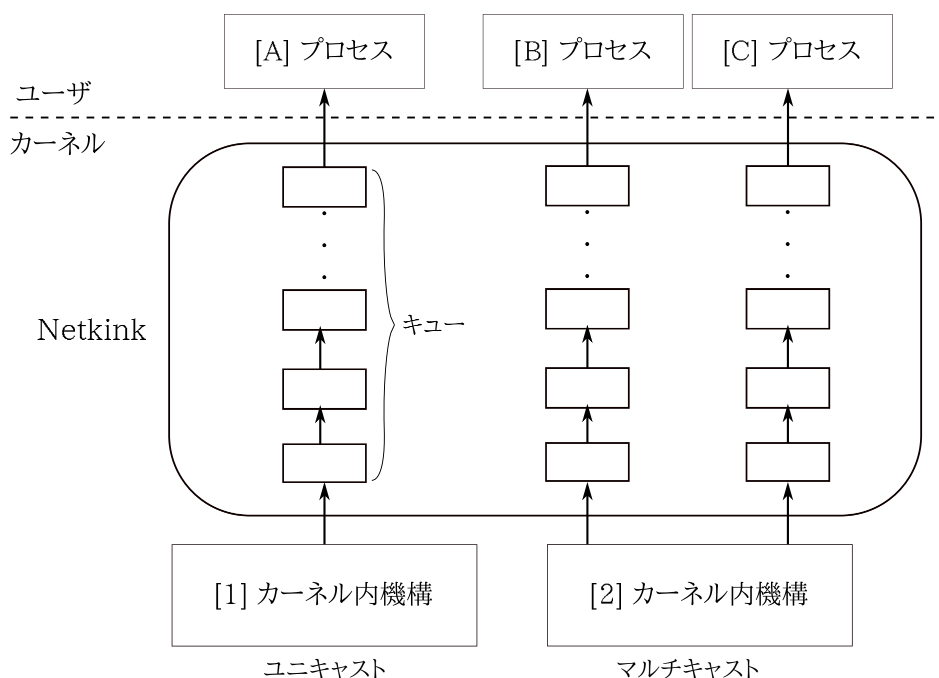

図 2: Netlink を用いた通信の概要

2.4 既存手法の問題点

上記に，カーネルとプロセスの間に存在する代表的な通信機構の概念について明らかにした．本章では，それらの既存手法が大容量なデータが通信される環境において抱える問題を分析する．

仮想ファイルシステムやソケットを用いた機構は，カーネル空間とユーザ空間でデータを双方向に通信する手段として提供されている．しかし，データを受信する場合は，受け取ったデータを自身のアドレス空間にコピーするためデータの送信速度に遅延が発生する問題を抱えている．送信する場合も上記と同様にデータのコピーが発生する．そのためカーネル内機構とプロセスが大容量なデータを通信する状況での使用には適さない．また，多くの仮想ファイルシステムでは，書込みバッファにメモリの 1 ページ分までの制限が設けられており，Linux では 4096 バイトまでのデータしかプロセスに提供できない．仮想ファイルシステムを用いて，制限を超えるデータを通信するとデータの欠落が起こる．そのため，仮想ファイルシステムは大容量のデータをプロセスへ提供する用途として最初から向いていない．また，Netlink を用いた通信ではメモリの使用に上限が設けられており，大容量データの通信を行うとなるとメモリ枯渇の問題が発生

する．さらに，カーネル空間からユーザ空間へデータの通信を行うとき，バッファオーバーランが発生する可能性がある [1]．これは，カーネル内機構が送信データをキューに書き込む速度と比較し，プロセスのデータの読み出しが非常に遅いため，キューのサイズを超える送信データの書き込みが起こりバッファオーバーランが生じる．確保するキューの領域が小さいと発生する確率は高くなるため，キュー領域を拡張することで一時的にバッファオーバーランを回避することが可能だが，いずれにせよ，こうした欠陥はセキュリティ上の深刻な問題になりかねない．

3　提案手法

本章では，カーネル空間からユーザ空間へ共有メモリを用いた大容量データを通信するための Memlink を提案する．Memlink の概要と実装について述べる．また，カーネル内機構が共有メモリを持つとき，その共有メモリ内のデータをプロセスへ効率的に通信するための，共有メモリを管理する機能を提案する．

3.1　メモリを共有した通信機構

本節で前章の問題を考慮した手法として，カーネル内機構とプロセスがメモリを共有しデータの通信を行う手法がある．本手法では，共有メモリ機構を利用しカーネル内機構の持つデータをプロセスへ提供することで上記の問題に対処し，既存の通信機構より大容量なデータの通信を実現し，さらにデータの通信速度の向上を図る．

共有メモリは，カーネル内機構とプロセスで一つのメモリ領域を共有し，効率的に共有メモリのデータを参照することができる手法である．共有メモリを用いることで，read や write といったシステムコールの発行を行わないため，プロセスのアドレス空間にデータのコピー処理が発生せず，データの読み書き性能が向上し，高速で大容量なデータの通信が可能となる．

