Linux 版の MBCF 通信機構について

松本　尚 [1]

概要：筆者のオリジナル並列分散 OS SSS-PC の基幹となる通信同期機構 MBCF の Linux OS 版である MBCF/Linux の開発を行った．高スループットと低レイテンシを両立し，高機能な共有メモリ操作を実現し，ロックフリーの不可分共有メモリ操作が可能な MBCF 通信機構を Linux 上で動かすことに成功した．本稿では，MBCF 自体を振り返り，MBCF の 64bit アドレス拡張，Linux 版の開発基本方針と OS の違いから派生する実装上の注意点について述べる．

キーワード：MBCF，MBCF/Linux，分散共有メモリ，通信同期機構，SSS-PC

Memory-Based Communication Facility in the Linux kernel

TAKASHI MATSUMOTO†1

Abstract: We have developed MBCF/Linux, which is the Linux OS version of MBCF, which is the core communication synchronization mechanism of our original parallel distributed OS SSS-PC. It achieves both high throughput and low latency and realizes high-performance shared memory operations, and also achieves lock-free atomic shared memory operations on Linux. In this paper, we look back on MBCF itself, and describe the 64-bit address extension of MBCF, the basic development policy of the Linux version, and the precautions for implementation derived from the differences in two OSs.

Keywords: MBCF，MBCF/Linux，Distributed Shared Memory，communication and synchronization mechanism，SSS-PC

1. はじめに

筆者のオリジナル並列分散 OS である SSS-PC[2] および SSS-CORE[1] の基盤となる通信同期機構として採用していた Memory Based Communication Facility (**MBCF**)[3][4]の Linux OS 版を開発するとともに，MBCF を 64bit アドレスに対応させた MBCF64 を新たに規定して，64bit アドレスの Linux OS にも対応可能にした．本稿では，MBCF 自体がどんなものであったか振り返っていただき，64bit アドレス対応のための拡張方式を解説し，Linux 版の開発基本方針と実装時の注意点について述べる．

2. MBCF とは

2.1 MBCF の概要

MBCF は Gigabit Ethernet のような高速ネットワークで接続された PC クラスタやワークステーションクラスタにおいてマシン境界を越えた共有メモリアクセス環境を実現するための通信同期機構として 1994 年に考案され，1996 年に発表された[1]ものである． MBCF を実装するに当たっては，特殊なハードウェアを必要としない．通信のための API(Application Program Interface)が共有メモリ操作となっている点が，socket に代表されるこれまでの通信 API とは大幅に異なる点である．OS が提供する通信機構自体が共有メモリ操作を基本とする API を直接的に提供することにより，高機能，低遅延，低オーバヘッド，ロックフリーの通信同期機構の実現が可能になる．

1 奈良女子大学
Nara Women's University

2.2 MBCF 研究開発の経緯

1994 年，ワークステーションクラスタ上において並列処理アプリケーションを効率良く実行可能な並列分散オペレーティングシステム SSS-CORE 開発中の筆者によって MBCF は考案された．当時の OS は，通信システムコールの時間的オーバヘッドコストが大きく，OS を介した通信では通信ハードウェアの性能を引き出すことができないでいた．そういう状況であったため，共有メモリ型の並列計算機では，OS を介して通信同期を行うのではなく，共有メモリ上でデータを受け渡し，細粒度で同期が必要な場合には共有メモリ上の同期フラグをスピンウェイトすることで同期を実現していた．しかし，ワークステーションクラスタ上にはマシン境界を越えた物理的な共有メモリは存在しない．そこで，ワークステーションクラスタ環境においても共有メモリを効率良く実現する方法についての研究を行った．まず，OS を介した通信には大きなオーバヘッドコストが掛かるというその当時の状況が改善不可能なのか，という疑問を持った．システムコールを利用する第一義的な目的は，特権モードに移行してユーザの要求する機能を実現する OS コードを呼び出して実行することである．特権モードへの移行は CPU 内部の 1bit の状態遷移に過ぎず，同時処理する（インフライト中の）命令数が少なかった 1994 年当時の CPU では, 1 クロックで十分に切り替えることが可能であった．また，Ethernet MAC に送信用のデータ領域を設定して，DMA による送信動作を起動することに大きなオーバヘッドコストがかかるとは考えられない．この送信処理自体に大きなオーバヘッドコストが内在するの

ⓒ 2021 Information Processing Society of Japan

であれば，1994 年当時市販されていた専用通信ハードウェアを持った分散メモリ型の並列計算機が低オーバヘッドで通信できることに矛盾を生じる．なぜなら，通信ハードウェアにパケットを送信させる制御手順は，専用通信ハードウェアも汎用の Ethernet MAC にも大差がないからである．これらのことから，送信時のシステムコールにおいて，大きなオーバヘッドコストが発生するとすれば，それは通信処理の本質とは無関係なものであるに違いないと思われた．よって，高速通信専用のシステムコールを新設して，余分なコピーや余分なことは一切行わないことにすれば，OS を介しても低オーバヘッドで送信ができると考えた．受信側に関しては，通常の Ethernet MAC であれば，受信したパケットを DMA でメモリ上に受け取り，パケットを受信したことを CPU に割込みで報告する．従来の OS では，受信割込みルーチンにおいて，通信プロトコルスタック（IP パケットなら IP プロトコルスタック）を起動する準備のみを行って，割込みから復帰する．この後，階層構造の通信プロトコルスタックの受信関連処理がスケジューリングされていき，パケットの通信対象となるプログラムにパケット内のデータを引き継ぐ準備がなされる．最終的に，通信対象のプログラムがスケジューリングされて受信用システムコール（例えば，recv や read）が実行された時に，カーネル空間からユーザプロセス空間にパケットで運ばれたデータがコピーされる．この受信側の一連の処理は，オーバヘッドコストがかなり大きくなると思われる．そこで，通信パケット内に通信システムが理解できる形で，通信相手プロセス内のデータが配送されるべき最終目的アドレスが通信ヘッダに書かれてあれば，受信割込みルーチンにおいて，ユーザプロセス空間内への書き込みまで終わらせることができると考えた．通信相手プロセスの識別子も当然ヘッダ内に含まれているものとする．Ethernet MAC チップがパケット受信によって起動するハードウェア割込みルーチンは，当然特権モードで実行されるため，通信先プロセスのメモリアドレス空間への切替えを行うことが可能である．もちろん，割込みルーチン終了前には元のアドレス空間に戻す．

このような思考実験の下，MBCF_WRITE の原型となるマシン境界を越えた他プロセス内のメモリ領域へのデータ書き込みを実装してみた．最初の実装は，10BASE-T Ethernet によるものだった[1]ため，データ転送能力は高くなかったが，送信ルーチンと受信ルーチンのオーバヘッドコストを十分に小さく抑えられることが実証できた．送信時も受信時もソフトウェアのオーバヘッドコストは当時の CPU 性能において，数マイクロ秒のオーダーに抑えることが可能となった（当時のシステムコール経由の送受信オーバヘッドコストは約 1 ミリ秒）．1997 年に Ethernet として 100BASE-TX を使用して MBCF を実装した[5]ところ，データ転送能力と遅延の両方において市販の並列計算機の専用通信機構[6][7][8]と勝るとも劣らない性能を達成することができた．

2.3 中粒度の共有メモリアクセス機構

MBCF を考案した当時，ハードウェアの支援機構を持った分散共有メモリ型並列計算機が研究開発され[9][10]，一部市販されていた．これらは，CPU のメモリアクセス（load/store 命令）を必要であればそのまま遠隔メモリの読み書きとしてハードウェアによって自動的に実現してくれる並列計算機である．これらのデータ転送単位は CPU のキャッシュブロックサイズに設定されていた．CPU の load/store 命令は 1byte から 8byte 程度の大きさのデータを扱うが，CPU のキャッシュはキャッシュブロック単位（32byte や 64byte）でデータ転送を行う．キャッシュを使わないと CPU 性能が大幅にダウンするため，分散共有メモリマシンのデータ転送は load/store 命令を契機として，キャッシュブロック単位でデータが転送されていた．これに対して，MBCF は既存のシステムコールベースの通信機構よりは大幅にオーバヘッドが削減されていたが，それでも数百から数千命令規模のソフトウェアによるオーバヘッドコストが存在している．このため，キャッシュブロック単位のデータ転送では，ソフトウェアオーバヘッドが性能を大きく阻害してしまう．このため，MBCF では，大きな粒度でデータ転送が可能な場合には，なるべくデータを一回にまとめて転送を行う必要がある．ただし，Ethernet を使用する場合であれば，ジャンボフレームを考慮しなければ，1 パケットの最大データ量は 1500byte であり，それ以上には大きくできない．このことから，MBCF は中粒度の共有メモリアクセスを実現するための機構と考えることができる．また，一回の操作におけるデータ転送サイズの上限が抑えられているため，割込みルーチン内でデータ転送のためのユーザプロセス空間へのメモリコピーを行ったとしても，割込み実行時間が長時間に亘って持続してシステム全体を困らせるような事態にはならないと考えられる．

2.4 MBCF の共有メモリアドレス空間

共有メモリ操作の要求元タスク（UNIX/Linux におけるプロセスと同義）から見ると，**（論理タスク ID, 論理アドレス）**の組が要求先のメモリ領域を規定する．論理タスク ID は要求元タスク内でプライベートに規定された識別子であり，その実体がどのノード（マシン）のどの物理タスクであるかは OS が要求元タスクごとに管理する．論理タスク ID と（物理ノード ID, 物理タスク ID）を対応させる管理表のことを**タスク表**（または **task table**）と呼ぶ．物理ノード ID は IP アドレスやマシン名に対応し，物理タスク ID は SSS-CORE / SSS-PC におけるタスク（UNIX/Linux のプロセス）の識別子に対応する．ただし，SSS-CORE / SSS-PC におけるタスク識別子はユーザが明示的に特定の値を指定することが可能であり，OS が値を自動的に割り当てる UNIX/Linux におけるプロセス ID とは異なる側面がある．この辺りの事情やこの違いが使い勝手に及ぼす影響につい

ⓒ 2021 Information Processing Society of Japan

ては，5.1 節で改めて議論する．要求元タスクが物理タスク ID を直接使用しないことにより，タスク表の書き換えにより,要求先タスクの実体を変更することが可能になる．SSS-PC OS ではこの機能を使って，タスクが別のノードにマイグレーション（移送）されても，継続して通信が行える機能を実現している．マイグレーション機能を持たない場合においても，プログラム内に論理タスク ID を埋め込むスタイルのコーディングを行っていても，OS レベルのタスク表の書き換えによって，実際に通信して操作する対象を別のノード（マシン）で動かすことが可能になる．極端なケースでは，要求元と要求先を同一ノード内に割り当てることや，両者を同一タスクとすることも可能である．同一ノードに割り当てられている場合は，同一マシン内で MBCF の処理が閉じるため，実際のマシン間の通信は発生しない．通信が発生しない分だけ，同一ノード内の MBCF 操作要求は遅延が小さくなる．

要求先タスク内の操作対象メモリ領域のアドレスは，要求先タスクのメモリ空間内の論理アドレスで指定される．構造体やメモリ領域の論理アドレスは，C 言語等であればユーザレベルで容易に獲得することが可能であり，遠隔メモリ操作を行いたい相手方のタスクに自分のアドレス情報を伝えることにより，MBCF を使った共有メモリ操作要求を行わせることが可能になる．

（論理タスク ID, 論理アドレス）の組は以下の本稿において**(Ltask, Laddr)**とも記述される．

2.5 MBCF 操作コマンドのバリエーション

MBCF による共有メモリ操作要求では要求先タスクの操作対象となる論理アドレスが指定されている．中粒度のメモリ操作オペレーション，例えば，共有メモリ書き込み（MBCF_WRITE）であれば，(Ltask, Laddr)の組と実際に書き込まれるデータサイズ(nbyte)とデータ列がメモリ操作要求パケット内に格納される．要求元の API としては, (Ltask, Laddr)とデータサイズ n と書き込むべきデータが格納されている領域のポインタ（SrcAddr）を指定することになる．共有メモリ操作であるため，書き込みのみではなく，データを読み出したい側（つまり要求側）しか必要なデータの場所が判らない場合には，読み出し操作があると便利である．読み出し操作が存在しない場合には，読み出して欲しいデータのアドレス等と内容を送りつけて欲しいメモリ領域の先頭アドレスを読み出したい側から書き込んで，その書き込みに返答する形でデータを持っているタスクに遠隔書き込みを行ってもらう必要がある．これでは，要求先でも要求に対応するためのユーザレベルのプログラムコードの実行が必要になり，コード生成にも手間がかかり，実行時間のオーバヘッドコストも大きい．読み出しコマンドが実現されて，要求先ノードにおける受信割込みルーチン内でユーザ空間の読み出しが完了すれば，要求先でユーザレベルのプログラムコードが実行される必要はない．この読み出しコマンドの実現には，要求先ノードの受信割込みルーチン内に，要求先タスクのメモリ空間からデータを読み込んだパケットを返送する機能を作り込む必要がある．ただし，メモリ操作要求パケットが失敗する可能性がある場合，その成否を要求元に伝えるためには，やはり返送パケットによる送信機能が割り込みルーチン内に必要となる．このような理由から，MBCF の受信割込みルーチンにおいては，要求元に返送パケットを送出する能力を実装している．この返送パケット送出能力と操作対象メモリの直接指定能力により，MBCF においては多数の操作コマンドバリエーションを作り出すことが可能となっている．なお，遠隔読み出しである MBCF_READ コマンドの場合は, 要求元は読み出し要求と読み出したデータを受け取る要求元側のバッファアドレスを指定して要求先に送りつけるだけであり，実際のデータ移動は要求先からの返送パケットにより要求元に送られることになる. 通常の send-recv 型の通信では，要求元が常にデータの送信側であり，MBCF ではこの辺りの概念も変わってくる．操作コマンドバリエーションの詳細については記述が長くなるので別稿に譲る．