しかし，これらのシステムコールでは，ファイルオフセットの調整とデータの読み書き操作はアトミックな処理として実行される．つまり，共有メモリを用いた通信によるデータの信頼性を確保するため，カーネル内機構とプロセスが共有メモリへ同時にアクセスしないように制御する機能が必要となる．また，カーネルとプロセスが同期を取る機能の実装が必要である．共有メモリへアクセスを制御する機能は次節で，さらに同期処理の機能については 3 章の 4 節で提案ている．

3.2　Memlink の概要

本節では，開発を行ったカーネル内機構から大容量データをプロセスへ共有メモリ用いて通信を行う Memlink について，概要と動作を述べる．Memlink では，共有メモリを用いてカーネル内機構のデータをプロセスに通信することで，前章で取り上げた問題点に対処し，大容量なデータ通信を実現し，さらにデータ通信速度の向上を図る．

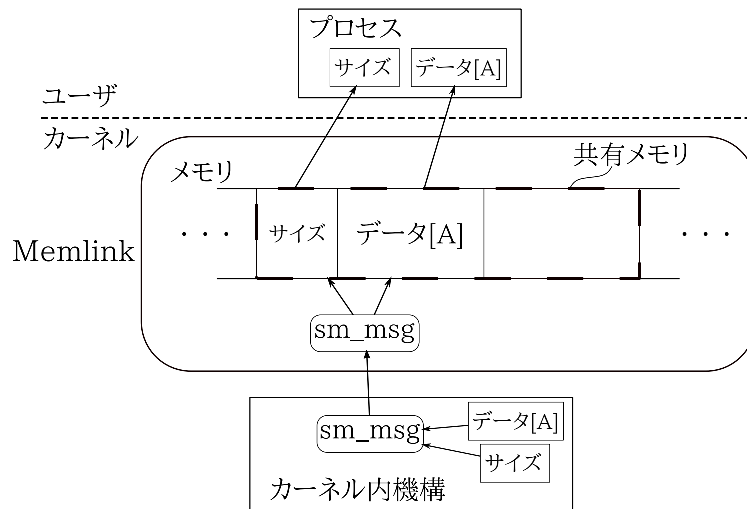

図 3: Memlink を用いた通信の概要

Memlink を用いたカーネル内機構とプロセスの通信の概要を図 3 に示す．

Memlink は共有メモリを確保し，そのメモリにカーネル内機構が書き込んだデータをプロセスへ提供するための機構である．カーネル内機構はプロセスにデータを提供するため，Memlink が提供する sm_msg に送信データと送信データのサイズを書き込む．sm_msg が構成する要素として，カーネル内機構の持つ送信データの長さを登録する len と送信データを登録する data 配列がある．共有メモリには，カーネル内機構から sm_msg 内に登録されたサイズと送信データが書き込まれる．プロセスは Memlink を用いることで共有メモリにアクセスし，カーネル内機構の持つデータを参照することができる．Memlink が提供する sm_msg を用いて通信することで，カーネル内機構とプロセスへ排他制御の機能を提供する．つまり，プロセスとカーネル内機構は競合することなく共有メモリを読み書きすることができる．しかし，sm_msg を用いると，競合を避けるためにロック処理が行われ，性能が低下する恐れがある．そのため，sm_msg を用いることなくカーネル内機構から直接，共有メモリにデータを書き込むための機能も提供している．そして，Memlink では共有メモリの最後尾までデータを書き込むと，再び共有メモリの先頭から順にデータを上書きし始めるため，データが途切れることなく書き込みを続けることができる．

Memlink はカーネル内機構が使うインターフェースとして，my_create_file 関数と my_trans_data 関数を提供している．my_create_file 関数では，カーネル内機構からファイル名と確保する共有メモリのサイズを受け取る．Memlink は，my_create_file 関数で

カーネル内機構の指定したサイズで共有メモリを確保し，debug ファイルシステムを用いてファイルを作成した後，共有メモリ領域をファイルにアタッチする．そして，カーネル内機構に作成した共有メモリの先頭アドレスを返す．my_create_file 関数で共有メモリ領域をファイルにアタッチすることで，プロセスがそのファイルのファイルディスクリプタを用いて共有メモリの先頭アドレスを取得することができる．my_trans_data 関数では，カーネル内機構から sm_msg を受け取り，共有メモリに送信データのサイズと送信データを書き込む処理が行われる．また，プロセスと Memlink が同時に共有メモリへアクセスしないように排他制御の機能が備わっている．