2.6 MBCF のメモリコンシステンシモデルと到着保障順序保障

MBCF は分散メモリ環境に共有メモリアクセス手段を提供するための機構である．そして，MBCF は直接メモリの実体に対して操作することのみを考えており，メモリの内容を遠隔ノードにキャッシュする仕組みは MBCF の枠組みの中では考慮していない．マシン境界を越えたキャッシュを想定しないため，メモリコンシステンシモデルは単純である．このため，同一要求元からのパケットの操作順序さえ保障されれば，先行するメモリ操作要求の完了を待つことなく，次の要求を実行することができる．通常の分散共有メモリ機構では操作完了を待つべき不可分操作に関しても MBCF の作法に則る限り操作完了を待つ必要はない．この点に関しては，2.7 節において詳述する．先にも触れた操作順序保障の必要性は以下のことに由来する．任意の二つのノードが要求元と要求先になった場合に，要求先の先行する要求が後続の要求に追い越されることがあっては，共有メモリモデルとしては甚だ使い勝手が悪い．つまり，A フラグをセットしてから B フラグをセットしたはずなのに，B フラグが先にセットされたと他タスクから観測されることがあるようでは誤動作を招きかねない．また，85 であった変数 X に 300 を書いたはずなのに, 自分で変数 X を読み出してみたら, 他の誰も変数 X に上書きしていないのに，書き込み前の 85 が読み出されたのでは，通常のメモリとしての使用は不可能である．この結果，MBCF 機構では 2 ノード間の MBCF に通信（操作要求と返信）の到着保障と順序保障を行う必要がある．順序保障に関しては，メモリ操作の順序まで含まれるので，要求先ノードの受信割込みルーチン内において，同一の要求先タスクに対する

MBCF 操作要求は到着順に処理する必要がある．なお，厳密に言えば，到着保障と順序保障は制約が厳しすぎる．計算機はメモリ順序モデルとしてプロセッサコンシステンシを保障すれば十分であるため，書き込み同士間と読み出し同士間で順序が入れ替わらず，異なる領域に対する書き込みと読み出しに関しては順序を保障する必要はない．ただし，CPU 内部のライトバッファであれば，領域の一致や重なりを検出してライトバッファを優先してフラッシュする必要性を判定することが容易に可能であるが，MBCF においては,異なる要求間での領域の重なりの判定が難しい（オーバヘッドコストを増大させる可能性がある）ため，より強い制約の到着保障と順序保障を行うことにより，ストロングコンシステンシを実現する．

2.7 ロックフリーの MBCF 不可分操作

通常はロック操作が必要な Read-Modify-Write 型の不可分操作に対して，MBCF は筆者が Memory Based Processor (MBP)[11][12]で 1992 年に提唱した「メモリベース不可分操作」が適用できる．MBP は主記憶の中に設置されるプロセッサであり，分散メモリ環境では分散されたメモリ領域ごとに MBP が存在している．物理的な距離が近く遠隔アクセスが不要であるため, MBP は自分が存在しているメモリ領域内のデータに対する Read-Modify-Write 処理を低レイテンシで実現することが可能である．例えば，あるメモリデータ X の内容が現在 1000 だとして，遠隔ノードのタスク 1 が X に 200 加算して，別ノードのタスク 2 が X に 300 加算するというケースを考える．通常の分散共有メモリ型並列計算機では X に対するロックを獲得してから，X を読み出してそれぞれの値を加算した内容を X に書き戻して，ロックを解除するという手順を踏むことになる．MBP が存在すれば，X の存在する領域を担当する MBP に対して，タスク 1 は X に 200 を加算する要求パケット（コマンド）を発行し，タスク 2 は X に 300 を加算する要求パケットを発行する．タスク 1 とタスク 2 の間には排他制御も同期も必要とされない. MBP は要求パケットを受け取った順に一つずつ排他的に処理する. MBP のレベルでローカルに排他制御は行われているが，MBP が変数 X にアクセスするコストにはノード間通信が含まれないため，このローカルな排他制御のコストは極めて低い．排他制御が必要な領域にローカルな演算装置を用意して，データとコマンドを投げつけることで，ノード間の排他制御を不要にする方式をメモリベース不可分操作と筆者らは呼んでいる．

MBCF では, MBP と同様にメモリベース不可分操作を実現できる．前出の例であれば，操作対象アドレスのデータに加算を行う MBCF_ADD コマンドを設けておいて，タスク 1 とタスク 2 は MBCF_ADD 要求パケットを要求先タスクの X のアドレスを操作対象にして発行すればよい．もちろん，タスク 1 とタスク 2 の間には同期も排他制御も不要である．任意のアドレスにユーザが用意した fifo キューにパケット単位のデータを登録する MBCF_FIFO コマンドは，まさにメモリベース不可分操作の典型例の一つである．操作対象アドレスには fifo キューを構成するための複数のポインタからなる構造体（FIFO 構造体）が存在し，パケット内に搭載して運んできたデータを fifo キューのバッファ領域に格納後，ポインタつまり FIFO 構造体を更新する．この処理を分散共有メモリ型の並列計算機で実行させる場合は，「まず FIFO 構造体のロックを取って，FIFO 構造体への排他的アクセス権を確保して, FIFO 構造体内のポインタを読み出して，データを FIFO キューのバッファ領域に格納してから，FIFO 構造体のポインタを更新して，FIFO 構造体へのロックを解除する」という手順になる．

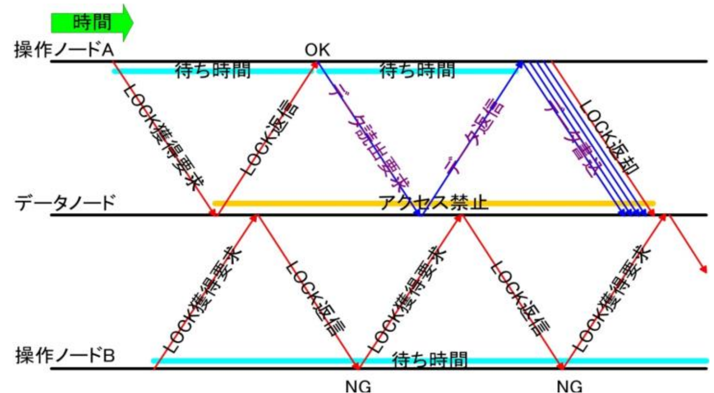

図 1 分散共有メモリ上の不可分操作（fifo キューへのデータ格納）

この分散共有メモリ上の不可分操作である FIFO キューへのデータ格納操作を図 1 に示す. 図の左から右へは時間経過を示し，データノードに FIFO 構造体が存在し，操作ノード A 上のタスクと操作ノード B 上のタスクから FIFO 構造体で規定される fifo キューにデータを登録しようとしている. ノード A とノード B のタスクはともに最初に FIFO 構造体のためのロックを獲得しに行き，図 1 ではノード A のタスクが先にロックを獲得している．ノード A のタスクはデータノード上で一度も拒絶されることなくロックを獲得しているが，ノード A とデータノード間の往復の通信遅延時間が待ち時間のオーバヘッドコストになっている．ロックの獲得に成功したので FIFO 構造体を読み出しに行くが，読み出し結果がノード A に戻るまでにまた往復の通信遅延時間がかかる．そこで得たポインタ情報を基に，格納すべきデータを書き込んで, FIFO 構造体も新たな内容で更新する．この後に，ロックを解除して他のタスクから FIFO 構造体へのアクセスが可能にする．この時点までは，ノード B のタスクからロック要求がいくらデータノードの FIFO 構造体に届いても，拒絶されて認められない．このように, ノード A のタスクにもノード B のタスクにも通信遅延に基づく大きなオーバヘッドコストを発生させてしまう.

これに対して，図 2 にまったく同じ処理をメモリベース不可分操作である MBCF_FIFO を使って実現した場合についての時間経過を示す．MBCF_FIFO コマンドを発行する前にノード A のタスクもノード B のタスクも排他制御も同

期も不要である．共に fifo キューに格納したいデータを載せた MBCF_FIFO 操作要求パケットをデータノードの要求先タスクの FIFO 構造体を宛先にして発行するだけである．通信遅延に基づく待ち時間はどのノードにも発生しない．データノードにおいてパケットの到着順に不可分に FIFO 構造体に対して処理がなされるが，この不可分操作はノードローカルな操作であるためほとんどオーバヘッドコストを発生させない．

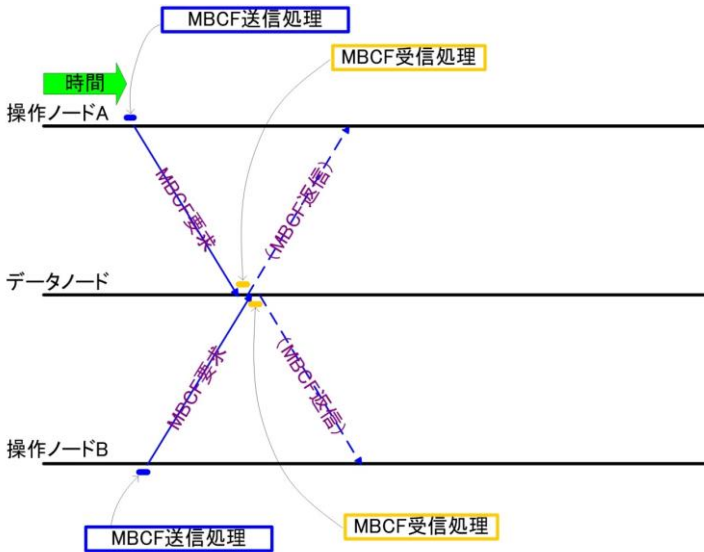

図 2 MBCF_FIFO による fifo キューへのデータ格納

なお，データノード上のタスク（たとえ要求先タスク自身であっても）が MBCF_FIFO 操作要求をデータノード内の要求先タスクに対して発行したい場合は，MBCF 要求発行システムコールを必ず使用する必要がある．MBCF 要求発行システムコール内で要求元と同じノードへの MBCF 受信処理は行われることになるが，このときに受信割込みルーチン内の MBCF 受信ルーチンとアドレス空間ごとに排他制御を行う．要求元が要求先と同一タスクであれば，プロセッサによるユーザレベルの LOAD/STORE 命令によって直接 fifo キュー操作を実現できてしまうが，これを行ってしまうと，要求先タスクにおける順序性や排他性が保たれなくなる．

このように，MBCF を利用した分散共有メモリ操作では，ロックを使った排他制御を行うことなしに分散共有メモリ操作が行える可能性が高い．ロックが必要になるのは MBCF パケット一つでは搭載不可能な大量のパラメータやデータが必要な不可分処理だけである．ロック操作が不要になる大規模分散共有メモリの実現手法である MBCF は夢の分散共有メモリ実現方法と呼べるかもしれない．

2.8 MBCF のタスク間同期方法

2.2 節で述べたように MBCF は 1994 年の考案当初はメモリ上の同期フラグによるスピンウェイト（ビジーウェイト）を分散メモリ環境でも効率良く行うための通信方法がないかという視点から考え出された．つまり，プロセッサのスケジューリングとしては並列実行するタスクが同時にプロセッサに割り付けられていることを前提にしていた．しかし，タスク数がプロセッサ数よりも多い状況や，並列処理だけではなく分散処理的な要素があり低速の I/O 待ちがあるような状況においては，スピンウェイトによる同期はプロセッサ資源の無駄な浪費を招く可能性がある．もう少し具体的に述べると，同期条件を完了させるためにまだプロセッサに割り当てられるべきタスク A がプロセッサの実行権が割り当てられるのを待っているにも関わらず，タスク B がタスク A からの同期完了のフラグ書き込みを待ってスピンウェイトを行う状況は単なるプロセッサの無駄使いに過ぎない．タスク B がタスク A に切り替え可能であれば即刻切り替えるべきであるし，走行するノードが異なる等の事情で切り替えが困難な場合は，タスク A が次にプロセッサが割り当てられるまでずっとタスク B が無駄にスピンウェイトのループを回っているよりも，タスク B と同じノードの他の実行待ちタスクに実行権を譲った方がプロセッサを有効に使える見込みが高い．このような状況に対応するため，MBCF にはタスク間同期を支援するためのオプションや MBCF コマンドが提供されている．同期オプションは各種 MBCF コマンドに付加的に適用することが可能になっている．以下に現在の MBCF システムにおいて同期のために提供されているタスク間同期オプションと MBCF コマンドの代表的なものを二つ挙げる．

- **sleep 状態タスクの wakeup**
  無駄なスピンウェイトを止めて sleep 状態に移行した要求先タスクを起こすためのオプション．このオプションがついた MBCF コマンドが到着した時に要求先タスクが sleep 状態ではない場合はこのオプションは無効である．MBCF のタスク間同期の最も一般的な方法．
- **ユーザ関数の非同期呼び出し**
  MBCF_SIGNAL に代表される非同期ユーザ関数の起動を伴う MBCF 操作要求コマンド．非同期ユーザ関数を起動すること自体がタスク間で同期を行う行為の一つであるが，起動されたユーザ関数内でタスク間同期を行うためのシステムコール等を実行することにより，要求先タスクの状態を変化させるタスク間同期が可能である．MBCF の非同期ユーザ関数呼び出しの特徴として，操作対象アドレス，操作コマンド，送信元論理タスク ID が起動される関数の引数として渡すことができる．MBCF_SIGNAL はそれに加えて，要求元タスクからのパラメータを要求パケットに搭載可能な範囲で引き渡すことが可能である．

2.9 MBCF が高速実装可能な理由

要求元タスク側でシステムコールを介して MBCF 操作要求パケットの送信を行っているが，この方式が低オーバヘッドで実現できたのは，パケット送信処理に不要な処理をシステムコールから完全に取り除き，コードの実装に階

ⓒ 2021 Information Processing Society of Japan

層構造といったオーバヘッドコストを持ち込むようなものは極力排除してコード最適化を実施した結果である．

同様に，要求先タスク側つまりパケットの受信側ノードにおいて，MBCF ではユーザプログラムが関与しなくても処理が完結する形式を採っている．つまり，OS カーネル内の受信処理で基本的な MBCF 操作要求は要求先タスクのメモリ空間に対して実行される．要求先タスクがスケジューリングされていない状況においても，メモリ空間を切り替えて操作対象アドレスのメモリに操作を加える必要がある．このアドレス空間切替えや特権モードからのユーザ権限アクセスのコストを，1994 年頃の CPU から実装された機能により，大幅に低く抑えることが可能になった．