プロセスには，init_memlink ライブラリ関数を提供している．プロセスは init_memlink にカーネル内機構で指定したファイル名と共有するメモリのサイズを引数に与える．init_memlink では，引数のファイル名を open してファイルディスクリプタを取得し，そのファイルディスクリプタを手掛かりに Memlink が確保した共有メモリのページをマップする．また，プロセスは，spin_memlock 関数と spin_memunlock 関数を使用することができる．これは，カーネル内機構と排他制御を実現するための機能である．spin_memlock 関数を用いることで，Memlink と同期し，カーネル内機構から受け取った送信データを Memlink が共有メモリに書き込みを行っているとき，プロセスから読み込みが発生しないように制御する．同様にプロセスが共有メモリのデータを参照している場合，Memlink から共有メモリにデータを書き込む処理がロックされる．spin_memunlock 関数は，Memlink に対するロックを解除する処理である．そして，close_file 関数では，プロセスの終了時にファイルのクローズやマップした共有メモリをアンマップする処理を提供する．排他制御の概要を 4 図に示す．

次にロック制御値について，その役割を表 1 に示す．

表 1: フラグに対する共有メモリ領域の状態

フラグ	読み書きの状態	メモリのロック状態
0	読み書き無し	ロック無し
1	データを書き込み中	共有メモリ領域の読み込みをロック
2	データを読み込み中	共有メモリ領域の書き込みをロック

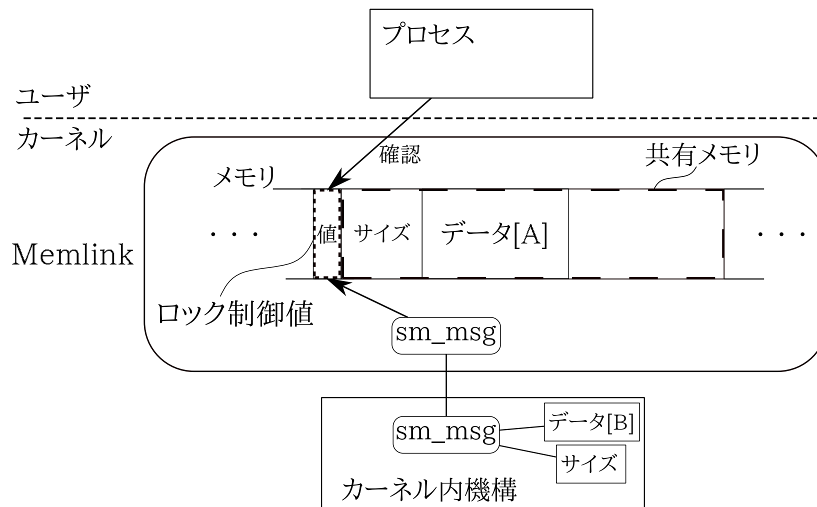

図 4: Memlink の提供する排他制御の概要

3.3 Memlink の動作

本節で Memlink の動作について述べる．カーネル内機構が Memlink を用いてプロセスへデータを通信する手順を以下に示す.

- カーネル内機構 (my_trans_data を用いてデータを書き込む場合)

[1] my_create_file を用いて共有メモリを確保する

[2] sm_msg に送信データとそのサイズを登録する

[3] my_trans_data を用いて共有メモリにデータを書き込む

- カーネル内機構 (カーネル内機構から直接共有メモリに書き込む場合)

[1] my_create_file を用いて共有メモリを確保する

[2] my_create_file から返る共有メモリの先頭アドレスを用いて，直接共有メモリに送信データを書き込む

カーネル内機構は，my_create_file 関数を用いて共有メモリのサイズとファイル名を決める．Memlink は，my_create_file でカーネル内機構が決めたサイズで共有メモリを確保する．そして，Memlink でカーネル内機構から指定されたファイル名のファイルを debug ファイルシステムを用いて作成する．ファイルの作成と共有メモリの確保に

成功すると my_create_file は，カーネル内機構に共有メモリの先頭アドレスをカーネル内機構に返す．次にカーネル内機構は，sm_msg の len に送信データのサイズと data 配列に送信データを登録する．カーネル内機構から my_trans_data を用いて sm_msg を Memlink に渡す．Memlink はカーネル内機構から受け取った sm_msg の len に登録された値を共有メモリに書き込み，len の型のバイト分進んだメモリのアドレスから len の値分のデータを配列から共有メモリに書き込む．このとき，Memlink は共有メモリの最後に書き込んだアドレスを記憶しているため，カーネル内機構は [2] から [3] の動作を繰り返すだけで順番に共有メモリにデータを書き込むことができる．また，排他制御が必要ない場合，my_trans_data を用いずに my_create_file が返す共有メモリの先頭アドレスを用いることで，カーネル内機構から直接共有メモリにデータを書き込むことができる．

次にプロセスが Memlink を用いてカーネル内機構の送信データを参照する手順を以下に示す．

- プロセス

[1] init_memlink を用いて共有メモリの先頭アドレスを取得する

[2] 必要に応じて spin_memlock を実装する

[3] 共有メモリ領域からデータを取得する

[4] spin_memlock を実装した場合，spin_memunlock を実装する

[5] close_file を実行する

プロセスから init_memlink にカーネル内機構で指定したファイル名と共有したいメモリのサイズを与えることで共有メモリの先頭アドレスを取得することができる．カーネル内機構で my_trans_data 関数を用いて送信データを共有メモリに書き込んでいる場合，必要に応じて spin_memlock を実装することができる．プロセスは init_memlink の返す共有メモリの先頭アドレスを用いて共有メモリからカーネル内機構のデータを参照する．プロセスは共有メモリの先頭から送信データのサイズを参照し，そのサイズの値を記憶する．そして，共有メモリから sm_msg の len の型のバイト数を進んだアドレスから送信データのサイズの値だけ，共有メモリを参照することでカーネル内機