これらのことから，MBCF 通信機構は大きな自由度を持ち，ロックフリーという特性を持ちながら，低オーバヘッドコストで実装が可能である．

3. MBCF の 64bit アドレス拡張

SSS-PC/SSS-CORE 用の通信同期機構として MBCF を開発していた当時（1990 年代から 2000 年代前半）には，32bit アドレスの CPU が主流であり，64bit CPU を搭載したマシンであっても，4GByte を越えるメモリを搭載したマシンはほとんど存在しなかった．このため，MBCF が取り扱うタスク内の論理アドレスを 32bit 長に設定していた．しかし，2021 年現在では，PC の多くに 64bit CPU と 64bit OS が搭載され，実メモリも 4GByte を越えて実装されることが普通になっている．MBCF の大きな魅力の一つは，新たなマシンをネットワークに追加するだけで，アプリケーションで使用する分散共有メモリ領域が拡大できることにあるにも関わらず，32bit 論理アドレスでは同一マシン内のメモリすら一つのアプリケーションからは使い切ることができない．このことから，MBCF の 64bit 論理アドレス対応は不可避だと考えて，MBCF のパケットフォーマットを見直して，64bit アドレス対応を行った．以下，64bit アドレスの MBCF を MBCF64，従来の 32bit アドレスの MBCF を MBCF32 と呼称し，双方に共通な議論や特にアドレスサイズを区別する必要のない場合は，単に MBCF と呼称する．

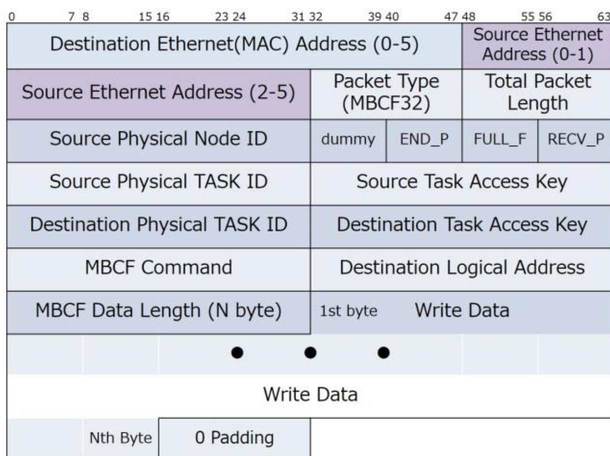

図 3 MBCF32 の MBCF_WRITE パケットフォーマット

図 3 に従来の MBCF32 において遠隔書き込みである MBCF_WRITE コマンドのパケットフォーマットを示す．MBCF は Ethernet 上での実装が行われてきており，Ethernet ヘッダが 14byte という限の悪い大きさでも alignment が適切に行われるようにパケットフォーマットが設計されている．このため，図には Ethernet ヘッダ部分も含んでいる．Ethernet ヘッダの Packet Type には MBCF32 を表わすプロトコル番号が入っている．Total Packet Length は Ethernet パケット全体のサイズを格納し，パケットが欠損していないかチェックするためのデータを供給する．Destination Physical Node ID はこのパケットが該当ノードの MAC アドレスによって操作対象ノードまで到達しているために，省略可能であると考え，パケット内部には格納されていない．
END_P，FULL_F，RECV_P の各 1byte のフィールドが MBCF32 の到着保障順序保障プロトコルと通信の効率を上げるためのバッファリング機能を制御している．END_P はそのパケットを送信元ノードが送信先ノードに送ったパケット番号を示している．8bit しかフィールドがないため，厳密には送信番号を 256 で割った剰余の値である．RECV_P は送信元ノードが送信先ノードから正しく受け取った最後のパケット番号を示している．END_P と同じく RECV_P も受信パケット番号を 256 で割った剰余の値である．この説明から判るように，MBCF は二つのノード間でバッファリングと到着保障および順序保障を行っている．しかも，受信側ではユーザ空間のメモリを直接アクセスするため，バッファリングはパケット再送に備えるために送信側においてだけノード単位に行われている．このことは，通信チャネルごとに送受信の両側においてバッファを必要とする TCP/IP よりも通信用バッファの必要量という点において圧倒的に優れている．FULL_F は送信元ノードへの確認応答（ACK: Acknowledge）の必要性を示す情報で，ACK 返答の緊急度に応じて 5 レベルが定義されている．

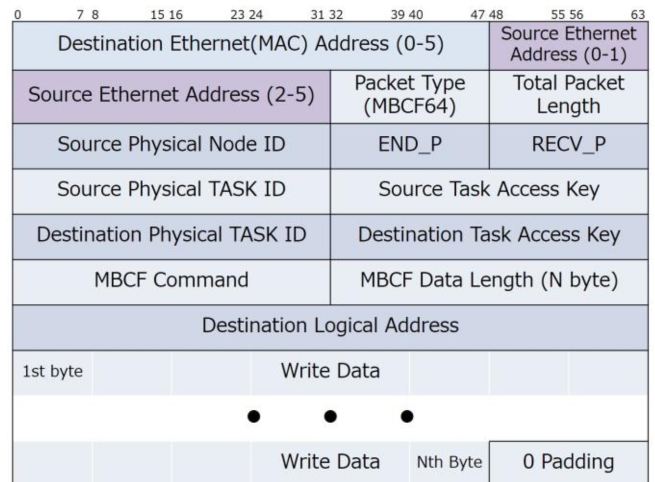

図 4 MBCF64 の MBCF_WRITE パケットフォーマット

MBCF32 のフォーマットは MBCF Data Length のフィールドまではすべての MBCF コマンドで共通であり，それ以降の付加情報のサイズは MBCF コマンドの種類のよって可変である．

MBCF_WRITE は対象領域に書き込むデータ以外の追加情報は不要であるため，MBCF Data Length のすぐ後に，データ列が続く形式となる．

図 4 に MBCF64 に MBCF_WRITE コマンドのパケットフォーマットを示す. MBCF32 のフォーマットと同様に Ethernet ヘッダ部をフォーマット内に含んでいる．MBCF64 のパケットフォーマット設計時に，プログラム書き換えの手間を考慮して，MBCF32 のフォーマットと極力同じ形式にするように注意した．Ethernet ヘッダの Packet Type フィールドは当然新たな MBCF64 用のプロトコル番号を割り当てた．64bit の論理アドレスを格納する Destination Logical Address は 8byte 境界であることが望ましいため, MBCF Data Length と Destination Logical Address のパケット内の順序を逆にした．あと，END_P と RECV_P のフィールドサイズを 8bit から 16bit に拡大した．8bit のパケット番号では，再送用のパケットが最大 256 個（実質的には半分の 128 個）しか保存しておいても区別がつかない．ということは，確認応答を待つことなく送信できるパケット数が 128 個未満であることを意味する．MBCF32 は 10BASE-T 時代に開発が開始され，Gigabit Ethernet 程度までは，この制約があってもハードウェアの性能限界までスループットを引き出すことが可能であった．しかし，SSS-PC の MBCF32 では使用したことがない 10GbE や 100GbE も Linux 版の MBCF64 の対象となるため，確認応答なしで発行可能なパケット数を増加させるために，フィールドサイズを 8bit から 16bit に拡大した．FULL_F のフィールドが MBCF64 のフォーマットでは図中に見え無くなっているが，この情報を 5 レベルから 4 レベルに集約して，MBCF Command フィールド内のカーネルが使用する部分の 2bit の領域に移設した．この変更によって，MBCF_WRITE コマンドのパケットは MBCF32 形式に比べてアドレスサイズの増加分の 4byte 増のみで形成可能となった．

図 5 に MBCF64 の MBCF_WRITE_F コマンドのパケットフォーマットを示す．MBCF_WRITE_F は MBCF のコマンドバリエーションの一つであり，対象論理アドレスから指定されたサイズのデータを書き込んだ後で，指定されたアドレスにあるフラグに指定された値を書き込むコマンドである．このため，MBCF_WRITE でも必要とされる操作対象論理アドレス（Destination Logical Address）とデータサイズ（MBCF Data Length）の他に，フラグの論理アドレス（Destination Logical Address2/Data1）と設定するフラグの値（Destination Logical Address3/Data2）が必要になる．追加のフィールドは論理アドレスでもデータでも保持できるように 8byte ずつのサイズが確保されている．この MBCF 要求パケットでは 4byte のフラグ値が使用されるため，4byte 分は未使用領域となる. この追加部分はコマンドの種類によって適宜割り当てられる．

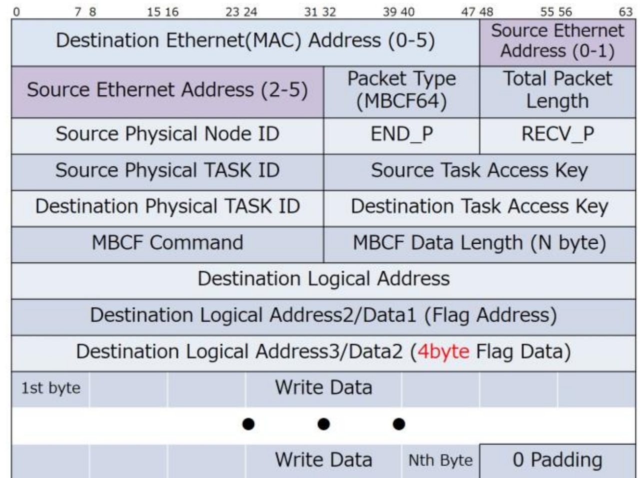

図 5 MBCF64 の MBCF_WRITE_F パケットフォーマット

4. MBCF/Linux の開発基本方針

Linux 上に実装した MBCF 通信同期機構を SSS-PC 上のオリジナル MBCF と区別するために MBCF/Linux と呼ぶことにする．MBCF/Linux の開発に当たって，以下の方針を掲げることにした．

1). API は極力オリジナルの MBCF と共通にする．
2). Linux カーネルの改変は極力回避する．
3). MBCF プロトコルスタックは Loadable Module によるデバイスドライバとして実装する．
4). MBCF プロトコルスタックをキャラクタデバイスとして実装し，MBCF のシステムコールは ioctl で実装する．
5). 二次記憶にスワップアウトされたメモリは操作対象にしない．
6). sk_buff 構造体によるパケット送受信を行う．

1).の項目は SSS-PC において開発された MBCF 使用技術をそのまま適用可能にするためである．2).の項目は MBCF を可能な限り多くの人々に使用してもらうためには，Linux カーネルへの MBCF 専用の変更はない方がよいと判断した．3).の項目も 2).と同じ理由とともに，MBCF は筆者が一人で考案しコードを書いているが, コードの一部の権利は大学発ベンチャー企業が保有しているため, 筆者の一存ではオープンソース化ができないという理由もある．4).の項目は, MBCF の API 操作は明らかに open, read, write, close の UNIX 流 I/O 操作や socket 操作とは異なっているため，新規にシステムコールを追加するのでなければ，ユーザ空間から多くのパラメータを一度にカーネルに引き渡せる ioctl による実装にならざるを得ない．また，ブロックデバイスを選択する理由が皆無であるため, より一般的なキャラクタデバイスとして実装する．ただし，MBCF

のキャラクタデバイス実装には，バイト単位に I/O にアクセスするという意味はない．5).の項目に関しては, SSS-PC OS もスワップ機能を実装していないが，サーバ用途や組込み用途の OS としては, なんの不都合もなかった. MBCF は 1 台のマシンではメモリが足りないような大規模アプリケーションでこそ活かされる通信機構であるので, 個人利用を想定したスワップ機能への対応は不要と判断した．MBCF で操作するメモリに関しては，mlock によって主記憶にピンダウンしておくものとする．SSS-PC では常にメモリのピンダウンは保障されていたため，オリジナル MBCF と MBCF/Linux のユーザ API 上の最大の差となる．6).の項目は NIC（Network Interface Card）の Linux 用デバイスドライバはすべて sk_buff と呼ばれる構造体に基づいてパケットの送受信を行っている．このため，sk_buff とその処理関数に基づいた実装を行えば，Linux 用のデバイスドライバが存在するすべての NIC が MBCF で使用可能ということになる．SSS-PC では利用したい NIC のデバイスドライバを逐一開発しなくてはならなかったため, その手間が無くなるだけでも MBCF/Linux 開発の意味がある．

5. MBCF/Linux 実装上の注意点

オリジナルの MBCF が実装された SSS-PC は完全にオリジナルかつゼロから研究開発された OS であるため，UNIX や Linux とは相違点が多数存在する．MBCF/Linux の開発においては，これらの相違点に注意を払う必要がある．以下に，MBCF/Linux 実装上の主な注意点を挙げる．

5.1 プロセス ID と物理タスク ID の違い

ノード内のアドレス空間を代表する識別子として SSS-PC OS においては物理タスク ID が使われている．UNIX/Linux においては同じ用途の識別子としてプロセス ID がある．ただし，この両者は生成方法（名前決定方法）に違いがある. 物理タスク ID は OS に自動生成させることも可能であるが，ユーザやタスクは物理タスク ID を明示的に指定して新しいタスクを生成することが可能である(ただし，該当 ID が未使用であることが必要)．この命名規則には大きな意味があり，特定用途のサーバタスク（Linux のサーバプロセスに相当）に対して，特定の物理タスク ID を命名することができるということである．MBCF においては物理タスク ID が要求先タスクの識別子であるため，特定のサーバタスクに決められた物理タスク ID を付与できることが非常に望ましい．つまり，SSS-PC OS において物理タスク ID は TCP/IP における well known ポート番号と同じ機能を果たすことができる．決まったサーバタスクに対してある決まった物理タスク ID が割り当てられていれば，そのサーバタスクを要求先タスクとして使用する場合には，その決められた物理タスク ID（とサーバタスクが動く物理ノード ID）を使って要求先の指定が可能になる．しかし，Linux のプロセス ID では，OS が自動的に適当な番号を割り当てることしかできないので，この運用はできない．MBCF で通信したいサーバプロセスにどんなプロセス ID が付けられるかは OS が ID を割り振るまで確定しない．つまり，要求元タスクはサーバプロセスのプロセス ID を知るための仕組みがないと通信を開始することができない．TCP/IP ではポート番号を明示的に指定する（bind する）ことができるため，ある機能を果たすサーバプロセスの通信ポートを well known ポートとして特定できるようになっている．以上の考察から，Linux のプロセス ID を物理タスク ID の代わりに使用することはできないと結論される. 明示的に名前を指定可能な「物理タスク ID」を MBCF/Linux において，プロセス ID の別名として，設定可能にする．