構の送信データを取得できる．spin_memlock を実装した場合，共有メモリのデータを参照したあとに spin_memunlock を行う必要がある．また，プロセスとカーネル内機構の競合を避けるため，共有メモリを読み込む前に spin_memlock 関数を使用することができる．プロセスが共有メモリを読み込みを終えると spin_memunlock 関数を呼び出す必要がある．プロセスを終了するとき，close_file 関数を呼び出す．

3.4 共有メモリの管理を行う Memlink の提案

前節では Memlink が共有メモリを確保し，その共有メモリを用いてカーネル内機構のデータをプロセスが参照する手法を紹介した．しかし，この機構ではカーネル内機構で確保したメモリをプロセスが参照できる機能を提供していない．カーネル内機構とプロセスのメモリアクセスの競合は，前節で説明したように Memlink が提供する my_trans_data 関数や spin_memlock 関数を用いると回避することができる．しかし，Memlink は，カーネル内機構が持つ送信データを共有メモリの最後尾まで書き込みが完了すると，共有メモリの先頭から順に古いデータを新しいデータに上書きする．その結果，プロセスの読み込みが遅いと途中から新しいデータに書き換わってしまう事態に陥る．そのため，プロセスと完全な同期を取れるような共有メモリを管理する機能が必要である．そこで本節では，プロセスがカーネル内機構の持つ共有メモリにコピーを伴うことなく効率的に参照するための，共有メモリの管理を行う Memlink の機能を提案する．

メモリを管理する機能を持つ Memlink を用いた，カーネル内機構とプロセス間の通信の概要を図 5 に示す．この Memlink は，カーネル内機構が持つ共有メモリをプロセスへ提供するための機構であり，data list 及び free list の 2 つのリストを用いて共有メモリを管理する．data list は，カーネル内機構の共有メモリの情報を持つ sm_info の管理を行う．sm_info は，共有するメモリの先頭アドレスとサイズを持つ．free list では，プロセスからの参照が完了し，使われていない共有メモリの sm_info を管理する．カーネル内機構は共有メモリを確保し，Memlink を用いてプロセスへメモリ内のデータを提供する．プロセスは，Memlink を用いることでカーネル内機構の持つメモリを共有し，そのデータを参照することができる．Memlink は，カーネル内機構が持つ共有メモリ内の大容量なデータをプロセスへ，コピーを伴うことなく効率的に通信すること

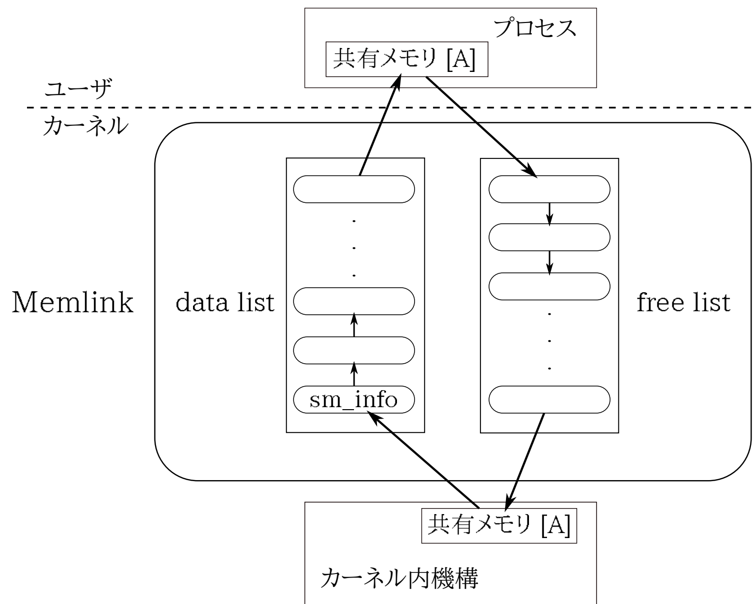

図 5: Memlink を使った通信の概要

ができる．

Memlink を用いた，カーネル内機構とプロセスのデータの通信手順を以下に示す．カーネル内機構は共有メモリを確保し，そこにプロセスへ提供するための送信データを置く．そして，その共有メモリの情報を sm_info に入れ，data list の先頭に登録する．Memlink はプロセスからデータの要求が発生すると，data list の最後尾の sm_info に対応する共有メモリをプロセスのアドレス空間内に割りつける．プロセスは共有メモリからデータを取り出し，共有メモリ領域の開放を Memlink に要求する．そして，Memlink によって対応する sm_info が free list の先頭から順番に登録される．カーネル内機構は Memlink の機能を用いて free list から sm_info を取り出し共有メモリの情報の登録を行う．data list に sm_info がないときプロセスは待ち状態になり，共有メモリの情報が data list に登録されるとプロセスは Memlink に起こされ，データの要求を行い始める．こうして，カーネルとプロセスが同期して通信を行うことができる．以降，上記の手順を繰り返し行うことでプロセスは，カーネル内機構の持つ共有メモリのデータに直接参照することができる．