5.2 Linux プロセスの MBCF タスク化

前節で述べたように，MBCF 通信を行う Linux プロセスにはプロセス ID とは別に，物理タスク ID(4byte 正整数の値)を割り当てる必要がある．物理タスク ID が規定されて初めて Linux プロセスは MBCF による操作対象タスク（プロセス）となる．他のタスクに対する MBCF 操作要求パケットも物理タスク ID が設定された後にしか発行できない．MBCF 通信においてはマシン内のプロセスを物理タスク ID によって識別するため，物理タスク ID はマシン（ノード）内で一意であることが保障される必要がある．MBCF デバイスに対する ioctl コマンドによって物理タスク ID が設定できるようにする．一度設定したら，物理タスク ID を変更することは許可しない．物理タスク ID 設定後に，そのプロセスから MBCF パケットの送信と, そのプロセスへの MBCF パケットの受信が可能になる. MBCF デバイスの明示的な close によって，それ以降の MBCF パケットの送受信はできなくなる．明示的な close を行わない場合，つまり明示的な MBCF デバイスの close なしにプロセスが終了ないしは kill された場合には，プロセスのメモリ資源の回収に先立って暗示的なデバイスの close が行われるため，この時点で MBCF 操作が不可能になる．

MBCF の操作対象領域をピンダウンしたり，初期値を代入したりする操作は，この物理タスク ID 設定前に行うことにより, 外部タスクからの MBCF 操作とプロセス自身による領域初期化の競合が回避できる．

5.3 物理ノード ID の選択

5.1 節において，MBCF 内に物理タスク ID を新設し，MBCF を使用する各プロセスと対応させると述べた．MBCF はノード（マシン）を跨いで実行される共有メモリ操作であり，要求先タスクの指定はあくまでも（物理ノード ID, 物理タスク ID）の組である．物理ノード ID に相当するものとして，Linux の世界ではマシン名や IP アドレスがすぐに連想される．MBCF/Linux は Ethernet 上の実装を想定しているため，最終的には通信先ノードの MAC アドレスが獲得される必要がある. マシン名は, DNS や/etc/hosts で解決されて IP アドレスに変換されるため, マシン名から

の変換は手間が大きく，物理ノード ID としての採用に適していない．それでは，IP アドレスを直接採用すればよいのかと言うと，それもあまりうまくない．IP アドレス自体が桁数が多く意味のない数字であり，ユーザが記憶することが困難である．また，IP アドレスを採用すると MAC アドレスとの対応は TCP/IP の ARP 表から得られることになる．ARP 表にエントリがない場合は，TCP/IP に従えば ARP パケットでデータリンク内に問い合わせることになる．ARP 表や ARP パケットの取り扱いは TCP/IP プロトコルスタックと密に関連しており，MBCF 向けに改変することは 4 章の基本方針と相容れない．そこで，MBCF デバイスのために別の MBCF 専用の ARP 表を用意することにすれば，物理ノード ID として IP アドレスを選択する必要性はなくなる．このため，SSS-PC 同様に 32bit の 1 から始まる物理ノード番号を物理ノード ID として各マシンにユーザが付与できるようにして，物理ノード ID と MAC アドレスの対応表（ノード表:MBCF の ARP 表）を各ノードに用意する．そして，Ethernet のブロードキャスト機能を利用して到着保障順序保障を行わない MBCF_ARP_REQ コマンドと，同様に到着保障順序保障を行わず ARP 要求元に MAC アドレスを返信する MBCF_ARP_REPLY コマンドを用意して，MBCF システムでノード表を管理する．なお，SSS-PC においては，OS 起動時に物理ノード ID としての物理ノード番号が自動的に割り当てられ，ノード表と ARP 関連 MBCF コマンドも実装済みである．

5.4 MBCF デバイス

4 章の基本方針で示したように，Linux において MBCF 関連システムコールはキャラクタデバイスである MBCF デバイスに対する ioctl システムコールとして記述される．64bit Linux に対しては MBCF デバイスとして 64bit 論理アドレスの MBCF64 デバイスを使用し，32bit Linux に対しては 32bit 論理アドレスの MBCF32 デバイスを使用する．これ以外の組み合わせも実現できないわけではないが，正式サポート対象外とする．

MBCF デバイスはマシンに一つのみ存在して，MBCF デバイスの属性として以下のパラメータを持つ．

(1). 「クラスタ名」: 63 文字以内の ASCII コードからなる MBCF デバイスが存在するマシンが属するクラスタを表わす名称．クラスタ名により，同一 Ethernet セグメント内に無関係のクラスタ複数組を混在させることが可能である．

(2). 「物理ノード ID」: 1 番から始まる 32bit の正の整数（物理ノード識別子）であり，所属するクラスタ内では一意である必要がある．

(3). 「使用 Ethernet デバイス」: 現行の SSS-PC 用の MBCF では一つのマシンにおいて 4 つの Ethernet デバイスまで使用することが可能になっており，故障時の使用デバイスの切り替え等にも対応している．MBCF/Linux では，当面の間は 1 つのマシンにつき 1 つの Ethernet デバイスのみを使用することとする．複数の Ethernet NIC を有する Linux マシンにおいては，MBCF 通信に使用する Ethernet NIC を指定する API を用意する．

これらのパラメータは各ノード（マシン）に固有のものであり，ユーザプログラム毎に自由に変更する性質のものではない．このため，これらのパラメータの設定には root 権限を要求することにする．また，これらのパラメータに設定された値は，root 権限ですべてのパラメータを未設定状態にリセットするか，マシンを reboot することによってすべてを未設定状態に戻すしか，変更する方法を認めないことにする．これらのパラメータがすべて設定されて初めて MBCF デバイスによる MBCF プロトコルスタックは動作を始めて，受信した MBCF パケットの処理を開始する．

5.5 プロセス構造体とネットワークデバイス構造体

Linux のプロセス構造体の情報のみでは，MBCF パケットの送受信を処理することはできない．典型的には，本稿の 2.4 節で説明したタスク表がパケット送信には必要となる．また，操作対象アドレスを指定すること無しに使用可能であり，タスク固有の fifo キューにデータグラムを登録する MBCF_WTASKQ コマンドの実装には，タスク固有の fifo キューとその fifo キューを規定するポインタ類，といった構造体やメモリ領域がタスク（プロセス）単位に必要となる．SSS-PC OS では，これらの構造体はタスク構造体（Linux のプロセス構造体:struct task_struct に相当）の中に含まれているか，もしくはタスク構造体内のポインタから辿ることができるようになっている．Linux のプロセス構造体自体を拡張することは原理的には可能であるが，このことは Linux カーネルの大幅な書き換えを招き，4 章の基本方針の 2).と 3).に矛盾する．Linux のプロセス構造体のみでは不足する情報は，MBCF/Linux のための新たなタスク構造体（struct L_taskinfo）のメンバとして定義して，このタスク構造体から Linux のプロセス構造体へのポインタを張ることにする．MBCF/Linux ドライバ内ではこの新しいタスク構造体をプロセス構造体として扱って，Linux のプロセス構造体のメンバを参照するときは，1 段ポインタを辿ることで該当メンバへのアクセスを得る．