表 2: 実計算機の仕様表

Name	Description
OS	Ubuntu 16.04 (64bit)
カーネル	Linux-4.4.0
CPU	Core i7-4790 3.60GHz
SSD	512GB
Memory	32GB

表 3: 仮想計算機の仕様表

Name	Description
OS	Ubuntu 16.04 (64bit)
カーネル	Linux-4.4.1
SSD	50GB
Memory	16GB

4 評価

本章では，提案手法が既存の通信機構より，大容量データを効率的にカーネル内機構からプロセスへ通信できるか確認するための評価実験について記述する．評価実験の結果とその考察について述べる．実験マシンの構成を表 2 に示す．

4.1 評価方法

Memlink は，カーネル内機構から大容量なデータをプロセスへ通信するための共有メモリを提供する．共有メモリを用いることでプロセスは，カーネル内機構が持つデータにコピーを伴うことなく参照することが可能となった．Memlink の効果を確認するため，カーネル内機構で用意した大容量データをプロセスへ通信し，カーネル内機構がデータを書き込む速度，及びプロセスがデータを参照する速度を測定する実験を行った．実験は表 3 の仮想実験機上で行った．

カーネル内機構から通信するデータは，1byte のデータを大量に書き込んだメモリの情報を用いる．そのデータを 0.5GB，1GB，5GB に変化させ，カーネル内機構からプロセスへ提供する．Memlink が提供する排他制御の機能を使用せず，直接カーネル内機

構からデータを共有メモリに書き込む手法で実験を行う．カーネル内機構は Memlink を用いて，共有メモリの作成から実際に送信データを書き込むまでの時間を測定する．カーネル内機構が共有メモリにデータを書き終えてから，プロセスが共有メモリの情報を取得し，すべてのデータを参照するまでの時間を測定した．それぞれ，実験を 10 回行い，測定した速度から平均を取る．

また，カーネルとユーザ空間で通信を行える既存の機構を用いて上記と同様の実験を行う．今回は，ソケットベースの Netlink を用いた通信と比較する．カーネル内機構が Netlink を用いて，ソケットバッファに送信データを書き込むまでの時間を測定する．そして，プロセスがソケット上のデータをすべて受信するまでの時間を測定する．

4.2 実験結果

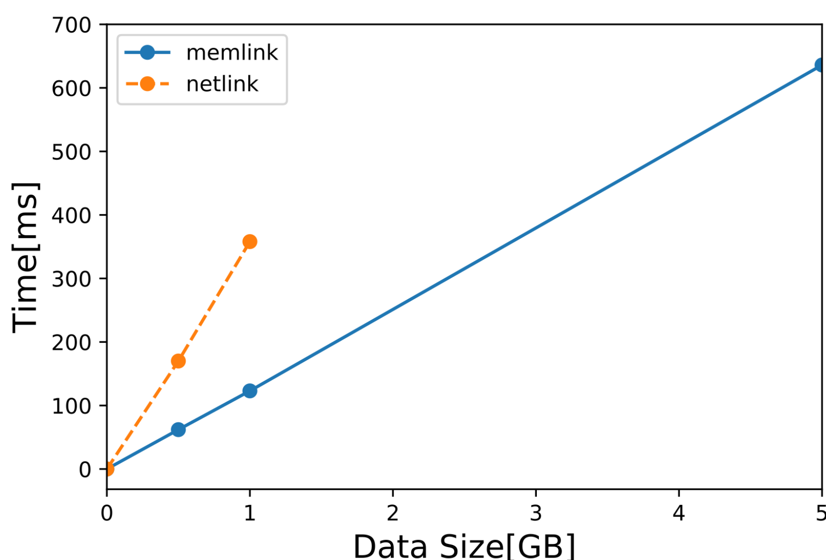

図 6: カーネル内機構によるデータの平均書き込み速度

表 4: カーネル内機構によるデータの平均書き込み速度

通信サイズ [GB]	Memlink[ms]	Netlink[ms]
0.5	62.80	170.20
1	123.23	358.39
5	627.97	

カーネル内機構が送信データを書き終えるまでの測定結果を表 4 と数値データを棒グラフにしたものを図 6 に示す．図 6 は，縦軸がカーネル内機構から送信データを書き込み終えるまでの処理時間の平均，横軸が送信データのサイズである．結果から，カーネル内機構の持つデータを書き込み終えるまでの時間は，Netlink に比べ Memlink は 0.5GB で約 2.7 倍，1GB で約 3 倍の速さで処理を終えていることが確認できる．5GB のデータを通信する実験において，Memlink では 1GB のおよそ 5 倍の処理時間を費やしたが，Netlink ではプロセスのバッファ空間が不足してプロセスが終了した．Netlink は，データの送信先をロストし完全にデータをプロセスに通信することができなかった．そのため，カーネル内機構がデータを書き込む処理時間を図ることができなくなった．