同様に，ネットワークデバイス構造体も MBCF を実装するためには不足するメンバが存在する．このため，新たなネットワークデバイス構造体（struct s3net_device）を設けて，これらのメンバを格納し，Linux のネットワークデバイス構造体へのポインタを格納する．MBCF/Linux ドライバ内ではこの新しいネットワークデバイス構造体をネットワークデバイス構造体として扱って，Linux のネットワークデバイス構造体のメンバを参照するときは，1 段ポインタを辿ることで該当メンバへのアクセスを得る．ただし，受信割込みルーチン（厳密には受信割り込みで起動される

ⓒ 2021 Information Processing Society of Japan

タスクレット）において引数として渡される Linux のネットワークデバイス構造体へのポインタは，Ethernet デバイスの割り込みルーチン周りに手を入れない限り変更できない．このため，MBCF 受信割込みルーチンにおいて，Linux のネットワークデバイス構造体へのポインタから，どの新しいネットワークデバイス構造体（struct s3net_device）に対する受信割込みであるかを特定した後に，受信割込み処理を実行する．

5.6 受信割り込み時のメモリ操作方法

MBCF では，MBCF パケットの受信割り込み時に，操作対象タスクのメモリ空間に切り替えを行って，ユーザ権限でメモリを読み書きする．この機能を Linux において実現するために，Linux のコンテクスト切り替えと，カーネル内からユーザ空間のメモリをアクセスする方法について，カーネルソースの調査を行った．その結果，メモリ空間の切り替えに関しては CPU 依存のコードになってしまうが，どちらの機能も受信割り込み時に起動される MBCF 受信のためのタスクレット内で実現できることが判った．詳細については，記述が長くなるため，別稿に譲る．

5.7 MBCF による SPMD 実行モデル

SSS-PC 上の並列実行アプリケーションプログラムは SPMD（Single Program Multiple Data）モデルを主に採用していた．SPMD モデルでは，まったく同じ実行コードを使って複数ノードにおいて複数のタスクを実行することにより，静的に割り当てられる変数や構造体のアドレスはすべての実行タスクで同じ論理アドレスに存在することになり，アドレス情報をお互いに通知することなく，いきなり遠隔メモリ操作要求を行うことが可能になる．しかし，最近の KASLR(Kernel Address Space Layout Randomization)機能が ON の Linux カーネルでは，BSS および DATA セクション上のグローバル変数や構造体の論理アドレスが，メモリにプロセスがマップされる度に，ランダムに変更されるため，この議論が成立しない．SPMD モデルである利点を MBCF が享受するためには，Linux カーネルの KASLR 機能を OFF にする必要がある．

6. おわりに

筆者のオリジナル並列分散 OS SSS-PC の基幹となる通信同期機構 MBCF の Linux OS 版である MBCF/Linux の開発を行った．64bit の Linux OS に対応するために，MBCF 機構も 64bit 論理アドレスへの対応を行った．本稿では，MBCF 自体を振り返るとともに，MBCF の 64bit アドレス拡張と Linux 版の開発基本方針と OS の違いから派生する実装上の注意点について述べた．intel CPU の載った PC を使用して，MBCF32/Linux は GbE では，MTU の値に関わらず，理論性能の上限スループットを達成し，一方 1440byte 遠隔書き込みの片道レイテンシは 120μsec 程度である．MBCF64/Linux は 10GbE を使った通常の MTU=1500 でも，理論性能値相当のスループットを達成し，1440byte 遠隔書き込みの片道レイテンシは 20μsec 程度である．なお，性能測定の詳細については，別稿で改めて報告する予定である．このように，高スループットと低レイテンシを両立し，高機能な共有メモリ操作を実現し，ロックフリーの不可分共有メモリ操作が可能な MBCF 通信機構を広く使用されている Linux 上で動かすことに成功した．なお，サポートした CPU としては IA32, AMD64 (intel64) のみではなく，ARM の Aarch32 と Aarch64 にも対応した．興味のある方々には，Loadable Module の配布を計画している．

謝辞　MBCF/Linux を本格的に開発する契機となった共同研究を実施していただいた三菱電機株式会社先端技術総合研究所のみなさまに深く感謝申し上げます．また，Linux の KASLR 機能について教えていただいた筑波大学の新城靖准教授にも深くお礼申し上げます．日頃の研究活動と研究室運営を手助けしてくれている松本研究室の学生のみなさまにも心より感謝いたします．

参考文献

[1] 松本 尚, 他: 汎用超並列オペレーティングシステム: SSS−CORE --- ワークステーションクラスタにおける実現 ---. システムソフトウェアとオペレーティングシステム研究会報告 No.73-20, 情報処理学会, pp.115--120 (August 1996).
[2] 松本 尚: 次世代オペレーティングシステム SSS-PC の開発 --- IA32 用カーネルアーキテクチャ ---. ITX2002 Summer 論文集（CDROM），情報処理振興事業協会, (June 2002)
[3] Matsumoto, T. et al.: MBCF: A Protected and Virtualized High-Speed User-Level Memory-Based Communication Facility. Proc. of the 1998 ACM Int. Conf. on Supercomputing, pp.259--266 (July 1998).
[4] Matsumoto, T.: A Study on Memory-Based Communications and Synchronization in Distributed-Memory Systems. Dissertation Thesis, Graduate School of Science, Univ. of Tokyo (February 2001).
[5] 松本 尚, 他: 100BASE-TX によるメモリベース通信の性能評価. コンピュータシステムシンポジウム論文集, 情報処理学会, pp.101--108 (November 1997).
[6] C. B. Stunkel, D. G. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao. The SP1 High-Performance Switch. In Proc. of SHPCC ‘94, May 1994.
[7] M. Snir et al. The communication and parallel environment of IBM SP2. IBM Systems Journal, 34(2), 1995.
[8] T. von Eicken, et al. Active Messages: A Mechanism for Integrated Communication and Computation. In Proc. of the 19th ISCA, pages 256-266, May 1992.
[9] D. E. Lenoski et al. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In Proc. of the 17th Int. Symp. on Computer Architecture, pages 148-159, May 1990.
[10] J. Kuskin et al. The Stanford FLASH Multiprocessor. In Proc. of the 21st Int. Symp. on Computer Architecture, pages 302-313, April 1994..
[11] 松本 尚, 他: 超並列計算機上の共有メモリアーキテクチャ. 信技報, CPSY 92-26, pp.47--55 (August 1992).
[12] 松本 尚: Memory-Based Processor を使用した汎用超並列計算機の基本アーキテクチャ. 並列処理シンポジウム JSPP ‘94 論文集, pp.409--418 (May 1994).

ⓒ 2021 Information Processing Society of Japan