次に，プロセスがカーネル内機構の持つ送信データを参照し終えるまでの測定結果を表 5 と数値データを棒グラフにしたものを図 7 に示す．

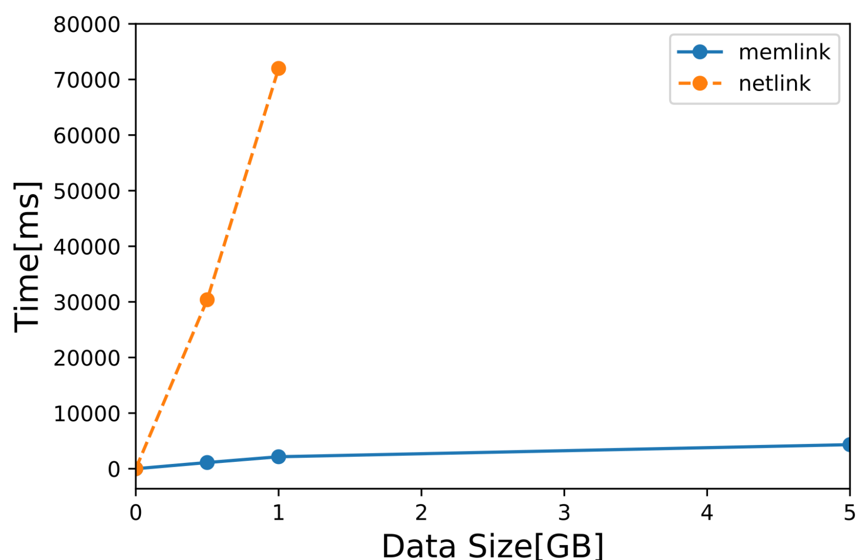

図 7: プロセスによるデータの平均参照速度

表 5: プロセスによるデータの平均参照速度

通信サイズ [GB]	Memlink[ms]	Netlink[ms]
0.5	1112	30384
1	2153	71983
5	4328	

図 7 の縦軸は，プロセスがカーネル内機構の持つ送信データを参照し終えるまでの処理時間の平均，横軸が参照するデータのサイズである．Netlink を用いて大容量のデータを通信するため，ソケットのバッファサイズを 1600MB まで拡張して実験を行っている．しかし，Netlink を用いて 5GB のデータサイズで実験を行ったところ，バッファ空間の不足によりデータの参照に失敗しプロセスが終了した．そのため，カーネル内機構と同様にプロセス側でも 5GB の実験結果を得ることができなかった．プロセスがデータを参照し終えるまでの測定結果から，0.5GB では約 27 倍，1GB ではおよそ 33 倍，Memlink のほうが短い時間で参照できることが確認できる．Netlink による通信では，データサイズの増加に伴い処理時間が大きく増加する結果となった．

4.3 考察

評価実験の結果，カーネル内機構からプロセスへ Memlink を用いた通信は，既存の機構より高速に大容量なデータを通信できることが確認された．カーネル内機構から送信するデータの増加に伴い，Memlink と Netlink の通信に費やされる時間の差が大きくなっている．これは，2 章で紹介したようにソケットなどの既存の通信機構に，メモリ内でコピーを伴うシステムコールを用いられていることが原因として考えられる．プロセスはカーネル内機構から送信されたデータをシステムコールの read や recv を用いて取得する．これらのシステムコールは，カーネル内機構からバッファ空間に書き込まれた送信データをユーザ空間があらかじめ確保したメモリにコピーしている [2]．この動作により，プロセスが送信データを読み込む際に遅延が発生すると考えられる．

今回の実験では，Memlink を実装すると同時にカーネル内機構から共有メモリに 1 度だけデータの書き込みが行われる．そのため，プロセスが起動する前にカーネル内機構がデータの書き込みを終えているため，カーネル内機構とプロセスが共有メモリ上で競合するような状況に陥らない．Memlink が提供するロック機能を使用せず，カーネル内機構から直接共有メモリにデータ書き込んでいるため性能に影響が生じることなく高速な通信が行えた．カーネル内機構とプロセスが競合する状況で通信を行うと性能が低下し，処理に多少の遅延が生じていたと考えられる．また，競合が発生する状況で送信データに欠損が発生していないか，十分に検証を行う必要ある．

5　おわりに

本論文では，カーネル内機構が持つデータをプロセスに通信するため，Memlink により共有メモリを提供することで通信の効率化を図った．共有メモリを用いることで既存の通信機構より大容量データを効率的に通信することを実現している．また，共有メモリを用いることでプロセスが送信データを読み込む際にコピーを伴わないため既存手法より高速な通信を実現していることが実験の結果から確認できた．

今後の課題として，第 3 章の 3 節で記述した，カーネル内機構が持つメモリをプロセスへ提供するためのメモリ管理機能を Memlink に実装し，その有用性を確認する必要がある．この機能を実装することで，Memlink の抱えるカーネル内機構とプロセスの同期処理の問題を解決することが可能となる．また，提案手法を実際に，動画や画像などの大容量データをプロセスと通信するカーネルシステムに実装し，性能が向上することを検証する必要がある．

謝辞

本研究を行うにあたり，終始熱心な御指導と御鞭撻を頂いた，三好力教授，芝公仁助教に心から感謝致します．また，日頃参考となる貴重な御意見や御協力を頂いた芝研究室の皆様に深く感謝致します．

参考文献

[1] Pablo Neira-Ayuso, RafelM．Gasca, Laurent Lefevre. Communicating between the kernel and user-space in linux using netlink sockets. 2010.

[2] 沖勝. クラウドコンピューティングテクノロジー:SDN ソフトウェアスイッチ「Lagopus」. 2014.

A 付録

Memlink のヘッダーファイル

```
#ifndef MEMLINK
#define MEMLINK

struct sm_msg{
  unsigned int  len;
  unsigned char data[0];
};

extern int (*my_trans_data) (struct sm_msg*, size_t memsize);
extern char* (*my_create_file) (char *file, size_t memsize);

char *create_file(char *filename, size_t memsize);
int trans_data(struct sm_msg*, size_t memsize);

#endif
```

Memlink のカーネルモジュール

```
#include <linux/module.h>
#include <linux/kernel.h>   //for printk
#include <linux/init.h>     //for __init __exit modules
#include <linux/fs.h>       //for file_operations
#include <linux/debugfs.h>  //for create debugfile
#include <linux/cdev.h>     //for create devfile
#include <linux/vmalloc.h>  //for vmalloc
#include <linux/mm.h>       //for mmap related stuff
#include <linux/wait.h>     //for wait_queue_head_t
#include <linux/sched.h>    //for wait_event_interruptible
#include <linux/spinlock.h> //for spin_lock
#include <linux/memlink.h>  //for recieve ksm , use sm_msg
#include <linux/poll.h>     //for poll_wait

#define DPRINT(...) printk(KERN_DEBUG __VA_ARGS__);
static int
mmap_fault(struct vm_area_struct *vma, struct vm_fault *vmf);
int
my_mmap(struct file *filp, struct vm_area_struct *vma);
static long
my_ioctl(struct file *filp, unsigned int cmd, unsigned long args);
void
my_mutex_lock(void);
void
my_mutex_unlock(void);
static unsigned int
my_poll(struct file *file, poll_table *wait);

DECLARE_WAIT_QUEUE_HEAD(wq);
void *page_ptr;
int devfd;
int mutexVal = 0;
static unsigned int writeSum;
unsigned int sync;
char *memmap;
char *head;
char *lock;
char *phyaddr;
size_t share_memsize;
unsigned long offset;
dev_t dev;
struct page *page;
struct cdev *devfile;
struct dentry *debugfile;
static wait_queue_head_t read_q;
struct vm_operations_struct mmap_vm_ops = {
  .fault = mmap_fault,
};

static const
struct file_operations my_fops = {
  .mmap           = my_mmap,
  .unlocked_ioctl = my_ioctl,
  .poll           = my_poll,
};

static int
mmap_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
{
  page_ptr = NULL;
  if((NULL == vma) || (NULL == head)){
```

```
    DPRINT("ERROR : VM_FAULT_SIGBUS\n");
    return VM_FAULT_SIGBUS;
  }
  offset = (unsigned long)vmf->virtual_address - vma->vm_start;
  if(offset >= share_memsize){
    DPRINT("ERROR : VM_FAULT_SIGBUS\n");
    return VM_FAULT_SIGBUS;
  }
  page_ptr = head + offset;
  page = vmalloc_to_page(page_ptr);
  get_page(page);
  vmf->page = page;
  return 0;
}

int
my_mmap(struct file *filp, struct vm_area_struct *vma)
{
  vma->vm_ops = &mmap_vm_ops;
  vma->vm_flags |= (VM_DONTEXPAND | VM_DONTDUMP);
  vma->vm_private_data = memmap;
  return 0;
}

int
trans_data(struct sm_msg *msg, size_t memsize)
{
  if(memsize  < writeSum + 1){
    writeSum = 0;
  }
  DPRINT("MESSAGE : Lock shared memory \n");
  if (__sync_val_compare_and_swap(lock, 0, 1) == 0){
  memcpy(head + writeSum, msg, sizeof(unsigned int) + msg->len);
  writeSum += sizeof(unsigned int) + msg->len;
  *lock = 0;
  }
  DPRINT("MESSAGE : UnLock shared memory \n");
  return 0;
}

char
*create_file(char *file, size_t memsize)
{
  DPRINT("MESSAGE : %s \n", file);
  share_memsize = memsize;
  debugfile = debugfs_create_file(file, 0644, NULL, NULL, &my_fops);
  if(debugfile == NULL) {
    DPRINT("ERROR : can't create debugfile \n");
  } else
    DPRINT("MESSAGE : Create debugfile -> %s\n", file);
  memmap = vmalloc(memsize);
  lock = memmap;
  head = memmap + 1;
  phyaddr = NULL;
  memset(lock, 0, memsize);
  DPRINT("MESSAGE : vmalloc -> %zd\n", memsize);
  DPRINT("MESSAGE : lock address -> %p\n", lock);
  DPRINT("MESSAGE : head address -> %p\n", head);
  return memmap;
}

void
my_mutex_lock(void)
{
  spin_lock(&wq.lock);
  wait_event_interruptible_locked(wq, mutexVal == 0);
  mutexVal = 1;
  spin_unlock(&wq.lock);
}

void
my_mutex_unlock(void)
{
  spin_lock(&wq.lock);
  mutexVal = 0;
  wake_up(&wq);
  spin_unlock(&wq.lock);
}

static unsigned int
my_poll(struct file *file, poll_table *wait)
{
```

```
  unsigned int mask = 0;

  poll_wait(file, &read_q, wait);

  if(sync != 0){
    mask |= (POLLIN | POLLRDNORM);
    DPRINT("MASSAGE : POLLIN | POLLRDNORM")
  }
  return mask;
}

static long
my_ioctl(struct file *filp, unsigned int cmd, unsigned long args)
{
  switch(cmd){
    case 1 : my_mutex_lock();
    break;
    case 2 : my_mutex_unlock();
    break;
  }
  return 0;
}

static int
__init mmapfile_module_init(void)
{
  DPRINT("-----Init memlink module-----\n");
  devfd = alloc_chrdev_region(&dev, 0, 1, "memlink_ctl");
  if(devfd < 0){
    DPRINT("ERROR : Can't create dev file\n");
    return -1;
  }
  devfile = cdev_alloc();
  devfile->owner = THIS_MODULE;
  devfile->ops = &my_fops;
  if(cdev_add(devfile, dev, 1)){
    DPRINT("ERROR : Can't Create devfile \n");
  } else {
    DPRINT("MESSAGE : Create devfile \n");
  }
  init_waitqueue_head(&read_q);
  DPRINT("MESSAGE : read q init\n");
  my_create_file = create_file;
  my_trans_data = trans_data;
  return 0;
}

static void
__exit mmapfile_module_exit(void)
{
  DPRINT("-----Exit memlink modul------\n");
  my_trans_data = NULL;
  my_create_file = NULL;
  debugfs_remove(debugfile);
  cdev_del(devfile);
  unregister_chrdev_region(dev, 1);
  vfree(memmap);
  DPRINT("MESSAGE : remove, free \n");
}
module_init(mmapfile_module_init);
module_exit(mmapfile_module_exit);
MODULE_LICENSE("GPL");
```

プロセスのヘッダーファイル

```
char *filename;

char *init_memlink(char *filename, size_t memsize);
char *spin_memlock(void);
int spin_memunlock(void);
void wait_receive(void);
int close_file(int debugfd, char *head, size_t memsize);
```

プロセスのライブラリ

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include "memlink.h"

int devfd;
int debugfd;
char *head = NULL;
char *lock;

char *init_memlink(char *filename, size_t memsize)
{
  char *debugpath;
  debugpath = (char *)malloc(sizeof(char) * 256);
  sprintf(debugpath, "/sys/kernel/debug/%s", filename);
  debugfd = open(debugpath, O_RDWR);
  if(debugfd < 0){
    perror("ERROR : can't open debugfile");
    return NULL;
  }
    devfd = open("/dev/memlink_ctl", O_RDWR);
  if(devfd < 0){
    perror("ERROR : can't open devfile");
    return NULL;
  }
  head = mmap(NULL, memsize, PROT_READ|PROT_WRITE, MAP_SHARED, debugfd, 0);
  if (head == MAP_FAILED){
    perror("ERROR : can't mmap");
    return NULL;
  }
  lock = head;
  head++;
  return head;
}

char *spin_memlock(void){
  while(1){
    if(__sync_val_compare_and_swap(lock, 0, 2) == 0)
      break;
  }
  return head + 1;
}

int spin_memunlock(void){
  *lock = 0;
}

void wait_receive(void){
  fd_set fds;
  FD_ZERO(&fds);
  FD_SET(devfd, &fds);
  if ( select( devfd + 1, &fds, NULL, NULL, NULL ) < 0 ) {
    perror( "ERROR : can't select" );
  }
}

int close_file(int debugfd, char *head, size_t memsize)
{
  if(close(debugfd) < 0){
     perror("ERROR : can't debugfile close");
    return -1;
  }
  if(munmap(head, memsize) < 0){
    perror("ERROR : can't munmap");
    return -1;
  }
}
```